UNIVERSITY
OF TRENTO

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

38050 Povo – Trento (Italy), Via Sommarive 14
http://www.dit.unitn.it

OPENKNOWLEDGE DELIVERABLE 3.4.:
SPECIFICATION OF ONTOLOGY MATCHING COMPONENT

Mikalai Yatskevich, Fausto Giunchiglia, Fiona McNeill
and Pavel Shvaiko

August 2007

Technical Report DIT-07-063

OpenKnowledge* Deliverable 3.4.: Specification of ontology matching component**

Coordinator: Mikalai Yatskevich[1]
with contributions from
Fausto Giunchiglia[1], Fiona McNeill[2], and Pavel Shvaiko[1]

[1] Department of Information and Communication Technology (DIT), University of Trento, Povo, Trento, Italy
`{yatskevi|fausto|pavel}@dit.unitn.it`
[2] The University of Edinburgh, Edinburgh, UK
`f.j.mcneill@ed.ac.uk`

Abstract This document provides a technical specification of the OpenKnowledge (OK) ontology Matching Component (MC). In particular, it discusses: (*i*) the MC logical architecture along with its constituent parts, (*ii*) the MC external interface to the other components of the OK system, and finally (*iii*) the MC physical architecture.

1 Introduction

The OpenKnowledge system is a peer-to-peer network of knowledge or service providers. Each computer in the network is a peer which can offer services to other peers. OK is viewed as an infrastructure, where we only provide some core services which are shared by all the peers, while all kinds of application services are to be plugged on top of it. These plug-in applications are called the OK Components (OKCs). Notice that the OKCs link services to the OK infrastructure and may not actually contain the services themselves.

Interaction between OKCs is a very important part of the architecture. By using the Lightweight Coordination Calculus (LCC) [13], developers are able to define the Interaction Models (IMs) that specify the protocol that must be followed in order to offer or use a service. OKCs are the ones in charge of playing the IM roles. Since there is no *a priori* semantic agreement (other than the IM), the ontology matching component is used to automatically make semantic commitments between the interacting parts.

The goal of this deliverable is to provide technical specification of the ontology matching component of the OK system. MC is designed to solve the

* OpenKnowledge is a 3 year long STREP project financed under the European Commission's 6th Framework Programme. See, `http://www.openk.org/` for details.

** The originally planned title of this deliverable as from the project proposal was "Specification of composite mapping engine". However, this new title better reflects current contents of the deliverable and needs of the project, and therefore, is used here.

semantic heterogeneity problem on the various stages of the OK interaction lifecycle. Specifically, in this document we focus on matching: (*i*) keywords in the IM annotations, (*ii*) terms in the role descriptions, and (*iii*) contents of the messages. In order to solve these particular matching problems we propose to exploit an architecture which is based on three major categories of matchers. These are *label*, *node*, and *structure preserving* matchers. Finally, we provide technical account of MC, by discussing its external interface to the other components of the OK system as well as its physical architecture.

The rest of the deliverable is structured as follows. Section 2 presents the lifecycle of interaction within the OK system. Section 3 describes the logical architecture of MC. Section 4 presents an external interface to MC as well as the data model for its constituent parts. Section 5 discusses the MC physical architecture. Finally, Section 6 summarizes the major findings of the deliverable.

2 Lifecycle of interaction

The OpenKnowledge system is designed to allow peers to search on a network for IMs that describe the interaction which they wish to initiate and to locate other peers to play the necessary roles in the interaction (with the initiating peer usually playing at least one role). Neither the IM nor the other peers need to be known beforehand, though this can be the case if desired. The matching algorithms described in the OK Deliverable 4.1 [6] facilitate the automatic interpretation of these IMs so that the initiating peer can determine whether the IM is really appropriate for its needs and peers potentially suitable for playing other roles can determine how they are able to fulfill the roles and whether the consequences of that role are compatible with their goals. The lifecycle of an interaction can be described in five steps as follows:

Step 1. The initiating peer must first locate an appropriate IM that results in the goals it wishes to satisfy. This IM can either be already known to it or can be found via the discovery component. A discovery component, using lightweight matching techniques, such as keyword matching, returns IMs satisfying the request. The description of a discovery component is out of scope of this deliverable (see the OK Deliverable 2.2 [15] for details). Once these potentially suitable IMs have been located, semantic matching is used to determine if they are appropriate for the task at hand.

Step 2. The discovery component (discussed in Step 1) will find potentially suitable peers through matching the role description in the IM with the descriptions that peers give to their capabilities.

Step 3. Potentially suitable peers are contacted and, if they are available and willing, will be sent a copy of the IM.

Step 4. Each peer will perform semantic matching to interpret the requirements and effects of the interaction. If they are happy with the consequences of the role and are able to fulfill the constraints, they will return this information.

Step 5. The suitable peers are ranked according to the trust values associated with them, and, in the advanced case of approximate matching, their matching scores. This trust value may come from the results of previous interactions with the peers and is out of scope of this deliverable (see [14] for details). The highest ranked peers are approached to play the roles in the IM.

3 The ontology matching component

The ontology matching component solves the semantic heterogeneity problem among different knowledge representation formalisms. This component offers match routines which produce correspondences between the labels, nodes of the graph-like structures and the LCC constraints [13]. This functionality is exploited by the OK system (in particular, by the Control Manager component) in at least three different phases:

- *Keyword matching* deals with the semantic heterogeneity in the IM descriptions, i.e., with matching keywords in the IM description and user query.
- *Term matching* deals with the structural heterogeneity in a role description. For example, with matching of methods: get_address(Full_Name) and get_address(Name, Surname).
- *Query/Answer matching* deals with the semantic heterogeneity arising from the statement of a query and the values returned in its answers. For example, the matching of needed for interaction module operation get_address(Stephen Salter) and the operation get_address(Salter, Stephen) that a particular peer can actually perform.

The ontology matching component is composed of the matchers of three kinds, namely:

- *Structure preserving matchers* are intended to match the LCC constraints or first order terms. Thus, for example, finding that journal(publication) = magazine(publication). They are employed in the peer recruiting process.
- *Node matchers* are intended to match the elements of the LCC constraints in the particular context. Thus, for example, finding that car $\neq$ automobile, if car is actually *part-of* train. Their results are further exploited by structure preserving matchers.
- *Label (element level) matchers* are intended to match the labels in the IM annotations, role elements and message contents. Thus, for example, finding that car = automobile. Their results are further exploited in interaction model selection and message content matching processes and reused by node matchers.

Figure 1 shows the logical architecture of the matching component. Solid lines stand for control flow. Small three rectangles stand for an external to the ontology matching component parts of the OK system, while everything inside the large rectangle represents the ontology matching component.

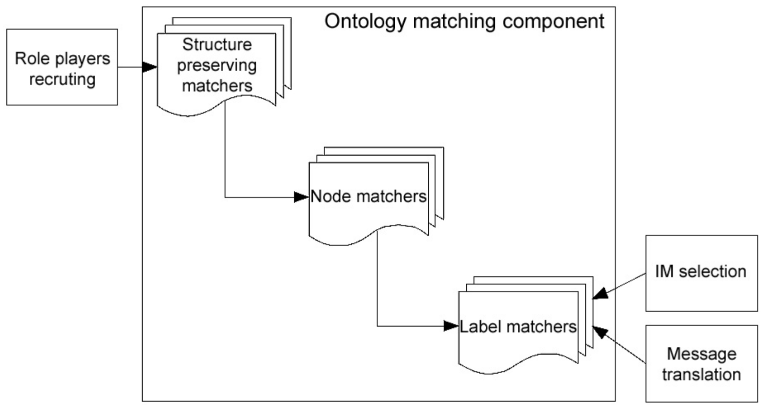

Figure 1. The MC logical architecture.

3.1 Element level matchers

Element level matchers are organized in a library. Currently, the library contains 18 matchers. Let us discuss these in some detail, by describing mostly their inputs and outputs, see [6,8,3,4] for more details.

Prefix is a string-based matcher. It checks whether one input label starts with the other one and returns the equivalence relation in this case.

Suffix is a string-based matcher. It checks whether one input label ends with the other one and returns the equivalence relation in this case.

Edit distance is a string-based matcher. It calculates the edit distance measure between two labels [17]. The calculation includes counting the number of the simple editing operations, such as delete, insert and replace needed to convert one label into another one and dividing the obtained number of operations with *max(length(label1),length(label2))*. If the value exceeds a given threshold the equivalence relation is returned.

NGram is a string-based matcher. It counts the number of the same ngrams (i.e., sequences of n characters) in the input labels. If the value exceeds a given threshold the equivalence relation is returned.

WordNet matcher is a knowledge-based matcher. It translates the relations provided by a lexical database WordNet [19,5] to semantic relations according to the rules described in detail in the OK Deliverable 4.1 [6] and in [10].

Leacock Chodorow matcher is a knowledge-based matcher. It exploits Leacock Chodorow semantic similarity measure [16]. It returns equivalence if the measure exceeds a given threshold.

Resnik matcher is a knowledge-based matcher. It exploits Resnik semantic similarity measure [21]. It returns equivalence if the measure exceeds a given threshold.

Jiang Conrath matcher is a knowledge-based matcher. It exploits Jiang Conrath semantic similarity measure [12]. It returns equivalence if the measure exceeds a given threshold.

Lin matcher is a knowledge-based matcher. It exploits Lin semantic similarity measure [18]. It returns equivalence if the measure exceeds a given threshold.

Hirst-St.Onge matcher is a knowledge-based matcher. It exploits Hirst-St.Onge semantic similarity measure [11]. It returns equivalence if the measure exceeds a given threshold.

Context vectors matcher is a knowledge-based matcher. It exploits context vectors semantic similarity measure [20]. It returns equivalence if the measure exceeds a given threshold.

WordNet gloss is a gloss-based matcher [8,7]. It compares the labels of the first input sense with the WordNet gloss of the second one. Specifically, it extracts the labels of the first input sense from WordNet. Then, it computes the number of their occurrences in the gloss of the second input sense. If this number exceeds a given threshold, the less general relation is returned.

WordNet extended gloss is a gloss-based matcher [8,7]. It compares the labels of the first input sense with the extended gloss of the second one. This extended gloss is obtained from the input sense descendants (or ancestors) descriptions in the *is-a* (or *part-of*) WordNet hierarchy. A threshold determines the maximum allowed distance between these descriptions. The type of relation produced depends on the glosses we use to build the extended gloss. If the extended gloss is built from descendant (or ancestor) glosses, then the more general (or less general) relation is produced.

Gloss comparison is a gloss-based matcher [8,7]. Within the matcher the number of the same words occurring in the two input glosses increases the similarity value. The equivalence relation is returned if the resulting similarity value exceeds a given threshold.

Extended gloss comparison is a gloss-based matcher [8,7]. It compares two extended glosses built from the input senses. Thus, if the first gloss has a number of words in common with descendant glosses of the second one (this is controlled by a threshold) then the first sense is more general than the second one or vice versa. If the corpuses (extended glosses) formed from descendant (or ancestor) glosses of both labels have a number of words in common then the equivalence relation is returned.

Semantic gloss comparison is a gloss-based matcher [8]. The key idea is to maintain statistics not only for the same words in the input glosses of the senses (like in the Gloss comparison matcher) but also for words which are connected through *is-a* and *part-of* relationships in WordNet. In semantic gloss comparison we consider synonyms, less general and more general concepts which may lead to better results.

MatchMiner is an approximate knowledge-based matcher. It uses the Semantic Web as a source of background knowledge in ontology matching. The core idea of this method is that, given two terms, an appropriate correspondence will be discovered by inspecting how these terms are related in ontologies available online.

PowerMap is an approximate knowledge-based matcher. It uses the ontologies on the Semantic Web as a background knowledge. The main differentiating features from other methods is that in PowerMap the matching process is driven by the task that has to be performed, more concretely by the input query asked by the user.

3.2 Node matchers

Currently, the node matchers library contains the only node matcher of the S-Match system [10]. S-Match determines the semantic relations holding among the nodes of the tree-like structures (e.g., classifications) by analyzing the meaning (concepts, not labels), which is codified in the elements and the structures of the input models. In particular, labels at nodes, written in natural language, are translated into propositional formulas which explicitly codify the labels intended meaning. This allows for a translation of the matching problem into a propositional validity problem, which is then efficiently resolved using (sound and complete) state of the art propositional satisfiability deciders.

3.3 Structure preserving matchers

Currently, the structure preserving matchers library contains 2 matchers [9]:

Exact structure preserving matcher is intended to match the trees with identical structures.

Approximate structure preserving matcher exploits the theory of abstraction as theoretical foundation and a tree edit distance algorithm for similarity computation between trees.

4 External interface

The ontology matching component offers the following interface to the other components of the OK system.

```
public interface Matcher {
// Matching methods

public MappingElement[] match (MappingElement[] previous, String
source, String target, Properties settings);

public MappingElement[] match (MappingElement[] previous, Object
source, Object target, STRUCTURE_TYPE type, Properties settings);
}
```

We call the former routine the string match routine and the latter, the object match routine. These routines provide a unified way to deal with different types of heterogeneity. The string match routine takes two strings (source and target), automatically recognizes implicitly described structures inside them, and produces the semantic relations between these structures as encoded within the mapping elements (MappingElement[]). These represent the matching result.

A mapping element (ME) is a 5-tuple $\langle ID_{ij}, N_i, N_j, R, C \rangle$, where ID_{ij} is a unique identifier of the given mapping element; N_i is the i-th object of the source structure; N_j is the j-th object of the target structure; R specifies a semantic relation (within, more or less general, equivalence and disjointness relations) which may hold between the concepts at nodes N_i and N_j; C is a similarity coefficient between 0 and 1 that stands for the plausibility of ME.

The object match routine takes source and target structures as Objects and the type of these Objects as the type parameter. Currently supported values of the type parameter are summarized in Table 1. For example, if the type parameter is set to LCC_CONSTRAINT, the matching component expects two tree representations of the LCC constraints as input parameters. The object match routine produces an array of MEs between the Objects of source and target structures.

Type name	**Description**
STRING	Parameters are interpreted as java strings to be matched by a label matcher.
LCC_CONSTRAINT	Parameters are interpreted as tree representations of the LCC constraints or as classes implementing the ITreeAccessor interface (see Figure 2 for more details).
PREPROCESED_TREE_AS_A_STRING	Parameters are interpreted as trees with linguistic preprocessing information attached; serialized into a XML string in the CTXML format [2].

Table 1. Types supported by the object match routine.

In the case that the given structures have already been partially matched (i.e., there is a subarray of MEs), both routines may reuse this information by exploiting the previous parameter in the routines definition in order to produce the rest of mapping elements faster.

Finally, the settings parameter contains the matcher specific properties to be passed to a particular matcher or a matching component as a whole.

Notice that the label, node and tree matchers have to implement the Matcher interface in order to be plugged into the OK system. Since the matching component itself implements the Matcher interface this allows the combination of the results of various matchers in a composite fashion.

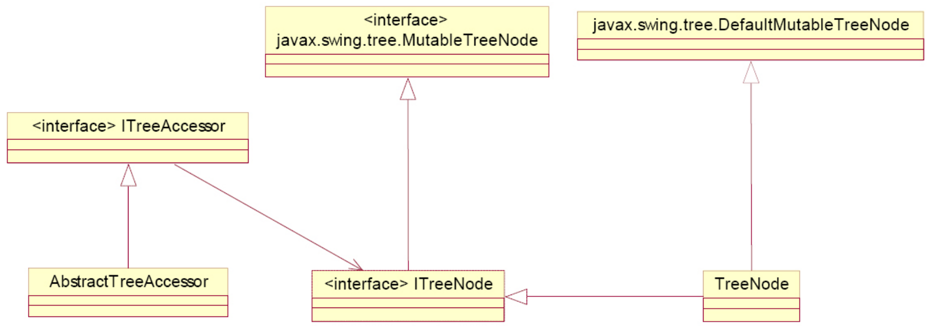

Figure 2. Class diagram of the OK matching component tree model.

Figure 2 provides a class diagram [1] of the tree model adapted by the OK matching component. The specific tree model is necessary since different node and tree matchers may store the trees they operate with, exploiting various implementations of the tree data structure. In order to guarantee the uniform access to various tree implementations, the ITreeAccessor interface is defined. Specifically, it provides the basic functionalities for managing trees. AbstractTreeAccessor provides the basic implementation of ITreeAccessor. The matcher developers are expected to provide their own implementations of ITreeAccessor. The implementations may however inherit AbstractTreeAccessor and reuse the basic functionalities implemented in it.

Below, we provide description of the methods of the ITreeAccessor interface:

- boolean contains(ITreeNode node)
  checks whether the given node is contained in the tree baked by ITreeAccessor;
- Set<ITreeNode> getAncestors(ITreeNode node)
  returns ancestors of the given node;
- Set<ITreeNode> getDescendants (ITreeNode node)
  returns descendants of the given node;
- Set<ITreeNode> getParents(ITreeNode node)
  returns parents of the given node;
- Set<ITreeNode> getChildren (ITreeNode node))
  returns children of the given node;
- ITreeNode getRoot()
  returns root of the tree;
- ITreeNode getMostRecentCommonAncestor(ITreeNode node1, ITreeNode node2)
  returns least common subsumer of the two given nodes;
- List<Object> getPostorderSequence()
  returns postorder tree traversal;
- List<Object> getPreorderSequence()
  returns preorder tree traversal.

The ITreeNode interface is defined to abstract matcher-specific tree node implementations. It extends standard SWING javax.swing.tree.MutableTreeNode interface. The basic implementation of ITreeNode, which is TreeNode, extends javax.swing.tree.DefaultMutableTreeNode class (see also Figure 2). Its functionalities may be inherited by the matcher-specific node implementations.

Below, we provide description of the ITreeNode interface methods:

- add(ITreeNode node1)
  removes node1 from its parent and makes it a child of this tree node;
- ITreeNode getFirstChild()
  returns the first child of the tree node;
- ITreeNode getLastChild()
  returns the last child of the tree node;
- ITreeNode getChildAfter(ITreeNode node1)
  returns the child of the tree node that goes after node1;
- ITreeNode getPreviousSibling()
  returns previous (left) sibling of the node;
- ITreeNode getRoot()
  returns the root of the tree;
- Object getUserObject()
  returns a user object stored in the node;
- boolean isRoot()
  checks whether the node is a root of the tree;
- Enumeration postorderEnumeration()
  returns postorder traversal of the tree starting from the node;
- Enumeration preorderEnumeration()
  returns preorder traversal of the tree starting from the node.

The ontology matching component works as an independent component and does not exploit any functionalities from the other OK components.

5 Physical architecture

The matching component is assumed to be deployed on any peer in the network. However, some peers, such as mobile terminals, may not have enough computational resources to perform the ontology matching operation. These peers may ask the dedicated ontology matching services deployed within the network to perform the ontology matching operation. Therefore, MC can be viewed as: (*i*) an integral part of the OK infrastructure on any peer, (*ii*) a service available to the other peers.

The UML deployment diagram [1] for the case (*i*) is depicted in Figure 3. Here, the peers are deemed to solve the matching problems locally while communicating with each other.

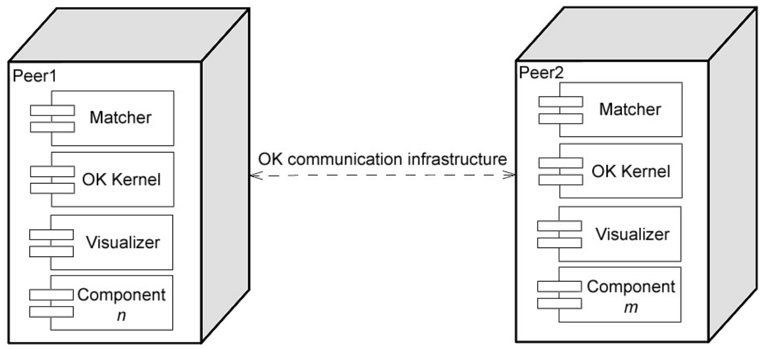

Figure 3. MC as part of the OK infrastructure on every peer.

The UML deployment diagram for the case (*ii*) is shown in Figure 4.

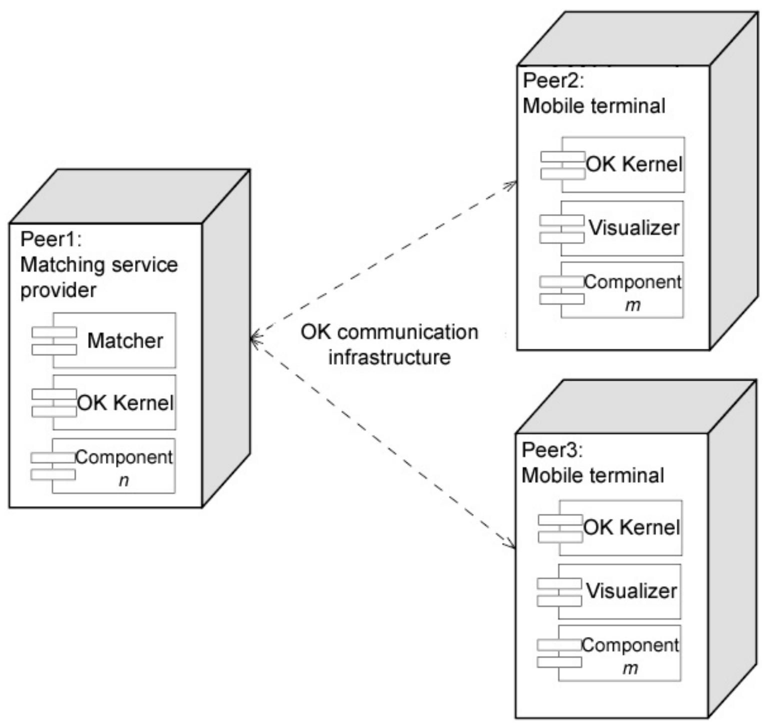

Figure 4. MC as a service available for the other peers.

Peer1 in Figure 4 has committed some of its computational resources to serve as a matching service provider. The peers who have not enough computational power to perform the resource consuming matching process, such as mobile terminals (Peer2 and Peer3), may request the matching service to perform the matching tasks for them. In this case all computations are performed on matching service provider and the results in terms of mapping elements are sent back to the requesting peers.

6 Conclusions

This document has provided a technical specification for the ontology matching component of the OpenKnowledge system. The component is designed to be easily extensible and exploits three categories of matchers: label, node, and structure preserving matchers. The technologies exploited for concrete matcher implementations are novel, especially those which concern the structure preserving matching. Through the first prototype we will be able to test these technologies in order to gain a better understanding of how they can fit together with the core idea of satisfaction of the OK system end-user requests.

References

1. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language user guide*. Addison-Wesley, Reading (MA US), 1998.
2. Paolo Bouquet, Antonia Donà, Luciano Serafini, and Stefano Zanobini. ConTeXtualized local ontology specification via CTXML. In *Proc. of the AAAI workshop on Meaning Negotiation*, Edmonton (CA), 2002.
3. Marco Calderan. *Semantic similarity library*. Bachelor thesis, The University of Trento, Technical report DIT-06-036, 2006.
4. Jérôme Euzenat and Pavel Shvaiko. *Ontology matching*. Springer, Heidelberg (DE), 2007.
5. Christiane Fellbaum. *WordNet: an electronic lexical database*. The MIT Press, Cambridge (MA US), 1998.
6. Fausto Giunchiglia, Fiona McNeill, Mikalai Yatskevich, Zharko Alekovski, Alan Bundy, Frank van Harmelen, Spyros Kotoulas, Vanessa Lopez, Marta Sabou, Ronny Siebes, and Annette ten Tejie. *OpenKnowledge Deliverable 4.1: Approximate Semantic Tree Matching in OpenKnowledge*. `http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D4.1.pdf`, 2006.
7. Fausto Giunchiglia, Pavel Shvaiko, and Mikalai Yatskevich. Discovering missing background knowledge in ontology matching. In *Proc. 16th European Conference on Artificial Intelligence (ECAI)*, pages 382–386, Riva del Garda (IT), 2006.
8. Fausto Giunchiglia and Mikalai Yatskevich. Element level semantic matching. In *Proc. ISWC Meaning Coordination and Negotiation workshop*, pages 37–48, Hiroshima (JP), 2004.
9. Fausto Giunchiglia, Mikalai Yatskevich, and Fiona McNeill. *Structure preserving semantic matching*. The University of Trento, Technical report, 2007.
10. Fausto Giunchiglia, Mikalai Yatskevich, and Pavel Shvaiko. Semantic matching: Algorithms and implementation. *Journal on Data Semantics*, IX:1–38, 2007.
11. Graeme Hirst and David St-Onge. Lexical chains as representation of context for the detection and correction malapropisms. In Christiane Fellbaum, editor, *WordNet: An electronic lexical database and some of its applications*. The MIT Press, Cambrige (MA), 1997.
12. Jay J. Jiang and David W. Conrath. Semantic similarity based on corpus statistics and lexical taxonomy. *The Computing Research Repository*, 1997.

13. Sindhu Joseph, Adrian Perreau de Pinninck, Dave Robertson, Carles Sierra, and Chris Walton. *OpenKnowledge Deliverable 1.1: Interaction Model Language Definition*. http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D1.1.pdf, 2006.
14. Sindhu Joseph, Carles Sierra, and Fausto Giunchiglia. *OpenKnowledge Deliverable 4.7: Trust models for open MAS*. http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D4.7.pdf, 2007.
15. Spyros Kotoulas and Ronny Siebes. *OpenKnowledge Deliverable 2.2: Adaptive routing in structured peer-to-peer overlays*. http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D2.2.pdf, 2006.
16. Claudia Leacock, Martin Chodorow, and George Miller. Using corpus statistics and WordNet relations for sense identification. *Computational Linguistics*, 24(1):1–40, 1998.
17. Vladimir Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Doklady akademii nauk SSSR*, 163(4):845–848, 1965. In Russian. English Translation in Soviet Physics Doklady, 10(8) p. 707–710, 1966.
18. Dekang Lin. Automatic retrieval and clustering of similar words. In *Proc. 17th International Conference of Computational Linguistics (CoLing)*, pages 768–774, Montréal (CA), 1998.
19. George Miller. WordNet: A lexical database for English. *Communications of the ACM*, 38(11):39–41, 1995.
20. Siddharth Patwardhan and Ted Pedersen. Using wordnet-based context vectors to estimate the semantic relatedness of concepts. In *Proc. of the EACL workshop on Making Sense of Sense: Bringing Computational Linguistics and Psycholinguistics Together*, Trento (IT), 2006.
21. Philipp Resnik. Using information content to evaluate semantic similarity in a taxonomy. In *Proc. 14th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 448–453, Montréal (CA), 1995.