Meta–Level Service Type Specifications

Arno Puder[1] and Kurt Geihs[2]

[1] Deutsche Telekom AG, Research Centre
P.O. Box 100003
64276 Darmstadt, Germany
puder@tzd.telekom.de

[2] Department of Computer Science
University of Frankfurt
60054 Frankfurt, Germany
geihs@informatik.uni-frankfurt.de

Abstract

The concept of a *service type* plays a fundamental role in specifying and trading client requests and service offers. Existing middleware architectures use syntactic interface type specifications based on specific IDLs, while ODP standards do not prescribe any particular IDL. Neither approach can handle semantical information beyond interface structure and quantitative service attributes. We present a meta–level type specification technique that is generic and flexible to express any kind of information related to a type, not only its operational interface structure. The technique uses a knowledge representation notation called *conceptual graphs*. We have integrated this technique into a trading system that can translate different IDL specifications to conceptual graphs and that can decide upon their potential subtype relationship. The trader can be configured flexibly to handle and match any kind of information that can be represented as a conceptual graph.

Keywords: Open Distributed Processing, conceptual graphs, type management, service types, trading, IDL

1 Introduction

A *service type* in an open distributed system denotes a set of objects which offer the same service at their interfaces. With a type, certain characteristics and semantics are associated that objects of that type should exhibit. The notion of a type is required in order to select and distinguish objects, and in order to offer and request a specific functionality. In Open Distributed Processing (ODP) (see [3]) the concept of a type plays an elementary role in the trading function (see [4]). Matching service offers and service requests is done based on type information that is provided by exporters and importers.

In todays distributed system platforms type specifications are based on syntactical specifications of the operational interfaces. Operational interface specifications are necessary in order to enable the invocation of an object's functionality. There exist several different notations for the definition of operational interfaces called *interface definition languages*, IDL for short. Examples of such IDLs are the DCE IDL [8] and the CORBA IDL [7]. Obviously, such a syntactical specification lacks means to express the semantics of the object's functionality explicitly.

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. Thus, ODP allows to employ different IDLs to specify operational ODP interfaces. However, no formal notation is given to define the syntactical structure of an interface on an IDL independent meta–level, and consequentially no means are available to compare two operational interfaces that are specified in two different IDLs, e.g. one in DCE IDL and the other in CORBA IDL. Likewise, the specification of semantic properties of an ODP object that expresses more than just quantitative attributes of a type is for further study in ODP.

Tied to an IDL there is a type conformance definition that defines a substitutability relationship for objects of different types. This leads to more flexibility in matching service offers and service requests. The subtype relationship constitute a type hierarchy based on inclusion polymorphism. For example, ODP, CORBA and DCE work with different subtype definitions, as will be described later. A general survey of different subtyping rules is given in [1].

In this paper we address the issues of meta–level type specifications containing syntactical as well as semantical type information. We will present a specification technique that is capable of representing conventional IDLs as well as additional semantical type information. Our approach is based on a knowledge representation notation called *conceptual graphs*. In an earlier paper (see [10]) we had discussed the requirements and our basic approach of building a trading system for open service markets. In this paper we concentrate on the type compatibility issues in different type domains and on the translation of interface specifications using conceptual graphs as a meta–level specification technique. In Section 2 of this paper we discuss the necessary fundamentals of type specifications and conceptual graphs. Section 3 demonstrates the usage of conceptual graphs for the specification of operational interfaces, and describes the translation between specifications as well as type conformance checking. In Section 4 we present the design of a trading system that uses these techniques, and we give a brief description of our prototype. Section 5 contains our conclusions and outlook.

2 Type Specifications and Conceptual Graphs

A 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 behavioural properties of a service type. The intension is not to be confused with a concrete physical representation of a type, rather it is something we can think about. Any physical representation of an intension, such as a formal specification, an operational interface specification or a computer program, 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) is the only artefact which can be transported between users. Within an open distributed system with a variety of service providers and service clients, it is likely that two users develop different extensions (or views) for the same intension (see Figure 1).

Figure 1 illustrates that a type's extension can be at different levels of abstraction. Whereas one extension of a type might be the specification of an operational interface, another extension might give an informal description of the service type's semantics. With the emergence of user oriented services as found e.g. in the World Wide Web (WWW), it is not sufficient to deal with (low level)

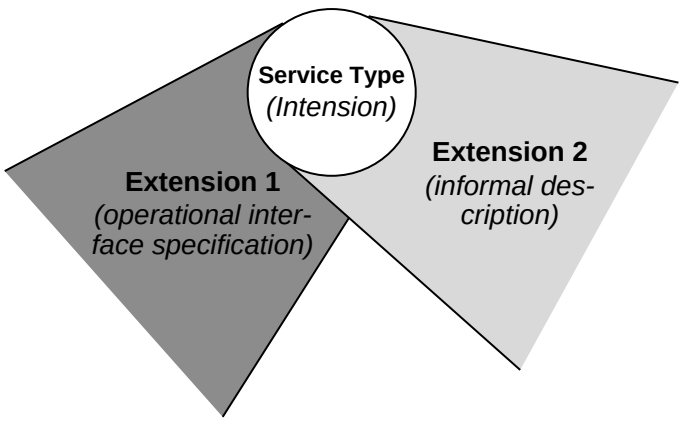


Figure 1: Extension and intension of a type.

operational interface specifications. An application user (i.e. *not* a programmer) must deal with service types to lookup matching service providers, and higher level, semantically oriented service type specifications are needed.

In the WWW there exist several pragmatic solutions that are based on a sort of high–level type specification. So–called web–crawlers traverse the web and generate indices from the textual parts of HTML documents. With respect to the ODP terminology, the export of services (i.e. HTML–documents) is done implicitly. The import is done by a user who poses a query through a set of words. The web–crawler (i.e. trader) searches for HTML–pages which contain all or parts of the words. Some web–crawlers allow more sophisticated queries, e.g. through boolean operators.

Such "type specifications" based on a set of words have several drawbacks:

- The words contained in a query are semantically unrelated, i.e. two documents that contain the same words have the same type.
- There is no way to express a relationship between a set of HTML–documents and a service type.
- Only alphanumerical information is exploited. Other information contained in HTML–documents, such as audio, graphics or Java applets, cannot be used.

Thus, different type specification notations serve different needs. While IDLs concentrate on structural, interoperability issues, natural language descriptions try to capture the semantics of a type. The question is whether we can construct a type specification language which is scalable and generic in the sense that it can express all of the existing type specification notations. With such a notation, generic type management and repository components could be defined which could be re–used in any type domain. In the following we will present such a generic specification technique based on conceptual graphs.

2.1 Conceptual Graphs

Conceptual graphs have been developed to model the semantics of natural language and to support the machine–based processing of natural language (see [11]). 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.

The semantics of a specification based on conceptual graphs is primarily defined through the relation nodes. Whereas concepts model objects of our perception, a relation node expresses a specific relationship between concept nodes. A relation has a well–defined semantic which clearly states the relationship that one concept has to another. Removing the relations in a conceptual graph just leaves an unordered set of concepts. Clearly, a conceptual graph is therefore more expressive than just a simple set of words as described above. The syntax of conceptual graphs forms a meta–grammar, because the set of relation names which may be used for conceptual relation nodes is not part of the grammar. The grammar for conceptual graphs only prescribes that two concepts may be connected only via a relation node. Depending on what kind of specification has to be coded as a conceptual graph, a set of specific relation names can be introduced. The semantics of those relations has to be well defined and all participants of an open distributed environment have to agree upon their semantics *a priori*.

Next we will show through an example how conceptual graphs (CG) can be used to specify different extensions of a type, e.g. the operational interface as well as informal information on what a service of that type does.

2.2 Example

Consider a travel service offering touristic arrangements. In our example the service allows to make and cancel reservations for a certain type of vacation with specific activities. First of all the operational interface of an object offering this kind of service would be formally specified in an IDL. The following interface specification is written in CORBA IDL:

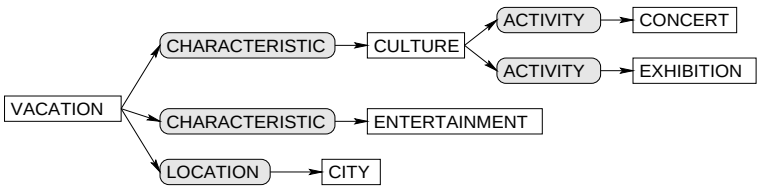
```
interface VacationService {

    typedef enum culture {
        concert, exhibition
    } Culture;

    typedef long Date;

    void MakeReservation( in Date date, in Culture type, out long key );
    void CancelReservation( in long key );
};
```

An object of type `VacationService` offers two functions at its operational interface. With the first function reservations for different activities can be made. In order to make a reservation, the date and the kind of the activity have to be provided as input parameters. The result of this function is a reservation key. An activity may be cancelled by calling the second function and

making reference to the reservation key. Apart from this "explanation" of the low level details of the operational interface, the provider of `VacationService` could give an abstract, more informal description on what is offered:

> *"The service offered is a vacation. During the vacation a cultural program is offered. The cultural activities include a concert and an exhibition. During the vacation entertainment is offered. The vacation is located at a city."*

Note that this informal description is not simply an explanation of the operational interface but rather an abstract description a user might give of this service type. The informal description above can be translated into a conceptual graph, which yields the following result written down here in the textual notation for conceptual graphs (concept nodes are denoted by white rectangles and relation nodes by black rectangles):



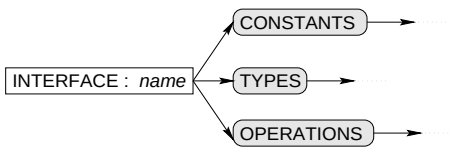
The semantics of the above conceptual graph is intuitively understandable. It corresponds to what a user might expect from an object with the interface `VacationService`. The relations `CHARACTERISTIC`, `ACTIVITY` and `LOCATION` have their specific semantics which are described in the relational catalog in [11]. The relation `LOCATION` for example links a concept `SOMETHING` to a concept `PLACE` or specialization of theirs. The semantics of this relation is that the first concept is located at a particular place. Note that on the one hand the CG is readable intuitively and close to natural language, while on the other hand it is a formally defined, machine–processable structure.

With a conceptual and relational catalogue the semantics of a conceptual graph can easily be determined. Since conceptual graphs allow the representation of arbitrary knowledge, it is a logical next step to represent an operational interface specification as a CG and combine it with the informal specification from above. Given a conceptual and relational catalogue to provide the required "vocabulary", an operational interface specification can be translated according to the following template:

Operational Interface:

`interface` *name*
{
    Constant declarations
    Type declarations
    Operations
}

Conceptual Graph:



According to these translation rules, the above interface for `VacationService` would be converted to a CG–based interface specification. Due to the lack of space we cannot show the complete mapping here, since many more subgraphs are needed to capture all details of an interface specification.

Basically all details found in an IDL contribute to a corresponding representation through conceptual subgraphs. This way all features found in the CORBA IDL and DCE IDL such as exceptions in CORBA or pipe declarations in DCE are mapped to conceptual subgraphs (see [5]).

To summarize this section, with CGs as the meta–notation for operational as well as semantical interface specifications we are able to extend the trading information very flexibly in a generic manner. This is the basis of an extended trading system, as will be described next.

3 Trading between Type Domains

A conventional IDL allows the specification of a set of functions, which an object offers 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. They rather define the characteristics that an IDL for open distributed systems should exhibit.

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 the subtype–relation. 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 uses no runtime information except an UUID and a version number 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 with contra–variance of input parameters and co–variance of output parameters as well as recursive subtyping.

A *type domain* is defined by a specific type notation and subtyping rule. If a trader wants to operate in all of the type domains introduced above, namely DCE, CORBA and ODP, an appropriate translation is needed that converts interface specifications between type domains and decides upon the subtype relationship. In our knowledge–based trading system, whose architecture will be described in the next section, we use conceptual graphs to specify types independent from their original type domain and to convert from one domain to another.

The structure of a conceptual graph which contains a specification of an operational interface 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 a given subtyping definition. The different subtyping rules are stored in the trader's type matching rule database. An interesting situation occurs, if 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 2. Service provider P located in the CORBA type domain exports its service to trader T. The trader itself is located in the type domain that operates with conceptual graphs. The interceptor I_P is responsible to convert the parameters of P's export operation. In particular interceptor I_P has to translate the CORBA IDL specification contained in the export operation to a conceptual graph. I_P behaves towards P like the trader T. From T's perspective, I_P is just another service provider. Service requester R, who is located in a DCE type domain, uses interceptor I_R to issue an import to trader T. The interceptor I_R has to translate the DCE IDL contained in the import operation to a conceptual graph. Trader T responds to

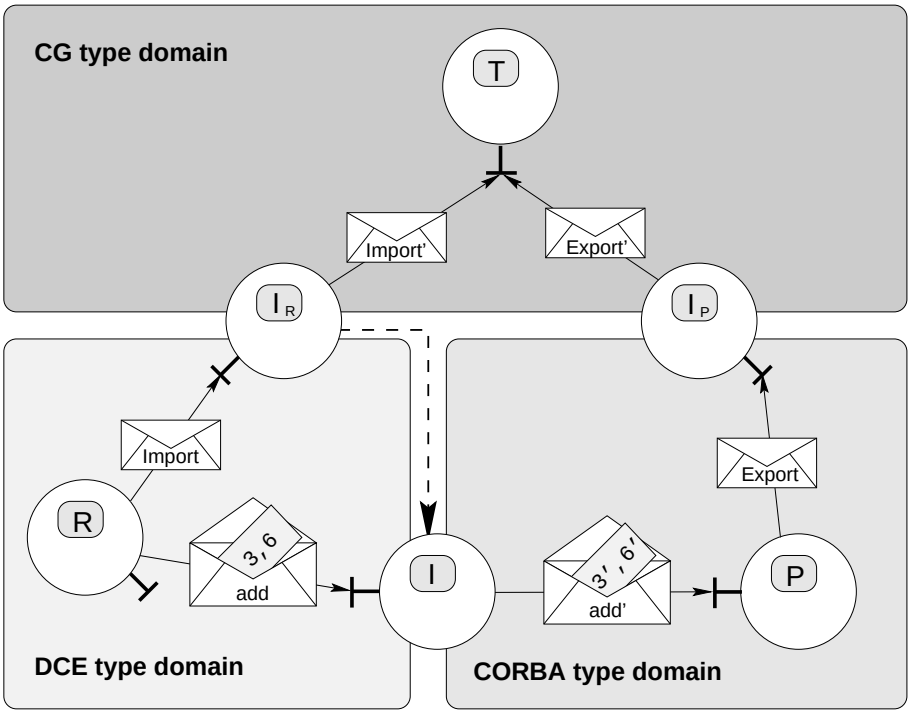


Figure 2: Interceptors overcome type boundaries.

the import operation with a reference to the interface of P. As the reference to P crosses the interceptor I_R, special care has to be taken as otherwise the endpoints of the reference would lie in different type domains. Interceptor I_R instantiates a new interceptor I (dotted line in Figure 2). I_R 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 the interceptor I can convert the parameters of a function call according to the contra– and co–variance rules, then trader T may employ the ODP subtyping rule. In the scenario depicted in Figure 2 the export is done using CORBA IDL, import is done using DCE IDL while the matching within the trader uses ODP's syntactic subtyping rule.

Note that in the above IDL–conversion example the conceptual graphs are only used for representing the operational interface of a type. Obviously, this is just a rather limited use, since much more information on a type could be represented in additional subgraphs of the same CG. Thus, IDL information as well as other information (e.g. the semantics of the service type) can be integrated into the same conceptual structure.

4 Prototype and Tool Support

A prototype of a knowledge–based trader has been implemented and placed in the public domain (see [9]). 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 handling of CGs and the user interactions with the

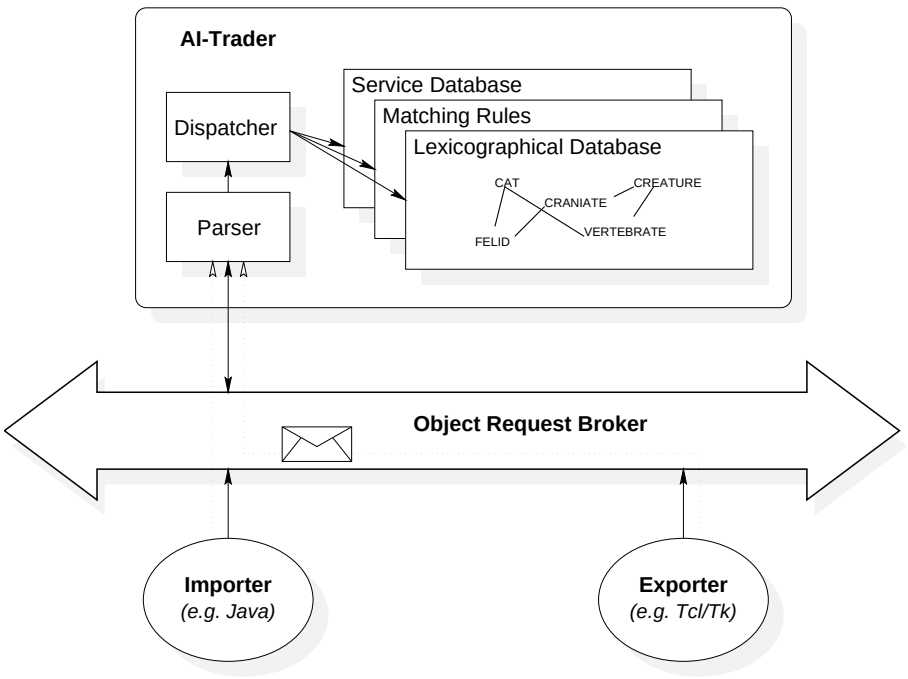


Figure 3: Implementation of the knowledge–based trader.

trading system. The trader is implemented as a separate Unix process running on top of the Orbix 2.0 Object Request Broker 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 CGs 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 knowledge–based trader consists of three components: a *service database*, a repository for *matching rules* and a *lexicographical database* (see Figure 3). The service database stores service types based on 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 use of conceptual graph matching rules to model different syntactic subtyping rules was demonstrated in the previous section. The lexicographical database maintains a comprehensive semantical network with linguistic information that is used by the matching rules. Thus, the lexicographical database provides "background" knowledge needed to match two conceptual graphs.

A knowledge–based trader builds up a knowledge base where each exported 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 CGs. 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 (i.e. extensions) 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 trading of types in open environments.

For the lexicographical database, we have converted the WordNet dictionary (see [6]) and extracted over 87.000 nouns from the English language with more than 95.000 relations between them. The linguistic matching rules make use of antonym, synonym and hyponym (i.e. specialization) relations found in WordNet. For example, the lexicographical database contains entries such as *printer* `IS_A` *hardware* and *love* `ANTONYM` *hate*. We have added another relation called `SUBTYPE` to store parameter subtype information (such as *short* `SUBTYPE` *long*). This relation is used within the ODP subtype rule to implement contra– and co–variance.

As part of the trading system, we have build tools for translating DCE and CORBA IDL to and from CGs. 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 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 the German Telekom. In the project 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.

4.1 Results

We have done some performance 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 task which has to be performed by any infrastructure offering some mediation functionality. As pointed out in section 3, 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 specification (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* interfaces. 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

	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.

all three subtyping rules, since T_2 lacks some specifications which only T_1 has). The performance tests have been performed on an IBM RS/6000 model 580.

The results reflect the complexity of matching two conceptual graphs. The ODP subtyping rule is more complex than the other two as it includes structural subtyping as well as contra– and co–variance. 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 that 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.

5 Conclusions

We have discussed the requirements for type specifications in open distributed processing environments, and we have presented a generic specification technique that can handle all known IDL specifications and that is suited to specify semantic type information which goes beyond the capabilities of IDL specifications. The technique is based on conceptual graphs, which were developed for the representation and machine–based processing of natural language. IDL specifications are translated to and from conceptual graphs automatically. We have integrated the specification technique into a knowledge–based trading system that is capable to operate in different type domains.

As part of the ODP standardisations a type repository is being defined. The repository stores information on the types that exist within some type domain. Based on our proposal for a meta–level specification technique it seems reasonable to define a generic type repository. It would use

a meta–notation to specify operational interfaces and type semantics in order to be independent of any concrete IDL technology, but would still enable the repository to provide specific IDLs by converting the meta–level definition. The reader is invited to test the specification and matching methodology by visiting our web–page (see [9]).

References

[1] W. Brookes, J. Indulska, A. Bond, and Z.Yang. Interoperability of Distributed Platforms: a Compatibility Perspective. In *3rd International IFIP TC6 Conference on Open Distributed Processing (ICODP'95)*, Brisbane, Australia, 20–24 February 1995. Chapman and Hall.

[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] C. Kaiser. Spezifikation operationaler Schnittstellen mittels kanonischer Konzeptgraphen–Templates. Diplomarbeit, Fachbereich Informatik, Goethe Universität, Frankfurt, August 1996.

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

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

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

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

[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.