Service Trading Using Conceptual Structures*

A. Puder, S. Markwitz and F. Gudermann

Department of Computer Science
University of Frankfurt
D–60054 Frankfurt, Germany
{puder,markwitz,florian}@informatik.uni-frankfurt.de

Abstract. An open distributed environment can be perceived as a service market where services are freely offered and requested. Any infrastructure which seeks to provide appropriate mechanisms for such environments has to include some mediator functionality to bring together matching service requests and service offers. The matching algorithm that the mediator must perform commonly builds upon an IDL–based type definition for service specification. We propose a type specification notation based upon conceptual graphs to support the cognitive domain of application users. In our framework the trader implements a matching algorithm as well as a learning algorithm which are tailored to service trading in open environments.

Keywords: Open distributed environments, type graphs, trading, service matching, service knowledge base, conceptual graphs.

1 Introduction

An open distributed environment can be perceived as a service market where services are offered and requested. A crucial aspect of these environments is the way services are *typed*. A type specification abstracts from specific internal details as it decouples the service usage from its implementation. Service types play an integral role during the mediation process. A software component known as a *trader* matches service offers and service requests based on their types.

There are several ongoing standardization efforts to develop an appropriate infrastructure supporting these environments (for example [5, 4, 6]). All these infrastructures focus on *interface definition languages* (IDL for short) as the main technique of service typing. An IDL is a low level specification of a precisely defined programming interface describing function names and input/output parameters in terms of signatures. These kind of types are meant to be handled by programmers and therefore are not suitable for application users.

With the emergence of high level services as found in the World Wide Web (see [1]) the need for an appropriate typing mechanism becomes evident. An application user (i.e. *not* a programmer) must deal with service types to lookup matching service providers. So far there exist only simple keyword based traders which allow only for a limited expressiveness of service types. We have chosen to develop a type specification based on conceptual graphs which are particularly useful to support the cognitive domain of the users.

* Appeared in *International Conference on Conceptual Structures (ICCS'95).* Springer

The paper is organized as follows: in section 2 we introduce concepts from the field of open distributed environments necessary for the scope of this paper. We present definitions for the terms *object graph* and *trader object* to motivate the need for type specifications of *User Types*. In section 3.1 we identify the requirements of a *learning model* and a *matching model* which will then be defined in sections 3.2 and 3.3 respectively. In these sections we also present appropriate algorithms. Finally a summary in section 4 concludes this paper.

2 Environment

The goal of this section is to motivate the need for an abstract type specification notation based on conceptual graphs. The starting point of our discussion will be the object graph model which serves as a suitable abstraction for an open service market where services are freely offered and requested. A crucial aspect of the object graph model is the typing of objects which is essential for supporting the matching process of service requests and offers. Finally we distinguish two levels of abstraction with respect to who — the programmer or the application user — is exposed to the type specifications. The premise of this paper is that currently there exists no suitable type notation which supports the cognitive domain of an application user.

2.1 Object graph model

The environment, which serves as a basis for this paper, builds upon the classical definition of an object (see [2]): *an object has state, behavior and identity; the structure and behavior of similar objects are defined in their common class. The terms instance and object are interchangeable.* Using this definition, a problem domain may be decomposed as a set of interacting and co–operating objects. A snapshot of such an object–based computation may be visualized as a directed graph, where nodes represent objects and arcs represent references. A *reference* (or arc) is therefore a referral of an object's identity. The direction of the arc determines whose identity is known to whom. For an object to hold a reference to another object means to know about the existence of this particular instance, allowing *operation invocations* (also commonly called method invocations). Thus a directed arc between two nodes (objects) represents the ability to invoke operations along the direction of this arc (i.e. the *service provider* is at the arc head, and the *requester* is at the tail). Service providers are also called *server objects* and service requesters are called *client objects.* A client object may be controlled by a (human) *user* who eventually decides the course of actions. The directed graph will be called an *object graph.*

An important requirement of the object model is that an object encapsulates data and code. The role of a *type specification* is therefore crucial in the sense that it should provide enough information to describe an object's behavior, yet conceal any implementation specific details. We assume that both references and objects are typed. Implementation details of a server object are irrelevant to a client. From a client object's perspective (the one holding the reference) a reference guarantees a contract that the server object must fulfill. Polymorphism occurs when the type of the reference is a super–type of the object to which it points. The server object therefore is a specialization to what the client expects, if it can fulfill the contract[2].

2.2 Trading and the role of types

In this section we assume some kind of type notation which allows the typing of references and objects as discussed above. Furthermore the object graph will be seen in the context of an open, distributed environment. By *open* we mean an environment spanning several administrative domains. Thus, the object graph and its modifications are to be seen as an abstraction of a service market, where services are freely provided and requested by independent parties.

In a distributed environment the object graph will generally be partitioned. A client object has only a limited view on the object graph, as global knowledge of its structure is generally impossible to acquire. References between objects induce a "knows–about–relation", it is also clear that without appropriate support from an underlying infrastructure a client can't see beyond a transitive closure of the references it holds (i.e. the partition of the object graph in which the client is embedded).

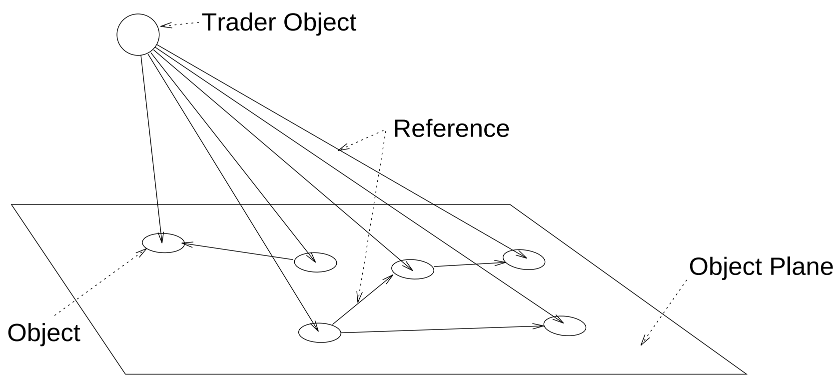

Fig. 1. The Trader Object possesses global knowledge of the object graph.

These considerations have led to system components like the ODP Trader (see [7]), which serve as a mediator between service requesters and service providers

[2] Polymorphism is often referred to as the "principle of substitutability" where an object of type A may be substituted by an object of type B without anything "bad" happening.

and therefore bridge the knowledge/visibility gap. The *trader* matches service requests with previously stored service offers and thereby helps to establish references in the object graph. The match process heavily depends on the precise definition of the type specification notation.

Within the object graph model introduced above, a trader can be modeled by an object as well (called the *trader object*, see figure 1). The trader object has a well known identity, so every object is given an implicit reference to this object (this fact is not shown in figure 1 to simplify the depiction). For a more detailed discussion see [9].

Conceptually the trader maintains a database of all service providers who have registered themselves previously. The database holds tuples each containing a *type specification* as one argument. The second argument represents a non–empty set of *addresses* of appropriate server objects providing the service described by the first argument. If a match of a request and a service offer succeeds, the trader uses the address to construct a reference which will be given to the client as the result.

2.3 System Types vs User Types

One can distinguish two different cases with respect to *when* a type specification is initially required: at *compile time* or at *runtime*. For compile time type notations there exists a wide range of notations based on interface signatures defined in some *interface definition language* (IDL for short). A type specification written in an IDL commonly lists a set of methods which are implemented by the server object. Special tools generate so–called *stubs* or *proxies* which eventually get linked to the client object. In terms of level of abstraction, an IDL is intended for programmers and is generally not seen or handled by users of the client object at runtime. In the following compile time types will be called *System Types*. Common examples for such notations are the CORBA–IDL and the DCE–IDL (see [5] and [4] resp).

In contrast a type specification notation used during runtime must be designed under different assumptions. The level of abstraction of the underlying objects is higher in the sense that the *user* determines at runtime the kind of service he or she wishes to use. Thus the programmer of a client object does not know at compile time the services the user eventually will use at runtime. Type notations suitable for runtime service specification will be called *User Types* (UT for short). A common technique for building such systems are *generic client objects* which are able to communicate with an a priori unknown server. Examples for such systems are the World Wide Web (WWW) or OpenDoc (see [1, 8]).

These systems have in common that different services can be provided at runtime without the need for a specific service type at compile time. Instead, a service provider has means to dynamically convey its particular user interface to the client via some sort of *graphical user interface* (GUI) description. The generic client is able to interpret these descriptions and to build and present an appropriate GUI for the end user. The user may interact with the generic client to invoke operations and to provide parameters embedded as widgets such as edit fields or check–boxes appearing in the GUI. These parameters are transfered to the service provider who takes appropriate actions to perform the request. For example, within the WWW system a GUI description is based on the *hypertext markup language* (HTML) which the generic client is able to translate into a visual presentation.

It is not immediately clear how a notation for User Type specifications should be defined. A type specification in this environment is more abstract than an IDL based specification. In particular it should support the cognitive domain of the users and not of the programmers. Our choice for User Type specification notation are *conceptual graphs* which in particular support the cognitive domain of the users. We do not compare conceptual graphs with other knowledge representation approaches. The reason for choosing conceptual graphs was their close relationship to natural language. We focus upon the non trivial mechanisms which the *trader* must perform during the mediation process in order to match service descriptions based on conceptual graphs. These mechanisms which the trader builds upon will be discussed thoroughly in the next section. The following table summarizes the differences between System Types and User Types.

	System Types	**User Types (UT)**
When initially required	At compile time.	At runtime.
Level of abstraction	Low. Supports fine grained objects manipulated by *programmers.*	High. Supports cognitive domain of *users* independent of implementation specific details.
Level of detail	High. Strict defined object interface based on signatures.	Possibly low. Vague and unprecise information due to the world of discourse.
Trading	Two phase trading: first service lookup then service usage. The traded service is always the desired one.	Multi phase trading: user may decide to switch back and forth between service lookup and service usage in order to obtain the desired service.
Supported kind of objects	Fine grained IDL–based typing of objects with a priori known service types.	Generic clients which handle a priori unknown service types (WWW, OLE2, OpenDoc).
Example notations	CORBA–IDL, DCE–IDL.	-

Table 1: Differences between System Types and User Types.

3 Trading through learning and matching

The basic interface that the trader offers its clients is that of a *service export* and a *service import* (see figure 2). A provider first has to export the type of service it wishes to advertise via the trader. The trader maintains a knowledge base of all registered service providers and therefore is able to map service types to appropriate providers. A requester on the other hands queries the trader for a suitable provider via a service import. An approximate portrayal of a service type is represented on the implementation level as a *conceptual graph*. Both import and export operations to the trader are formulated using conceptual graphs. In the following the service description passed as a parameter for an export operation will be called a *type graph* and the description passed as an argument for an import operation a *query graph* respectively.

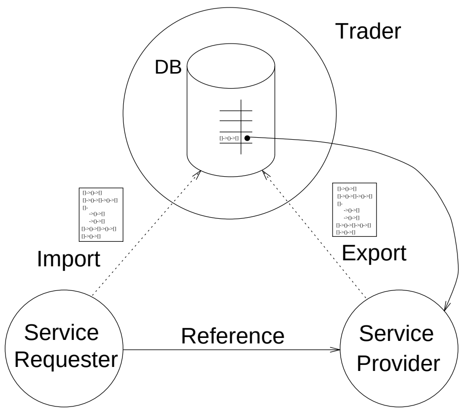

Fig. 2. Architectural overview: export and import of services.

The trader has to implement the import and the export operations. Translated to an AI context, a service export can be seen as a step of a *learning* process. Successive exports increase the quality of service descriptions in terms of decreasing semantic distance to a User Type. A service import on the other hand is implemented within the trader as a *match* process of conceptual graphs. The trader has to cope with the problem that the size (in terms of nodes) of a type graph may grow considerably due to the diversity of service descriptions. To handle this situation we have devised a *learning model* which represents a formal specification of the export operation described in section 3.3. Furthermore there is a need for a *matching model* which defines a metric expressing the degree of similarity between a query and a type graph and which serves as a basis for a formal specification of the import operation. This match is the major task that the trader must perform. The practical relevance of our proposal mainly rests upon the quality of this matching process. A step by step development of the match algorithm will be discussed in detail in section 3.2. The basic idea lies in the determination of an *akin index* which is a measure of the semantic difference of a query graph and a type graph. The best match (i.e. most appropriate service) is then decided by means of an *akin taxonomy* defined over all type graphs contained in the trader's knowledge base.

3.1 Requirements of the matching– and learning model

There is an immediate relation between the matching model and the learning model. Prior to the match of a query graph and a type graph (defined by the matching model), the trader must build up an approximation of a service type obtained by a sequence of learning phases (defined by the learning model). In section 3.2 we show that existing definitions of conceptual graphs do not satisfy the special requirements of a trader. One implication will be that weights need to be assigned to the nodes of a type graph. We identify the following requirements which a learning model must satisfy:

L1: The initial type graph describing a UT must be a "good" first approximation.
L2: One execution of a learning algorithm must either extend the type graph or adapt it in terms of a better approximation to the UT. The weights assigned to the nodes of a type graph should converge asymptotically to a final value.
L3: Eventually a sufficient approximation of the UT will be reached.

The semantic distance of a query graph and a type graph will by characterized by an akin index. We identify the following requirements for a matching model:

M1: The akin index denotes the quality of identical semantic context within a type graph and a query graph. A high akin index reflects major identical components and vice versa.
M2: The size of a type graph has no influence on the akin index. Increasing the size of a type graph merely reflects a more detailed description of a service (i.e. a closer approximation to the UT).
M3: The difference in akin indices is a true measure of the real semantic difference.
M4: It is possible to assign weights to the nodes of a query graph which will be taken into account by the match algorithm.

The matching model builds upon the learning model. The learning algorithm has to be repeated due to missing details or wrong weights (i.e. the approximation to a UT is insufficient), if the evaluation of the akin index does not reflect the real semantic distance. In order to properly motivate the way we assign weights to nodes in a conceptual graph we will first discuss the matching model prior to the learning model.

3.2 Fuzzy context matching

Practical experience has shown that *atomic conceptual graphs* (see [3]) are sufficient to express type graphs for approximation of UTs. The semantics of a simple conceptual graph can be defined through a mapping to a first order predicate formula (see ϕ operator defined in [11]). Using this mapping, example 1 shows a type graph and a query graph, as well as their semantics in terms of the existentially quantified first order formulas F_1 and F_2. The informal semantic of the type graph G_1 is that *there is a relational database which runs on AIX machines and possesses an SQL-interpreter*. The informal semantic of the query graph G_2 expresses the need for *something which supplies data security, runs on AIX machines and possesses an SQL-interpreter*[3].

Example 1. Let G_1 be a type graph and G_2 be a query graph.

G_1 :
```
[DATABASE]-
    ->(CHRC)->[RELATIONAL]
    ->(LOC)->[HOST:AIX]
    ->(PART)->[SQL-INTERPRETER]
```

$$F_1 : \exists x_1 \exists x_2 \exists x_3 \Big(DATABASE(x_1) \wedge RELATIONAL(x_2) \wedge HOST(AIX) \wedge SQL-INTERPRETER(x_3) \wedge CHRC(x_1, x_2) \wedge LOC(x_1, AIX) \wedge PART(x_1, x_3) \Big)$$

G_2 :
```
[SOMETHING]-
    ->(SUPP)->[DATA-SECURITY]
    ->(LOC)->[HOST:AIX]
    ->(PART)->[SQL-INTERPRETER]
```

$$F_2 : \exists x'_1 \exists x'_2 \exists x'_3 \Big(SOMETHING(x'_1) \wedge DATA-SECURITY(x'_2) \wedge HOST(AIX) \wedge SQL-INTERPRETER(x'_3) \wedge SUPP(x'_1, x'_2) \wedge LOC(x'_1, AIX) \wedge PART(x'_1, x'_3) \Big)$$

Every first order formula and therefore every conceptual graph has an intended semantic defined by an interpretation I over domain D. The interpretation maps every constant to a distinct element in D and all ground instances of every predicate symbol to *true* or *false*. Together with the usual semantics of the connectives, a first order formula can be assigned a truth value under an interpretation. Naturally the intended interpretation of a formula should be a model (i.e. evaluate the formula to *true*).

The akin index is obtained by determining the common subgraphs of both graphs. The predicate logic provides a proper framework for this operation. Given two conceptual graphs the common subgraphs are computed by mutually eliminating concept and relation nodes which are not contained in the other graph. Two nodes may be similar after a reduction specializing the nodes. This elimination process is sound due to the fact that the corresponding first order formulas only contain conjunctions. If the intended interpretation is a model for a formula then it is also a model for the formula obtained by removing one or more predicates. In example 2 the result of this elimination process is shown. The remaining common parts of the type graph and the query graph denote the *semantic intersection*. The akin index is computed based on the semantic intersection and the original query graph. This computation ignores the root concepts of those two graphs as the akin index should infer a metric of the similarity of two descriptions rather than of their root concept names.

Example 2. Let G_3 be the semantic intersection of G_1 and G_2

[3] To keep the size of this and the following examples within reasonable size, the service descriptions are not further refined. One such possible refinement for the database example would be some kind of product information.

```
G3 :  [SOMETHING]-                    (concurring context of G1 and G2)
        ->(LOC)->[HOST:AIX]
        ->(PART)->[SQL-INTERPRETER]
```

$$F_3 : \exists x_1 \exists x_2 \Big((SOMETHING(x_1) \wedge HOST(AIX) \wedge LOC(x_1, AIX)) \wedge (SOMETHING(x_1) \wedge SQL - INTERPRETER(x_2) \wedge PART(x_1, x_2)) \Big)$$

The similarity of the semantic intersection G_3 and the type graph G_1 can be computed by comparing their sizes in terms of number of nodes (see [13]). Employing this method it can be seen that (i) G_1 has been reduced in size to 66% to result in G_3 and (ii) to 66% of G_2 match with G_3 respectively. In example 3 a different type graph (labeled G_4) is given. Type graph G_4 also can be reduced to G_3 if it is to be compared with G_2. Now there is a 50% correspondence in size between G_3 and G_4 while there is still a 66% match between G_1 and G_3. The deviation obviously results from the higher level of detail of G_4. The measure presented in (i) violates the requirement **M2** of the matching model and is therefore not suitable.

Example 3. Let G_4 be a type graph.

```
G4 :  [DATABASE]-
        ->(LOC)->[HOST:AIX]
        ->(PART)->[SQL-INTERPRETER]
        ->(ATTR)->[OPTIMIZED-ACCESS]
        ->(SUPP)->[SCHEMA-EVOLUTION]
```

On the other hand the measure (ii) also is insufficient, as there is no justification that the akin index is solely based on the size of the semantic intersection. Summing up the arguments presented above, the mere size in terms of nodes of the semantic intersection is not sufficient as it disregards the context of the query graph within the type graph.

Our solution to this problem is the assignment of weights to the nodes in a conceptual graph. Naturally the weights reflect the importance of each node in a graph. In example 4 the notation is slightly extended to allow for a weight specification of each node. The nodes of the type graph G_1 and the semantical intersection G_3 are now assigned individual weights.

Example 4. Let G_1' and G_2' be conceptual graphs with associated weights.

```
G1' : [DATABASE]-
        ->(CHRC,w1)->[RELATIONAL,w2]
        ->(LOC,w3)->[HOST:AIX,w4]
        ->(PART,w5)->[SQL-INTERPRETER,w6]

G2' : [SOMETHING]-
        ->(SUPP,w1')->[DATA-SECURITY,w2']
        ->(LOC,w3')->[HOST:AIX,w4']
        ->(PART,w5')->[SQL-INTERPRETER,w6']
```

The root concept nodes of a graph do not get assigned a weight because they are ignored during the matching process as explained above. There are two distinct ways to assign weights to the nodes of a conceptual graph, depending on whether all the weights of the graph satisfy a global invariant or not. A global invariant might be the requirement that for each node the sum of all the weights of its successors must be equal to one (see [10]). The consequence of this invariant is that the absolute weight of a particular node may decrease if the graph is extended by further nodes, as shown in example 5. Therefore the akin index of this extended graph and the query graph decreases in the same sense due to the different weights in the semantic intersection. This clearly violates requirement **M2** of the learning model.

Example 5. Let G_5 be a type graph and G_5' an extension assigned with weights.

G_5 :
```
[DATABASE]-
     ->(CHRC,1/2)->[RELATIONAL,1]
     ->(LOC,1/4)->[HOST:AIX,1]
     ->(PART,1/4)->[SQL-INTERPRETER,1]
```

G_5' :
```
[DATABASE]-
     ->(CHRC,2/5)->[RELATIONAL,1]
     ->(LOC,1/5)->[HOST:AIX,1]
     ->(PART,1/5)->[SQL-INTERPRETER,1]
     ->(SUPP,1/5)->[DATA-SECURITY,1]
```

This undesired effect of a decreasing akin index by an increased level of detail of the type graph can be eliminated by assigning weights which do not adhere to a global invariant. We therefore propose an independent assignment of weights to the nodes of a conceptual graph. In order to reach this independence, the first order formula of the semantic intersection has to be interpreted differently. All direct successors of a node represent alternative descriptions of this node. In the formula some conjunctive junctors have to be replaced by disjunctions. Example 6 shows how to interpret the semantic intersection.

Example 6. Let G_3' be the semantic intersection of G_1' and G_2' and F_3' the new interpretation.

G_3' :
```
[SOMETHING]-                    (concurring context)
   ->(LOC,w1'')->[HOST:AIX,w2'']
   ->(PART,w3'')->[SQL-INTERPRETER,w4'']
```

$$F_3' : \exists x_1 \exists x_2 \exists x_3 \Big((SOMETHING(x_1) \wedge HOST(AIX) \wedge LOC(x_1, AIX)) \vee (SOMETHING(x_1) \wedge SQL-INTERPRETER(x_3) \wedge PART(x_1, x_3)) \Big)$$

Derived from our practical experiences we assign each node (except the root node) of a conceptual graph a natural number ranging from $1, \ldots, 100$. This scale is divided into three categories depending on the importance of the node for the graph it appears in: (I) $1, \ldots, 33$ minor importance, (II) $34, \ldots, 64$ indifferent, (III) $65, \ldots, 100$ major importance. The default value of a node is 50.

The new interpretation intuitively divides a conceptual graph as a set of alternating subgraphs which overlap each other (figure 3 depicts this idea). The semantic of each subgraph is given by the ϕ operator. The akin index is determined by matching all possible combinations of all subgraphs.

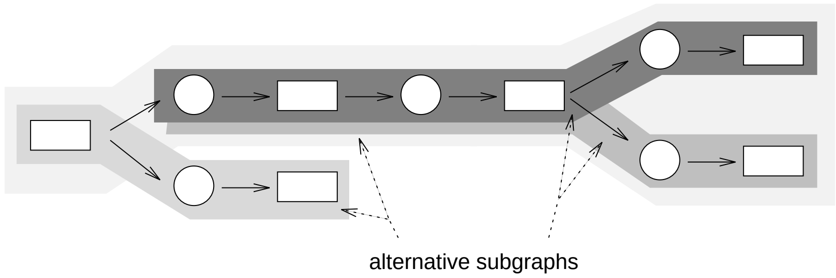

Fig. 3. Alternative subgraphs of a conceptual graph.

The root node of each subgraph is explained by its successors and the weight of this subgraph therefore has to be reduced proportionally. This can be accomplished by a simple multiplication of weights. The weights of different subgraphs are computed by a *maximum operator* known from the Fuzzy Set Theory (see [14]). This operator satisfies the requirement of the independence of weights. Given a query graph, an increase in level of detail in a type graph can only result in a bigger semantic intersection. Together with the maximum operator the akin index can therefore only increase as well. The best alternative subgraph is chosen and the remaining graphs discarded. This scheme evaluates the quality of an alternative within the semantic intersection compared to the context of the original conceptual graph. Altogether the akin index is computed recursively. The complete match algorithm is given below.

Experiments have shown that two UTs may have a very close semantical relationship. To further differentiate between similar semantics we integrated the possibility to include counter examples with the keyword `NOT`. In this case the weight of the contradictory context is set to zero.

Algorithm 1: compute the akin index of a query graph and a type graph

1. **Input**: G_T type graph and G_Q query graph.
2. $Restrict_C$, the restrict–operation taken from the *canonical formation rules* (see [11]), determines the common subtype of the concept nodes. $Restrict_R$ the restrict–operation for relation nodes respectively.
3. *weight* determines the weight of a node. Let the weight of the root nodes of G_T and G_Q be 100.
4. $index_C(n_T, n_Q)$ determines the akin index of every pair of concept nodes from G_T and G_Q, starting with n_T and n_Q respectively. $index_R$ is computed analogously.

5. Let n_T, n_Q be concept or relation nodes of G_T, G_Q. $index_C$ and $index_R$ are defined as follows:

   $index_C(n_T, n_Q) =_{df}$

   (a) $\frac{weight(G_T, n_T) \cdot 100}{weight(G_Q, n_Q)}$, if $Restrict_C(n_T, n_Q) \neq \perp$ or
   (b) $\max(\max\{\frac{weight(G_T, n_T) \cdot 100}{weight(G_Q, n_Q)} \cdot index_R(n_T^+, n_Q^+)\}, 0) \cdot 1/100$, if $Restrict_C(n_T, n_Q) = \perp$ and for all direct successor nodes n_T^+, n_Q^+ : $index_R(n_T^+, n_Q^+) \neq -1$ or
   (c) 0, otherwise

   $index_R(n_T, n_Q) =_{df}$

   (a) -1, if $Restrict_R(n_T, n_Q) \neq \perp$ but n_T contradicts with n_Q and for all $n_T^+, n_Q^+ : index_C(n_T^+, n_Q^+) > 0$ or
   (b) $\max\{\frac{weight(G_T, n_T) \cdot 100}{weight(G_Q, n_Q)} \cdot index_C(n_T^+, n_Q^+)\} \cdot 1/100$, if $Restrict_R(n_T, n_Q) \neq \perp$ or
   (c) 0, otherwise

6. **Output**: $index(G_T, G_Q) =_{df} index_C(n_T, n_Q)$ determines the akin index of G_T and G_Q with n_T, n_Q the root concept nodes of G_T, G_Q.

To determine the best service provider, the maximum of all the akin indices resulting from the query graph and all the type graphs stored in the trader's knowledge has to be computed. The best possible match for a query graph G_Q with the smallest semantic distance is therefore given by $match(G_Q) =_{df} max\{index(G, G_Q) | G$ type graph of the knowledge base$\}$. The following example shows in detail a computation of an akin index according to algorithm 1.

Example 7. Determine the akin index of the semantic intersection of G_1' and G_2' with `w1` = 50, `w2` = `w4` = `w5` = `w6` = 100, `w3` = 60, `w1'...w6'` = 100.

$$
\begin{aligned}
index(G_1', G_2') &= index_C(DATABASE, SOMETHING)\\
&= \max(\max\{100 \cdot 100/100 \cdot index_R(CHRC, SUPP),\\
&\quad 100 \cdot 100/100 \cdot index_R(CHRC, LOC),\\
&\quad 100 \cdot 100/100 \cdot index_R(CHRC, PART),\\
&\quad 100 \cdot 100/100 \cdot index_R(LOC, SUPP),\\
&\quad 100 \cdot 100/100 \cdot index_R(LOC, LOC),\\
&\quad 100 \cdot 100/100 \cdot index_R(LOC, PART),\\
&\quad 100 \cdot 100/100 \cdot index_R(PART, SUPP),\\
&\quad 100 \cdot 100/100 \cdot index_R(PART, LOC),\\
&\quad 100 \cdot 100/100 \cdot index_R(PART, PART)\}, 0)\\
&= \max(\max\{0, 0, 0, 0, 60, 0, 0, 0, 100\}, 0\}\\
&= \mathbf{100}, \text{ with}
\end{aligned}
$$

$index_R(CHRC, SUPP) = index_R(CHRC, LOC) = index_R(CHRC, PART) =$

$$
\begin{aligned}
index_R(LOC, SUPP) &= index_R(LOC, PART) = index_R(PART, SUPP) = \\
index_R(PART, LOC) &= 0 \\
index_R(LOC, LOC) &= \max\{60 \cdot 100/100 \cdot \\
&\qquad index_C(HOST : AIX, HOST : AIX)\} = 60 \\
index_R(PART, PART) &= \max\{100 \cdot 100/100 \cdot \\
&= index_C(SQL - INTERPRETER, \\
&\qquad SQL - INTERPRETER)\} = 100
\end{aligned}
$$

3.3 Fuzzy context learning

The assignment of weights to the nodes of a conceptual graph is central in our model for the computation of an akin index. We have devised a learning model tailored to the matching model introduced above. The learning model allows the incremental growth of type graphs which results in a better approximation to UTs. Within the framework of this model there needs to be a first type graph as initial approximation of a UT representing a valid semantic context (canonical, see [12]). The semantic distance between this first type graph to the UT may be high. Using the join–operation (as described in [11]) the type graph will be successively refined by further graphs. The semantic context remains valid. Example 8 demonstrates one application of the learning algorithm by combining G_6 and G_7 yielding G_8.

Example 8. G_6 is refined by joining it with G_7 resulting in G_8.

G_6 :
```
[DATABASE]-
    ->(CHRC,w1)->[RELATIONAL,w2]
    ->(LOC,w3)->[HOST:AIX,w4]
```

G_7 :
```
[DATABASE]-
    ->(LOC,w1')->[HOST:AIX,w2']
    ->(PART,w3')->[SQL-INTERPRETER,w4']
```

G_8 :
```
[DATABASE]-
    ->(CHRC,w1)->[RELATIONAL,w2]
    ->(LOC,w')->[HOST:AIX,w'']
    ->(PART,w3')->[SQL-INTERPRETER,w4']
```

The nodes of the example graphs can be attributed with weights, reflecting their relevance within the context of the graphs. The weight of those nodes which extend the type graph during a join operation remain unchanged. Synonym nodes or nodes resulting from a restrict operation require a re–computation of the weights. This step has to take into account how often the restrict operation already has been performed on this node in order to accomplish an asymptotic behavior of the learning process (requirement **L2** of the learning model). Therefore a counter is associated with every node and is being increased by one with every restrict during a join operation. The new weight is computed according to formula 1.

$$\text{new weight} = \frac{\text{counter} \cdot \text{old weight} + \text{weight of the example graph}}{\text{counter} + 1} \quad (1)$$

New weights only have a small influence on old weights with a high counter. They converge within smaller getting bounds. As a consequence examples joined with a type graph early in the course of time have a higher impact on the weights than at a later point of time. The influence of bad examples on type graphs decreases accordingly which results in a fault tolerant behavior of the learning model.

The converging of weights assigned to nodes implies a criterion to decide when the weight of a node has reached a fix point and therefore is not subject to further changes. Further examples have no more influence on this weight. The learning can be seen as partially finished if some counters reach a given bound. The best possible approximation is reached towards the end of the learning phase. The weights then reflect the importance of nodes which serve as a basis for the matching model.

Algorithm 2: adopted join

1. **Input**: Enter initial type graph.
2. Successive extension of the type graph through examples. This requires the re–computation of the type graph's weights according to formula 1, if those nodes are already contained in the type graph. Step 2 can be repeated an arbitrary number of times.
3. **Output**: The resulting type graph denotes a semantic approximation of a UT, where the weights of the nodes represent the importance for the context of embedded components of the graph.

4 Conclusion

Open distributed environments may be seen as a service market where services are freely offered and requested. The mediation of these services is done by a designated system component known as a *trader*. Current traders primarily base their matching algorithm of services upon IDL-based type notations. We have shown how conceptual graphs can be used for abstract type specification to support the cognitive domain of application users. This approach is particularly useful as the trader *learns* various ways of describing a service. We have developed formal *learning*– and *matching models* which serve as a basis of our implementation.

We have implemented the algorithms described in this paper as well as a *trading protocol* (see [9]). The complete source, using various C++ PD-class libraries and a Tcl/Tk–based GUI front end, are placed in the public domain and may be requested from the authors. Our implementation of an AI-based trader maintains a database of *uniform resource locators* (URL) of the World Wide Web.

Acknowledgements

We thank Wayne Brookes (University of Queensland, Australia) and an anonymous referee for their comments and discussions on this article.

References

1. Tim Berners-Lee et al. The World–Wide Web. *Communications of the Association for Computing Machinery*, 37(8):76–82, August 1994.
2. G. Booch. *Object Oriented Design with Applications*. BenjaminCummings Publishing Company, Inc, Redwood City, California, 1991.
3. Gerard Ellis. Compiled hierarchical retrieval. In T. E. Nagle, J. A. Nagle, L. L. Gerholz, and P. W. Eklund, editors, *Conceptual Structures, current research and practice*, chapter 31, pages 595–604. Ellis Horwood Limited, 1992.
4. Open Software Foundation. *Introduction to OSF DCE*. Prentice–Hall, Englewood Cliffs, New Jersey, 1992.
5. Object Management Group. *The Common Object Request Broker: Architecture and Specification Revision 1.1*. 1991.
6. ISO/IEC. Information Technology – Basic Reference Model of Open Distributed Processing – Part I. ISO/IEC COMMITTEE DRAFT ITU-T RECOMMENDATION X.901, 1993. ISO/IEC CD 10746–1.
7. ISO/IEC. *ODP–Trader, Document Title ISO/IEC JTC 1/SC 21 N 8192*. 1993.
8. Component Integration Laboratories. Shaping tomorrow's software (white paper). Technical report, cil.org:/pub/opendoc–interest/OD–overview.ps, 1994.
9. Arno Puder, Stefan Markwitz, Florian Gudermann, and Kurt Geihs. AI–based Trading in Open Distributed Environments. In *International Conference on Distributed Processing (ICODP'95)*. Chapman and Hall, 1995.
10. A. L. Ralescu and J. F. Baldwin. Concept learning from examples and counter examples. In B. R. Gaines and J. H. Boose, editors, *Machine Learning and Uncertain Reasoning, Knowledge–Based Systems*, pages 49–74. Academic Press, 1990.
11. John F. Sowa. *Conceptual Structures, Information Processing in Mind and Machine*. Addison–Wesley Publishing Company, 1984.
12. M. Wermelinger and J. Lopes. Basic Conceptual Structures Theory. In William M. Tepfenhart, Judith P. Dick, and John F. Sowa, editors, *Conceptual Structures: Current Practices. Second International Conference on Conceptual Structures, ICCS'94, College Park, Maryland, USA, August 1994*. Springer, 1994.
13. G. Yang, Y. Choi, and J. Oh. Cgma: A Novel Conceptual Graph Matching Algorithm. In H. D. Pfeiffer and T. E. Neagle, editors, *Conceptual Structures: Theory and Implementation, 7th Annual Workshop*, pages 252–261. Springer, 1992.
14. Hans Jürgen Zimmermann. *Fuzzy Set Theory and Its Applications*. Kluwer Academic Publishers, second, revised edition, 1992.