Cooperation Support for an Open Service Market[1]

M. Merz, W. Lamersdorf

Hamburg University - Department of Computer Science - Databases and Information Systems,
Vogt-Kölln-Str. 30 - D-22527 Hamburg, [merz|lamersd]@dbis1.informatik.uni-hamburg.de

Open communications technology allows the interconnecting of great multitudes of client applications with varieties of services in what can be considered a *Common Open Service Market (COSM)*. In a COSM, application development can profit from existing services used as building blocks for the development of individual integrated applications. Decisive for the success of this software development process is the identification and relating of conforming cooperation partners with each other through an *appropriate trading mechanism*. The additional effort for a client to utilize remote servers or to switch between different providers of a distinct service is called *transition effort*. This effort should be reduced by the underlying support system as much as possible.

This paper seeks to derive design principles for distributed application development from an economics open market analogy. A *Service Interface Description Language (SIDL)* is presented as the basis for minimizing transition costs for distributed applications. It is used for the trading process as well as the creation of graphical local *user interfaces* for arbitrary remote services at binding time. Finally, the paper outlines the current status of a distributed prototype system which implements cooperation support for a COSM.

Keyword Codes: C.2.4, H.4, H.5.0

Keywords: Distributed Systems, Communications Applications, Information Interfaces and Presentation, General

1. INTRODUCTION

Today's wide-spread availability and use of high-bandwidth communications technology makes various kinds of innovative *open computing environments* for cost-effective distributed application programming feasible and desirable. Open distributed computing environments provide a basis for increased specialization of offered services. Increased specialization, however, requires adequate communication and cooperation support in order to allow nodes to offer their respective services uniformly to one another.

Major problems of realizing open distributed computing environments with minimal effort for each single application component are caused by the high number and heterogeneity of nodes which may potentially participate. Solutions for such problems require improved and additional system components in the areas of *communication* and *cooperation* in open distributed systems.

---

[1] Appeared in: Proc. Int. IFIP Congress on ODP, North-Holland, 1993

Making specialized services on dedicated nodes available to utilizing applications requires considerable effort for combining a given set of basic functions into an integrated distributed application. In traditional environments, this integration effort is often due to a multitude of - in most cases independently developed - software modules which have to be configured and adapted explicitly for each actual client/server relationship. In this process, a great variety of different interface standards has to be taken into account [ODP92a]. This leads to considerable adaptation (or: *transition*) costs for each single component in such an integrated open systems environment as depicted in Figure 1a [Herb91]. As depicted in Figure 1b, he COSM approach tries to tie together semantics of the user interface, client and infrastructure components based on an adequate interface description of remote services.

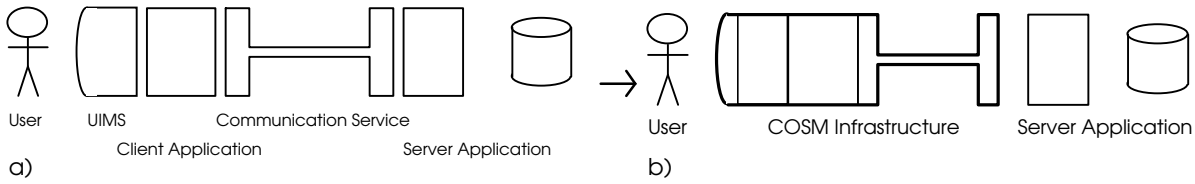

Fig. 1: Components of remote client/server interaction: a) traditional vs. b) COSM approach

The rest of the paper is organized as follows: The second section motivates system support for a COSM based on an analogy taken from economic science. The following section first describes basic concepts and components of a distributed application-oriented COSM infrastructure and then combines them into a corresponding system software architecture. Finally, section 4 first presents some details of a SIDL which aims to support service classification as an important prerequisite for service acquisition in a COSM. It also gives an overview of a respective distributed prototype system.

2. REDUCING TRANSITION COSTS IN CLIENT/SERVER RELATIONSHIPS

Given a great magnitude of available services which may enter and leave the service market autonomously, an infrastructure is required that facilitates transitions between client/server-relationships in an adequate way. Such transitions occur if, e.g., human users prefer to interact with alternative servers due to reduced utilization costs, or if a binding to a newly obtained service is to be established. Reconfiguring client/server relations may require conceptual and implementation effort at prohibitively high costs that prevent any change. Further, independence from a single (master) service should be given in order to enhance *participation autonomy* of clients and servers, which leads to a market-of-services model as introduced in [WoTs90]. Therefore, before going into technical details, we would like to compare the COSM model to an analogous balanced market model as defined in terms of economic science. The following three examples shall demonstrate some basic constituents of a market and the circumstances under which such an ideal market is balanced (which means that all ressources are optimally allocated). We then derive from that model some conditions which must be satisfied in order to realize an infrastructure for open systems in which client/server relationships are optimally supported.

According to economics theory, the model of *perfect competition* is built on the following assumptions [Hick43]: First, on both sides of a market there is an *atomistic granularity* of participants (consumers as well as suppliers). Second, *no* personal, spatial or other *preferences* exist in any consumer/supplier relationship. Third, one single market price for

distinct services is *transparent* to all participants, and, finally, *free market access* is always provided for all consumers as well as suppliers. In this example, the characteristics of perfect competition may serve as an analogon of COSM participation autonomy. If we interpret the respective conditions in an open computer cooperation context, we may recognize obstacles to realizing such an 'ideal' COSM for computer and software services: For example existing network services as well as clients are often difficult to access via a common communication medium. Therefore, only few services are offered and - due to high setup costs for service provision - consumers are reluctant to switch between alternative providers within a specific service category, even if they exist.

A second concept from economic market theory addresses the aspect of *added-value* generation: In an open service market, service suppliers may also play the client role to further servers acting as a link in a value-adding chain. Translated into an open computer communication context, this analogy shows that an open service environment should specifically support easy market access of producers of 'added-values' or improved service for the benefit of other market participants, if it appears to be profitable. In a similar way, the effort to interact with an alternative server should always be as small as possible. Only then can the *make-or-buy-decision* to provide a service 'internally' or to obtain it from a supplier in the market be freed from any bias against cooperation [WaWe84]. In existing systems, such bias is frequently caused by the prohibitively high transition costs in case of purchasing from external sources.

In summary, all constitutional elements that lead to a well functioning, balanced market are closely related to the overall goal of *reducing transition costs*. In an open market, such transitions may occur frequently: Entering a market, e.g., is a transition for either a client or a server, changing from self-production to purchasing from an external supplier is a transition, and changing the supplier is a transition as well. In all cases, one of the most important prerequisite for reduced transition costs is some form of uniform agreement on how to cooperate. This, however, means to agree on dedicated interaction standards which all cooperation partners can jointly and consistently rely upon. Accordingly, a first goal of a COSM realization is to identify aspects of service interfaces which need to be described and standardized. Then, a suitable interface specification technique and dedicated system software have to be designed and implemented. All such components together will finally make up a common COSM cooperation infrastructure that supports distributed service access and management in an open market of services.

3. COSM DESIGN

3.1. Design Principles

The architecture of the COSM system support platform as presented below is based on two simple design principles: First, at any level of abstraction of the overall system, a generic client/server model is uniformly applied to separate cooperating components horizontally into always a 'requesting' and a 'responding' entity, respectively. Secondly, a strict separation is applied between the 'application' layer, which uses a service, and the supporting service layer, which provides the corresponding service. Figure 2 presents this generic client/server model graphically.

In the overall design of the COSM support system, this model is applied recursively at three distinct levels (see Figure 2):

1. The *operating system level* may comprise extended operating system components like specific file system or memory management services, which can be separated from an OS (micro-) kernel as dedicated systems components.
2. The *COSM support service level* (*support level)* provides specific modules like interface repository or security services, based on environments like OSF/DCE (Distributed Computing Environment). Further support level components provide distribution transparency to the application level (similar to transparency mechanisms in [ANSA91]).
3. The *COSM application level* supports dedicated distributed application components such as application-specific or generic clients and dedicated application servers. Application layer components also interact with the *human user* to efficiently support the interaction with remote service applications.

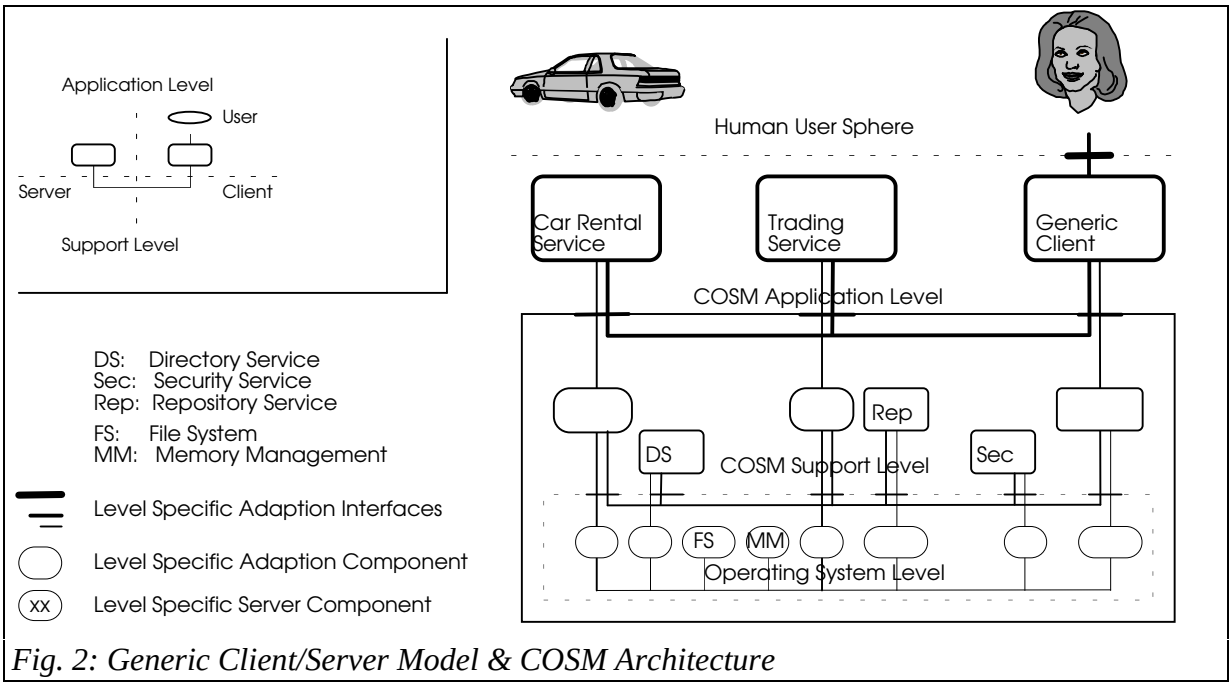

Fig. 2: Generic Client/Server Model & COSM Architecture

As mentioned before, an important design goal for the COSM architecture is to reduce transition efforts, i.e., application programming efforts for connecting application level components to their supporting common cooperation services. This leads to a strong and well supported application level *modularization* (additional functionality should be *added on* at the application layer) rather than a monolithic architecture where all services are provided internally at the support level (*build-in* approach).

3.2. Requirements for a Common Open Service Market

Service Interface Description

COSM participants act in various roles, either as clients or as servers. In general, application components are developed independently from one another and at different points of time and space. A client is thus unspecific to a server, so the assumption of conforming client/server

interfaces, as made implicitly by traditional stub generators, does not hold for COSM service invocations. Instead, COSM servers are required to identify themselves and have to supply their own *service descriptions* explicitly to the underlying support system. This allows COSM clients to browse and import such service specifications. This, however, means that COSM client and server applications require interpreting stubs in order to avoid recompilation of stub modules whenever a new client/server relationship is established, comparable with the OMG CORBA 'Dynamic Invocation Interface' [OMG91]. The trading service itself acts as a COSM application and, therefore, exports its own service description directly to potential client applications.

Explicit service descriptions are to be transferred between different application level components. They may serve not only for marshalling purposes but also for assuring conformity of parameter values as actually transferred with their type specifications in the respective service descriptions. Since human users may also be involved in service acquisition processes, natural language annotations may also be added to the service description. In summary, any basic *Service Interface Description Language (SIDL)* [MeLa93] for a COSM requires at least the following three components:

- parameter type declarations for interpreting stub control and conformance assurance,
- (remote) procedure descriptions, and
- natural language annotations to support the human user at the service selection process.

Some aspects of such a service description language are presented in the following sections.

Generic Clients

Generic client components in COSM are defined without immediate relationship to a specific server, i.e. without knowledge about a the application functionality of a server. This information has to be imported explicitly at binding time. If a human user of a client component is involved in service selection, binding, and interaction, the client functionality can be reduced to a *generic* interface component which supports general service description browsing, acquisition, and remote service invocation tasks. In COSM, the client components are called *generic clients* and may utilize elements of explicit service descriptions also to generate a server-specific graphical user interface [JCKa92, NaKa92]. Therefore, an important property of generic clients in COSM is a well-defined relationship of linguistic (SIDL) service description elements to corresponding user interface management system (UIMS) components at the client site. This allows client application development to benefit automatically from SIDL service specifications, as provided by a remote server.*

Components of SIDL service descriptions like types, operations and textual annotations result in respective UIMS components. Therefore, type-specific value forms are generated that allow to present or edit data values. Other elements (like buttons or list items), that can be activated by mouse events, are related to respective remote operations defined in a SIDL service interface description. Furthermore, for each UIMS element, the user can be supported through additional information given by respective natural language text annotations.

---

* For example, users are not required to learn the different dialog control and interface styles used by various services. Further, the semantics of user interaction with servers can remain unchanged - even if visual presentation elements of alternate user interface systems may vary. This enables an implementation of generic clients independent of hardware and UIMS as presented in section 4.

In result, service integration, as proposed for COSM based on uniform SIDL specifications, substantially reduces the multitude of heterogeneous interfaces between human users and actual applications. After integration within a COSM environment, the only remaining interfaces to be related (*bound*) together are the human user interface on the one side, and the server application programming interface, on the other (Figure 1b).

In a COSM, the notion of *binding* has a broader meaning than just the assignment of a name to a remote service instance. A client/server binding may also comprise an agreement on the "terms of trade" of the service, as specified in the explicit service description.

Service Integration

A COSM system support architecture should facilitate the integration of services with minimal possible effort, that is the realization of an added value with minimal overhead. A cooperation support infrastructure for such *service integration* motivates the development of specialized 'mediation' services. In general, service mediation can be done either by *service referral* or by *service chaining*: In the first case, a mediator first acts as a switch to remote services and then allows users to bind directly to these services. In the latter, the mediator service acts itself as a client for service integration and the binding is realized between the mediator and correspondingly connected remote servers.

In order to support service mediation by referral, a COSM requires a naming schema which uniquely identifies services world-wide. A service name in our COSM prototype environment is currently composed of the server's node address and a local name. This information remains transparent to human users of generic clients since service names are values of a special base type SERVICE in the service interface description language. Records, sets, lists or variants may contain elements of type SERVICE. Remote operations, e.g., of the form

Service GetProxy(**Service**); // reply an alternative service

accept service names as parameter and deliver results of type SERVICE.

Since a generic client provides a conforming UIMS component for each service description element, values of the type SERVICE also have their individual representation: either as a button element or as a list item if specified like SEQUENCEOF SERVICE in the SIDL service specification. Thus, a client/server binding can be invoked easily by a user-interface event. Applied to aspects of service mediation, the COSM communication infrastructure should support a corresponding nesting of server binding representations at the (graphical) user interface level.

Trading

Trading is the process of matching service requests with service offers based on attributes which characterize quality requirements of a client and, respectively, quality assurances of a server [ODP92b]. A classification of the service domain requested can be characterized by specific service property lists. In the COSM environment, trading can be considered as a mapping from a list of required service properties and a list of supported properties to a list of service references (values of type SERVICE). If this mapping is carried out by a distinct application component, this component is called a *trader* [TsWW92]. The trader service may be provided by a federation of server instances. In the context of COSM applications, the process of trading can be viewed as a mediating (value-adding) service either by referral or by chaining. In the first case, different traders may supply varying individual interfaces. By

using the generic client, a human user is enabled to import the service description of each trader at binding time and to adapt to specific interfaces. Informally, such a trader service interface could have the following structure:

```
Type ServiceList SequenceOf { Service };
Type Spec Record {
                Int maxResults;                          // limit returned list of services
                SequenceOf {
                        String property;                 // demanded properties
                ...
                        } properties;
                };                                       // import description
ServiceList     ListAllServices( ),
                    Comment "List of all services currently
                                registered at trader";
ServiceList     SearchMatchingServices( Spec );
Service         SerachBestService( Spec );
```

4. SIDL: A SERVICE INTERFACE DESCRIPTION LANGUAGE

In this section, we first present an overview of the base version of SIDL as defined in [MeLa93] and [Merz92]. Then, we introduce some SIDL extensions and a first version of a corresponding COSM prototype implementation.

For the description of COSM service applications, a minimal description language is required to supply a formal description of parameter types and procedures at the server interface. Any further formal specification aspect is considered an extension to these base elements. In particular, service descriptions can be extended to

- support a *finite automaton* description to model the service behaviour.
- involve additional primitives, e.g., to support a *transaction*-based execution of remote procedures,
- embed *further* formal description techniques.

As far as informal descriptions are concerned, natural language annotations to the syntactical elements of the interface descriptions are allowed. This leads to different granularity levels of text annotations, spanning from a description of the overall service functionality to single parameter value annotations.

4.1. Basic Elements of SIDL

This section focuses on a base version of SIDL that contains service description aspects as motivated above. A car rental service is used as an application example for human user activities. Several questions may arise in such an application scenario, for example: How are data entry forms generated and presented by a remote generic client? Which functionality can be specified formally and what further information is to be given to the user by means of textual annotations? How can transferred data values be type-checked against their service description and be validated?

Figure 3 shows the main aspects to be described at the server interface: RPC procedures which represent transitions between server states (*Init* and *Selected*) as well as parameter and

result types. Accordingly, the SIDL code derived from this specification contains type, procedure, and state description sections. Furthermore, the EXPORT section contains a list of service properties, used by the trader for service selection tasks.

```
TYPE  SelectCarT RECORD{...};                    // see below
TYPE  SCResultT ...;                             // Booking confirmation
TYPE  BookCarT ...;
...
SERVICE     SelectCar { REQUEST  SelectCarT; RESULT SCReturnT };
SERVICE     BookCar { REQUEST BookCarT; RESULT ResultT };
SERVICE     Abort { REQUEST BookCarT; RESULT ResultT };

STATES {
        INIT:   SelectCar ->    S2;     // A car
has to be selected at least once, then the selection
        S2:     SelectCar ->    S2;     // can be
confirmed or cancelled.
        S2:     BookCar ->      INIT;
        S2:     Abort ->        INIT;
        };
EXPORT {
        Category: "CarRental";
        ChargeMethod: "PerInvocation";
        ChargeAmount: "5";
        ChargeCurrency: "USD";
        ... };
```

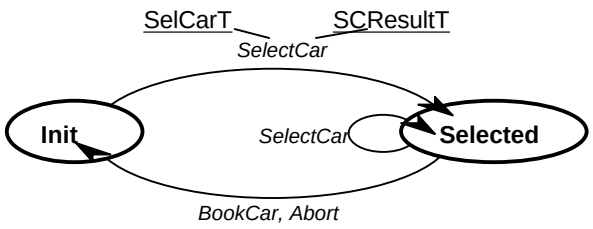

Fig. 3: SIDL service description and the corresponding service model

4.1.1. SIDL Data Type Definitions

Any remote procedure call may require structured parameter or return values to be transmitted between clients and servers. Accordingly, the SIDL type system provides the following types:

Basic types:	INTEGER, FLOAT, CHAR, STRING, TEXT
Structured types:	RECORD{...}, CHOICE{...}, ARRAY {...}, SEQUENCE [OF]{...}
Dynamic type:	ANY

The TEXT type refers to a text file on the local workstation which can be embedded as an RPC parameter. CHOICE specifies the variant part of a RECORD discriminated by a type tag. A SEQUENCE OF type denotes a repetition of identical element types. The special type ANY describes dynamic types, i.e. values of type ANY are transmitted dynamically at runtime but not checked for conformance since their actual type cannot be anticipated at binding establishment. According to the SIDL syntax definition, a type declaration can be extended optionally by a list of attribute/value pairs. These may concern subrange restrictions of a type or hints for the user interface representation. The following parameter type defines a record type that contains nested basic and structured types:

```
TYPE SelectCarT RECORD {                                    --->
        INTEGER,    LABEL "Mileage",                            STRING,   LABEL "Customer Name";
                    RANGE TINY 50 5000;                         STRING,   LABEL "First Name";
        STRING,     LABEL "Booking Date";                       STRING,   LABEL "Street";
        INTEGER,    LABEL "# Days",                             STRING,   LABEL "City";
                    RANGE TINY 1 50;                            CHOICE {
        INTEGER, LABEL "Model",                                         INTEGER  LABEL "Visa #";
                    COMM "For a broader range                           INTEGER  LABEL
                    of models consult our                                       "MasterCard #";
                    service at main branch",                            INTEGER  LABEL "Amex";
                    RANGE CHOICE 3                                      InvoiceT   LABEL "Invoice";
                    "BMW 323" "VW Golf"                         } LABEL "Payment";
                    "Fiat UNO";                         } Label "Select Car Form";
                                    --->
```

Some integers are constrained types restricted to a subrange. Thus, range constraints can be considered by a generic user interface in order to reject input of data values that do not satisfy the type constraints. For the automatic generation of user interfaces, however, text annotations like `LABEL` and `COMM` are treated as hints since they may not necessarily be considered by the generic client. An example for a type-specific value editor, created by the generic client UIMS is shown in Figure 4.

4.1.2. User Interface Description

The user interface specification of a remote service in open systems provides some additional hints for a generic client for automatic (graphical or window) presentation of the typed data values. Such hints have to be specified abstractly in order to enable a wide range of window managers to support a generic user interface implementation. Type-specific editors of such interfaces may vary in their visual appearance, e.g. the type '`TINY` integer' may be represented graphically as a slider or as an entry field.

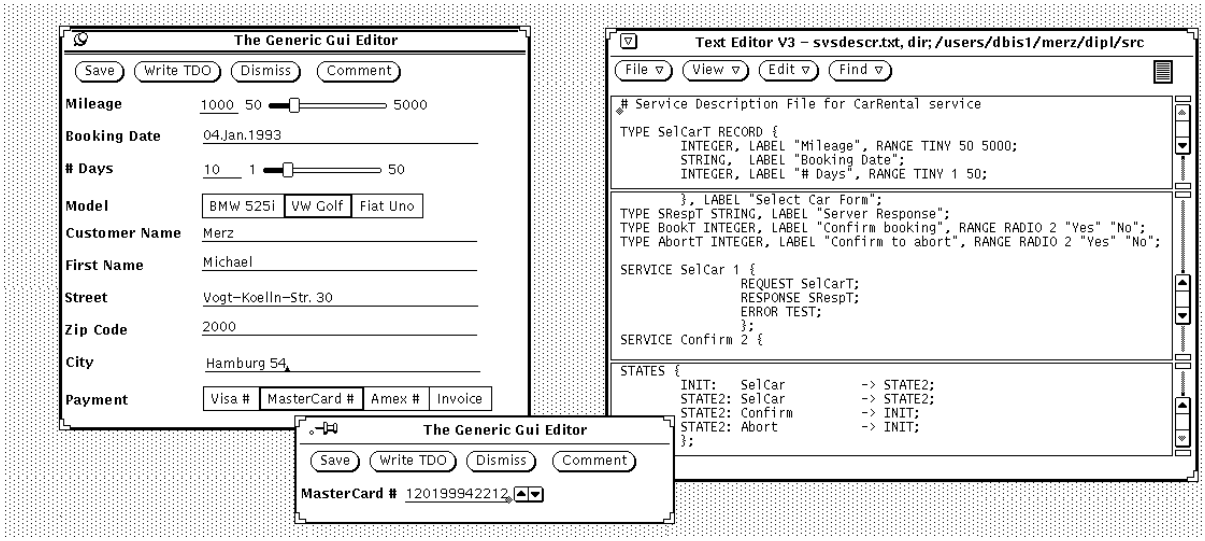

Figure 4: Service description and a generic client user interface

Figure 4 gives an example on how the user interface specification could be utilized for automatically generating a query form from the SIDL service interface description. For example, the right window of Figure 4 shows a service description file where the previously

introduced type `SelCarT` is defined and used as a parameter type for the `SelCar` service procedure. On the left side, the generic client application is shown after binding to the car rental service which supplies this procedure. The form windows in the left part of Figure 4 represent the parameter value for the procedure invocation. The actual parameter transfer is initiated by pressing the "Write TDO" button: A Tagged Data Object (TDO) is generated from the current data value and sent to the server.

4.2. Checking Type Conformity

Following the overview of the base version of SIDL, some examples of *dynamic* extensions shall be given. In order to realize dynamic extensibility, service descriptions can themselves be considered as data values which are transferable between network nodes. However, if such service descriptions are defined by means of monomorphic programming languages like Pascal or C, *all* participating applications have to interchange values of *exactly the same* data type among each other. If the structure of such a service description value is to be extended by additional descriptional elements, either additional service descriptions have to be explicitly defined or an extended version has to be 'standardized' among all applications. In summary, in monomorphic type systems it is not possible to accept data values of different types, even if they belong to a common *supertype*. Polymorphic programming languages, however, support exactly this aspect. Therefore, the extensibility requirement for service descriptions leads to a consideration of languages extensions like, e.g., *Laura* [Tolk92] or polymorphic programming environments like *Quest* [Card89] or *Tycoon* [ScMa92] which implicitly support implicit subtyping rules. Therefore, SIDL elements, which are required for type conformity checking, could be transformed into *Tycoon Language* (TL) code in order to enable a dedicated trader component to carry out the conformity checking process automatically.

4.3. Prototype Implementation Overview

Finally, we present an overview of a first prototype implementation of a COSM infrastructure which supports SIDL service descriptions and automatic user interface generation. The system model for our COSM implementation prototype involves four kinds of components: a generic client, a client agent (CAG), a server agent (SAG), and a server. The purpose of both kinds of agents is to protect their applications against potential type mismatches between actual parameters transferred and the data type specified in the service description.

The process of *client/server-binding* implies the selection of a server as well as the import of the server's service description. Before binding, the service description is stored at the server's site after being converted from an external representation. At binding time, the service description is transferred to both agents where it is stored as long as client and server are bound. At the actual *service invocation,* RPC parameters are transferred via both agents in order to perform the necessary conformance checks. If there is a mismatch between specified types and the parameter types transferred, this is detected by the local agent of each site and an error code is returned.

Instead of involving a specific client application, parameter values are mapped directly on the user interface level. Therefore, the generic user interface supports user functions to select an appropriate server, to examine the service procedures offered by this server, and, finally, to invoke selected procedures. Thus, the process of binding between client and server is reflected at the user level by this service selection process. The actual service invocation

requires the user to supply the RPC with parameter values. Therefore, the generic user interface generates a typed form for parameter entry (Figure 4). The required type description is retrieved from the local CAG. Return values are presented in the same way.

The prototype was developed on a heterogeneous workstation cluster, consisting of Sun SPARC stations as well as IBM RS/6000 AIX workstations. Currently, a standardized RPC interface serves as a common communication basis. Following the model described above, the prototype supports the integration of user interface and service description aspects. Developing a new server application just requires to code service procedures based on the server communication interface and to describe these procedures by means of a SIDL service description: the formal parts as type, procedure, state and export description and, optionally, the informal part of the user interface description as natural language annotations.

5. CONCLUDING REMARKS

This papers aims at improved system support for flexible client/server integration in modern distributed and heterogeneous open systems. Specifically, it addresses problems of matching distributed application program *client requests* with generic remote *server interface functions* as provided at dedicated server nodes anywhere in an open network environment. The goal here is not just to support a *specific* client/server cooperation but rather to design a *generic* architecture for flexible service management in open systems. This architecture should help reduce transition costs and facilitate decisions as to whether to 'make' or to 'buy' different application components.

According to software abstraction principles, the COSM architecture separates strictly between application components on the one side and supporting services, which are hidden to the application, on the other. Below this borderline, the COSM cooperation support system infrastructure is implemented. At the application level, service descriptions are to be used easily in order to utilize generic components of the platform for specific applications needs. As shown in Figure 1, the COSM infrastructure helps considerably to reduce the number of different and separated interfaces between the human user and the server.

Currently, work on the COSM prototype implementation concentrates on using and extending standardized X.500 Directory Services [ISO88] into a globally accessible open *repository* service for storing, managing, and making available COSM service descriptions which have been specified with SIDL. In addition to managing such *static* service descriptions, later versions of the prototype may take into account additional information on component services (e.g., *dynamic* status information) available for client use. This can be based on standardized open 'Systems Management' [ISO90] and function, in the way as proposed in [PoTT91]. Here, various problems of managing both static and dynamic systems management information in high quantities, at different location, and with high efficiency requirements as necessary in large scale open systems are still unresolved.

Acknowledgement

The authors thank Florian Matthes for valuable remarks on a former version of this paper.

References

[ANSA91] ANSA: A System Designer´s Introduction to the Architecture, APM Ltd, 1991

[Card89] L. Cardelli: Typeful Programming, DEC SRC Research Report #45, Palo Alto, 1989

[Herb91] A. Herbert: The ANSA Project and Standards, in: S. Mullender (Editor): Distributed Systems, ACM Press, New York, 1991, pp.391-399

[Hick43] J.R. Hicks: Value and Capital, an Inquiry into some Fundamental Principles of Economic Theorie, 2nd Ed., Oxford, 1943

[ISO88] ISO/ IEC JTC 1 SC 21, International Standard IS 9594: "The Directory", 1988

[ISO90] ISO/ IEC JTC 1 SC 21, International Standard IS 10040: "OSI - Systems Management Overview", 1991

[JCKa92] Jagannathan, J. Cleetus, R. Kannan: Application Message Interface, in: IEEE Phoenix Conference on Computer and Communications, 1992, pp. 493-500

[Merz92] M. Merz: Generic Support for Distributed Client/Server-Cooperation in Open Systems (in German), Masters Thesis, Dept. of Computer Science, Hamburg University, 1992

[MeLa93] M. Merz, W. Lamersdorf: Generic Interfaces to Remote Applications in Open Systems, in: Proc. Intern. IFIP Workshop on Interfaces in Industrial Production and Engineering Systems, North-Holland, 1993, pp 267-281

[NaKa92] R. V. Narender, R. Kannan: Dynamic RPC for Extensibility, in: IEEE Phoenix Conference on Computer and Communications, IEEE Computer Soc. Press, 1992, pp. 93-100

[ODP92a] ISO/IEC JTC1 SC21 WG7: Basic Reference Model of Open Distributed Processing, Working Document N7053, 1992

[ODP92b] ISO/IEC JTC1 SC21 WG7: Trader, Working Document N7047, 1992

[OMG91] The Common Object Request Broker: Architecture and Specification, OMG Document No. 91.12.1, 1991

[PoTT91] R. Popescu-Zeletin, V. Tschammer, M. Tsichholz: 'Y' Distributed Application Platform, IEEE Computer Communication, vol. 14, no. 6, 1991, pp 366-374

[ScMa93] J.W: Schmidt, F. Matthes: Lean Languages and Models: Towards an Interoperable Kernel for Persistent Object Systems, Procs. Int. IEEE/RIDE Workshop on Interoperability, IEEE Computer Soc. Press, Los Alamitos 1993

[Tolk92] R. Tolksdorf: Laura: A Coordination Language for Open Distributed Systems, Report 1992/35, TU Berlin, 1992

[TsWW92] V. Tschammer, A. Wolisz, M. Walch: The Performance of Multiple Traders Operating in the Same Domain, in: IEEE Workshop on Future Trends of Distributed Computing Systems, IEEE Computer Soc. Press, Los Alamitos 1992, pp. 122-128

[WaWe84] G. Walker, D. Weber: A Transaction Cost Approach to Make-or-Buy Decision, Administrative Science Quarterly, 29, 1984, pp 373-91

[WoTs90] A. Wolisz, V. Tschammer: Service Provider Selection in an Open Services Environment, in: 2nd IEEE Workshop on Future Trends of Distributed Computing Systems, Los Alamitos, IEEE Computer Soc. Press, Los Alamitos 1990, pp. 229-235