New Concepts for Qualitative Trader Cooperation

Arno Puder[1] and Cora Burger[2] ‡

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

[2] Institute of Parallel and Distributed High–Performance Systems (IPVR)
University of Stuttgart
Breitwiesenstr. 20–22
70565 Stuttgart, Germany
caburger@informatik.uni-stuttgart.de

Abstract

Traders serve as mediators between service providers and users. The trader returns to a service requester by delivering a set of suitable providers that satisfy the request. Because of the size and complexity of distributed systems, one trader manages a certain part of the whole set of providers only. To enlarge this restricted set of potential service providers, a lightweight form of cooperation (called interworking) between two or more traders was introduced. Therefore, the main purpose of such cooperations was concerned with quantitative aspects, neglecting the possibilities of more general concepts of cooperation. A need for qualitative trader cooperations arises. Traditional trader approaches focus upon the operational interface of providers as a basis for type descriptions. But in general, application users in an open electronic market require mediation at different levels of abstraction. Suitable traders are specialized in one of these levels. By combining these isolated solutions, a qualitative trader cooperation can be achieved.

In the following, we take a closer look at type descriptions at different levels of abstraction. Similarities will be discovered that can be used to derive appropriate mappings. In particular we demonstrate that ODP–types based on an arbitrary interface definition language can be described in terms of conceptual graphs that are used in knowledge–based trading. From this relationship, a general type manager can be derived, coping with both type concepts. As a consequence, the trading of operational interfaces and the trading based on a knowledge representation scheme can be combined. Together they exhibit a more powerful trading service by offering different interfaces to different kinds of users.

Keywords: Trading, type manager, cooperation

1 Introduction

In future open distributed systems, mediation functionality will be required at very different levels and in many different ways. For instance, application users will be looking for suitable

‡Supported by *German Hochschulsonderprogramm II*

partners to work on a certain topic, they will search documents or services fulfilling certain criteria. Robots and so–called intelligent agents need suitable partners to solve problems, and some client programs query traders to find suitable servers for late binding at runtime. The mediation functionality required in such cases is comparable. This suggests the need to search for similarities between current approaches and to try to combine them to achieve a general solution to mediation problems.

Up to now, the combination of mediating components was used in a rather restricted sense (cf. [2], [11], [10], [23], [7]). Its main purpose was to cope with the size of distributed systems and with the existence of different organizational domains, each one having its own mediating component. These components cooperate by sharing information about potential partners across domain boundaries (also called *interworking* of traders). By enlarging the set of offering partners and potential requesters, only an amelioration in quantitative aspects has been achieved. In [5] we have already shown, that cooperation between mediating components can concern qualitative aspects as well (e.g. by realizing some kind of load balancing between domains). In the following, we will develop an example by first comparing and then combining different mediation concepts.

Solutions to the mediation problem are based on characterizing providers and on matching such characterizations. To enable selection processes, equivalence classes are introduced by means of types and their relationships. For instance, videos could be subsumed under types like "documentation video" or "movie", printer servers under the type "printer". Each person or object offering a certain facility as well as others requesting a facility must specify its type. Thus, the typing of services plays a central role in mediation. To be able to combine different approaches, these aspects must be studied in more detail.

The most popular approach is the one of ODP–trading that is going to be standardized by ISO and ITU and which aims at supporting client–server binding at runtime (cf. [10]). Traders based on ODP use an interface definition language (e.g. ANSA–, DCE– or CORBA–IDL) along with a set of properties to define a standardized service type (cf. [8], [17], [23]). One drawback of this approach consists in relying heavily on a common understanding about the semantics of service types. Furthermore, conversions between each language and all other languages are needed to enable world wide trading (cf. [23]), which leads to an unnecessary overhead.

In the field of distributed artificial intelligence, capabilities and needs of intelligent agents are described using a fixed set of characterizations (cf. [22]) or the combination of KQML (knowledge query and manipulation language) and KIF (knowledge interchange format) as described in [7]. To support the interaction with application users, either generic dialogues (cf. [14]) or conceptual graphs (cf. [20]) are currently used. The latter approach is called knowledge–based trading and will be used for cooperation with ODP–based traders to obtain a mediation service of higher quality. As will be shown in subsection 4.3, service types defined in an IDL can be expressed by conceptual graphs. This fact enables the combination of ODP–based and knowledge–based trading to support software components as well as application users.

The structure of this paper is as follows: First the basic architecture for traders is introduced (section 2). Main components of this architecture are the type manager and the function for matching and selection, both working on types. The corresponding definitions of types are studied for the case of ODP–based and knowledge–based trading. Their differences are demonstrated by a common example: a print service is characterized by means of the IDL approach and by means of conceptual graphs. Section 3 discusses the differences in detail. Based on the results, section 4 outlines how service types being defined in an IDL can be expressed by conceptual graphs and derives a general type manager which copes with both kinds of service types. Furthermore, consequences for matching components are explained. Section 5 depicts the background for a realization of the new concepts and presents a conclusion and outlook.

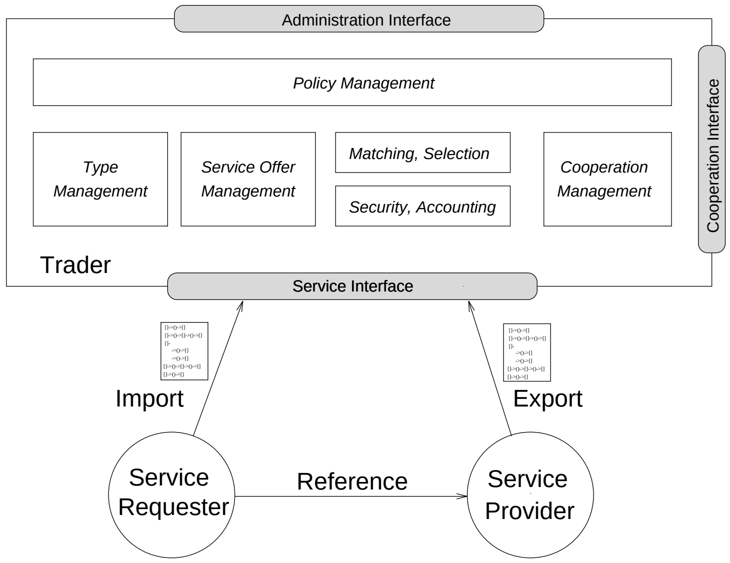

Figure 1: Basic architecture of a single trader.

2 Basic Trader Architecture and Type Definitions

A first approach to the cooperation of ODP–based and knowledge–based trading consists in classifying client interactions according to their abstraction level (ODP–conformant or knowledge–based by application users) and in introducing a distribution component. This component has to recognize the level of abstraction of a certain offer or request and pass it to a suitable trader. I.e., all data from "low level" clients which require operational interfaces are sent to ODP–based traders and all data from "high level" clients seeking services based on generic interfaces are sent to knowledge–based traders. This distribution component could be integrated in existing traders or remain separated. In the integrated case, traders have to interact to exchange offers and requests from clients of the other level of abstraction. Thus, a loose cooperation between traders takes place.

To develop a more comprehensive solution, the architecture of traders has to be studied. From current trader implementations (cf. e.g., [1], [12], [13], [15]), some requirements for such an architecture can be derived. It must contain the following components (cf. figure 1):

- Interfaces for the trading service, for administration and for cooperation with other traders
- Management of policies, of cooperations, of offered services and of types
- Matching and selection
- Security and accounting of trading service

The functionality and collaboration of these components is straightforward. On one hand, various interfaces support interactions with appropriate partners (clients, administrators or other traders), on the other hand, management components take care of storage, modification and retrieval of objects. For instance, administration components control the trader's behavior by entering, modifying or removing policies that are managed by the policy management, trader components negotiate a certain kind of cooperation that is managed by the cooperation management of each participant. All partners of a particular trader have to be authenticated and possibly some of the interaction messages have to be enciphered resp. deciphered. Besides these tasks of a security component, resource consumption for serving requests of clients and other traders could be accounted for.

Having in mind that the cooperation of traders should allow different levels of abstraction in terms of their usage, only a few components of the trader's architecture are relevant. To identify them, the interactions between a trader and its clients have to be explained in more detail. There are two roles of clients, service requesters and service providers. The latter *export* their service offers to be registered by the service offer management in the trader's database. Further interactions can take place to modify or retract these offers or to update the values of dynamic service properties. On the other hand, service requesters search for services fulfilling certain criteria, i.e. they want to *import* service descriptions to be able to contact *a priori* unknown service providers. Such requests are treated by the component for matching and selection.

To structure and organize the search for suitable services in an efficient way, the type manager takes care of the management of service types and their relationships (cf. [8]). If there is no offer available in the trader's database for a requested type, the type manager can be asked for a functionally equivalent service type. Such relationships between types will be discussed in more detail in the following subsection.

Main differences between ODP– and knowledge–based trading arise from the management of service offers, from type management and from the component responsible for matching and selection. This is due to the fact that ODP– and knowledge–based trading use different notations to describe services and their types. Whereas in ODP the description of services focuses around operational interfaces based on interface definition languages (IDL), knowledge–based trading applies a knowledge representation notation based on *conceptual graphs* (see [21]). At first sight, these concepts look totally different. But by studying them in more detail, similarities can be found. In the following we concentrate on the description of service types and their relationships and depict the underlying concepts for ODP– and knowledge–based trading.

2.1 ODP–conformant Type Definition

According to ODP, different kinds of types are involved in the description of services. Whereas *service types* describe the abstract functionality of a service, an interface type defines the computational behavior of the according interface. The computational behavior itself is defined via a set of signatures each describing a method name and zero or more input/output parameters (parameter types). Figure 2 shows the composition of a service being an instance of the service type.

Besides this basic characterization of a service, a more fine–grained specification can be obtained by adding a set of general and specific properties like expiration date or physical characteristics of the underlying hardware (e.g. the DPI a printer supports). These properties are described by a pair of (name, value) and can be static or dynamic. For dynamic properties, a service offer evaluator interface has to be defined to be able to get current values[1].

Interface types are defined in an arbitrary *interface definition language* (IDL). For example, in CORBA–IDL (cf. [16]) the interface of a print service could be defined as follows:

[1] In [12] this was even driven a step further by enabling the service offer evaluator as well as the trader to activate the refresh of values depending on the frequency of access and changes.

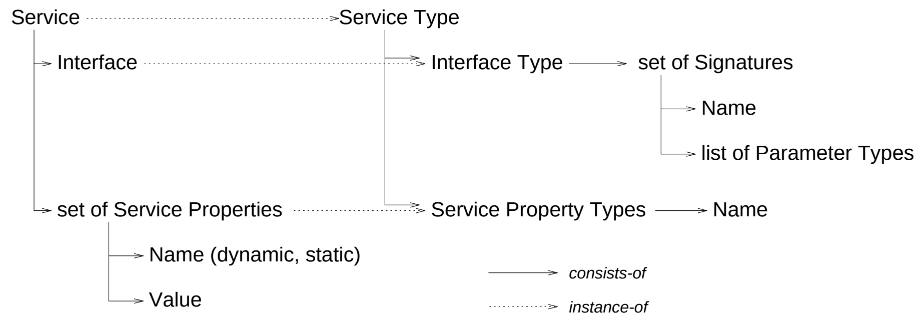

Figure 2: Relationship of Service, Interface and Service Property.

```
interface PrintServiceInterface  {
    // Submit Job operation
    void SubmitJob (
      in  string  Data,
      in  integer NumberOfCopies,
      out integer JobNumber
    );

    // Kill Job operation
    void KillJob (
      in integer JobNumber
    );
}
```

This means, that the print service has two operations, one to submit a job with a certain number of copies and another one to kill jobs. By using a hypothetical, but CORBA–like notation, the following property types can be added to this rudimentary description, where `QueryPrinterQoS` denotes the interface of the corresponding service offer evaluator:

```
properties PrinterQoS {
    static  string Name;
    static  int    DPI;
    dynamic int    QueueLength
                      queryinterface QueryPrinterQoS;
}
```

To enable flexible matching and selection processes, service types have to be compared by studying their relationships: all service types being related in a certain way are candidates for the same kind of requests, i.e. they can be substituted without breaking any expectations on the side of a client or a server. This property is one explication of *polymorphism* where a service is not accessed under its own type, but rather a polymorphic one[2].

Furthermore, types of different organizational domains can be functionally equivalent despite different characterization schemes. For instance, in one domain the service type "printer" holds, whereas in another domain the equivalent service is called "print". Mappings between equivalent types must be known if traders of different domains cooperate (cf. [4]). A special

[2] The conditions which must hold for polymorphism go beyond the scope of this paper. See [9] for a definition of a syntactic subtype relation rule.

case for equivalences in general are conversion relationships, e.g. one service uses parameters in centimeters, the other one in inches. These conversions play an important role for the binding between service requesters and providers. Finally, the grouping of interface types and sets of property types to service types as shown in figure 2 is itself a relationship. Services may be viewed as instances of this kind of relationship.

For the comparison with knowledge–based trading it should be noted, that relationships of ODP–types exhibit properties like reflexivity, symmetry, transitivity and the cardinality of involved types. Like relationships in general, type relationships can be instantiated either implicitly (by means of rules, contra– covariance or semantical subtyping) or explicitly by some application user, e.g. the administrator. In ODP it has not been defined yet, how the validity of types can be checked and how type relationships are obtained.

Furthermore, the ODP documents make no assumption about the precise nature of the definition of a service type. Although a service type and an interface are different in terms of level of abstraction, the tight coupling of them focuses on the operational behavior of the ODP–type definitions. The class of different services which all use the same interface type is completely disregarded (e.g. the services which are accessible via HTTP in the World Wide Web). Naturally this kind of type definition neglects application users which have a more abstract view on services. Thus ODP supports typing on an interface level and doesn't offer suitable abstractions for more abstract kind of type notations.

2.2 Knowledge–based Type Definition

With the emergence of high level services as found in the World Wide Web (see [3]) and *generic clients* (see [14] and [6]) a 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. The need for a service of a particular type arises during runtime and not at compile time. 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.

The knowledge–based trading scheme as described in [20] and [19] addresses these problems. It introduces a type description language based on *conceptual graphs* which are particularly useful to support the cognitive domain of application users (see [21]). Furthermore the introduction of a *machine learning algorithm* solves the problem of meta–trading: in the knowledge–based trading framework various ways to describe a service are being learned over time and therefore cope with individual views that different users might have on a service.

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

A concept node represents either a concrete or an abstract object in the world of discourse. As for the context of service types a concept may be a concrete object such as `PRINTER`, `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 conceptual graph describes a printer (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, visualizes information which can be either textual or graphical and it prints on paper or slides.*

```
[PRINTER] -
        -> (IS-A) -> [HARDWARE-DEVICE],
        -> (CONNECTED-TO) -> [COMPUTER],
        -> (VISUALIZES) -> [INFORMATION] -> (CAN-BE) -
                                               -> [TEXTUAL],
                                               -> [GRAPHICAL].,
        -> (PRINTS-ON) -
                -> [PAPER],
                -> [SLIDES]..
```

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.

3 Comparison of ODP– and knowledge–based Type Management

Each of the two kinds of traders introduced above, an ODP–based and an knowledge–based trader, have their specific clients at different levels of abstraction. An ODP–based trader is mainly used for type matching and checking at runtime. Although this trader allows programmers to browse the type domain in search for appropriate service types, it does not address important issues concerning type management in general. Set aside the fact that a knowledge–based trader typically only has human users, it is more concerned with questions on *how* the knowledge of a service type is obtained.

Whereas in the ODP context trading is more on an instance level, an knowledge–based trader mediates types to alleviate the need for an a priori knowledge of these. Meta–trading thus trades the knowledge of types and not of instances as in the ODP context. Despite those differences, the type managers of both traders are similar in the sense that the structure of their type repositories from an abstract point of view is the same.

3.1 Comparison of Type Definitions

The type repository of an ODP–based trader is able to store elementary type information based on the classification depicted in figure 2 such as *parameter types*, *interface types*, *service property types* and *service types*. Relationships between various types have been introduced to cope with important questions like polymorphism or federation of types. A bird's eye view of this type repository reveals a graph in the mathematical sense, where nodes represent types and arcs different kinds of relationships between these.

The type repository of a knowledge–based trader on the other hand stores a set of conceptual graphs representing its knowledge base. A conceptual graph stems from a mathematical domain and is a bipartite graph where the relation nodes may be viewed as labeled arcs between two or more concepts. Although a knowledge–based trader itself does not define elementary data types found in ODP, it will be shown in section 4.2 that these can easily be modeled by appropriate conceptual graph templates.

Each of the two type definitions is intended for different kinds of service requesters. An ODP–based trader focuses on type definitions which denote the precise operational behavior of service offers and provide only little support for type management on a cognitive level. A knowledge–based trader on the other hand supports human users which often only have a vague idea of the kind of service they desire. In particular it should be noted that ODP only helps programmers by means of a *browsing tool* as a solution to meta–trading.

3.2 Comparison of Interaction Patterns

A closer look on the interaction pattern between the two traders and their embedded type managers reveals a further difference. During the execution of the match operation the ODP trader performs queries to its type manager to check for example for appropriate subtypes. Interaction with the type manager is therefore interleaved with the actual match algorithm performed within the trader. A knowledge–based trader on the other hand first retrieves a complete subgraph — namely a conceptual graph — from its type manager before it starts processing the match operation to compute the semantic distance of a query and a type graph.

An ODP–based trader has a very focused view on the data structures that are held in the type repository and a knowledge–based trader treats whole subsets of these data structures as one semantic context. An ODP–based trader therefore has an *intentional interpretation* of type repository and a knowledge–based trader an *extensional interpretation.*

Apart from different interpretations of the information found in the type repository, the type manager could be shared among them. This results in a different architecture where the type manager is not part of a trader in general but rather an independent component which is being shared among several traders. We propose a qualitative cooperation between traders by defining a general type manager through its interface and its data model. This supports the current trend in ODP to separate type manager and trader.

4 A General Type Manager Model

This section describes a *General Type Manager* to achieve a qualitative cooperation of the two aforementioned different trading approaches. This cooperation is based upon the above discussion of the two type definitions. By sharing the type manager among different traders and therefore placing it outside the trader (see figure 3), we emphasize the importance of this component.

It will not be discussed whether further trader components — such as the service offer management — could be shared too. We focus upon the structure of the general type manager serving both an ODP–based and a knowledge–based trader. It will be described in the following two sections by defining its operational interface and its internal type repository (section 4.1 and 4.2 respectively). It should be noted that this description is by no means complete with respect to the current ODP–standard. A complete description would go beyond the scope of

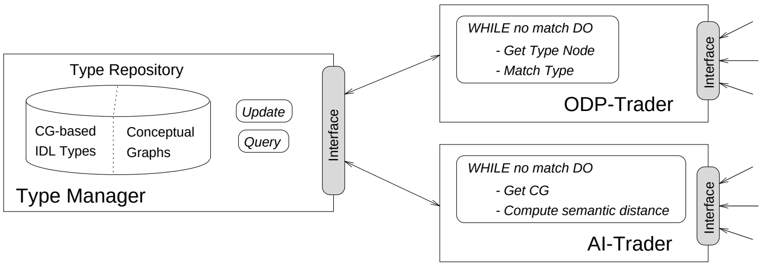

Figure 3: Qualitative trader cooperation via the *General Type Manager*.

this paper and therefore the following explication is meant to convey the idea on how the two different type definitions might be merged into one. Finally in section 4.3 an example is given how conceptual graphs may be used to describe an operational interface.

4.1 Operational Interface of the General Type Manager

As the general type manager is being shared among ODP–based and knowledge–based traders, its operational interface must suffice the requirements of both traders. The type manager maintains a type repository of all type graphs known in a particular domain. It must offer its clients (i.e. traders) operations to update and query its knowledge base. In section 3.2 it has been argued that an ODP–based trader has an intensional view on the type repository whereas a knowledge–based trader has an extensional view. This translates immediately to specific requirements of the operational interface. A suitable interface which the general type manager provides is defined by:

- retrieve a conceptual/relation node
- retrieve all successor nodes from a given node
- insert/update/remove a conceptual/relation node
- insert/remove directed arc between two existing nodes

The first two operations allow queries and the last two operations updates on the type repository (see also figure 3). It should be noted that all operations must satisfy invariants which hold for conceptual graphs (e.g. that all type graphs remain connected, directed and bipartite graphs). It might be feasible to extend the interface to cope with complete subgraphs as a parameter.

4.2 Common Type Repository

A central component of the type manager is its repository where all relevant information about types is stored. As has been suggested above, the information maintained by the type repositories from ODP–based and knowledge–based traders are structurally the same. The knowledge–based trading context imposes no restriction concerning the kind of nodes stored in its repository. A conceptual graph as a knowledge representation technique is therefore more general than an ODP–based type definition.

Node type	*Name*	*Description*
Concepts	SERVICE-TYPE	Every service is an instance of this concept type and is being described by the following sub–concepts.
	INTERFACE-TYPE	The operational behavior is described by an interface type.
	PROPERTY-TYPE	Service property types are attached to the service type via this concept type.
	SIGNATURE	Each method of an interface type is described by an instance of a signature concept node.
Relations	CONSISTS-OF	Maps service types with interface/service property types and interface types with signatures respectively.
	IN-PARAMETER	Link to the input parameters of a method.
	OUT-PARAMETER	Link to the output parameters of a method.
	INOUT-PARAMETER	Link to the input/output parameters of a method.
	STATIC-ATTRIBUTE	Lists all static properties of a service property type.
	DYNAMIC-ATTRIBUTE	Lists all dynamic properties of a service property type.
	EVAL-CALLBACK	Denotes evaluation callback interface for dynamic attributes.
	ANNOTATED-BY	Informal annotation of a service type in form of an embedded conceptual graph.

Table 1: Concept and relation node types used for service templates.

The proposed general type manager solely works with conceptual graphs. The question arises on how a traditional IDL based type definition can be handled by the type manager. One possible solution is to use *conceptual graph templates* to map a service type with its associated operational interface to a canonical representation based on a conceptual graph. The kinds of concept and relation nodes necessary to define such templates are listed in table 1. The concept and relation nodes as listed there allow the construction of templates which are sufficient to specify service types and interface types (an extensive example will be given in the following section). Two points are worth mentioning: the modeling of signatures and the possibility for annotations.

There exists a straightforward mapping of ODP terminology to appropriate concept nodes except for the signatures which are an essential part for the definition of an interface type. Service–, interface– and property types are denoted by identifiers in the ODP context and can therefore instantaneously be mapped to corresponding concept nodes. A specification of an interface type consists of a set of signatures which defines its computational behavior. A signature is represented as an atomic entity within the ODP context. Despite its atomicity, it contains complex information like types of input/output parameters and therefore will be modeled by a complete conceptual graph with several nodes.

The possibility of machine readable semantic annotations of a service type on the other hand clearly goes beyond the capabilities of the ODP trading framework. A service type may be annotated by a conceptual graph representing its informal semantics. A programmer may exploit this additional information to solve the none trivial problem of meta–trading by querying the type manager via an informal description of a service. So far an ODP–based trader only has offered a programmer the possibility of browsing the complete type hierarchy, which will not be feasible for large quantities of constantly changing service types.

4.3 Example: IDL–Specification through CG–Templates

This section provides a concrete example on how an IDL–based service type is mapped to a corresponding set of conceptual graphs. We refer to section 2.1 assuming a *print service* with two operations called `SubmitJob` and `KillJob` as well as three service properties called `Name`, `DPI` and `QueueLength` has to be mapped to appropriate conceptual graphs.

```
[SERVICE-TYPE:{"PrintService"}] -
          -> (CONSISTS-OF) -> [INTERFACE-TYPE:{"Printer"}],
          -> (CONSISTS-OF) -> [PROPERTY-TYPE:{"PrinterQoS"}].

[INTERFACE-TYPE:{"PrintServiceInterface"}] -
          -> (CONSISTS-OF) -> [SIGNATURE:{"SubmitJob,KillJob"}].

[SIGNATURE:{"SubmitJob"}] -
          -> (IN-PARAMETER) -
                    -> [STRING:{"Data"}],
                    -> [INTEGER:{"NumberOfCopies"}].,
          -> (OUT-PARAMETER) -
                    -> [INTEGER:{"JobNumber"}]..

[SIGNATURE:{"KillJob"}] -
          -> (IN-PARAMETER) -> [INTEGER:{"JobNumber"}].

[PROPERTY-TYPE:{"PrinterQoS"}] -
          -> (STATIC-ATTRIBUTE) -
                 -> [STRING:{"Name"}],
                 -> [INTEGER:{"DPI"}].,
          -> (DYNAMIC-ATTRIBUTE)
                 -> [INTEGER:{"QueueLength"}]
                       -> (EVAL-CALLBACK)
                            -> [INTERFACE-TYPE:{"QueryPrinterQoS"}].
```

The print service in this example consists of an interface type called `Printer` and a service property type called `PrinterQoS`. The interface type as well as the service properties are "explained" by their own graphs. The interface type `PrintServiceInterface` basically consists of a set of signatures, each being described separately by a conceptual graph. A complete specification of a signature is done by listing its input and output parameters along with their respective parameter types and names. The service property types finally are described by their static and dynamic attributes. Furthermore the service type `PrintService` may be annotated by simply embedding the informal description presented at the end of section 2.2 via the relation node `ANNOTATED-BY` (this embedding is not shown in the example above).

5 Conclusion

Because of similarities between two different schemes for type definitions, a general type manager has been developed enabling the cooperation of ODP–based trading and knowledge–based trading for application users. The prototypical implementation will be built on the ODP–based trader of the MELODY project (Management Environment for Large Open Distributed sYstems, cf. [12]) and the AI–Trader (cf. [18]).

It can be concluded, that in contrast to quantitative trader cooperation being achieved by mere interaction between existing traders, the qualitative cooperation as described in this paper is based on a redesign of the trader's architecture. The study of ODP– and knowledge–based trading results in separating type managers from traders, sharing them among different kinds of traders and choosing a knowledge representation technique as an appropriate abstraction for various type notations within the type manager. This contributes to the current discussion of ODP about developing type managers apart from traders.

As has been mentioned already, the definition of service offers should also be examined to derive further kinds of qualitative trader cooperation. Of special interest in this context is a combination of traders being able to handle dynamic properties with ones that are not. For instance, prices for services in electronic markets won't be fixed and thus should be updated from time to time. Therefore qualitative cooperation among specialized traders will play a more and more important role.

Acknowledgements

We thank Kurt Geihs (Univ. of Frankfurt), Scott M. King (Thinkage LTD, Canada) and Andreas Vogel (Univ. of Queensland) for their comments and discussions on this article.

References

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

[2] M. Bearman and K. Raymond. Federating Traders: An ODP Adventure. In *International IFIP Workshop on Open Distributed Processing*, October 1991.

[3] T. Berners-Lee et al. The World–Wide Web. *Communications of the Association for Computing Machinery*, 37(8):76–82, August 1994.

[4] 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.

[5] C. Burger. Cooperation policies for traders. In *3rd International IFIP TC6 Conference on Open Distributed Processing (ICODP'95)*, Brisbane, Australia, 20–24 February 1995. Chapman and Hall.

[6] Component Integration Laboratories (CIL), Shaping Tomorrow's Software (White Paper), ftp://cil.org:/pub/cilabs/tech/opendoc/OD–overview.ps, 1994.

[7] M.R. Genesereth and S.P. Ketchpel. Software Agents. *Communications of the Association for Computing Machinery*, 37(7):48–53, July 1994.

[8] J. Indulska, M. Bearman, and K. Raymond. A Type Management System for an ODP Trader. In *Proceedings of the IFIP TC6/WG6.1 International Conference on Open Distributed Processing*, pages 141–152, Berlin, Germany, 13–16 September 1993. North–Holland.

[9] ITU.TS Recommendation X.903 — ISO/IEC 10746–3: Basic Reference Model of Open Distributed Processing Part 3: Prescriptive Model, February 1994.

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

[11] L.A. De Paula Lima jr. and E.R.M. Madeira. A Model for a Federative Trader. In *3rd International IFIP TC6 Conference on Open Distributed Processing (ICODP'95)*, Brisbane, Australia, 20–24 February 1995. Chapman and Hall.

[12] E. Kovacs and S. Wirag. Trading and Distributed Application Management: An Integrated Approach. In *Proceedings of the Fifth IFIP/IEEE International Workshop on Distributed Systems: Operations & Management*, Toulouse, 1994.

[13] L. Kutvonen. Comparison of the DRYAD Trading System to ODP Trading Function Draft. Technical Report C–1994–51, Department of Computer Science, Univeristy of Helsinki, October 1994.

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

[15] K. Müller-Jones, M. Merz, and W. Lamersdorf. The TRADEr: Integrating Trading Into DCE. In *3rd International IFIP TC6 Conference on Open Distributed Processing (ICODP'95)*, Brisbane, Australia, 20–24 February 1995. Chapman and Hall.

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

[17] C. Popien and B. Meyer. A Service Request Description Language. In *Proceedings of FORTE'94*, Bern, Switzerland, October 1994.

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

[19] 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.

[20] 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.

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

[22] G. Tidhar and J.S. Rosenschein. A Contract Net with Consultants: An Alternative Architecture and Experimental Results. In B. Neumann, editor, *10th European Conference on Artificial Intelligence*, pages 219–223. John Wiley & Sons Ltd., 1992.

[23] A. Vogel, M. Bearman, and A. Beitz. Enabling Interworking of Traders. In *3rd International IFIP TC6 Conference on Open Distributed Processing (ICODP'95)*, Brisbane, Australia, 20–24 February 1995. Chapman and Hall.