Generic Interfaces to Remote Applications in Open Systems[1]

M. Merz and W. Lamersdorf

Department of Computer Science, University of Hamburg, Vogt-Kölln-Straße 30,
D-2000 Hamburg 54, Germany; eMail: [merz|lamersd]@dbis1.informatik.uni-hamburg.de

Abstract

Future industrial production and engineering environments will profit substantially from emerging *open distributed computer communication network* environments. They will, in principle, be able to benefit from a high potential of *services* available in such environments to support individual *client* applications. In practice, however, free and flexible client/ server cooperations are frequently hindered by the great and confusing variety of *interfaces* involved in accessing various and heterogeneous network services.

In order to support open client/ server cooperations in distributed systems, this contribution proposes a *unifying description mechanism* for remote services in computer networks. It describes an *application oriented generic communication service*, which facilitates client/ server cooperation in open systems. Most important basis for such a service is a *uniform service specification* mechanism for open server interfaces. Correspondingly, the paper first specifies a specific *service interface description language* (SIDL). It then shows how such a service interface description could also be used for automatic creation of server-specific local *human user interfaces*. In combination, a generic *network interface description language (NIDL)* specification, as proposed here, supports client applications in open systems by providing a *common* mechanism to access and utilize *any* service available *anywhere* in the network.

Keyword Codes: C.2.4, H.4, H.5.0
Keywords: Distributed Systems, Communications Applications, Information Interfaces and Presentation, General

1. INTRODUCTION

Since the integration of formerly - logically and geographically - separated and heterogeneous software systems has become an issue for research and industrial implementation, new concepts to support this kind of 'interoperability' have gradually emerged: programming languages have been extended to cover communication requirements between separated modules [1]; advanced database systems started to support distribution schemata for the allocation of

[1]Appeared in: Proc. IFIP Working Conference on Interfaces in Industrial Systems for Production and Engineering, North-Holland, 1993, pp 267-281

distributed objects [2]; and specific application oriented communication standards have been developed for, e.g., accessing remote database services in open network environments [3]. As a result, the system designer in the context of such a 'distributed application' scenario is now confronted with a large number of different kinds of interfaces to various services offered anywhere in the network. The variety of such interfaces can be described between the human user, on the one hand, and a (potential) multitude of heterogeneous and different remote applications in the open systems environment, on the other hand. In such a scenario, the potential cooperation of different users and system component is, in practice however, hindered by the multitude of different servers and interfaces as generally offered in the heterogeneous open system network environment.

Consider, as a simple example, an access facility to remote database services in open systems, which involves the following (different) interfaces (see Figure 1): the user interface between the human user and a (local) front-end application software, the interface between this software and appropriate network communication services, its counterpart at the remote site and the interface between the remote application and its local resources. For each of these interfaces there are several de-jure and de-facto standards available, but in most cases these standards have been specified for the requirements of some specialized application contexts (e.g. electronic mail or document interchange standards [4]) or they concentrate on system-oriented services, like communication standards, adhering to the ISO reference model for Open System Interconnection (OSI). Only recently, the interdependence of several of such interfaces between the human user, on one hand, and remote applications in open systems, on the other hand, has become an issue for international standardization in the context of the fundamental work on a general framework for 'Open Distributed Processing' (ODP) of ISO and CCITT [5].

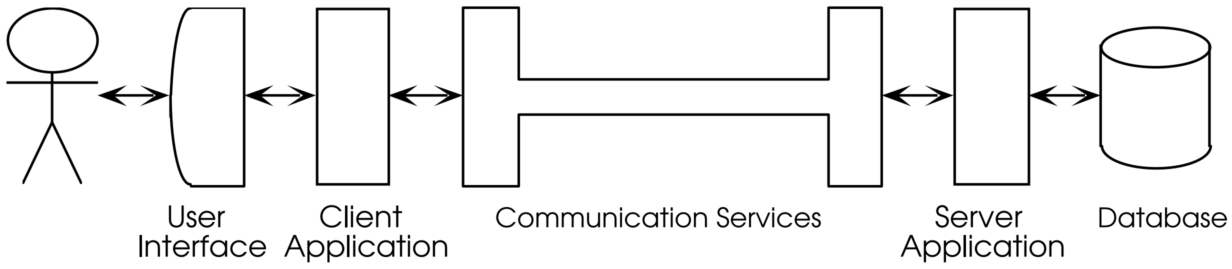


Figure 1: Interfaces between the human user and a remote server

The situation becomes even more complex if we consider future developments of high-speed communication systems: in future distributed open systems, interconnected by high-speed networks, a vast number of services will be easily accessible at a high transfer rate and a high level of distribution transparency [6]. In such environments, remote applications will play the role of dedicated servers, performing specialized task, rather than monolithic software systems like today's host applications. Here, remote servers are accessible through specific server interfaces in a similar way to both local and external communication partners [7].

Especially in a scenario as sketched above, it would create great confusion for human users if the human interface to such a fine-grained "market of services" was reflected by a similarly fine-grained structure of different (!) server interfaces. Therefore, access to various services in open systems could be greatly supported and improved if all accessible services could be described in terms of a unified, standardized, and commonly known formal notation. Such a unifying formalism for network (service) interfaces of any kind is called a 'Network Interface Definition Language' (NIDL) and serves to provide a corresponding service description for each server that is directly or indirectly accessible for any (remote) client node [8].

This paper focuses on the joint and unified design and description of both communication and user interfaces in a heterogeneous open system scenario. The goal aimed at here is a generic system software component, which dynamically generates the required user interfaces from any specific 'Service Interface Description' (SID) expressed in terms of the standardized NIDL (see Figure 2). In such a scenario, the SID could serve both for determining the network interface of a remote server and for its (e.g. graphical) appearance at the presentation level of a local human user interface [9].

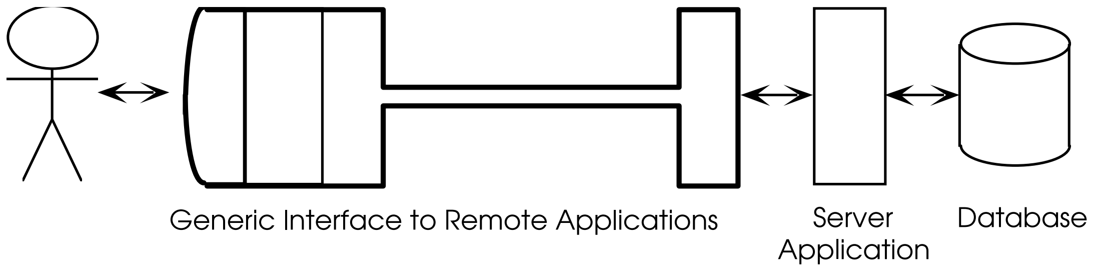


Figure 2: Integration of interfaces and components in the client/server-model

The paper is organized as follows: section 2 gives a survey about potential problems when applying current client/server interfacing techniques in the context of an open systems background. Based on these facts, requirements for a successfull client/server interaction are elaborated in Section 3. Afterwards, a client/server interaction model is presented and, based on this model, a prototype implementation, including an introduction to the service interface description language SIDL (Section 4). Some conclusions are finally presented in Section 5.

2. INTERFACING PROBLEMS IN OPEN SYSTEMS

As a motivating example, a car reservation service is considered that a car rental company offeres to its clients based on a distributed application service. Several questions arise within this context: how are data entry forms to be presented at a remote client's site? How is the service access to be supported for a client? What does the client's software need to know about the service considered? And how can the validity of the data transferred be assured? Finally, how does the client software adjust to a possible release change of the server interface?

As a first step, the following sections give a short classification of these interfacing problems at client/server-systems in open distributed environments. This classification focusses on problems arising specifically in an open systems context.

2.1 Heterogeneity Problems

There are several levels of heterogeneity in open distributed processing that emerge from the integration of different multi-vendor hardware and software components. At a distinct level, heterogeneity is even demanded, as the specialization of software systems requires "non-standard" implementations. In general, however, heterogeneity hinders the desired cooperative interaction of distinct distributed applications.

At the lowest level of heterogeneity, different hardware implementations and, thus, varying physical representations of data values at each respective local system have to be integrated by corresponding mapping mechanisms between heterogeneous system components. In the context of the ISO OSI reference model for open system communication, this transformation task is performed by the presentation services at level 6 of the ISO OSI reference model [10].

At a higher level of abstraction, heterogeneity problems address differences in services and resource management functions: communicating applications first have to agree about a jointly supported protocol as well as about certain communication quality attributes. Local resources, like file systems, databases or operation system services, can not be accessed from remote systems if different and inconsistent interfaces are supplied. At last, applications semantics may vary themselves, even though their syntax and semantics may seem virtually identical at the interface level.

In general, the most promising approach to tackle heterogeneity problems is to standardize interfaces or applications as a whole. A standardization procedure, however, is a time consuming process and hinders an immediate "publication" of new services. Therefore, generic application oriented communication standards may be well suited for defining the basic communication protocols necessary for each class of similar applications (like, e.g., Remote Database Access, RDA)[11].

In our case of integrating communication and user interface description techniques, the standardization may also cover the syntax of a service description that is then transmitted as an individual 'protocol data unit' (i.e. standardized message) over the network. Thus, each individual server is able to export such a description of its services provided to any of its potential clients in a unified way.

2.2 Service Access Problems

The next important question to be addressed in such a scenario is, for example, how to initiate an access to a remote application before actually interact with it?

In the context of a Local Area Network (LAN), e.g., local 'context servers' may be accessible via a dedicated 'name server'. Using such a server, a client may send a service ID to the name server and then receive all (or some, the 'best possible', etc.) information necessary to perform a remote service invocation. Here, the name server's task is to check whether a service is registered under the given ID by, e.g., looking up a local table. In this simple case, only required and registered service IDs have to be matched, since developers of client and server applications took specific care for these IDs to conform to each other. In more complex scenarios the match between client requirements and server potentials could be provided by a specific distributed system service (a so called 'trader' or 'broker' component) based, for example, upon a more extensive formal specification of both requestor and server functionality [12].

In open service environments at a global scale, an ID-based service selection is possible but not satisfactory for client service users: IDs have to be centrally reserved for each service offered, and IDs have to be known in advance by clients for all accessible servers that provide potentially useful functions for clients in addition to the specific server interfaces. This situation could be greatly improved if services could be described and identified by potential clients *semantically*, i.e. not only by a more or less characteristic service name or ID, but in terms of a certain specification of their semantic *functionality* and respective properties. If such a service description is available, the service selection mechanism (which is a part of the 'trader' function - see below) for the client would not be restricted anymore to using IDs as the only attribute for specifying and identifying services but could be based on a limited match of the attribute *semantics* of both the client request and the respective server function [13]. However, even this simple task of attribute based matching of client requests to registered services requires additional support by the new distributed service component which supports matching client request to the 'right' server functions available anywhere in the network. In advanced distributed open system scenarios, such a component is called a 'trader' [14].

2.3 Conformance Problems

In heterogeneous distributed systems, client calls are usually transmitted to remote server functions via an abstract communication mechanism: the *Remote Procedure Call* (RPC) component [15]. The RPC communication mechanism aims at hiding nearly all distribution and communication problems to RPC users - even in certain (e.g. message transmission) error situations. For a local system component, which uses RPC - for example a client calling a remote server - all server functions are locally represented by a so called 'stub' function. This stub component, which is a part of any distributed RPC based communication function, then does all necessary transformation from local to communication contexts (resp. vice versa) and cares for message transmission and reception.

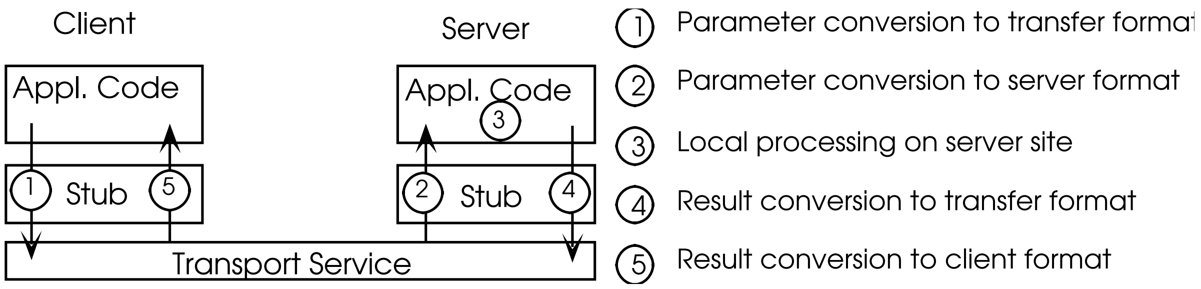

Client
Server
Appl. Code
Appl. Code
3
1
Stub
5
2
Stub
4
Transport Service
1 Parameter conversion to transfer forma
2 Parameter conversion to server format
3 Local processing on server site
4 Result conversion to transfer format
5 Result conversion to client format

Figure 3: Remote procedure calling phases

In advanced distributed systems based on RPC, the necessary local stub can be *automatically generated*, based on a respective local and remote *interface specification*. In addition, when using such 'stub generators' to automatically create client and server interface code, client and server parameter types will implicitly match since they are derived from the same interface specification.

In open systems environments, however, the situation is somewhat more difficult: developers of client and servers applications are, in general, remote and unknown to each other; clients are not supposed to supply specific compilers for client's interface stubs. Further, they can not rely on an interface description as imported from a remote server site since data types, as defined at the server node, may not conform to the server's actual interface data types - or the exporting component may be faulty or even malicious.

In order to automatically support the development of interfaces between both client and server components in open systems, at least two requirements have to be fulfilled: first, the protocol data unit (i.e. the 'message') that carries the request has to contain type information about the parameter values it contains and, second, a service description has to be imported by the client from the server in order to (type) check the conformance of the actual request parameters and types with those of the server as specified in its service description. To avoid using uncompiled source code stubs, the service description notation should be interpreted directly by a generic client stub.

2.4 Standardization Problems

As a consequence of what was addressed in the preceding paragraphs, an extended, formalized, and unified - i.e. *standardized* - *service interface description technique* is a necessary prerequisite for any effort - human or system supported - to match client requests and server offerings in an open systems scenario. Such a standard service description mechanism could then become a basis for a more elaborate trader service component that supports client and server matches in open system environments, where client and server functions are not only provided on distinct nodes of the network, but also independently developed and, in many cases, unrelated and unknown to each other.

The issue in this case is to commit to a reasonable scope of standardization aspects: are only syntactical aspects to be defined as it is the case with ASN.1 [10] or are service primitives to be covered as well? At what extent is the interdependence between user interface elements, data types and service primitives a standardization matter: a standard could prescribe types and their appearance at the user interface level or it could be more appropriate to design a generic service with the option for application specific extensions.

In the following sections we concentrate on an *executable typed* protocol description.

3. ELEMENTS OF AN INTERFACE DESCRIPTION LANGUAGE FOR OPEN SYSTEM SERVICES

As stated above, RPC provides an appropriate cooperation paradigm and communication mechanism for client/ server applications in distributed environments. For open networking system at a global scale, however, an additional important question is to be raised: how can an RPC interface description (as known, e.g., from Sun RPC) be conceptually extended to satisfy at least the following problems:

- First, an abstraction layer has to be created to cover those heterogeneity problems, which arise from any mismatch of the involved interfaces as shown in Figure 1.
- Second, means to express the functionality of a server to both, human and software clients, are to be supplied to a common trading service.
- Third, since clients and servers do not provide implicit knowledge about each other's interfaces, it has to be explicified dynamically to prevent mismatching interaction.
- Finally, a sufficient level of generality is required if the previous aspects are covered by an integrating standard.

Applied to the car rental example, a service interface description notation should define all necessary elements that enable the car booking task to human clients: different user interfaces might be involved, invalid data values and types are to be rejected. The user may first enter a specification form of the required car and thereafter acknowledge a final order form. Therefore, several structural and behavioral constraints have to be explicated within the service description.

Consequently, we have identified the following elements of an extended interface description:

3.1 Type and Procedure Description

The necessary information about service functionality and interface could, in a RPC scenario, be provided based on a unified and standardized service interface description language (SIDL) for the remote procedure call interfaces of any remote service available to any client in the network.

Automatic stub generation is well applicable in local area network environments, where both client and server code is written and used by a limited group of closely related developers [16]. In open system environments, however, a user is not expected to be able to compile and link client interface code. Under such conditions, a generic client interface with an *interpreting stub* would be more appropriate. But having only a *generic communication interface*, the client application code still remains server specific. By replacing this code with a *generic user interface*, the client application as a whole is able to *adapt dynamically* (i.e. 'automatically') to any interface as required by a server. Therefore, the following aspects have to be formalized as well in order to be interpretable by generic client applications:

3.2 Export Description

A server's location is, in general, not known to a client in the open network. If a user searches for a service to utilize in such a scenario, a communication connection can not be established between client and server unless it has acquired a matching service description. In advanced distributed application environments, this task of selecting an appropriate service and providing the communication link to it is supported by a *trader* component or service. The trader's task is to register service descriptions, which are to be received from servers from anywhere in the network, and then to facilitate a client's search for a specific service according to its request. In an *attribute list* based service description technique as proposed here, the server provides two alternate service descriptions techniques: the *export description* and *natural language tags*. Correspondingly, a trader offers two alternate service acquisition techniques in our model:

1. A trading mechanism, based on *attribute lists* (the *export description*) [17].
2. An interactive *browsing* mechanism through registered services by the user based on *natural language tags*.

To support the task of service selection, the service interface description as proposed in this paper is extended by an *export description*, which characterizes the interface of a server as a whole on the basis of a server specific (formalized and standardized) *attribute list.*

3.3 User Interface Description

In our service interface description technique, the interface description items (such as type, procedure, state and export description) are extended by a *label* and a *comment* attribute for each service description. These extensions may be used to supply natural language annotations of the item they are bound to. They serve as an additional redundant, user-oriented *tag* to support an interactive analysis of service characteristics, which are available and potentially useful for specific client request in the network.

Since a generic user interface is driven by the service interface description, *type-specific editors* for data values can help to prevent potential type mismatches. Each type that can be defined within the service description is automatically mapped into a specific editor structure. Further constraints, like subrange types for integers, should be possible to define. These constraints should be reflected by specific user interface objects, which prevent input of not type conforming data values. Thus, the generic client application is able to prevent transmission of faulty parameters to remote servers.

In order to reflect, e.g., a subrange type, several user interface objects are selectable; therefore, the service description language is additionally extended by various *presentation hints* for the respective data types. These descriptions can then be used to automatically create a (graphical) user interface representation of the respective remote service interface values on a local I/O device - independent of the server and its function accessed in the open network environment.

3.4 State description

In general, servers can be classified as *stateless* and *statefull* servers. Services of a stateless server can be invoked in any order, while in the case of statefull servers only a subset of all possible invocation sequences is allowed. A specification of this subset of allowed state transitions is part of the protocol description of that respective server. If a formal service description contains information about those 'legal' server states, remote clients could also acquire that description and restrict client behaviour to only that which is allowed at the server interface.

Consequently, the last part of our service interface description is a formal description of a *state transition automaton* which specifies legal server states and transitions as initiated by server functions and service requests. By defining communication states as a part of the service description interpreted at run time, they are not "hard wired" within client and server instances. Therefore, the state description serves as an application protocol specification.

3.5 Implementation Architecture Overview

In the following, the general system architecture of a *prototype system environment* is presented, which implements in a small example the basic components of a distributed client/server environment with formally and uniquely defined interfaces as presented above. A first *prototype* of the respective main system components has been implemented in the context of a locally distributed heterogeneous open system network environment.

The corresponding system model involves four kinds of components: client, client agent (CAG), server agent (SAG) and server. Clients and servers consist of application code and an interface to their respective local agents. The interaction between clients and servers can be divided into 5 phases: start-up of the components, binding between client and server, service invocation, unbinding, and shut-down of the components. The process of *binding* implies the selection of a server as well as the import of the server's service description. As shown in Figure 4, at the beginning of a binding the service description is stored at the server's site after being converted from an external representation. The next step is the transfer of the service description to both agents, where it is stored persistently. Only the internal representation of a service description needs to be standardized since it is interchanged between heterogeneous components.

During a *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, it is discovered by the agent local to the sender, and an error code is returned.

Instead of involving a specific client application, parameter values are mapped directly to the user interface level. Therefore, the generic user interface supports functions for the user to select an appropriate server, to examine the service procedures offered by this server and, finally, to invoke selected procedures. Thus, the communication-oriented 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 8). The required type description is retrieved from the local CAG. Return values are presented in the same way.

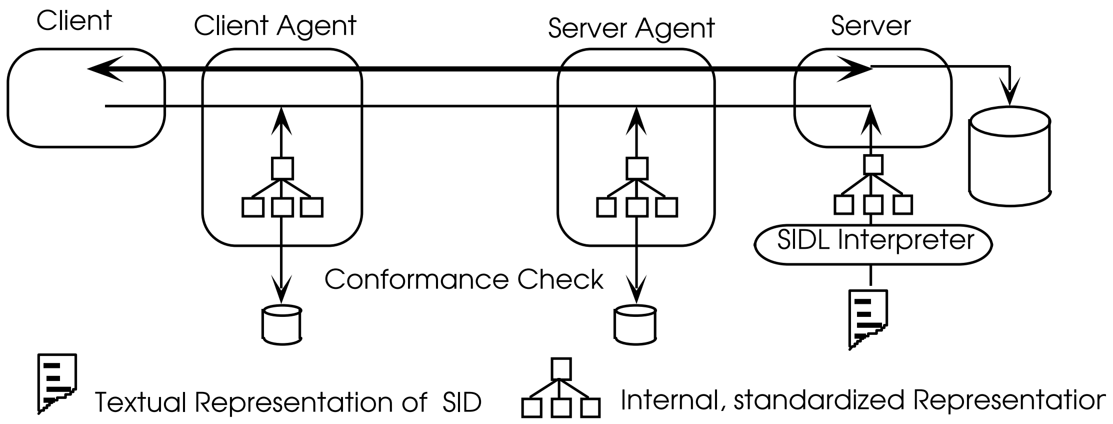


Figure 4: Transfer of service description from server to other components involved

By involving this generic style of user interface to remote services, a conformance between the client and server interfaces is given implicitly. The possibility of non-conformance is, however, left at the semantic level of an application. The effort of client developement is, therefore, reduced to only one implemention per hardware and software platform.

The prototype, implemented according to the model described above, supports the integration of user interface and communication service aspects. Developing a new server application requires solely to code service procedures upon the server communication interface and to describe these prococedures 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 tags.

The prototype implementation was developed on the basis of the SUN RPC library using XDR (eXternal Data Representation) [16] as presentation service. The technical environment consists of IBM RS/6000 and Sun SPARC Stations running AIX and SunOS as operating systems. In an implementation designed for heterogeneous and open system scenarios, the allocation of the involved client and server components to hardware systems is not restricted to the example configuration as shown in Figure 5.

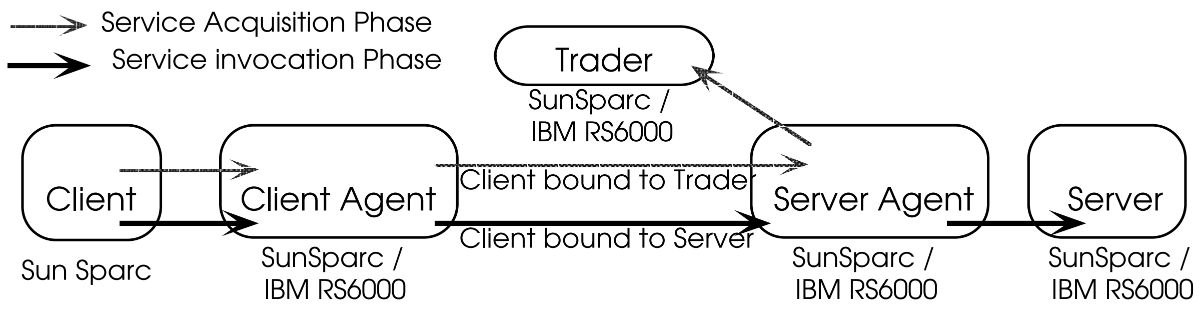


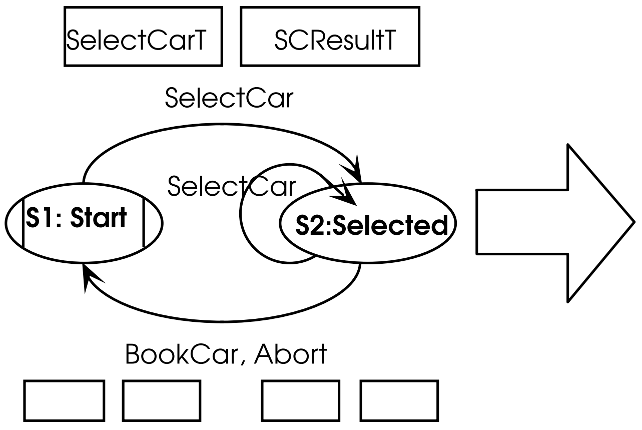
Figure 5: Prototype implementation

4. A SERVICE INTERFACE DESCRIPTION LANGUAGE

This section presents some technical details of the Service Interface Description Language (SIDL) as proposed in this paper. As a consequence of the service description elements as presented above, a SIDL service interface description contains four main components:

- The *type description*, defining at least one type for RPC parameters to be transferred,
- the *procedure description*, which describes remote services as a procedural interface,
- an optional *state description* in case of statefull servers, and
- an *export description*, which classifies the service exported on basis of attribute lists.

The remaining user interface description is syntactically integrated into these components.



```
TYPE SelectCarT RECORD { ... };
TYPE SCResultT ...;
TYPE BookCarT ...;
...
SERVICE SelectCar {
                REQUEST SelectCarT
                RESULT    SCReturnT;
                ...
                };
...
STATES {
                INIT: SelectCar -> S2
                S2: SelectCar -> S2;
                S2: BookCar -> INIT;
                S2: Abort -> INIT;
                };
EXPORT { ... };
```

Figure 6: A service interface and the description in SIDL notation

4.1 Data Type Declarations

Any remote procedure call may require structured parameter or return values to be transmitted to and from client and server interfaces. Therefore, an orthogonal and complete type system is necessary to describe these types uniquely. In distributed systems, pointer or reference types are not allowed since their values are invalid outside of their local context. However, transformation functions have to be supplied to transform between tree or list structures as a local representation and unique bit sequences as a general transfer representation.

Accordingly, the SIDL type system contains the following types:

Basic types:

```
INTEGER, DATE, CARDINAL, FLOAT, CHAR, STRING and TEXT
```

Structured types:

```
RECORD { ... }, CHOICE { ... } and SEQUENCE {...}
```

Opaque type:

```
ANY
```

The TEXT type refers to a text file on the local workstation, which can be embedded into a RPC parameter. CHOICE specifies the variant part of a RECORD discriminated by a type tag. A SEQUENCE type denotes a repetition of identical subtypes. The opaque type ANY allows dynamic types, which are dynamically received at runtime but not checked for conformance, since their actual type can not be anticipated at binding time.

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

```
TYPE SelectCarT RECORD {
             STRING,    LABEL "Booking Date";
             INTEGER, LABEL "Mileage", RANGE TINY 50 10000;
             INTEGER, LABEL "# Days", RANGE TINY 1 100;
             INTEGER, LABEL "Model", COMM "For a broader range
                                      of models consult our
                                      service at main branch",
                 RANGE RADIO 3 "BMW 323" "VW Golf" "Fiat UNO";
             STRING,  LABEL "Customer Name";
             STRING,  LABEL "First Name";
             STRING, LABEL "Street";
             STRING, LABEL "Zip Code";
             STRING, LABEL "City";
             CHOICE {
                     INTEGER LABEL "Visa";
                     INTEGER LABEL "Master";
                     INTEGER LABEL "Amex";
                     INTEGER LABEL "Invoice";
                     } LABEL "Payment";
};
```

defines a record type that contains nested structured and basic types. Some integers are constraint types restricted to a subrange of, e.g., 100 in the case of the "# Days" field. 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. Extension list attributes are a subject to standardization in order to be interpreted correctly at heterogeneous sites. For the automatic generation of user interfaces, however, they are treated as hints, since they may not necessarily be considered by the generator.

4.2 Service Procedure Interface Description

Service procedures may differ in parameter type or in call semantics. To define these procedures, they are supplied with an attribute list that contains at least the mandatory attributes REQUEST and RESULT, which refer to SIDL data types. Further attributes controlling the remote procedure call semantics can be supplied optionally.

The following example shows how natural language extensions are embedded into a procedure description. The standardized keyword COMM (for 'comment') is followed by an annotation that contains hints for the human user on the intended procedure semantics. This information should be accessable for a human user while gathering for a suitable service at a generic trader function.

```
PROCEDURE SelectCar {
        REQUEST SelectCar;
        RESULT  ResultType;
        TIMEOUT 120, COMM "Check availability takes time";
        /* more optional attributes */
        }, COMM "Claims reservation,
                     committed by CommitBooking";
```

The interactive trading mechanism requires natural language annotations as given by the COMM extension presented above. A server developer is encouraged to use annotations within the service description as well as a client's user is encouraged to browse through a trader directory when searching for appropriate services in the network. Finally, the generic user interface should provide the possibility to retrieve this additional information about the service procedures the user is currently working with.

4.3 State Description

As mentioned above, stateless servers can performe client requests at the server site in any order. Stateful servers, however, require a client to issue server calls in an distinct order. In the car rental example, this would mean that a reservation can only be committed by a user when preceeded by a car model selection. Such restrictions on how to use a specific server are, in general communication specifications, part of the *protocol* or *state* description of the respective server's behaviour. Usually, state-transition diagrams, resp. finite state machines, are used to model the range of valid sequences of potential service calls. Since states, as they appear at the RPC interface level, are specific for the server application, a individual application protocol specification has also to be supplied for each service as part of the service interface description (see Figure 7).

During the client/ server interaction, the server state is traced by the CAG and SAG in order to provide a state conformance check for further RPCs. State specifications may also contain an extension list with annotations for each transition, resp. for the state description as a whole.

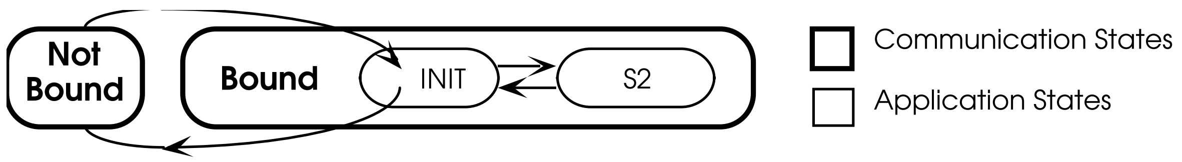


Figure 7: Server states and their definition in the state description

4.4 Export Interface Description

The export interface specification is an optional part of a service interface description and provides an attribute list which characterizes the service as a whole. In contrast to, e.g., extension list keywords, export definition attributes are not standardized within the scope of SIDL; they are currently restricted to providing an option for additional informal and server specific service descriptions (which, of course, could lateron be standardized as well). For example, the following export definition describes a car rental service:

```
EXPORT {
        SERVICE_CLASS            CAR_RENTAL;
        SERVICE_NAME             "RentACar";
        SERVICE_FEE_CATEGORY PER_INVOCATION;
        SERVICE_FEE_CURRENCY USD;
        SERVICE_FEE_CHARGE       0.1;
        };
```

4.5 User Interface Description

The user interface specification of a remote service in open systems provides some additional hints for a client, which may use it for an automatic (graphical or window) presentation of the typed data values. Such hints have to keep a distinct level of abstraction in order to allow a wide range of potential window managers to support an implementation of a generic user interface on top. Type-specific editors of such interfaces may vary in their visual appearance, e.g. the type '`TINY` integer' may be graphically represented as a slider or as an entry field. Figure 8 gives an example on how the user interface specification could be used for automatically generating a query form from the respective SIDL service interface description.

As an example, the upper right window of Figure 8 shows a service description file, where the type `SelCarT` is defined and used as 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 8 represent the parameter value for the procedure invocation. The actual parameter transfer is effected by pressing the "Write TDO" button: a Tagged Data Object (TDO) is generated from the current data value and sent to the server. The transfer syntax of this data object is checked for conformance by the CAG, resp. SAG component.

(See appendix)

Figure 8: Prototype application

5. CONCLUDING REMARKS

This contribution aimed at improved system support for the problem of matching specific application program *client requests* with arbitrary but appropriate generic remote *server interface functions* as provided at dedicated server nodes in modern distributed and heterogeneous computer network environments. Such open systems environments typically contain a multitude of heterogeneous and autonomous client and server components which occasionally cooperate in performing specific distributed application tasks.

In order to support application development for open client/ server environments, the paper addresses the important problem of appropriately describing the multitude of various and different (user and server) *interfaces* in a uniform, standardized and machine-readable way. Such a description represents a basic prerequisite for systematic computer support for distributed client/ server applications. This cooperation is, in practice however, often hindered by the lack of adequate (formal) interface description mechanisms. Therefore, the paper proposes a concept, describes a language and shortly mentioned a corresponding prototype implementation for a unifying *network/ service interface description* technique.

In result, the proposed *service interface description language* (SIDL) helps to reduce both, *complexity,* required to access heterogeneous services in open systems, as well as *implementation effort,* required for realizing open distributed applications considerably by providing the necessary system support for uniquely specifying all involved user client and server communication interfaces. Finally, we have demonstrated how such interface specifications can also be used for the *automatic* creation of a local human user interface to any remote server with a corresponding formal service description, as proposed in this paper.

Future work in this area concentrates, e.g., on a relaxation of the interaction restrictions between user oriented and communication oriented components. For example, the human user interface should be freed from acting as a visual parameter entry stub for RPC invocation while keeping the concept of generality via loaded service description at binding time. Therefore, a deeper examination of corresponding *User Interface Management System* (UIMS) technologies and their relationship to respective communication oriented services seems advisable, especially for future large-scale distributed information services in open systems.

6. REFERENCES

[1] W. Lamersdorf, et. al.: Database Programming for Distributed Office Systems, IEEE Office Automation Symposium, Los Alamitos, 1987

[2] M. Oeszu, P. Valduriez: Principals of Distributed Database Systems, Prentice Hall, New Jersey, 1991

[3] ISO / IEC JTC 1 / SC21 / WG 3: Information Processing Systems - Open System Interconnection (OSI): Remote Database Access (RDA), International Standard 9579, 1993

[4] J. Rosenberg et al.: Multi-media Document Translation - ODA and the EXPRESS Project, New York, 1991

[5] P. Linington: Introduction to the ODP Basic Reference Model, in: International IFIP Workshop on ODP, Berlin, 1991

[6] APM Ltd.: ANSA - An Application Programmer's Introduction to the Architecture, Cambridge, UK, 1991

[7] APM Ltd.: ANSA - An Engineer's Introduction to the Architecture, Cambridge, UK, 1989

[8] M.S. Verrall: Unity Doesn't Imply Unification or Overcoming Heterogeneity Problems in Distributed Software Engineering Environments, in: The Computer Journal, Vol. 34, No. 6, 1991

[9] M. Merz: Generische Unterstützung verteilter Client/ Server-Kooperation in offenen Systemen (Generic Support for Distributed Client/ Server Cooperation in Open Systems), Diploma Thesis, Dept. of Computer Science, University of Hamburg, 1992

[10] P. Gaudette: A Tutorial on ASN.1, Technical Report NCSL/SNA, 1989

[11] S. Pappe: Datenbankzugriff in offenen Rechnernetzen (Database Access in Open Systems), Springer-Verlag, Berlin, 1991

[12] M. Bearman, K. Raymond: Federating Traders: An ODP Adventure, in: Proceedings International IFIP Workshop on ODP, Berlin, 1991

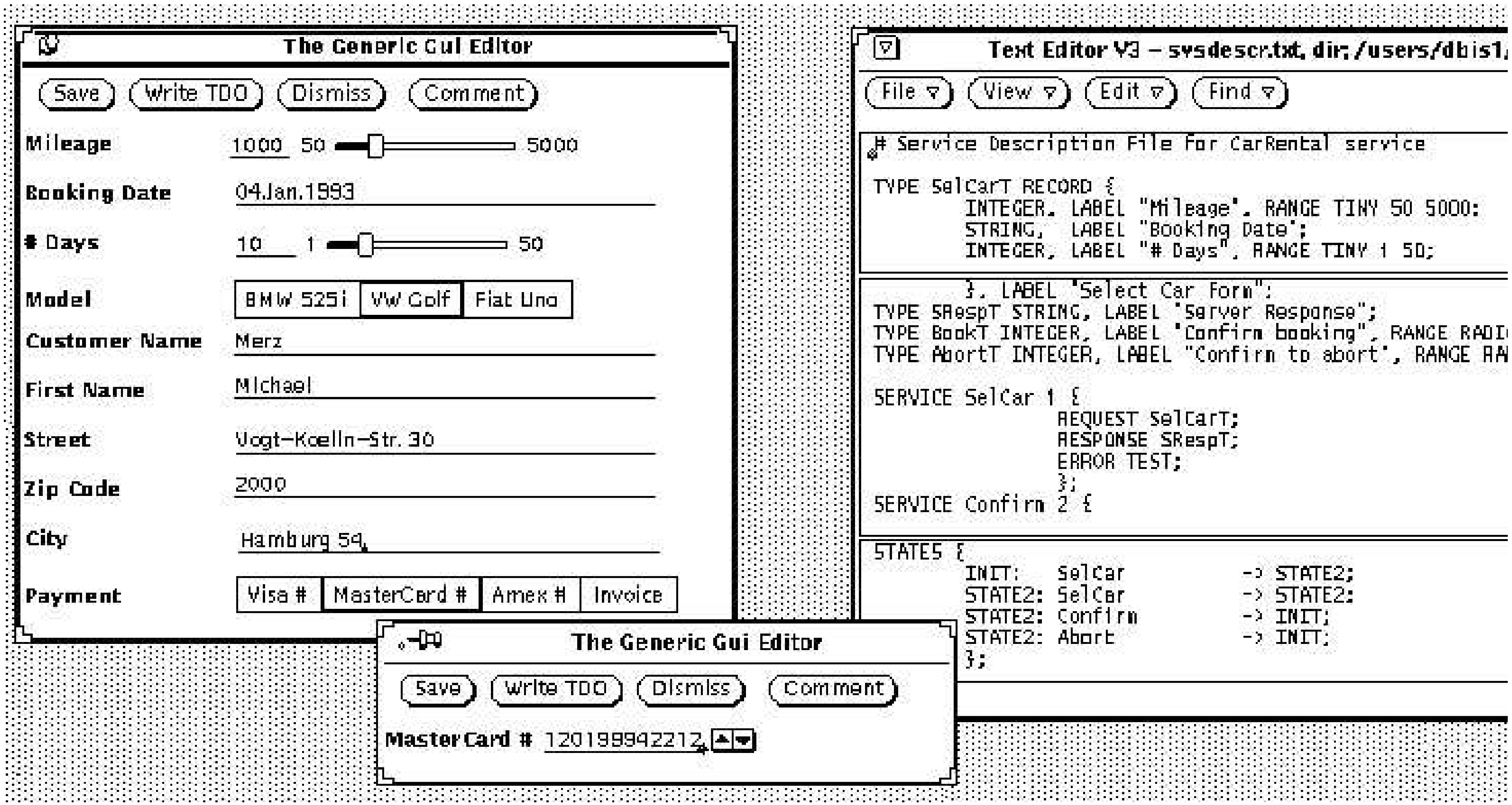
[13] A. Wolisz, V. Tschammer: Service Provider Selection in an Open Services Environment, in: Second IEEE Workshop on Future Trends of Distributed Computing Systems, IEEE Computer Society Press, Los Alamitos, 1990

[14] J. Nehmer, F. Mattern: Service Modeling in Distributed Operating Systems, in: Second IEEE Workshop on Future Trends of Distributed Computing Systems, IEEE Computer Society Press, Los Alamitos, 1990

[15] A. Birrel, B. J. Nelson: Implementing Remote Procedure Calls, in: ACM-TOCS, Vol. 2, 1984

[16] Sun Microsystems: Network Programming Guide, 1990

[17] R. N. Chang, C. V. Ravishankar: A Service Acquisition Mechanism for the Client/ Service Model in Cygnus, in: 11th International Conference on Distributed Computing Systems, Arlington, Texas, 1991



Mileage	1000 50 5000
Booking Date	04.Jan.1993
# Days	10 1 50
Model	BMW 525i | VW Golf | Fiat Uno
Customer Name	Merz
First Name	Michael
Street	Vogt-Koelln-Str. 30
Zip Code	2000
City	Hamburg 54
Payment	Visa # | MasterCard # | Amex # | Invoice

MasterCard # 120199942212

```
# Service Description File for CarRental service

TYPE SelCarT RECORD {
        INTEGER, LABEL "Mileage", RANGE TINY 50 5000;
        STRING,  LABEL "Booking Date";
        INTEGER, LABEL "# Days", RANGE TINY 1 50;

        }, LABEL "Select Car Form";
TYPE SRespT STRING, LABEL "Server Response";
TYPE BookT INTEGER, LABEL "Confirm booking", RANGE RADI
TYPE AbortT INTEGER, LABEL "Confirm to abort", RANGE RA

SERVICE SelCar 1 {
                REQUEST SelCarT;
                RESPONSE SRespT;
                ERROR TEST;
                };
SERVICE Confirm 2 {

STATES {
        INIT:   SelCar          -> STATE2;
        STATE2: SelCar          -> STATE2;
        STATE2: Confirm         -> INIT;
        STATE2: Abort           -> INIT;
        };
```