OpenKnowledge

FP6-027253

Extension of LCC as a Visualisation Language

David Dupplaw[1], Dave Robertson[2], Madalina Croitoru[1],
Srinandan Dasmahapatra[1], Bo Hu[1], Paul Lewis[1], and Liang Xiao[1]

[1] School of Electronics and Computer Science, University of Southampton, UK
[2] School of Informatics, University of Edinburgh, UK

Report Version: final
Report Preparation Date: 31.1.2007
Classification: deliverable D5.1
Contract Start Date: 1.1.2006 Duration: 36 months
Project Co-ordinator: University of Edinburgh (David Robertson)
Partners: IIIA(CSIC) Barcelona
Vrije Universiteit Amsterdam
University of Edinburgh
KMI, Open University
University of Southampton
University of Trento

Visualisation and Message Markup in OpenKnowledge

David Dupplaw
dpd@ecs.soton.ac.uk

Dave Robertson
dr@inf.ed.ac.uk

Madalina Croitoru
mc3@ecs.soton.ac.uk

Srinandan Dasmahapatra
sd@ecs.soton.ac.uk

Bo Hu
bh@ecs.soton.ac.uk

Paul Lewis
phl@ecs.soton.ac.uk

Liang Xiao
lx@ecs.soton.ac.uk

January 31, 2007

Abstract

OpenKnowledge relies on the interaction of peers in a peer-to-peer network to provide constraint satisfaction of constraints on interactions within an interaction model. Currently, if constraints in the interaction model fail, with no alternative during a back-tracking operation, the whole interaction fails. In certain cases this is the correct thing to do. In other cases a human could be introduced in the process to act as an agent that will solve the constraint manually, thereby allowing the interaction to continue.

In this document we suggest ways in which this interaction may occur. This has effects on the design of parts of the system. Therefore we describe how the markup of the interaction state should be passed through the messages so that the visualisation may appear correctly. An example involving multimedia markup is also presented.

We also describe other non-interactive visualisation operations, both online and offline, that can be implemented in the system.

1 Introduction

The issue of visualisation of information when sharing knowledge in distributed systems has not featured prominently in debates in the knowledge engineering community. One reason for this is the ubiquity of visualisation and the diversity of visualisation tools. Another reason is that one might assume the visualisation issue is readily solved: all we do is use our favourite visualisation tool for the

type of information we wish to view. In this paper we demonstrate that in true distributed knowledge sharing (of the sort we need for activities like large scale, open semantic webs) the visualisation problem is non-trivial - in the worst case insoluble - for conventional systems. We then provide a means of avoiding the worst case, to achieve a practical solution in a peer-to-peer setting.

Suppose that we have two existing Web services with API's available (*e.g.* via WSDL): the first is one that requires as input someone's weight and height and then calculates as output his or her body mass index; the second is a lifestyle recommendation service that will generate as ouptut some life-enhancing suggestions based on the input of a person's body mass index. As engineers, we would like to combine these services so that people can offer their weights and heights and get lifestyle recommendations without having to go to the trouble of accessing each service separately. The traditional method of tackling this problem is simply to build a new Web service that calls the other two, with a user interface appropriate to our new service. The user interface (all of it specific to our new service) might prompt the person for weight and height; then call the BMI calculation service; then call the lifestyle service; then display the recommendations back to the person. Stripped of the details necessary to get the job done in a Web environment, this simply is function composition:

$$lifestyle\_from\_bmi = display(lifestyle(bmi(input(weight), input(height))))$$

where $lifestyle\_from\_bmi$ is the function giving us our new service; $lifestyle$ and bmi are the functions corresponding to calls to our existing two services; $input$ is a function allowing user input; and $display$ is a function providing a visualistaion to the user of the $lifestyle$ output.

This works for a single application but it is limiting because the $input$ and $display$ functions are built purely for the $lifestyle\_from\_bmi$ service, so each time we think of a new service we must build the interface for it anew. There are far too few interface (and Web service) designers in the world to construct all the bespoke interfaces we need.

Service composition languages (such as OWL-S) offer a radically different solution to the service composition problem. In this view, the inputs and outputs of services would be typed, so an automated system could detect which input to bmi was of type $height$ and which was of type $weight$, and might also (modulo ontology matching) be able to infer that the output of bmi could supply the input to $lifestyle$. If we also could infer the visualisation necessary for communication with users then this would solve the problem of having to write visualisations. We might infer for example that an input of type $weight$ could be supplied via an interface that asked the question "What is your weight?" and accepted a response typed into an edit field. Unfortunately this is impractical as a general solution because:

- Type information must be very specific in order to identify visualisations reliably. In our example, perhaps a sliding scale might be better, since weight in kilos varies between 0 and (say) 500, but to infer this requires

more information than we could expect reliably to obtain in practice via typing.

- The interaction can influence the visualisation. In our example, we might want the display of lifestyle recommendations to say things like "based on your calculated BMI of 31, given that you said you were 1.7m tall we recommend you take more exercise". This requires the visualisation of the output of the $lifestyle$ service to be presented in the context of the inputs and output of the bmi service.

From the above, some aspects of the visualisation of knowledge must be specified in addition to bare knowledge services and, furthermore, the specification of visualisation may apply across service interactions. A language in which interactions are described therefore appears essential in order to tackle this problem. One such language is the Lightweight Coodination Calculus being used as part of the OpenKnowledge project.

OpenKnowledge is a system for the execution of distributed applications, where the applications' functionalities reside on different computer systems. The network connecting these systems is a peer-to-peer network, although OpenKnowledge is largely agnostic to the underlying implementation. Interaction models are defined as constrained message passing operations; that is, message passing occurs between two peers in the network if some constraint can be satisfied. In effect, this pushes the implementation of the service functionalities into a constraint on the sending of the result message.

In OpenKnowledge we make these interaction models the currency of the system, sharing, transmitting and composing them. A simple interaction model formalised in LCC is shown in Model 1. This has an agent in role1 sending a message to an agent in role2, if the variable C can be created. The agent in role 2 simply accepts the message.

$$\begin{array}{l} \mathbf{a}(\text{role1}, ID) :: \\ \quad m1(C) \;\Rightarrow\; \mathbf{a}(\text{role2}, ID2) \leftarrow \mathbf{find}(C) \\ \\ \mathbf{a}(\text{role2}, ID2) :: \\ \quad m1(C) \;\Leftarrow\; \mathbf{a}(\text{role1}, ID) \end{array} \tag{1}$$

During this simple interaction we have a constraint on the message sending operation. In the execution of such a protocol the constraint will invoke some code to side-effect C. Depending on the nature of the code, this may be achievable automatically, but in the case that it is not we have a number of choices:

- Constraint is unsatisfiable: If the code cannot find a value for C, the code may simply say the constraint is unsatisfiable, and the interaction protocol will need to back-track, or in Model 1 it will fail.

- Do some matching: If C is of a particular type, but we are unable to create objects of that type, the constraint code may invoke an ontology matcher to mediate the variable and the constraint code [ZAvH06].
- Ask someone: Perhaps the code is unable to generate C automatically, but failure of the interaction model is undesirable, so we may invoke a visualisation to ask a user for the value of C.

There are a number of difficulties associated with such visualisation; for example, where should the visualisation appear (on which peer in the network) and how should the visualisation appear?

Defining how the visualisation should appear relies very much on the content of the object to visualise. The content of such objects needs to be passed to the visualiser via the interaction state. Markup of the interaction state will involve the markup of those objects which the visualiser will be able to use to formulate an appropriate visualisation. In section 5 we describe how the markup for objects can be achieved.

In the following sections, we shall introduce the reason for visualisation and how it has an impact on the way in which messages and objects are marked up. In section 7 we also discuss related visualisation techniques for visualisation of the state of the interaction, for user feedback purposes. As a preliminary to this, we describe in the next section the basic style of visualisation proposed for OpenKnowledge, using our BMI example for illustration.

2 Basic Visualisation in LCC

In the previous section we explained why it is difficult, when composing services, to divorce visualisation from the interaction of which it is a part. The issue then is whether we can include with our specification of interaction a specification of its accompanying visualisation. In doing so, we prefer to maintain some separation between the specification of the dynamics of interaction and its visual aspects, so that we can re-use interactions with different visualisations. We also want to use an abstract specification language for the visualisation component, to avoid the need to have numerous device-specific visualisations for each interaction model. The most direct way to achieve this is to associate visualisation specifications with constraints in the interaction model, as we shall explain using our BMI example.

Suppose that the engineer of our BMI service (the first of the two in our earlier example) decides to use LCC to specify a preferred form of interaction with the service. There are two roles: the role of the BMI supplying service ($getBMI$) and the role of its client ($bmiClient$). In the interaction the client asks for a BMI, supplying height and weight information as part of the request, and receives a BMI estimate calculated by the service.

$$
\begin{array}{l}
\mathbf{a}(\text{getBMI}, X) :: \\
\quad ask\_bmi(H, W) \Leftarrow \mathbf{a}(\text{bmiClient}, Y)\ then \\
\quad bmi\_estimate(B) \Rightarrow \mathbf{a}(\text{bmiClient}, Y) \leftarrow bmi(H, W, B) \\
\\
\mathbf{a}(\text{bmiClient}, Y) :: \\
\quad ask\_bmi(H, W) \Rightarrow \mathbf{a}(\text{getBMI}, X) \leftarrow weight(W)\ and\ height(H)\ then \\
\quad bmi\_estimate(B) \Leftarrow \mathbf{a}(\text{getBMI}, X)
\end{array}
\tag{2}
$$

The definition above is independent of visualisation but we can identify visualisation related components of the specification (constraints that might be satisfied interactively or messages that might be visualised upon receipt). We use the relation $visual(Term, Visualisation)$ to connect a term from the interaction model to an appropriate abstract description of visualisation. For our example this might be:

$$
\begin{array}{l}
visual(weight(W), qask('What\ is\ your\ weight\ in\ kilos?', W)) \\
visual(height(H), qask('What\ is\ your\ height\ in\ metres?', H)) \\
visual(bmi\_estimate(B), message(['Your\ BMI\ is', B]))
\end{array}
\tag{3}
$$

These visualisation descriptions now allow the LCC interpreter to know when a visualisation needs to occur and the form of visualisation that is important. However, this visual relation does not explicitly state *how* the visualisation should be presented to the user.

Following a similar approach to the lifestyle advisor service gives us the following interaction model

$$
\begin{array}{l}
\mathbf{a}(\text{lifestyleAdvisor}, X) :: \\
\quad tell\_bmi(B) \Leftarrow \mathbf{a}(\text{lifestyleClient}, Y)\ then \\
\quad lifestyle\_advice(L) \Rightarrow \mathbf{a}(\text{lifestyleClient}, Y) \leftarrow lifestyle(B, L) \\
\\
\mathbf{a}(\text{lifestyleClient}, Y) :: \\
\quad tell\_bmi(B) \Rightarrow \mathbf{a}(\text{lifestyleAdvisor}, X) \leftarrow bmi(B)\ then \\
\quad advised(L) \leftarrow lifestyle\_advice(L) \Leftarrow \mathbf{a}(\text{lifestyleAdvisor}, X)
\end{array}
\tag{4}
$$

and we could add to it the following visualisation definitions in order that the interpreter can question the user for their BMI and present the list, L, of lifestyle advice.

$$
\begin{array}{l}
visual(bmi(B), qask('What\ is\ your\ bmi?', B)) \\
visual(advised(L), message(L))
\end{array}
\tag{5}
$$

Now suppose that we want to describe the more complex interaction described in the Introduction involving both services. The roles for $getBMI$

and $lifestyleAdvisor$ need not change but we can write a new role for client, $bmiLifestyleClient$, that allows it to interact in sequence with both services:

$$\begin{array}{l}\mathbf{a}(\text{bmiLifestyleClient}, X) :: \\ \quad ask\_bmi(H, W) \Rightarrow \mathbf{a}(\text{getBMI}, X1) \leftarrow weight(W) \; and \; height(H) \; then \\ \quad bmi\_estimate(B) \Leftarrow \mathbf{a}(\text{getBMI}, X1) \; then \\ \quad tell\_bmi(B) \Rightarrow \mathbf{a}(\text{lifestyleAdvisor}, X2) \; then \\ \quad advised(H, L) \leftarrow lifestyle\_advice(L) \Leftarrow \mathbf{a}(\text{lifestyleAdvisor}, X2)\end{array} \tag{6}$$

We then add the following visualisation definition (replacing those above) to give:

$$\begin{array}{l}visual(weight(W), qask('What \; is \; your \; weight \; in \; kilos?', W)) \\ visual(height(H), qask('What \; is \; your \; height \; in \; metres?', H)) \\ visual(advised(H, L), message(['Based \; on \; height', H, L]))\end{array} \tag{7}$$

This describes the basic form of visualisation used to accompany LCC in OpenKnowledge. It provides a simple, yet effective, way to allow interleaving of interaction with visualisation - thus allowing LCC itself to provide a standard backbone for attaching visualisation to services and (importantly) also to compositions of services.

It is important to note that these visual relations do not specify how the visualisation will occur, they simply provide an archetype of the visualisation that is required. It is important that the peer that will be performing the visualisation is able to choose the best way to perform the visualisation, and the semantic description that accompanies any objects may be used to infer the specifics of the visualisation procedure.

In the following sections we describe in more detail pros and cons of the approach.

3 Complex Visualisation

In the previous section we described how specific visualisation cues can be provided as part of an interaction protocol by specifying relations between a term in the interaction model and some visualisation declaration. The visualisation declaration is related to the interaction model but is not part of it; this is important as the interaction may or may not invoke the visualisation, depending on whether that constraint can be automatically satisfied or not. It is only in the case when the constraint cannot be satisfied automatically that visualisation occurs based on this relation.

In the previous section the term *qask* was invented to represent the visualisation that would ask a user a question. However, it is foreseen that richer forms of visualisation will be required for richer objects.

Let us take the example of a service, deployed as an OpenKnowledge component, that provides a place where users can watch and comment on videos. The basic function is that a user may search for video files based on some keyword tagging, watch the video and highlight parts of the video providing their comments. The interaction model is given in Model 8.

$$
\begin{array}{l}
\mathbf{a}(\text{videoServer}, V) :: \\
\quad (findVideo(K) \Leftarrow \mathbf{a}(\text{videoFinder}, VF)\ then \\
\quad videos(V_L) \Rightarrow \mathbf{a}(\text{videoFinder}, VF) \leftarrow searchVideoDB(K, V_L)) \\
\quad or \\
\quad (comment(C, V) \Leftarrow \mathbf{a}(\text{videoFinder}, VC)\ then \\
\quad commentSaved(C, V) \Rightarrow \mathbf{a}(\text{videoFinder}, VC) \leftarrow saveComment(C, V)) \\
\quad then \\
\quad \mathbf{a}(\text{videoServer}, V) \\
\\
\mathbf{a}(\text{videoFinder}, VF) :: \\
\quad findVideo(K) \Rightarrow \mathbf{a}(\text{videoFinder}, VF) \leftarrow getKeywords(K)\ then \\
\quad videos(V_L) \Leftarrow \mathbf{a}(\text{videoServer}, VS) \leftarrow selectVideo(V_L, V)\ then \\
\quad comment(C, V) \Rightarrow \mathbf{a}(\text{videoServer}, VS) \leftarrow getVideoComment(V, C)\ then \\
\quad commentSaved(C, V) \Leftarrow \mathbf{a}(\text{videoServer}, VS)
\end{array}
\tag{8}
$$

There are certain constraints in this model that can only be satisfied with the direct input of a user (*getVideoComment* for example). The visualisation relations may be configured for this model as follows:

$$
\begin{array}{l}
visual(getKeywords(K), qask(\text{`}Enter\ some\ keywords', K)) \\
visual(selectVideo(V_L, V), choose(\text{`}Select\ a\ video', V_L, V)) \\
visual(getVideoComment(V, C), vidServerDisplay(V, C))
\end{array}
\tag{9}
$$

The first two visual terms (*qask* and *choose*) can be considered part of the underlying OpenKnowledge visualisation library; they simply ask a question and accept a string answer, or allow the choice of one from a list. However, the very specialised video display that allows users to peruse a video, highlight parts and comment on them, is very likely to be visualisation that has been supplied by the service provider. Because it is not part of the underlying OpenKnowledge library it will have to be installed by the user. In section 4 we describe how specialised visual components can be handled by this visual relation abstraction.

4 Framework for Visualisation

The visual relation approach provides the ability to map visualisation requests to actual visualisation implementations. This provides a number of advantages:

- *Device Portability*: The device on which the visualisation is to take place can choose the best method for providing the interaction. For example,

qask may provide a text box on a workstation, but may prompt for voice input on a mobile phone.

- *User Preference Support*: A user may prefer particular ways in which to perform a particular type of interaction, or may deny particular types of visualisation. It can also provide multiple visualisations for particular visual terms, allowing the user to select (or perhaps automatically determine) which visualisation to use for particular visual terms.

- *Extensibility*: The language of visualisation can be extended, providing a means for adding new visualisation for the system. New implementations of visualisation can be made available for a user to download.

The framework for visualisation needs to provide the means for mapping the visualisation term to the implementation of the visualisation. Again, this can be provided by a simple set of relations that map the visual term to a set of implementations.

Should an interaction model contain a visual term that the client does not understand, the client can provide the user with the option to download an implementation from a list sourced from an external repository of implementations. Such downloads may be performed in advance of the interaction execution too (say, during subscription to a role). The repository in this case may be a validated set available from a trusted website (say the OpenKnowledge website), or potentially from a discovery procedure on the OpenKnowledge network itself.

The video server example in Model 8 uses a service-specific visualisation, and uses the service-specific visual term $vidServerDisplay(V, C)$. When the interaction model execution requires visualisation for this visual term, the user is prompted to download the visualisation implementation if they wish. If they decline the download, the interaction will fail. The download can be certificated to assure authorship. Because the visualisation is mapped via the visual term, the user is free to download third-party implementations of the visualisation, allowing community-driven growth of the visualisation repository.

The specific visualisation, $vidServerDisplay$, will require some video to present to the user. Unlike the simple *qask* visual term that takes a string (the question) and the variable to fill, the visualisation $vidServerDisplay(V, C)$, in Model 8, takes the term V which represents a video file. Clearly, video cannot be sent within the messages themselves, as it may be many mega-bytes in size. Similary, other applications dealing with large multimedia objects will need to transfer the objects, so in Section 5 we describe a means for marking up these large objects and other multimedia in such a way to minimise transfer of data, by only transferring meta-data until the raw data is needed. Section 6 considers some extensions to this system that might provide some automatic visualisation selection by reasoning.

5 Markup for Interaction and Visualisation

The OpenKnowledge system is all about sharing knowledge that is marked up using ontological structures. The base layer for ontological markup will be RDF/OWL [MvH04] which will allow ontological mapping to take place at various intersection points in the system. This markup is critical for facilitiating interoperation between disparate systems that are utilising different vocabularies. For example, two independent processes that understand different vocabularies will only be able to interoperate if an ontological mapping can be calculated that will allow the two systems to communicate. This interoperability is also necessary to allow visualisations of data that have been presented in a different language.

There is a tendency to consider multimedia data as different to other data, such as numerical or textual data; however, multimedia data can be marked up in much the same way as non-multimedia data. Indeed, the less explicit nature of the semantics of multimedia data means that it is almost always necessary that it is marked up. There are already well established methods for marking up such data and as long as ontological expressions of these markup languages can be sought, the multimedia data can be used much like any other. For example, MPEG-7 [MKP02] provides a good framework for the markup of multimedia data. Initially, this was defined as XML-schemas, however, work has been undertaken that resulted in an OWL version of the MPEG-7 schema [Hun01] and this could provide the basis of a multimedia markup language for OpenKnowledge. Of course, if another language was utilised by some other component, interoperability can still be obtained as long as a mapping from that language to MPEG-7/OWL is available or computable.

It is perhaps worth mentioning that, in most cases, data can be transmitted alongside the markup - a floating point number representing a monetary value is probably smaller in bytes than the markup to indicate it represents currency. However, in some cases and moreso with multimedia data, the data will be many megabytes but processes acting upon the data may only require the metadata: it could be unnecessary to pass the data around in the messages. Instead, the messages might contain the data markup with some extra information to indicate to the peer how to retrieve the large block of data. This information might well consist of another interaction model that provides a means to contact, negotiate and download the data. Such meta-models could be automatically instantiated when the deferred elements of the interaction state are accessed remotely, thereby providing fast throughput of messages when the large data items are not required, yet requiring no authoring overhead in the creation of the interaction model when such objects are required.

5.1 Message Markup

One of the important issues when designing the communication layer within the system, is to ensure that all markup is transmitted across the network in a portable and re-useable way; it is a fundamental functionality of the Open-

```
@prefix : <http://www.openk.org/message#> .
@prefix mm: <http://www.openk.org/multimedia#> .
@prefix bio: <http://www.inf.ed.ac.uk/onto/bioinf#> .

:messageContent
    :interactionModelType "LCC";
    :interactionModel """r( role1, initial )  a( role1, id ) :: ...""" ;
    :interactionState [
        :symbolTable [
            :symbol [
                :name "I";
                :value [
                    mm:type mm:Image;
                    mm:width 320;
                    mm:height 240;
                    bio:hasProtein "NC";
                ]]]] .
```

Listing 1: An OpenKnowledge message marked up with TURTLE

Knowledge system. Because all our markup will be in the form of RDF (even if that RDF is OWL), then it seems logical to use RDF for message markup, as it will keep all ontological markup as first-class citizens in the representation. Using XML is a possibility, but all markup would then be demoted to strings so that they may be passed around the system requiring further processing overhead if reasoning is to take place.

There are many ways to represent RDF, but TURTLE (Terse RDF Triple Language) [Bec] is by far the most usable. TURTLE is a simple, plain-text, punctuated representation of RDF that has low parsing overheads and a small size, both of with RDF/XML lacks [Bec03]. Using TURTLE allows our markup to remain first-class (ideal for reasoning), while also providing us with usability and low-overheads. TURTLE is widely supported by RDF libraries like Sesame [BKvH02] and Jena [CDD+03], so incurs almost no overhead in development.

Listing 1 shows an example of how a message might be encoded with TURTLE. This message contains an interaction state that contains objects that have ontological markup. The markup shows that the object I is an image (defined by the ontological type `mm:Image`), and has dimensions 320x240. It is also marked up with concepts from another ontology, the `bio` ontology that is defined in the TURTLE header. This gives some further information about the image. All these concepts and literals remain first-class in this message, because we are using RDF for the message interface. Unmarshalling of this message is straightforward when using RDF libraries like Sesame.

6 Visualisation Reasoning

In the previous sections we have introduced how the visualisation of constraints within the system is divided into two: the declaration of the visualisation and the device-specific implementation of the visualisation. In further versions of the system we might foresee reasoning taking place at either of these two interfaces to minimise the chances of the interaction model failing.

The use of the visual terms for abstracting the implementation from the declaration could allow for reasoning to take place in the case that an interaction model contains a constraint that, in the current context, is unsatisfiable and there has been no visualisation defined for the unsatisfiable constraint. It may be possible to reason over the current context and infer an appropriate visualisation to insert. Such a situation might arise if an interaction model has been authored inefficiently. Clearly, this will need prior knowledge of which visualisations are available and the types of data they deal with. The visualisation repository may be able to provide this.

Another situation where reasoning might be employed is in the case where a peer does not have an implementation to invoke for a particular visualisation request. Through inference and ontology mapping it may be possible to find a visualisation that the peer does have access to that may be able to provide the visualisation that is being requested. The final choice of visualisation will, of course, be user-driven.

7 Execution State Visualisation

Visualising the execution state is useful for user feedback and execution history perusal, both important for building a user's trust in the system, as well as for diagnosing faults during component authoring. This is a distinct problem to the problem of visualisation of constraints that are unsatisfiable automatically.

Allowing the user to see the execution state of a running interaction model is useful for providing the user with feedback to prove that their interaction model is actually executing, or for providing feedback on where the interaction model is slow. A simple block diagram of the roles within the model and some form of display that gives each role's progress would suffice. Because roles are orchestrated by a coordinator peer, peers in the model who wish to be informed of the current interaction state, for visualisation purposes, may quite simply register with the coordinator who will respond with interaction state objects when the state changes, allowing the user's peer to update its display. Using the coordinator event-driven paradigm could allow for peers outside the current interaction to 'listen-in' to the state of the interaction, and it may be necessary to enforce the coordinator to only provide state updates to those who are playing roles in the current interaction model.

Visualisation of execution history is a special case of visualisation of a running interaction model where the interaction model is at the final step of its interaction. The state should be visualisable in the same way as a running

execution model, allowing objects to be inspected and visualised using the visualisation mapping on the local peer.

It may be necessary to provide protection for interaction objects that are vunerable to malicious intent; for example, it is possible the interaction state may contain the maximum bid value for bidders in an auction, and these must not be visible to other bidders within the interaction. As always, careful building of interaction models may alleviate the problem, but in a large community-driven network, adding such support would make the system more trustable and easier to use.

8 Conclusions

In this document we have described the various methods of visualisation (user interaction) for the OpenKnowledge system. Visualisation during interaction execution may be invoked during constraint satisfaction when constraints cannot be satisfied automatically.

We have introduced the idea of dividing the definition of a visualisation into a declaration that declares an intent to visualise, and an implementation that supports the declaration on the specific device on which the peer is running. This allows the interaction model and implementation to remain discrete while maintaining maximum flexibility. We have described a framework in which this visualisation functionality can be provided in an extensible way encouraging open, community-driven expansion of the available visualisations.

To achieve the visualisation of complex, multimedia objects that cannot be delivered within the messages themselves, we have introduced the idea of RDF-based messaging that provides an extensible language into which ontological definitions of objects can be included where necessary. These definitions would provide meta-data for the multimedia objects and may include information on how to download the data, should the visualisation require it.

We have also described how the current and previous states of the interaction may be visualised at a peer, both for history perusal and for user feedback - both important for building users' trust of the system.

Acknowledgement

This work has been undertaken in the 'OpenKnowledge' Specific Targeted Research Project (STREP), sponsored by the European Commission under contract number FP6-027253.

References

[Bec] D. Beckett. Turtle - terse RDF triple language. *ILRT University of Bristol, first announced January 2004, http://www.ilrt.bris.ac.uk/discovery/2004/01/turtle/.*

[Bec03] D. Beckett. A retrospective on the development of the RDF/XML revised syntax. Technical Report 1017, ILRT, University of Bristol, 11 June 2003.

[BKvH02] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In I. Horrocks and J. Hendler, editors, *Proceedings of the First Internation Semantic Web Conference*, number 2342 in Lecture Notes in Computer Science, pages 54–68. Springer Verlag, July 2002.

[CDD+03] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: Implementing the semantic web recommendations. Technical report, HP Laboratories Bristol, HPL-2003-146, Dec. 24, 2003.

[Hun01] J. Hunter. Adding multimedia to the semantic web: Building an mpeg-7 ontology. In J. Euzenat I. F. Cruz, S. Decker and D. L. McGuinness, editors, *in SWWS*, pages 261–283, 2001.

[MKP02] JM Martinez, R Koenen, and F Pereira. Mpeg-7-the generic multimedia content description standard, part 1. 9(2):78–87, 2002.

[MvH04] Deborah L. Mcguinness and Frank van Harmelen. OWL web ontology language overview. W3C recommendation, W3C, February 2004.

[ZAvH06] W. ten Kate Z. Aleksovski, M. Klein and F. van Harmelen. Matching unstructured vocabularies using a background ontology. *Proceedings of Knowledge Engineering and Knowledge Management (EKAW)*, 2006.