System Support for Knowledge–Based Trading in Open Service Markets

A. Puder and K. Geihs
Department of Computer Science
University of Frankfurt
D–60054 Frankfurt, Germany
{puder,geihs}@informatik.uni-frankfurt.de

July 8, 1996

Abstract

An open distributed service 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 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 that supports the openness of the service environment, since it is more flexible than IDL–based definitions, and since it is closer to 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

The evolution of distributed system technology has led to global information processing systems that support ubiquitous dissemination of information and access to all kinds of services. They are used for scientific, commercial, private and other purposes, and they integrate different information media and communication technologies. The World Wide Web (WWW) of today is an example of such a system. There is a constantly growing number of independent service providers offering a variety of services to a large population of service consumers. In such an environment a client, looking for some service, will be able to choose from a number of similar service offers that differ e.g. in service properties, service pricing and quality of service guarantees. Let us call such a distributed system an *open service market* (OSM).

Characteristic for an open service market is the wide spectrum of *service types*, the dynamicity of service configurations, and the inherent decoupling of service providers and service customers. Therefore, an essential function of a supporting environment for an OSM is a *service browsing* and *mediation* function. The mediation is often called *service trading*, e.g. in the ANSA and in the ISO standardisation of Open Distributed Processing (ODP) (see [1] and [4]). In these approaches the notion of a service type is tightly coupled to an operational interface that is used to access the service. What is missing is conceptual information of the service semantics that complements the syntactic interface specification. We claim that in an OSM the notion of a service type has to be extended to include not only the specification of an operational interface but also a more abstract specification describing the service functionality from a users perspective.

In this paper we present a technique for service trading that is based on conceptual information represented by *conceptual graphs* — a knowledge–based methodology that has been applied so far primarily for

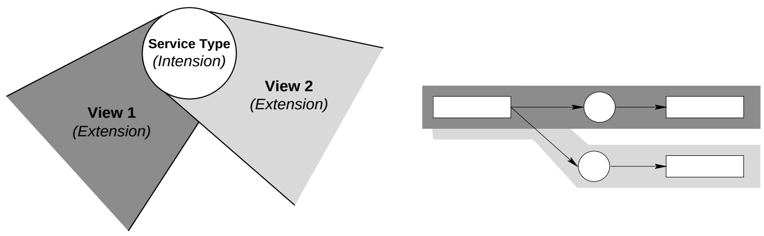

Figure 1: Extension and intension of a type.

knowledge representation and machine processing of natural language. In the following we will show how the specification technique works, and how the matching of service offer and service demand is done. Then, we will describe the tools we have developed to support the handling of conceptual graphs. Finally, we will comment on our experience with the application of our methodology.

2 Type Specification and Conceptual Graphs

The notion of a *service type* is fundamental for the discussion of service trading. A *trader* matches service offers and requests based on service types. A type implies an abstract understanding of the behaviour of instances of the type. A service type is independent from a particular implementation. In popular distributed system architectures, such as ANSA, CORBA, DCE and ODP (see [1], [6], [7] and [3]), a type denotes a specification of an operational interface, based on some *interface definition language* (IDL), and optional service attributes that provide additional information on the service properties. With the emergence of user oriented services as found e.g. in the World Wide Web a service type no longer denotes only a low level operational interface. An application user (i.e. *not* a programmer) must deal with service types to lookup matching service providers. Trading in this environment emphasizes the mediation of *service types* rather than *service instances* of a particular type. 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*. The process of distributing the knowledge of service types will be called *meta–trading* as it trades types and not instances of types.

A service type consists of an *intension* and an *extension*. The intension of a type is abstract in the sense that it serves as an abstract image of the properties of a service type. The intension is not to be confused with a physical representation, rather it is something we can think about. Any physical representation of an intension, such as a specification, is an extension of that intension. The extension provides a specific point of view of the intension of the service type. A service type specification (i.e. an extension) based on an intension is the only artefact which can be transported between users. Within an open service market, it is likely that two application users develop different extensions (or views) for the same intension (see left side of figure 1).

The knowledge–based trading scheme as described in [10] and [9] addresses the problem of abstract service type specifications supporting the cognitive domain of application users. It introduces a type description language based on *conceptual graphs* and a *machine learning algorithm* which solves the problem of meta–trading (see [11]): in the knowledge–based trading framework various service descriptions are being learned over time and therefore cope with the individual views of users. Therefore a conceptual graph as an extension of a service type serves as an approximation of the type's intension. 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 a mathematical 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, COMPUTER or PAPER as well as an abstract object such as INFORMATION with no physical representation. Whereas concepts model objects of our perception, a relation node expresses a specific relationship between concept nodes.

The following graph describes the concept *printer* in a textual notation for conceptual graphs (concept nodes are surrounded by square brackets and relations by round brackets). The informal semantic of the concept is: *A printer is a hardware device, connected to a computer and visualizes textual and graphical information.*

```
[PRINTER] -
     -> (IS-A) -> [HARDWARE-DEVICE],
     -> (CONNECTED-TO) -> [COMPUTER],
     -> (VISUALIZES) -> [INFORMATION] -> (REPRESENTATION) -
                                           -> [TEXTUAL],
                                           -> [GRAPHICAL].,
     -> (OPERATIONAL-INTERFACE) ->
            [INTERFACE-TYPE:{"PrintServiceInterface"}]
                -> (CONSISTS-OF) -
                       -> [SIGNATURE:{"SubmitJob"}]
                            -> (IN-PARAMETER) -
                                 -> [STRING:{"Data"}],
                                 -> [INTEGER:{"NumberOfCopies"}].,
                            -> (OUT-PARAMETER) -
                                 -> [INTEGER:{"JobNumber"}]..,
                       -> [SIGNATURE:{"KillJob"}]
                            -> (IN-PARAMETER) -> [INTEGER:{"JobNumber"}]..
```

Conceptual graphs can easily be visualized. In the *graphical notation*, concepts are denoted by rectangles and relations as circles (see right side of figure 1). Note that the above graph contains the specification of an operational interface as well. Embedded are the description of two remote operations called SubmitJob and KillJob. The operation SubmitJob has two input parameters and one output parameter, whereas the operation KillJob just has one input parameter. The way information of an operational interface is represented in a conceptual graph is independent of a particular IDL. A conceptual graph therefore allows the homogeneous representation of abstract (conceptual) knowledge from the users domain as well as specific low level knowledge in form of operational interfaces.

3 Trading of Service Types

A trader offers two basic primitives at its operational interface: service providers register themselves with the *export* operation, and a service requester uses the *import* operation to look for suitable providers. In our framework, service exports as well as service imports specify the types through conceptual graphs, called *type graphs* and *query graphs*, respectively. A knowledge–based trader builds up a knowledge base where each service description is being refined over time through successive export operations. The import operation computes the *semantic distance* of a query graph and all type graphs stored in the trader's knowledge base. The semantic distance induces a metric to determine a similarity index between conceptual graphs. Due to an intuitive common understanding of particular services, the likelihood of a service importer getting the desired service without knowing the precise type definition increases over time as the trader learns to incorporate various views of the service into the type graph. Thus the problem of an *a priori* knowledge of a service type is resolved to automate the process of meta–trading.

The knowledge–based trader consists of three components: a *service database*, a repository for *matching rules* and a *lexicographical database* (see figure 2). The service database stores service types based on

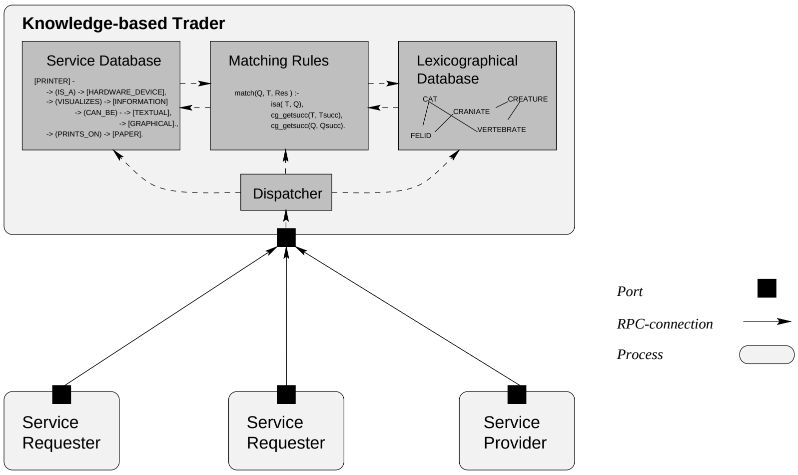

Figure 2: Architecture of the knowledge–based trader.

conceptual graphs. A set of service providers is associated with every service type. The service database provides a mapping of types to instances of these types. The repository of matching rules is used to define different metrics to compute the semantic distance between two graphs. There exist a wide range of different rules, e.g. generic rules from linguistics (such as specialization and negation) and rules for matching operational interfaces according to different subtyping rules. The next section demonstrates the use of conceptual graph matching rules to model different syntactic subtyping rules. The lexicographical database maintains a comprehensive semantical network with information that is used by the matching rules. Thus, the lexicographical database provides "background" knowledge needed to match two conceptual graphs. We will come back to this in section 3.2.

3.1 Syntactic subtyping

This section demonstrates the usage of conceptual graphs for the specification of operational interfaces. As mentioned before, an IDL allows the syntactic specification of a set of functions, which an object implements at its interface. Various IDLs have been standardized, such as the DCE–IDL or the CORBA–IDL. The ODP standards do not prescribe any particular notation for an IDL, nor do they make reference to any existing IDL. The ODP standards rather define the characteristics that an IDL for open distributed systems should exhibit. According to ODP, a service type consists of an interface type and a set of service property types. An interface type itself consists of a set of signatures each denoted by a name and an ordered list of parameter types.

Each IDL, including ODP's abstract characterisation of an IDL, is complemented by a (syntactic) subtyping rule. Syntactic subtyping guarantees the substitutability of interfaces based on inclusion polymorphism. DCE, CORBA and ODP have different definitions of precisely what constitues a subtype. In DCE an interface type T_1 is a subtype of an interface type T_2, iff all functions defined in T_2 are a subset of those defined in T_1 and appear in the same order. CORBA relaxes this rule by omitting the need for the concurring functions to be declared in the same order. The reason for this relaxation lies in CORBA's interface repository which stores runtime type information. Whereas DCE posseses no runtime information except an UUID assigned to each interface, CORBA has access to type information at runtime via its interface repository. ODP extents CORBA's subtyping rule by additionally allowing parameter subtyping by contra–variance of input parameters and covariance of output parameters.

A *type domain* is defined by a specific type notation and subtyping rule. Our knowledge–based trader can operate in all of the type domains introduced above, namely DCE, CORBA and ODP type domains.

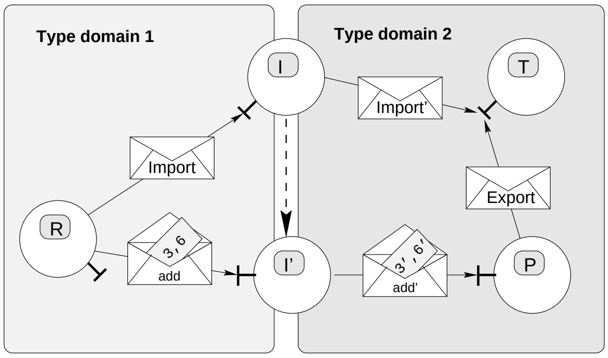

Figure 3: Interceptors overcome type boundaries.

Approriate translation tools convert an interface type to a conceptual graph. The structure of the conceptual graph is defined by a template similar to ODP's abstract service type definition. Once translated, the knowledge–based trader can compare two conceptual graphs using one of the subtyping rules from its matching rule database. An interesting situation occurs, if a service requester and provider are located in different type domains. In this case a particular object called *interceptor*, brigdes the border between type domains. Trading between different type domains is depicted in figure 3. Service provider P located in type domain 2 exports its service at trader T. Service requester R uses interceptor I to issue an import to trader T. The interceptor I in this example bridges type domain 1 and 2. I behaves towards R like the trader T. From T's perspective, I is just another service requester. Trader T responds to the import operation with a reference to the interface of P. As the reference to P crosses the interceptor I, special care has to be taken as otherwise the endpoints of the reference would lie in different type domains. Interceptor I instantiates a new interceptor I' (dotted line in figure 3). I returns a reference to I' as a response to R's import operation.

The capabilities of an interceptor with respect to the kind of parameter conversions it can perform, have influence on the trading process. If for example type domain 1 is a DCE domain and domain 2 is a CORBA domain, then the interceptors capabilities decide the kind of subtyping rule selected for an import operation. If an interceptor can convert the parameters of a function call according to the contra– and covariance rules, then trader T may employ the ODP–subtyping rule. In the scenario depicted in figure 3 export is done using CORBA–IDL, import is done with the DCE–IDL while the matching within the trader uses ODP's syntactic subtyping rule.

3.2 Tools and Applications

A prototype of a knowledge–based trader has been implemented and placed in the public domain (see [8]). The prototype is used primarily to investigate the specification and matching methodology based on conceptual graphs, to design and test the required protocols, and to build and evaluate appropriate tool support that facilitates the user interactions with the trading system. The trader is implemented as a separate Unix process using Sun–RPC to communicate with service importers and exporters. The Tcl/Tk toolkit has been used to implement a user friendly graphical frontend. It allows the construction and manipulation of conceptual graphs using point–and–click techniques. Besides the Tcl/Tk frontend, we have implemented a Java applet with the same capabilities. Thus, our knowledge–based trader can be accessed with any Java capable browser.

The lexicographical database stores "every day knowledge" which is being used during the match of two conceptual graphs. We have converted the WordNet dictionary (see [5]) and extracted over 87.000 nouns from the English language with more than 95.000 relations between them. For example, the lexicographical database contains entries such as *printer* `IS_A` *hardware* and *love* `ANTONYM` *hate*). The linguistic matching rules make use of antonym, synonym and hyponym (i.e. specialization) relations found in WordNet. We

	Verdict	DCE	CORBA	ODP
$T_1 \leq T_1$	Succeeds	6.72 sec	7.02 sec	9.39 sec
$T_2 \leq T_1$	Fails	7.21 sec	7.90 sec	3.83 sec
$T_1 \leq T_2$	Succeeds	3.51 sec	3.65 sec	5.60 sec
$T_2 \leq T_2$	Succeeds	3.61 sec	3.71 sec	4.59 sec

Table 1: Performance of different subtyping rules.

have added another relation called `SUBTYPE` to store parameter subtype information (such as *float* `SUBTYPE` *int*). This relation is used within the ODP–subtype rule to implement contra– and covariance.

We have build tools for translating DCE– and CORBA–IDL to and from conceptual graphs. This allows the trading of service types independent of the IDL that was used to specify their operational interfaces. Besides the benefit of providing a homogeneous framework for trading IDLs, with such a translation capability the trader can be used to translate specifications written in different IDLs. Three matching rules handle syntactic subtyping of IDLs according to the conventions defined by DCE, CORBA and ODP.

The type specification based on conceptual graphs as described in this paper is being integrated currently into a TINA (see [2]) compatible distribution platform in a joint project with a large telecommunications company. We investigate its suitability as a trading platform for telecommunication services. The knowledge–based trader will be used as a service navigation facility that supports the matching of telecommunications service providers and clients.

3.3 Performance

We have performed some measurements with our knowledge–based trader by testing the matching of two interface specifications using the DCE, CORBA and ODP subtyping rules. The matching of two operational interface specifications is a common task which has to be performed by any infrastructure offering some mediation functionality. As pointed out in section 3.1, there exist different IDLs as well as different subtyping rules. The first interface type we have used for the performance test — in the following called T_1 — is taken from Annex A of the ODP–Trader document (see [4]). T_1 denotes the operational interface of an ODP conformant trader written in CORBA–IDL. Interface type T_1 contains multiple interfaces and module definitions and makes extensive use of interface inheritance. The source file of T_1 is more than 600 lines of CORBA–IDL specification. Based on T_1 we have derived a second interface type T_2. This interface is the same as T_1 except that it does not contain the *export* and *management* interface. T_2 is about half the size of T_1 in terms of lines of IDL code.

We have converted both interface types to conceptual graphs using our translation tools. The T_1 specification yields a conceptual graph with 4.754 concept nodes and 2.740 relation nodes. T_2 has 2.638 concept nodes and 1.479 relation nodes. We have measured all four permutations of comparing T_1 and T_2 (see table 1). The notation $T_1 \leq T_2$ is to be read as "test if T_1 is a subtype of T_2". To test $T_1 \leq T_2$, we have first exported T_1 and then imported T_2 using one of the three subtyping rules. The results listed in table 1 denote the average time for one import–operation. The verdict column says, whether the subtyping test succeeded or failed (e.g. only $T_2 \leq T_1$ fails for all subtyping rules, since T_2 lacks some specifications which only T_1 has. The verdict for one row is independent of the particular subtyping rule applied). The performance tests have been performed on an IBM RS/6000 model 580.

The results reflect the complexity of matching two concpetual graphs. The ODP–subtyping rule is more complex than the other two as it includes structural subtyping as well as contra– and covariance. Two results are worth mentioning, both related to the second row of table 1. First of all it seems surprising that the DCE– and CORBA–subtyping rules take a long time to find out when two types are not in a subtype relationship. This is due to the fact that the matching rules use backtracking in order to test all possible permutations while matching conceptual graphs. Only when the last permutation fails, the knowledge–based trader can infer that the two types do not match. The ODP–rule on the other hand is able to infer the same result faster as it tries to match the structure of the type definitions within the type specifications. When a user–defined type (represented as a sub–graph within the conceptual graph) does not satisfy the subtype relationship, the knowledge–based trader can immediately infer the verdict.

The performance of the trader could be enhanced by maintaining a cache and computing hash values from conceptual graphs. The cache could be used to check if the comparison of two graphs with a particular rule had already been performed in the past. If so, the import–operation could immediately return the verdict. Furthermore, the performance of the matching rules could be enhanced by hard–coding the various rules in an efficient programming language. Currently, the matching rules are implemented in Prolog. The knowledge–based trader has an embedded Prolog subsystem which allows the insertion and changing of matching rules at runtime. This has obvious advantages such as extensibility and expressiveness. However, using Prolog is not as efficient as imperative languages.

4 Conclusions

Our basic assumption in this paper is that there will be an ongoing evolution of distributed system technology such that eventually we will see large service environments with a dynamically changing variety of service types, service providers and customer preferences. A look at the abundance of information sources and services available already in the Internet reveals that such an open service market may not be too far away.

In such environments support for service trading is needed. In terms of expressiveness, flexibility and openness, the traditional naming schemes found in distributed systems do not scale up with the expected growth and variation of an open service market. Also an application user will be exposed more and more with the non–trivial task of information discovery and selection. A type notation is required which supports the cognitive domain of users. We have presented an enhanced trading technique that is based on results from knowledge–based systems to specify and match service types, and to mediate between service requester and service provider. We have shown that the technique is upward compatible with existing trading systems, and that an efficient tool set can ease the handling substantially.

The reader is invited to test the specification and matching methodology based on conceptual graphs by visiting our web–page (see [8]).

References

[1] ANSAware Release 4.1, Manual Set, February 1993.

[2] F. Dupuy, G. Nilsson, and Y. Inoue. The TINA Consortium: Toward networking telecommunications information services. *IEEE Communications Magazine*, pages 78–83, November 1995.

[3] ITU.TS Recommendation X.901 — ISO/IEC 10746–1: Basic Reference Model of Open Distributed Processing Part 1: Overview and Guide to the use of the Reference Model, July 1994.

[4] ODP Trading Function, ITU/ISO Committee Draft Standard ISO/IEC DIS13235 Rec. X.9tr, May 1995.

[5] G.A. Miller et al. Five papers on WordNet. Technical Report CSL Report 43, Cognitive Science Laboratory, Princeton University, March 1993.

[6] Object Management Group (OMG), The Common Object Request Broker: Architecure and Specification, Revision 1.1, December 1991.

[7] Open Software Foundation. *Introduction to DCE.* Open Software Foundation, Inc., 1992.

[8] A. Puder. Introduction to the AI–Trader Project. http://www.vsb.informatik.uni–frankfurt.de/-projects/aitrader/intro.html, Computer Science Department, University of Frankfurt, 1995.

[9] 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 Verlag.

[10] A. Puder, S. Markwitz, F. Gudermann, and K. Geihs. AI–based Trading in Open Distributed Environments. In *3rd International IFIP TC6 Conference on Open Distributed Processing (ICODP'95)*, Brisbane, Australia, 20–24 February 1995. Chapman and Hall.

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