Generic Trading Service in Telecommunication Platforms

A. Puder[1] and K. R"omer[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
roemer@cs.uni-frankfurt.de

Abstract. Telecommunication companies are currently defining a *Telecommunications Information Networking Architecture* (TINA) to meet the future demands of electronic markets. As part of those standardization efforts generic services are being defined. We demonstrate how a knowledge–based approach using conceptual graphs helps to develop a generic service for the mediation of services. The prototype of a knowledge–based trader is build according to the client/server paradigm which allows access to this service in a heterogeneous environment. This paper describes service mediation in general and detailing the matching of service offers and requests.

Keywords: TINA, RM–ODP, type system, interface definition language, trading.

1 Introduction

The constant growth of digital networks in telecommunication systems has opened new opportunities with respect to the provision of new services. The telecommunication companies have an interest not only to transport data but to provide user–oriented end–services via these digital networks. This results in the migration of classical services like yellow–pages towards computer supported services in the digital network. Telecommunication companies place high demands on the properties of the underlying infrastructure which are generally not met by the current Internet technology. Aspects like reliability, broadband communication, accounting and security require new technologies and new software distribution platforms.

Such demands require standardization of the underlying infrastructure in order to support interoperability in a world wide heterogeneous network. Several institutions work on technology independent standards which define frameworks for open distributed systems. For example the *International Organization for Standardization* (ISO) and the *International Telecommunication Union*, *Telecommunication Standardization Sector* (ITU–T) have developed the *Reference Model for Open Distributed Processing* (RM–ODP) (see [2], [3], [4] and [5]). The RM–ODP describes an architecture which supports the distribution, interoperability and portability of software components. The telecommunication companies have adopted the RM–ODP and refined it according to their needs. The resulting standardization effort is led by the *Telecommunications Information Networking Architecture* (TINA) Consortium; an international collaboration of telecommunication and IT companies aiming at defining and validating an open architecture for information and telecommunication services, which meets the future market needs (see [1]). The TINA architecture is based on distributed computing and object orientation.

Generic services are of particular interest as they can be customized easily for different environments. The German Telekom is developing in collaboration with the University of Frankfurt a prototype of a particular service: a generic trading service. In this novel approach we make use of conceptual graphs as a service type specification language. The structure of this paper is as follows: In section 2 we first provide the necessary background of service trading and then show how conceptual graphs can be used for type specifications and the matching of service offers and requests. In section 3 we show in more detail how the matching of conceptual graphs works. The current prototype of our knowledge–based trader is described next in section 4. In section 5 we present a conclusion and an outlook for future work.

2 Service Mediation in Open Service Markets

2.1 The RM–ODP model of service trading

The RM–ODP as well as TINA demand the support of a trading function which assists in the search for services (see [6]). All participating parties in a telecommunication system assume the roles of *service providers* and *service requesters*, which have no *a priori* knowledge about each other. In particular they are not linked statically to each other, but rather dynamically when they decide to participate in the open service market. During the mediation of services another party assumes the role of a *trader*, whose task is to match service offers and requests (see figure 1). According to the RM–ODP a mediation process is divided into several steps. First a service provider exports its service offer with the trader (1). At a later point in time a service requester tries to import a particular service (2). If the trader finds a matching service offer, which has previously been exported, it responds a reference to the service requester pointing to the appropriate provider (3). After a successful mediation process the service requester and provider are bound to each other and start to interact (4). The trading of services as described by the RM–ODP easily scales up as one trader can delegate import operations to another trader.

The notion of a *type system* is essential for the discussion of a mediation process (see [11]). For one thing a type system defines a *type specification language*, which allows the specification of services in open distributed systems. A

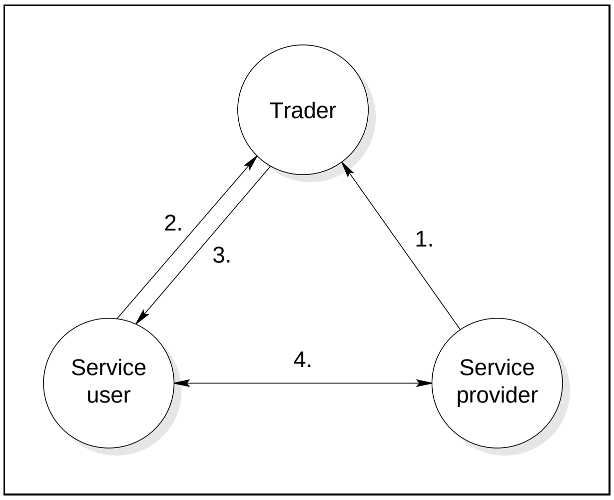

Fig. 1. Interactions during a mediation process.

type specification abstracts from specific internal details as it decouples the service usage from its implementation. Furthermore a type system defines *matching rules*, which allow the association of service offer and request, both represented as type specifications. 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.

Depending on the application domain, several different type systems have evolved. For example the type specification language of the RM–ODP as well as those of the middleware platforms DCE (see [9]) and CORBA (see [8]) focus on operational aspects of a type (i.e., how a service is accessible from a programming point of view). The corresponding type specification languages are commonly called *interface definition languages* (IDL). The matching rules for type specifications* written in these languages are quite different in the RM–ODP, DCE and CORBA (this will be explained in more detail in one of the following sections).

* For the domain of operational interfaces the matching rules are commonly called *syntactic subtyping rules*.

A general drawback of operational type specifications is the lack of semantics. The description of the semantics of a type specification is generally included in terms of informal comments which are ignored during service mediation. Although those comments are discarded by a middleware platform, they are nonetheless important during the mediation process where programmers browse IDL–repositories for appropriate interface specifications. It would therefore be useful to make use of the information contained in a comment during a mediation process.

This implies that service type specifications have different levels of abstraction with respect to the level of detail contained in the specifications (i.e., operational interface specification vs. informal comments). Existing service mediators are tailored for one of these levels of abstraction. In the following we demonstrate how conceptual graphs (see [13]) can help to build a generic trader which embraces those levels of abstraction.

2.2 Type specifications through CGs

In this section we concentrate on the specification of service types at different levels of abstraction and postpone the discussion of the matching of those specifications until a later section. Consider an object oriented decomposition of a bank application. There will most likely be a set of objects representing *accounts*. Conceptually an account object keeps track of a balance which can be altered using designated operations. With this informal description one could write an operational interface specification using the notation according to the CORBA–IDL:

```
/* Operational interface for a bank account */
interface Account {
    /* Deposit a certain amount to an account */
    void deposit( in unsigned long amount );
    /* Withdraw a certain amount from an account */
    void withdraw( in unsigned long amount );
    /* Show the current balance of an accout */
    void balance( out long amount );
};
```

An object of type `Account` offers three operations at its interface. The operation `deposit()` takes one input parameter which denotes the amount to be deposited. Likewise the second operation `withdraw()` withdraws a specific amount. Finally the third operation `balance()` returns the current total of the account as an output parameter. Since the balance can be negative, the type of the output parameter is `long` and not `unsigned long`.

The COBRA–IDL is tailored for the specifiation of operational interfaces. Programmers require a semantic specification of the behaviour as well. These are usually attached in form of comments throughout the interface specification (denoted through `/*` and `*/` within the CORBA–IDL for example). The comments are ignored by a CORBA compliant platform. We propose a generic type specification language based on conceptual graphs which can represent the informal description as well as the formal specification of the operational interface using one notation.

For the following analysis we limit the discussion to the specific details of an operation as defined by CORBA. As can be seen in the interface `Account`, an *operation* consists of an *operation name* and an ordered *set of parameters***. A parameter itself consists of a *directional attribute* (`in`, `out` or `inout`), a *parameter type* and an *identifier* denoting the name of the parameter. With this "anatomy" of an operation specification we can assemble a domain specific conceptual and relational catalogue. We have developed such a catalogue which provides the "vocabulary" to express the information contained in an operational interface specification (see [7]).

The conceptual graph depicted in figure 2 shows how to translate the operation specification for `deposit()` (concept nodes are denoted by white rectangles and relation nodes by grey rectangles). The `(ANNOTATION)` relation connects the formal specification of this operation to an informal description. The informal description is derived from the comment associated with the operation `deposit()` in the interface `Account`. Note that the conceptual graph shown in figure 2 is itself embedded in a conceptual graph representing the complete interface `Account`. This example shows how formal and informal descriptions representing specifications at different levels of abstractions are mapped onto one conceptual graph.

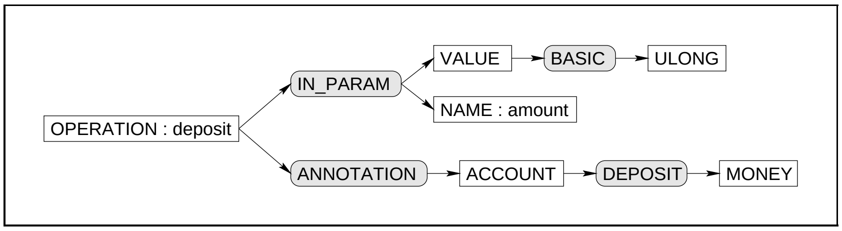

Fig. 2. Conceptual graph representing the specification of the operation `deposit()`.

2.3 Syntactic subtyping

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

** There are more components to an operation specification like exceptions or contexts which are of no importance for the scope of this paper. See [8] for a more detailed description.

DCE–IDL or the CORBA–IDL. The RM–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 RM–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 the RM–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. The RM–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.

Since operational interface specifications are translated to conceptual graphs during the mediation process as described in the previous section, there must exist different matching rules which are used by the trader during an import operation. The matching rule to be used depends on the context (i.e., the type systems the service user and provider employ). With our approach it is possible to match an operational interface written in DCE–IDL with another one written in CORBA–IDL using a matching rule according to the RM–ODP.

3 Matching Rules

As has been motivated above, the matching rules which are used to compare two conceptual graphs depend on the context. Thus during a mediation process three parameters are required: the service provider's type specification (in the following called *type graph*), a service user's type specification (in the following called *query graph*) and a set of rules which determine whether type and query graph match. The service user has to choose a subset of rules as a parameter of the import operation. Our prototype of a knowledge–based trader has six matching rules so far:

1. DCE subtyping
2. CORBA subtyping
3. RM–ODP subtyping
4. Specialization
5. Quantity
6. Negation

The first three rules model syntactic subtyping rules of various middleware platforms which have already been described in [12]. The last three rules model linguistic matching rules. The specialization rule matches a type graph with a query graph, if the specification contained in the former is more specific than the specification contained in the latter. The quantity rule matches a type graph `[OBJECT] -> (QUANTITY) -> [UNIT:`i_1`]` with a query graph `[OBJECT] -> (COMPARATOR) -> [UNIT:`i_2`]`, if COMPARATOR (i_1, i_2) holds (where COMPARATOR is something like *equal*, *less* or *greater*). The negation rule is non–monotonic in the sense that it reduces the set of matching type graphs. This rule determines if two sub–graphs of a type and query graph have opposite meaning. The opposite meaning is checked with the help of a lexicographical database which is part of the knowledge–based trader's architecture.

A matching rule has two input parameters — a type and a query graph — and one output parameter which reflects the *semantic distance* of those two graphs. The value of the output parameter is either -1 if the two graphs do not match or in the interval $[0,1]$. A value of 0 means that the semantic distance is unknown and the value of 1 that there is a best possible match between type and query graph (i.e., minimal semantic distance). If more than one matching rule is selected to determine the semantic distance, then the overall semantic distance is determined by:

$$MATCH(Q,T) =_{df} \begin{cases} \max_{1\leq i\leq n}\{MR_i(Q,T)\} & : \quad \forall i : MR_i(Q,T) \geq 0 \\ -1 & : \quad \text{otherwise} \end{cases}$$

where $MR_1, MR_2, \ldots, MR_k$ denote the matching rules which were selected. That is type and query graph do not match if at least one matching rule rejects them (i.e., returns -1), otherwise the overall result is the maximum match value determined by all the selected rules.

In the following sections we present formal specifications for the aforementioned linguistic matching rules. Although the matching rules return a semantic distance in terms of a numeric value, it should be noted that the individual nodes of the type and query graph are themselves not assigned a weight.

3.1 Specification of the Specialization Rule

The matching rule for specialization should obey the following design principles:

1. The more branches of a type and a query graph match modulo a lexicographical database, the better should the match be evaluated (i.e., the semantic distance should become less).
2. The influence on the semantic distance between type and query graph decreases for information deeply embedded within the type and query graphs. This guarantees that aspects of a service type specification located closer to the root node of a CG contributes more to the semantic distance than very specific details of a service deeply embedded in the CG.
3. If parts of a type graph are rejected (i.e., evaluated with -1), then this should not necessarily lead to a rejection of the type graph as a whole.

Now we present a formal specification of the specialization rule which follows the above design principles. The definition makes use of a 2–ary predicate

$\sim_{R_1,R_2,\ldots,R_l}(W_1, W_2)$ which is true iff the words W_1 and W_2 are related via the lexicographical relations $R_1, R_2, \ldots, R_l$ in the lexicographical database. In the following definition the general structure of a type and a query graph is depicted in figure 3. Only two levels of successors of the nodes Q and T are shown. Each node Q_{ij} and T_{kl} can have further subgraphs.

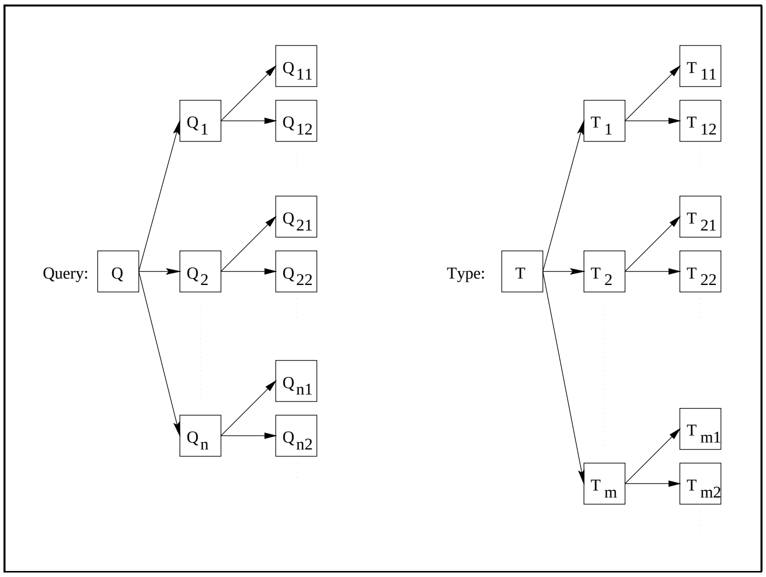

Fig. 3. General structure of a type and a query graph.

The formal definition of the matching rule for specialization is represented by a 2–ary function called *SPEC*. The value of *SPEC* is determined according to the following algorithm:

1. If not $\sim_{IS\_A,SYNONYM}(T, Q)$ then $SPEC(Q,T) =_{df} 0$, STOP.
2. If $n = 0$ (i.e., the query graph consists of only the root node) then $SPEC(Q,T) =_{df} 1$, STOP.
3. Calculate the semantic distances $w_1, \ldots, w_n$ of Q's subgraphs $Q_1, \ldots, Q_n$ as follows (note the recursive definition via the aforementioned function *MATCH*, which applies all selected matching rules to subgraphs):

$$w_i =_{df} \begin{cases} \max_{1\leq j\leq m}\{MATCH(Q_i,T_j)\} & : \ \forall j : MATCH(Q_i,T_j) \geq 0 \\ -1 & : \ \text{otherwise} \end{cases}$$

4. If Q denotes a concept node then $SPEC$ is defined as:

$$SPEC(Q,T) =_{df} \begin{cases} \frac{1}{n}\sum_{i=1}^{n} w_i & : \ \forall i : w_i \geq 0 \\ 0 & : \ \text{otherwise} \end{cases}$$

Otherwise Q denotes a relation node and *SPEC* is defined as:

$$SPEC(Q,T) =_{df} \begin{cases} \frac{1}{n}\sum_{i=1}^{n} w_i & : \quad \forall i : w_i > 0 \\ 0 & : \quad \text{otherwise} \end{cases}$$

STOP.

Example Let Q be a query graph and T_1 and T_2 be two type graphs as shown in figure 4. The semantic distances of $MATCH(Q,T_1)$ and $MATCH(Q,T_2)$ are to be computed where the set of selected matching rules is $\{SPEC\}$. Assume the lexicographical database contains entries such that $\sim_{IS\_A,SYNONYM}(X,X)$ is true for each $X \in \{a,b,\ldots i\}$.

During the execution of $MATCH(Q,T_1)$ $MATCH$ is recursively invoked on each node of the query graph. The numbers at the upper right corners of the nodes in figure 4 are the results of those recursive invocations of $MATCH$. That is 0.5 (the number at the root node) is the overall result of $MATCH(Q,T_1)$. Similarly 0.75 is the result of $MATCH(Q,T_2)$.

Those examples hint that design–goals (1) and (2) are met by the matching rule $SPEC$. Design goal (3) is met due to the fact that a reject (i.e., value of -1) is not propagated to the root but only one level up. 2

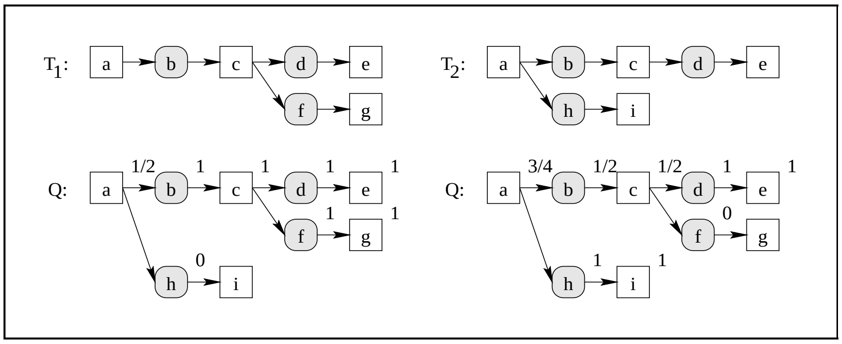

Fig. 4. Sample type and query graphs.

3.2 Specification of the Negation Rule

The design criteria for the negation rule are somewhat simpler than those for the specialization rule: reject (i.e., return -1) if type and query graph have opposite meaning with respect to the lexicographical database.

In contrast to the specialization rule both the negation and quantity rules are non–recursive in the sense that they do not apply themselves to subgraphs. But if used together with the specialization rule they are in fact applied recursively to all subgraphs due to the fact that the specialization rule calls $MATCH$ on subgraphs which in turn evaluates all selected matching rules on the subgraphs.

The formal definition of the matching rule for negation is represented by the 2–ary function *NEG*. Again the following definition refers to the general structure of a type and a query graph as depicted in figure 3. The value of *NEG* is determined according to the following algorithm:

1. If Q denotes a relation node then $NEG(Q,T) =_{df} 0$, STOP.
2. Calculate $NEG(Q,T)$ as follows:

$$NEG(Q,T) =_{df} \begin{cases} -1 & : \quad \exists i,j,k,l : isa(T,Q) \wedge antonym(T_k,Q_i) \wedge isa(T_{kl},Q_{ij}) \\ 0 & : \quad \text{otherwise} \end{cases}$$

where

$$isa(T,Q) =_{df} \sim_{IS\_A,SYNONYM} (T,Q)$$
$$antonym(T,Q) =_{df} \sim_{ANTONYM,SYNONYM} (T,Q)$$

STOP.

Example Consider the following type graph T and query graph Q:

```
T: [CAT]->(HATE)->[DOG]
Q: [ANIMAL]->(LOVE)->[DOG]
```

$MATCH(Q,T)$ with $\{NEG\}$ as the set of selected matching rules evaluates to -1, assuming the lexicographical database contains entries that make all of the following predicates evaluate to true:

$$\sim_{IS\_A,SYNONYM} (CAT, ANIMAL)$$
$$\sim_{ANTONYM,SYNONYM} (HATE, LOVE)$$
$$\sim_{IS\_A,SYNONYM} (DOG, DOG)$$

3.3 Specification of the Quantity Rule

Using concept nodes of the form `[UNIT:`val`]` one can specify a quantity of val units of `UNIT` (e.g., `[DAY:10]` corresponds to ten days). The quantity rule should accept (i.e., return 1) if the type graph specifies a quantity i_1 of some unit and the query graph specifies a quantity i_2 of the same unit together with a comparator r (e.g., $<, =, >$) and the condition $r(i_1, i_2)$ is fullfilled.

The formal definition of the quantity matching rule is represented by the 2–ary function *QUANT*. The following definition assumes the existence of a unary function $value(C)$ that returns the value specified within concept node C. If C contains no value 0 should be returned. Figure 3 shows the general structure of a type and a query graph. The value of $QUANT$ is determined according to the following algorithm:

1. If Q denotes a relation node then $QUANT(Q,T) =_{df} 0$, STOP.
2. Calculate $QUANT(Q,T)$ as follows:

$$QUANT(Q,T) =_{df} \begin{cases} 1 & : \quad \exists i,j,k,l : \begin{array}{c} isa(T,Q) \wedge \\ iscomp(Q_i) \wedge isquant(T_k) \wedge \\ syn(T_{kl}, Q_{ij}) \wedge \\ Q_i(value(T_{kl}), value(Q_{ij})) \end{array} \\ 0 & : \quad \text{otherwise} \end{cases}$$

where

$$\begin{aligned} isa(T,Q) &=_{df} \sim_{IS\_A,SYNONYM} (T,Q) \\ syn(T,Q) &=_{df} \sim_{SYNONYM} (T,Q) \\ isquant(Q) &=_{df} isa(Q, "QUANTITY") \\ iscomp(Q) &=_{df} isa(Q, "COMPARATOR") \end{aligned}$$

STOP.

Example Consider the following type graph T and query graph Q:

```
T: [CAT]->(AGE)->[YEAR:5]
Q: [ANIMAL]->(LESS)->[YEAR:10]
```

$MATCH(Q,T)$ with $\{QUANT\}$ as the set of selected matching rules evaluates to 1, assuming the lexicographical database contains entries that make all of the following predicates evaluate to true:

$$\begin{aligned} &\sim_{IS\_A,SYNONYM} (CAT, ANIMAL) \\ &\sim_{IS\_A,SYNONYM} (AGE, QUANTITY) \\ &\sim_{IS\_A,SYNONYM} (LESS, COMPARATOR) \\ &\sim_{SYNONYM} (YEAR, YEAR) \end{aligned}$$

4 Architecture of the Knowledge–based Trader

The knowledge–based trader consists of three components: a *service database*, a repository for *matching rules* and a *lexicographical database* (see figure 5). 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. All six matching rules mentioned before are implemented within the PROLOG sub–system of the knowledge–based trader. 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.

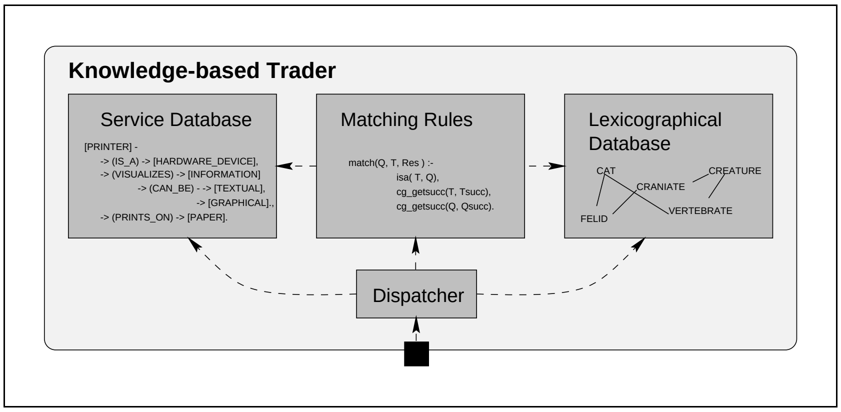

Fig. 5. Architecture of the knowledge–based trader.

The knowledge–based trader is generic in the sense that it works solely with conceptual graphs. Beside the service mediation specific translators allow the conversion between the DCE– and CORBA–IDL. 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. We have performed timings of the three matching rules which handle subtyping of IDLs according to the conventions defined by DCE, CORBA and RM–ODP (see [12]).

The implementation of a matching rule within the knowledge–based trader has access to the details of the type and query graphs to be matched via build–in predicates. The following six predicates are available for the implementation of matching rules:

match($+queryGraph, +typeGraph, -matchResult$)
: Entry point of a matching rule. Compares *queryGraph* with *typeGraph*. The result of the match (i.e., the semantic distance) has to be unified with *matchResult*.

cg_match($+queryGraph, +typeGraph, ?matchResult$)
: This predicate applies all selected matching rules recursively with *queryGraph* und *typeGraph*. The semantic of this predicate is identical with `match/3`.

cg_getsucc($+cgNode, ?successorList$)
: Unifies *successorList* with a list of sucessor nodes of *cgNode*.

cg_getname($+cgNode, ?nodeName$)
: Unifies *nodeName* with a readable string associated with *cgNode*.

cg_gettype($+cgNode, ?nodeType$)
: Determines whether *cgNode* is a concept or relation node. The variable *nodeType* is bound to `concept` or `relation` accordingly.

cg_matchword($+word1, +relationList, +word2$)
: Corresponds to $\sim_{R_1,R_2,\ldots,R_l}$ (`word1`, `word2`) where *relationList* contains $R_1, \ldots, R_l$. This predicate fails if no path between the words using the given lexicographical relations can be constructed.

A matching rule which makes use of the above predicates has access the other components of the knowledge–based trader. The predicate `cg_matchword/3` accesses information contained in the lexicographical database. The predicates `cg_getsucc/2`, `cg_getname/2` and `cg_gettype/2` access information in the service database. See appendix A for an example.

5 Conclusion and Outlook

In this paper we have presented the architecture of a generic trading service for telecommunication systems where conceptual graphs are used as a type specification language. As has been shown, this methodology allows service type specifications at different levels of abstraction. With the proper tool support, operational interface specification can be translated to conceptual graphs. The matching of import and export during a mediation process depends on the context which are modeled through different matching rules. Each rule is tailored for a specific context.

There remain a couple of refinements which need to be investigated. The performance measurements we have performed indicate that new approaches must be taken if the knowledge–based trader were to be exposed to a commercial environment. Especially promising seem to be techniques for computing hash values from CGs to support caches. Furthermore the human computer interface to the knowledge–based trader would benefit from natural language to CG translators to facilitate the formulation of import operations. Finally we intend to extend the RM–ODP model of service mediation to relax the strict 1–to–1 mapping of request and offer. It seems feasable to support a 1–to–n mapping where several service providers are "composed" to achieve in a combined effort the needs of a service requester.

Appendix A: Implementation of the Specialization Rule

In the following we demonstrate the implementation of the matching rule $SPEC$ as described in section 3.1. The predicates prefixed by `cg_` are explained in section 4. The code as shown below is taken without modification from the initialization files of our knowledge–based trader (see [10]). The code itself is located in the component for matching rules within the knowledge–based trader.

```
@define-matching-rule 'Specialization'

 isa(Node1, Node2) :-
         cg_getname(Node1, Name1), cg_getname(Node2, Name2),
         cg_matchword(Name1, ['IS_A', 'SYNONYM'], Name2).

 match(Q, T, Res) :-
         cg_gettype(Q, concept), isa(T, Q),
         cg_getsucc(T, Tsucc), cg_getsucc(Q, Qsucc),
         match_lists_con(Qsucc, Tsucc, Res, 0, 0).

 match(Q, T, Res) :-
         cg_gettype(Q, relation), isa(T, Q),
         cg_getsucc(T, Tsucc), cg_getsucc(Q, Qsucc),
         match_lists_rel(Qsucc, Tsucc, Res, 0, 0).

 best_match(_, [], MaxVal, MaxVal).
 best_match(Q, [T|Tlist], MaxVal, Res) :-
         (cg_match(Q, T, Val) ; Val is 0),
         (Val < 0
         ->      Res is Val
         ;       best_match(Q, Tlist, max(Val, MaxVal), Res)).

 match_lists_con([], _, 1000, _, 0) :- !.
 match_lists_con([], _, Res, Sum, NSucc) :-
         Res is round(Sum/NSucc), Res > 0.

 match_lists_con([Q|Qlist], Tlist, Res, Sum, NSucc) :-
         best_match(Q, Tlist, 0, BestMatch),
         (BestMatch > 0
         ->      NewSum is Sum+BestMatch
         ;       NewSum is Sum),
         match_lists_con(Qlist, Tlist, Res, NewSum, NSucc+1).

 match_lists_rel([], _, 1000, _, 0) :- !.
 match_lists_rel([], _, Res, Sum, NSucc) :-
```

```
37          Res is round(Sum/NSucc), Res > 0.
38
39   match_lists_rel([Q|Qlist], Tlist, Res, Sum, NSucc) :-
40          best_match(Q, Tlist, 0, BestMatch), BestMatch > 0,
41          match_lists_rel(Qlist, Tlist, Res, Sum+BestMatch, NSucc+1).
42
43 @end
```

The predicate `isa/2` in lines 3—5 implements $\sim_{IS\_A,SYNONYM}(Q,T)$. Predicate `match/3` defined in lines 7—15 is the entrypoint of the matching rule, see section 4 for details. To avoid using floating point numbers the result is scaled by 1000, that is `Res` $\in \{-1, 0, 1, 2, \ldots, 1000\}$. If `Q` denotes a concept node then the first branch of `match/3` in lines 7—10 is taken, otherwise the second branch in lines 12—15. `best_match/4` in lines 17—22 implements step 3 of the algorithm in section 3.1. The predicates `match_lists_con/5` and `match_lists_rel/5` in lines 28—33 and 35—41 implement the two cases of step 4 of the algorithm.

References

1. F. Dupuy, G. Nilsson, and Y. Inoue. The TINA Consortium: Toward networking telecommunications information services. *IEEE Communications Magazine*, pages 78–83, November 1995.
2. 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.
3. ITU.TS Recommendation X.902 — ISO/IEC 10746–2: Basic Reference Model of Open Distributed Processing Part 2: Descriptive Model, 1994.
4. ITU.TS Recommendation X.903 — ISO/IEC 10746–3: Basic Reference Model of Open Distributed Processing Part 3: Prescriptive Model, February 1994.
5. ITU.TS Recommendation X.904 — ISO/IEC 10746–4: Basic Reference Model of Open Distributed Processing Part 4: Architectural Semantics, 1994.
6. ODP Trading Function, ITU/ISO Committee Draft Standard ISO/IEC DIS13235 Rec. X.9tr, May 1995.
7. C. Kaiser. Spezifikation operationaler Schnittstellen mittels kanonischer Konzeptgraphen–Templates. Diplomarbeit, Fachbereich Informatik, Goethe Universität, Frankfurt, August 1996.
8. Object Management Group (OMG), The Common Object Request Broker: Architecure and Specification, Revision 2.0, July 1995.
9. Open Software Foundation. *Introduction to DCE.* Open Software Foundation, Inc., 1992.
10. A. Puder. Introduction to the AI–Trader Project. http://www.vsb.informatik.uni–frankfurt.de/projects/aitrader/, Computer Science Department, University of Frankfurt, 1995.
11. A. Puder and C. Burger. New concepts for qualitative trader cooperation. In A. Schill et al., editors, *International Conference on Distributed Platforms*, pages 301–313. Chapman & Hall, February 1996.
12. A. Puder and K. Geihs. System support for knowledge–based trading in open service markets. In *7th ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996.

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