AI–based Trading in Open Distributed Environments

A. Puder, S. Markwitz, F. Gudermann and K. Geihs
{*puder,markwitz,florian,geihs*}@*informatik.uni-frankfurt.de*

Department of Computer Science
University of Frankfurt
D–60054 Frankfurt, Germany

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 an environment has to include mediator functionality (i.e. a trader) that matches service requests and service offers. Commonly, the matching process is based upon some IDL–based service type definition, and the types of the various services have to be "standardized" and distributed *a priori* to all potential participants. We argue that such well defined "standards" are too inflexible and even contradict the idea of an open service market. Therefore we propose a new type notation based on *conceptual graphs*. The trader maintains a knowledge base about service types in form of conceptual graphs. During the trader operations the service type knowledge evolves as it is continuously refined and extended. Users of the trading service interact with the trader and formulate queries in a corresponding notation that allows for a conceptual specification of the desired service type. Adequate matching algorithms and protocols have been implemented.

Keywords: Trading, type specification, conceptual graphs.
Classification: C.2.4; D.1.5; I.2.4

1 Introduction

The emerging Reference Model of Open Distributed Processing (ODP) provides an architectural framework for the standardization of distributed system technology. It defines abstract concepts that are appropriate to reason about and specify general distributed systems.

The basic goal is to enable the interworking of heterogeneous systems. Furthermore, it is addressing the question of application portability and distribution transparencies.

One of the functions that will be standardized as part of the ODP activities is the trading function (see [ISO94]). It is concerned with the matching of service requesters and service providers. The matching is done based on the notion of a *service type*, which (informally) is something that expresses properties of an object. The trading function is provided by a component called *trader*. A service provider exports its service offer to the trader (called *service export*). The trader maintains a database of service exports. A service requester makes an inquiry to the trader for a particular service offer and — if available — receives a reference to a suitable service exporter. This is called *service import*.

The concept of a service type plays an important role in such environments. The notion of a type is well known from conventional procedural programming languages, where types are used in order to aid error checking and software maintenance. Static typing is the dominant approach in these languages. In object–oriented languages the notion of a type is somewhat more flexible because of subtype relationships (see [Ame90] and [BJ93]). Nevertheless, the programmer of an application has a rather precise knowledge about what kind of types to be used.

In a general, large, open distributed system with a variety of different service providers, service requesters and service types, there is much less knowledge about the set of (service) types that will be available during the lifetime of an application program. Clearly, an application needs to understand the basic semantics and the access rules of the services it is going to work with. However, in an open service environment many different "flavors" of a particular service type may be offered over time by different service providers using the same or very similar service interfaces.

Consequently, in such dynamic environments service providers and requesters need means to specify service types and to *learn* about new service types at runtime. We have developed a notation for expressing the knowledge about service types and thus to support the trading function in open distributed systems. Our approach is based on a knowledge representation technique called *conceptual graphs*. A conceptual graph captures the knowledge about a service type and allows the specification of a type using a powerful, extensible notation. The trader matches service imports and exports using the information contained in the conceptual graphs.

This paper motivates our approach and demonstrates its strengths. In Section 2 we describe our assumptions that result from the envisaged trading environment. Section 3 introduces the conceptual graph technique. We present an example and a formal theoretical framework that is adapted to the trading requirements. The specification of a type may evolve over time. Therefore, an algorithm is presented describing how the interacting entities can incrementally acquire more knowledge about a service type. In Section 4 we give an overview of the trading protocol which allows for an interactive service type negotiation. A matching algorithm and protocol have been implemented and are available. Section 5 contains further details as well as our conclusions.

2 Environment

We use a basic model called the *object graph* to motivate our definitions of type notations within open distributed environments. The discussion of object graphs in this section serves as a starting point for the conceptual graphs, which will be presented in the following sections.

2.1 Object graph model

Our model is based upon the classical definition of an object, as it can also be found in the ODP RM [ISO93a], i.e. *an object is characterized by its behaviour and, dually, by its state. An object is distinct from any other object.* 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*. The directed graph will be called an *object graph*. In terms of level of abstraction a *client* may be an object in the common sense or a human user interacting with a client object. The terms *client* and *user* will be used synonymously throughout this paper.

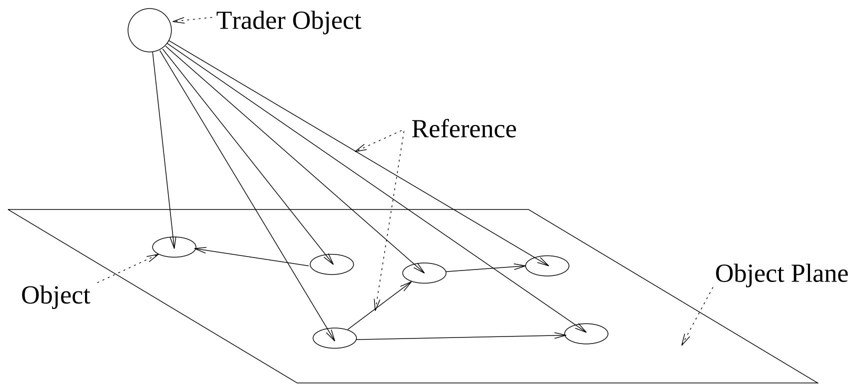

Figure 1: Trader object bridges the knowledge/visibility gap.

An important consequence 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 treaty that the server object must fulfill. Polymorphism here 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 treaty[1].

One can distinguish two different cases with respect to *when* a type specification is 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 some 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 transparent for users of the client object at runtime.

In contrast a type specification notation used during runtime must build upon different mechanisms. 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. 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), OLE2, OpenDoc or COSM (see [BL$^+$94], [Mic93], [Lab94] and [MML94] respectively).

All these systems have in common that different services can be provided at runtime without the need for a specific service interface definition 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)* descriptions. The generic client is able to interpret these descriptions and to build and present an appropiate GUI for the end user. The user may interact with the generic

[1]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.

client to invoke operations and to provide parameters embedded as widgets such as edit fields or checkboxes 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.

Via such mechanisms there may be a rich variety of different services accessible. However, there arise questions such as:

- How does a user specify its service requirement?
- How can a user find a suitable service provider for a desired service type?

It is not immediately clear what a notation for a type specification should look like. A type specification in this environment is more abstract and vague than an IDL based specification. In particular it should support the cognitive domain of the users and not of the programmers. In the following sections we propose a technique, called *conceptual graphs*, which is appropriate for runtime type specification.

2.2 Trading and the dualism of type definitions

In this section some consequences with regard to the role of a trader will be discussed. We assume the need for runtime type specifications as discussed above. Furthermore the object graph will be seen in the context of an open distributed environment. By *open* we mean an environment where all participating service providers are not known *a priori*. 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[2]. A client object has only a limited view on the object graph, as global knowledge of it's structure is generally impossible to acquire. As 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).

These considerations have led to proposals like the ODP Trader or CORBA's Request Broker (see [ISO93b] and [Gro91] respectively), which serve as a mediator between service requesters and service providers and therefore bridge the knowlegde/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. With respect to the categorization made in the previous section, current traders primarily match compile time type information.

It is important to think about the role of a type in such an environment. In order for the match algorithm within the trader to succeed, a type description must conform to a kind of "standard", which all participating parties have to agree upon *a priori*. This standard has to be defined well enough to be matched unambiguously against other types. Current traders, like the aforementioned ODP Trader, base their match algorithm mainly upon syntactic features of the interface. The implication is that the exact syntactic structure of a particular service signature must be communicated to all parties.

[2]Note that in *one address space* object systems the usual approach is that all except one designated root partition are subject to a garbage collector, as in Smalltalk.

The requirements of the definition of type specification notations can therefore be characterized as follows:

1. A notation should be based upon a *precisely defined syntax* to avoid ambiguities.

2. A notation should be *open* enough to avoid the need for an a priori standardization of service descriptions.

The first requirement originates from the fact that the trader must have solid grounds for a matching algorithm. This leads to an *explicit definition* of an object's type which necessarily must be known by all potential clients in advance. The second requirement on the other hand stems from a pragmatic point of view, whereas a client object should not need the a priori knowledge on *how* a server object has chosen to describe its type. This leads — contrary to the first requirement — to an *implicit definition* of a server object's type. The latter requirement clearly would be desirable as it would avoid the need to standardize every object type in advance.

We call the obvious contradiction the *duality* of the requirements of the notation for a type specification in open distributed environments. We have previously proposed a formal framework to solve this duality for compile time type notations (see [Pud94]). In the following section, we present a notation suitable for runtime typing, which in particular addresses the dualism mentioned above.

3 Towards AI–based trading

In contrast to compile time types, which are handled by a programmer, a type specification suitable for runtime represents an information artifact which is dealt with by a user. A notation therefore must adhere to the world of discourse of the user community with much less precisely defined syntax. On the other hand the notation should be flexible enough to allow for a broad expressiveness for a large variety of services as the experience with the WWW has shown.

Our approach — which copes with the aforementioned dualism — is based upon techniques which originated in the field of machine learning. There exists a wide range of literature on machine learning and various proposals have been made (see [Bol87] for an overview). Concerning the problem of AI–based trading, we have decided to build our framework upon a knowledge representation method called *conceptual graphs* (see [Sow84]). We have devised our own theoretical framework for conceptual graphs to suit the particular needs of a trader. In the following subsection the notion of a conceptual graph and a machine learning algorithm will be presented from a pragmatic point of view. Then a formal specification will be given.

3.1 AI–based trading: a pragmatic example

Conceptual graphs have been developed to model the semantics of natural language. Service descriptions based on conceptual graphs are therefore intuitive in the sense that there is a close relationship to the way human beings represent and organize their knowledge. From an abstract point of view a conceptual graph is a finite, connected, directed, bipartite graph. The nodes of the graph are either *concept* or *relation nodes*. Due to the bipartite nature of the graphs, two concept nodes may only be connected via a relation node.

A concept node represents either a concrete or an abstract object in the world of discourse. As for the context of service types a concept may be a concrete object such as `PRINTER`, `COMPILER` or `DATABASE` including specific instances (e.g. `HP-Laserjet`, `GCC`, `Ingres`, etc), as well as an abstract object such as PRINTING-SPEED or PROGRAMMING-LANGUAGE with no physical representation. Whereas concepts model objects of our perception, a relation node expresses a specific relationship between concept nodes. In the following examples a concept node is surrounded by square brackets and relations by round brackets, respectively.

The following conceptual graph labeled as CG1 describes an object oriented language called C++[3]. The informal semantic of the concept is: "Something which is a superset of a programming language called C, supports classes which themself consist of methods and a state. Furthermore the methods describe the behaviour of classes." The syntax of the following examples is according to a grammar which we have defined for AI–based trading and can be processed by our implementation.

```
CG1: [OO-LANGUAGE:{"C++"}] -
        -> (SUPERSET-OF) -> [PROGRAMMING-LANGUAGE:{"C"}],
        -> (SUPPORTS) -> [CLASSES] -
                          -> (HAVE) -> [METHODS] -> (DESCRIBE) -> [BEHAVIOR],
                          -> (HAVE) -> [STATE].
```

A concept can be recursively defined via subconcepts. The concept CG1 is therefore *explained* by two subconcepts which are connected to the *root concept* C++ with the relations SUPERSET-OF and SUPPORTS. A concept node itself is divided into a *type* and a possibly empty list of *instances* for that type. The root concept of CG1 therefore defines C++ as an instance of type OO-LANGUAGE. If the concept CG1 is regarded as a service which is offered by some provider, then the following conceptual graph would represent a query which matches with the previous service description:

```
CG2: [SOMETHING:*] -> (SUPPORTS) -> [CLASSES]
```

The informal semantics of CG2 is: "I need something which supports classes." As CG1 has previously been defined as something which actually does support classes, the trader would match these two descriptions. It should be noted that queries and service descriptions are formulated using the same notation. The root concept of CG2 [SOMETHING:*] introduces two new notions. The asterisk "*" denotes a *generic object* which will be matched with any other object. On the other hand it is not clear how SOMETHING is to be matched with OO-LANGUAGE. A concept node is a *typed entity* which may have arbitrary number of *instances*. In our notation a type is written left of a colon whereas the (possibly empty) instance list is written inside curly brackets to the right side. The set of all types T form a lattice with a partial ordering $\leq_T$ which denotes *specialization*. The type lattice used for this example is shown in figure 2.

The type SOMETHING as the top element of the lattice is *generic* in the sense that all other types are specializations of it. The matching process that the trader must perform can specialize types in a query. In order to match CG1 and CG2, the type SOMETHING is specialized or *reduced* to OO-LANGUAGE. Next consider a different query called CG3:

```
CG3: [SOMETHING:*] -> (ENCAPSULATE) -
                         -> [STATE],
                         -> [BEHAVIOR].
```

Some client wishes "something which encapsulates state and behaviour." As can been seen easily, even after a proper reduction of the root concept [SOMETHING:*], the requirement formulated in CG3 does not match CG1 although from a intuitive point of view they should. There is no

[3]For the purpose of this example we assume no further refinement of this service description (i.e. whether the service CG1 represents a language reference, product information or other).

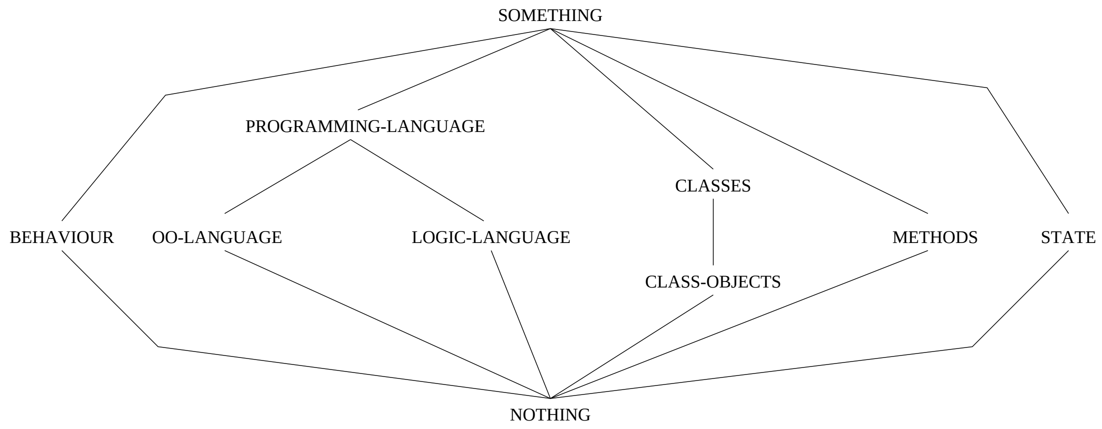

Figure 2: A possible explication of a type lattice.

way the trader can possibly match those two concept graphs because it doesn't have any notion of the underlying semantics. But if the trader were told that the two descriptions denote the same concept, then it could enhance `CG1` by *learning* the new features of the concept called C++:

```
CG4: [OO-LANGUAGE:{"C++"}] -
          -> (SUPERSET-OF) -> [PROGRAMMING-LANGUAGE:{"C"}],
          -> (SUPPORTS) -> [CLASSES] -
                           -> (HAVE) -> [METHODS] -
                                     -> (DESCRIBE) -> [BEHAVIOR].,
                           -> (HAVE) -> [STATE].,
          -> (ENCAPSULATE) -
                  -> [STATE],
                  -> [BEHAVIOR].
```

Obviously the query `CG3` will match the description in `CG4`. Next consider a different service provider registering a new service called Objective–C. The initial concept graph describing the service might look like:

```
CG5: [OO-LANGUAGE:{"Objective-C"}] -
          -> (SUPERSET-OF) -> [PROGRAMMING-LANGUAGE:{"C"}].
```

The previous concept graph is a subgraph of `CG4` and therefore the trader will ask the new service provider whether Objective–C is merley another instance of the type `OO-LANGUAGE` along with C++. If this should be the case, the trader will simply add the new instance to the root concept node. For the purpose of this example the service provider considers C++ different from Objective–C. In doing so he must augment his original conceptual graph by appropriate subconcepts which distinguish it from `CG4`. This augmentation results in the following graph `CG6` which states that "Objective–C is a superset of C and supports class objects":

```
CG6: [OO-LANGUAGE:{"Objective-C"}] -
          -> (SUPERSET-OF) -> [PROGRAMMING-LANGUAGE:{"C"}],
          -> (SUPPORTS) -> [CLASS-OBJECTS].
```

As the subconcept `[CLASS-OBJECTS]` distinguishes the two concepts, it is added as a counter example to `CG4` which yields the following new conceptual graph for C++:

```
CG7: [OO-LANGUAGE:{"C++"}] -
          -> (SUPERSET-OF) -> [PROGRAMMING-LANGUAGE:{"C"}],
          -> (SUPPORTS) -> [CLASSES] -
                              -> (HAVE) -> [METHODS] -
                                               -> (DESCRIBE) -> [BEHAVIOR].,
                              -> (HAVE) -> [STATE].,
          -> (ENCAPSULATE) -
                    -> [STATE],
                    -> [BEHAVIOR].,
          -> (NOT SUPPORTS) -> [CLASS-OBJECTS].
```

As the previous discussion suggests, a conceptual graph *explains* through an amalgamation of examples and counter examples. The trader can increase the quality of a concept over time as it incorporates new subconcepts. The quality of the matching process performed by the trader will therefore increase in the same way.

3.2 Formal Specification

The previous subsection has presented an extended example to demonstrate the power of a trader employing AI–techniques. In this section a formal framework for conceptual graphs, the join of two graphs as a learning mechanism and finally a match of two graphs will be given. We will start by defining the basic sets of the formal model:

Types T: Let T be a set of all types. The types in T and the partial ordering $\leq_T$ form a lattice $(T, \leq_T)$ with $\texttt{SOMETHING} \in T$ the top element and $\texttt{NOTHING} \in T$ the bottom element.

Relations R: Let R be the set of all relations. The relations R and the partial ordering $\leq_R$ form a lattice $(R, \leq_R)$ with $\texttt{LINK} \in R$ the top element and $\texttt{NO-LINK} \in R$ the bottom element.

Objects O: Let O be the set of objects of our perception. The objects are to be seen as instances of one or more types from T.

Concepts C_n: Let $C_n = T \times 2^O$ the set of all concepts. A concept is a tuple of a *type* and a subset of the set of all *objects*. The *generic object* (denoted by $*$) is a representative for any object and defined as O for formal reasons.

The set of *relations* is also organized in terms of a lattice with the partial ordering $\leq_R$. This will allow greater flexibility for the match operation. The set of all concepts may not be true with respect to the world of discourse. Therefore we introduce a *conformity relation* which provides a link to a higher order knowledge base. The relation *Conf* is not meant to be implemented, rather as a formal framework to argue about the *truth* of concepts. But there are nevertheless some properties which must hold. The decision of the meaningfulness of a concept eventually can only be decided outside the scope of the trader.

Conformity Relation: $\mathrm{Conf} : C_n \to \{true, false\}$

Let $(t, o) \in C_n$ with $t \in T$ and $o \in 2^O$.

1. $\Big(\mathrm{Conf}((t, o)) = true\Big) \Rightarrow \Big(\forall t' \in T : (t \leq t') \Rightarrow (\mathrm{Conf}((t', o)) = true)\Big)$.
2. $\mathrm{Conf}((\texttt{SOMETHING}, o)) = true$.

3. Conf((NOTHING, o)) $= false$.

4. Conf(($t, *$)) $= true$.

5. Conf(($t, \emptyset$)) $= false$.

If an object is an instance of a type, then it must also be an instance of all it's super types (i.e. more general types). All objects are instances of the top type SOMETHING and no object is an instance of the bottom type NOTHING. Finally every type has at least the generic object as an instance.

One important transformation of concept nodes is that of a *restrict operation*. A restrict specializes two concepts to their least common ancestor in terms of their types and instance lists. It is important to note that the restrict operation *does not* necessarily preserve truth (i.e. the result of a restrict operation on two true concept nodes, with respect to the conformity relation, must not necessarily be true). The join and match operation will use the restrict to transform a query for building maximal common subgraphs of two concepts.

The result of $\mathrm{Restrict_C}$ is the *minimal common subtype* (i.e. the least subtype which can be obtained by specializing two types). $\mathrm{Restrict_R}$ denoting the *minimal common relation* is defined analogously.

$\mathbf{Restrict_C}$: Let $\mathrm{Restrict_C} : C_n \times C_n \to C_n$ where

$$\mathrm{Restrict_C}((t_1, o_1), (t_2, o_2)) =_{df} \begin{cases} (t, o) & : \; (t \leq_T t_1 \wedge t \leq_T t_2) \wedge ((\forall d \in T) \\ & \quad ((d \leq_T t_1 \wedge d \leq_T t_2) \Rightarrow d \leq_T t)) \text{ and} \\ & \quad \text{with } t \neq \texttt{NOTHING} \text{ and } o = o_1 \cap o_2 \\ (\texttt{NOTHING}, \emptyset) & : \; \text{otherwise} \end{cases}$$

$\mathbf{Restrict_R}$: Let $\mathrm{Restrict_R} : R \times R \to R$ where $\mathrm{Restrict_R}(s, t) = u$ iff $u \leq_R s$ and $u \leq_R t$ and $(\forall w \in R)((w \leq_R s \wedge w \leq_R t) \Rightarrow w \leq_R u)$.

The first major definition is that of a *conceptual graph* as a graph containing concept and relation nodes.

Conceptual graph G: Let $N \subseteq \mathbb{N}$ be a finite, not empty set of node numbers and K be a set consisting of concepts and relations with $K \subseteq C_n \cup R$. There must be at least one concept in K and K is finite ($C_n \cap K \neq \emptyset$ and $|K| < \infty$). Let $m : N \to K$ be a total, not necessarily surjective numbering function. Let $V \subseteq N \times N$ be a set of vertices.

Let $G = (N, K, V, m)$ be a rooted, connected, acyclic and bipartite digraph with

(i) $\Big(\forall (n_1, n_2) \in V\Big)\Big(\Big|\{(n, n_1) | (n, n_1) \in V \wedge n \in N\}\Big| \leq 1\Big)$

(ii) $\Big(\dot{\exists}\, n \in N\Big)\Big(\Big|\{(n_1, n) |\; (n_1, n) \in V \wedge n_1 \in N\}\Big| = 0\Big)$ (n is called the root node number of the conceptual graph ($root(G) = n$))

(iii) $\Big(\forall (n_1, n_2) \in V\Big)\Big((m(n_1) \in C_n \wedge m(n_2) \in R) \vee (m(n_1) \in R \wedge m(n_2) \in C_n)\Big)$

(iv) $\Big(\forall (n_1, n_2) \in V\Big)\Big(\Big|\{(n_2, n) | (n_2, n) \in V \wedge n \in N\}\Big| = 0 \Rightarrow m(n_2) \in C_n\Big)$.

The set of all conceptual graphs is denoted by CG.

The join operation which is defined next merges two conceptual graphs into one. The join is not possible if the root concept nodes of the two graphs can't be restricted. Otherwise the resulting graph is obtained by recursively trying to overlay subconcepts as much as possible. The merging of two graphs is minimal in the sense that the joined graph is the smallest possible. The join operation is the basis for a machine learning algorithm. It should be noted that the result of a join necessarily has to be checked against the conformity relation. A join of two graphs can therefore only be a tool provided by the trader to aid a service provider augmenting and refining one of his or her service descriptions.

Join operation $Join : CG \times CG \to CG \cup \{nil\}$. The result of $Join(G_1, G_2)$ with $G_i = (N_i, K_i, V_i, m_i)$, w.l.o.g. $N_1 \cap N_2 = \emptyset$, $n_i = root(G_i)$, $k_i = m_i(n_i)$ the root concept node of G_i, $i = 1, 2$ is:

(i) nil if $\text{Restrict}_{\text{C}}(k_1, k_2) = (\texttt{NOTHING}, \emptyset)$ or

(ii) $G_J = (N_J, K_J, V_J, m_J)$. $\tilde{k} = \text{Restrict}_{\text{C}}(k_1, k_2)$ the new root node of G_J and $\tilde{n} = \max\{N_1 \cup N_2\} + 1$ the new root node number.

  (a) $N_J =_{df} N_1 \cup N_2 \cup \{\tilde{n}\} \setminus \{n_1, n_2\}$.

  (b) $K_J =_{df} K_1 \cup K_2 \cup \{\tilde{k}\}$.

  (c) $V_J =_{df} V_1 \cup V_2 \cup \{(\tilde{n}, n) | \exists (n_1, n) \in V_1 \vee \exists (n_2, n) \in V_2\} \setminus \{(\hat{n}, n) | (\hat{n} = n_1 \vee \hat{n} = n_2) \wedge n \in N_1 \cup N_2\}$.

  (d) Define m_J as follows:

$$m_J(n) =_{df} \begin{cases} m_1(n) & : \quad n \in N_1 \setminus \{n_1\} \\ m_2(n) & : \quad n \in N_2 \setminus \{n_2\} \\ \tilde{k} & : \quad n = \tilde{n} \end{cases}$$

(iii) Set $\hat{n} = \tilde{n}$.

(iv) If there exists direct successor nodes of $\hat{n}$: for all direct successors $\hat{n}_1, \hat{n}_2$ of $\hat{n}$:

  (a) if $m_J(\hat{n}) \in C_n$ and $\text{Restrict}_{\text{R}}(m_J(\hat{n}_1), m_J(\hat{n}_2)) = k' \neq \texttt{NO-LINK}$ then define $m_J(\hat{n}_1) =_{df} k'$ and connect all direct successors of $\hat{n}_2$ with $\hat{n}_1$. Define $m_J(\hat{n}_2) =_{df}$ undef. $\hat{n} = \hat{n}_1$. Go to (iv).

  (b) if $m_J(\hat{n}) \in R$ and $\text{Restrict}_{\text{C}}$ $(m_J(\hat{n}_1),\ m_J(\hat{n}_2)) = k' \neq (\texttt{NOTHING}, \emptyset)$ then define $m_J(\hat{n}_1) =_{df} k'$ and connect all direct successors of $\hat{n}_2$ with $\hat{n}_1$. Define $m_J(\hat{n}_2) =_{df}$ undef. $\hat{n} = \hat{n}_1$. Go to (iv).

The match operation is one of the key mechanisms of the AI–based trading concept. A match takes two conceptual graphs as input and produces their intersection.

Match operation $Match : CG \times CG \to CG$**:** Let G_S be a conceptual graph for a service description, G_Q a conceptual graph for a query. $Match(G_S, G_Q) = G_M$ the match graph is constructed as follows:

(i) $G_M =_{df} G_Q$, $n_i = root(G_i), i \in \{S, M\}$

(ii) If $m_S(n_S) \in C_n$ then

  (a) if there exists direct successor nodes of n_S:

if the number of direct successor nodes of n_M is greater 0 then for all direct successor nodes n'_M of n_M: if there exists no node n'_S (direct successor of n_S) with $\text{Restrict}_\text{R}(m_S(n'_S), m_M(n'_M)) \neq \texttt{NO-LINK}$ then delete n'_M and all existing successor nodes of n'_M else set $n_S = n'_S$ and $n_M = n'_M$, go to (ii)

(b) else delete all existing successor nodes of n_M; if $\text{Restrict}_\text{C}(m_S(n_S), m_M(n_M)) = (\texttt{NOTHING}, \emptyset)$ then delete n_M.

(iii) If $m_S(n_S) \in R$ then

(a) if there exists direct successor nodes of n_S:

if the number of direct successor nodes of n_M is greater 0 then for all direct successor nodes n'_M of n_M: if there exists no node n'_S (direct successor of n_S) with $\text{Restrict}_\text{C}(m_S(n'_S), m_M(n'_M)) \neq (\texttt{NOTHING}, \emptyset)$ then delete n'_M and all existing successor nodes of n'_M else set $n_S = n'_S$ and $n_M = n'_M$, go to (ii)

(b) else delete all existing successor nodes of n_M; if $\text{Restrict}_\text{R}(m_S(n_S), m_M(n_M)) = \texttt{NO-LINK}$ then delete n_M.

The trader decides the quality of a match by evaluating the result of a match operation according to some metric (for an in–depth discussion on this topic see [PMG95]). As we have just finished a prototype of the AI–based trader, this metric will be subject to modifications as we gain more experience. It should be clear that *wrong* answers to a query are possible if the quality of the service description or the query itself isn't sufficient. This will be discussed in greater detail in the following section.

4 AI–trading protocol

In this section we focus upon the *trading protocol* which embeds the trader as well as client and server objects into one framework. The justification for a designated protocol becomes clear when compared with the traditional task of service trading based on compile time type notations. The proposed type notation, introduced as *conceptual graphs*, does not rule out that the trader may make mistakes due to unprecise service descriptions. The interaction with a trader therefore goes beyond the one time matching of service requests. An AI–based trader may have to backtrack and refine previously stored descriptions through learning of new concepts and offering a client different services. It should be noted that a human user (i.e. *not* some software component) eventually recognizes a wrong service which may lead to further interaction with the trader providing a refined description.

We will discuss the protocol only on an informal level. Three distinguished roles may be identified which participate in the trading process: a *client* (respectively a *user*), a *server* and the *trader* itself. Each of these roles will be discussed separately.

Trader: Conceptually the trader maintains a database of all service providers who have registered themselves previously. The database holds tuples each containing a *conceptual graph* as one argument and *addresses* of one or more server objects providing the service described by the conceptual graph as another. 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. The precise structure of an address lies outside the framework.

Service provider: A service provider implements some service and wishes to export it through the trader. A suitable service description is formulated as a conceptual graph. The service provider may have to adjust his or her concept upon request from the trader. Eventually the quality of the service description will increase as the conceptual graph is refined over time.

Service requester: A service requester seeks a particular service and consults the trader for an appropriate reference. The desired service is specified again via a conceptual graph. As the trader's knowledgebase is incomplete initially, a service requester may not get what he or she intended. In this case the user has to further interact with the trader.

With respect to the AI–based trading protocol there are two distinct interactions with the trader: the *export* of a new service and the *import* of a particular service. A service provider initially exports a conceptual graph describing the service along with an address. The trader tries to match this graph with those already stored in its database. The service provider is presented a list of possible matches, i.e. services which are similar. The service provider either has to refine his initial description to increase the semantic distance to those matches or can decide that his service is just another instance of a service already registered.

A service requester formulates a query in terms of a conceptual graph which the trader tries to match with those services previously stored in its database. If there is no match the requester has to browse all services manually. If this search leads to the desired service, the trader then forwards the original query to the provider. It is the provider's task to augment his own conceptual graph accordingly, such that the formerly unsatisfied query will produce a match. In this case the learning process occurs on the side of the service provider.

If the service requester notices that an inadequate service was given to him or her then the query has to be re–formulated and posted again to the trader. Eventually the service requester will get the desired service. The learning process here occurs on the requester's side who has learned to precisely define what he or she wants. This scenario suggests that in terms of machine learning terminology the trader assumes the role of a teacher as well as a student (when new service descriptions are being taught), the client assumes the role of a student and the server may both act as a teacher and as a student.

5 Conclusion and outlook

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. In this paper we have proposed a new type notation which allows for abstract descriptions of arbitrary services. This notation — building upon techniques from the domain of machine learning — supports the cognitive domain of the users. The trader maintains a knowledgebase which is refined over time as the trader *learns* various ways of describing a service. The quality of a match therefore increases in the same sense, thus solving what we call the *dualism* of type notations.

We have implemented the algorithms and the protocol described in this paper. The complete source, using various C++ PD–class libraries, are placed in the public domain and may be obtained from the first author. Our implementation of an AI–based trader maintains a database of *uniform resource locators* (URL) of the World Wide Web. Future work will include a more comfortable GUI–based front end for conceptual graphs as well as experiments with various metrics for the match algorithm.

Acknowledgements

We thank Scott M. King (Thinkage LTD, Canada) and an anonymous referee for their comments and discussions on this article.

References

[Ame90] P. America. Designing an object oriented programming language with behavioural subtyping. In *REX School/Workshop, LNCS 489*. Springer, May/June 1990.

[BJ93] B. Liskov and J. Wing. A new definition of the subtype relation. In O. M. Nierstrasz, editor, *ECOOP'93: Object–Oriented Programming*. Springer, 1993.

[BL+94] Tim Berners-Lee et al. The World–Wide Web. *Communication of the Association for Computing Machinery*, 37(8):76–82, August 1994.

[Bol87] Leonard Bolc, editor. *Computational Models of Learning*. Springer, 1987.

[Gro91] Object Management Group. *The Common Object Request Broker: Architecture and Specification Revision 1.1*. 1991.

[ISO93a] ISO/IEC. Information Technology – Basic Reference Model of Open Distributed Processing – Part I. ISO/IEC COMMITTEE DRAFT ITU-T RECOMMENDATION X.902, 1993. ISO/IEC CD 10746–2.3.

[ISO93b] ISO/IEC. *ODP–Trader, Document Title ISO/IEC JTC 1/SC 21 N 8192*. 1993.

[ISO94] ISO/IEC. Working Document - ODP Trading Function, January 1994. ISO/IEC JTC1/SC21 N8409.

[Lab94] Component Integration Laboratories. Shaping tomorrow's software (white paper). Technical report, cil.org:/pub/cilabs/tech/opendoc/OD–overview.ps, 1994.

[Mic93] Microsoft. OLE 2.0 Design Specification. Technical report, ftp.microsoft.com /developr/drg/ole–info/OLE–2.01–docs/OLE2SPEC.ZIP, 1993.

[MML94] M. Merz, K. Müller, and W. Lamersdorf. Service trading and mediation in distributed computing environments. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS '94)*. IEEE Computer Society Press, 1994.

[PMG95] A. Puder, S. Markwitz, and F. Gudermann. Service trading using conceptual structures. In *3rd International Conference on Conceptual Structures (ICCS'95)*, Santa Cruz, University of California, 14–18 August 1995. Springer.

[Pud94] Arno Puder. A Declarative Extension of IDL–based Type Definitions within Open Distributed Environments. In *OOIS'94: Object–Oriented Information Systems*, South Bank University, London, 1994. Springer.

[Sow84] John F. Sowa. *Conceptual Structures, information processing mind and machine*. Addison–Wesley Publishing Company, 1984.