FOCS – A Toolkit For Flexible Telecommunication Systems

Jens-Peter Redlich <redlich@informatik.hu-berlin.de>

Abstract

One peculiarity of todays telecommunication systems is their tremendous heterogeneity which is mainly owed to their historical, evolutionary development and perpetual introduction of new devices and technologies. This makes it hard for a programmer / system designer to integrate many (sub-) systems into one application and to provide "flexible services"[1].

In this paper we will emphasize the necessity for a new telecommunication application framework that allows to integrate many different services and provides these applications with a mechanism allowing late / dynamic (re-) configuration.

FOCS[2] *is a recently started project that examines the applicability of object oriented design and implementation methods in the field of telecommunications. As many service providers offer equivalent*[3] *services, it seams reasonable to introduce service abstractions and to design applications in a way that makes it possible to reference these service abstractions instead of the actual service providers.*

I. Introduction

Todays telecommunication systems are characterized by a great variety of equipment connected through the network, many of them providing similar services. For example, the world wide public telephone network accommodates devices dating back to ancient times of telecommunications as well as modern digital ISDN devices.

At present time we can observe both the continuous output of new standards from standardization organizations and a perpetual change of the network itself (e.g. by adding new equipment or introducing new technologies).

For this reason the actual implementation of a particular service may be diverse (e.g. telephone: POTS, ISDN, or even videophone). Which of them we choose depends on things such as special requirements we may have, availability of special-purpose hardware or other resources (e.g. bandwidth), or service costs. Availability of resources may change over time relatively slowly (e.g. provision of a network access), less slowly (e.g. wait until purchase of new special-purpose hardware), or even comparatively fast (e.g. wait until the local router is running again or traffic congestions are resolved).

From the user's point of view, the different implementations of a particular service look all the same. It is the underlying mechanisms that changes (and that the user normally doesn't want to care about). These underlying mechanisms often themselves rely on (sub-) services, the implementation of which can be manifold too. Consider for example a service providing data transportation: Depending on which data encoding algorithm we use (or if we use one at all) we will get different implementations of that service. They all rely on different sub-services (here the data encoding algorithm) but provide one and the same service to the user, namely data transportation.

For *flexible services* it may be possible that the service is still available even if some sub-services are not (e.g. due to a lack of resources). However, the service has to change its form in this case (e.g. a videophone service might substitute its "continuous video" sub-service by a "slow motion video").

At Humboldt-University Berlin we are currently undertaking a project called *FOCS* which investigates the issue of building up complex telecommunication systems by using less complex components. These components correlate to the sub-services mentioned above and are used to obtain a maximum level of flexibility. The choice of a particular component may, for instance, be made at runtime from a set of equivalent service providers. Simultaneously we investigate to which extend the appliance of object-oriented techniques can simplify and speed up the design and implementation of communication applications.

Although only recently started, this (object-oriented) approach has already proven essential at this stage of the project. To support the objectives of FOCS in the best possible way we introduced a new inter-object communication model which will be described later.

The ultimate aim of FOCS is to provide a framework for a seamless integration of many different services and their implementations. Once a service provider has been

[1] With "flexible services" we mean services that are able to adapt to changing situations. How this adaptability can be achieved is one topic of this paper.

[2] "FOCS" = "Flexibility Oriented teleCommunication System".

[3] The term "equivalent service" makes only sense in respect to a certain (user-) viewpoint (what the user expects the service to do).

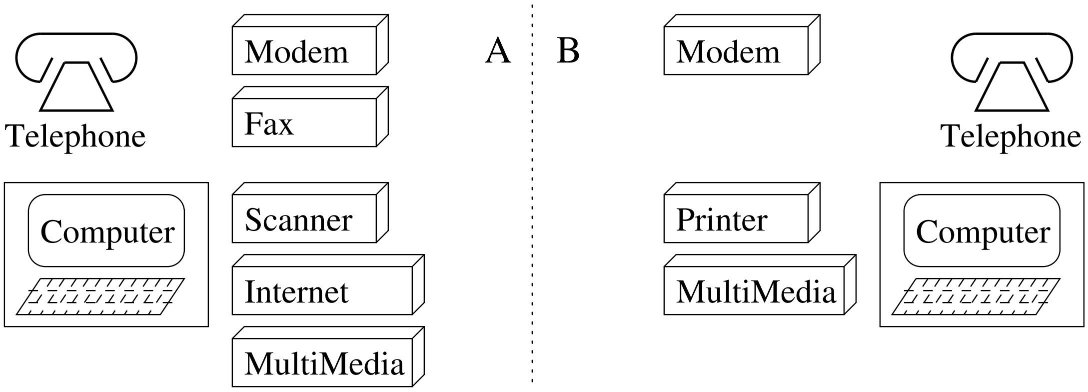

Figure 1: **A** *wants to send a message to* **B**. **A** *has the following technical equipment available: telephone, modem, fax, scanner and a computer with internet access and multimedia hardware.* **B** *owns: a telephone, a printer and also a computer with a multimedia extension. The question is: How can we get the message passed from* **A** *to* **B** *?*

implemented, it should be possible to reuse it under totally different circumstances.

FOCS' general approach to meet these goals is presented in section II of this paper. Basic concepts are: object based / oriented system design, encapsulation, service abstraction (introduction of ServiceID), service market. An example is given to illustrate that there may exist several ways to realize a certain (abstract) service, none of which is universal because of its dependence on resources that are not available everywhere.

The FOCS-Toolkit is introduced in section III. It incorporates the basic concepts mentioned above. We explain what service providers (SP's) are and how they establish a (local) service market. Brokers are presented as special SP's, which are to locate appropriate providers of (abstract) services on the service market.
The importance of a new inter object communication model is stressed, because it enables late configuration as requested above (by completely avoiding explicit references to service providers).

Although the project has only recently started and is just about to get into shape, a brief list of future plans is given in section IV, which concludes this paper.

In what follows we demonstrate realizations of the service described in figure 1. Note that some of the resources needed are of a dynamic kind (their availability may change fast), while others are not:

1. **Traditional Telephone Call**
   (a) Both A and B do have a telephone (static).
   (b) It is possible to get a telephone connection between A and B (dynamic).
   (c) The line is not busy; i.e. B does not use its telephone (dynamic).
   (d) B is at home (dynamic).
2. **Fax**
   We cannot fax from A to B as B does not have a fax receiver (static). We may wish to scan the letter instead and have it delivered as a computer processable file.
   (a) A has a scanner (static).
   (b) File delivery is possible. B has no Internet access so we have to use the modems instead.
      i. A and B do have a modem (static).
      ii. It is possible to get a telephone line from A to B (dynamic).
   (c) B has the software to process, and a printer to print the file (static).
3. **Computer Document**
   (a) The document can be created on A's computer using an editor or the scanner (static).
   (b) B has a computer and appropriate hard- and/or software to present the file (static).
   (c) File delivery is possible. As B has no Internet access, we have to use modems.
      i. A and B do have a modem (static).
      ii. It is possible to get a telephone line from A to B (dynamic).

II. Basic Concepts

Modern telecommunication systems are highly computer based systems. Many of todays services are implemented in software. Therefore it is important to study

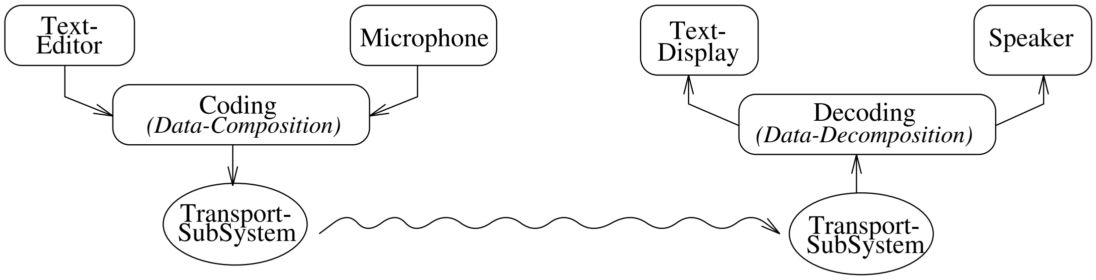

Figure 2: *Example of a simple application, allowing simultaneous recording, transmission and presentation of audio and text data. Important: The components shown in the picture above are abstractions, which have to be substituted by concrete objects. Possibly there exists a choice between many implementations for some of them.*

the applicability of modern software development methods, e.g. object orientation, for this field.

This approach is further motivated by the phenomenon that especially telecommunication systems have to undergo many changes and extensions (because of modified user requirements, or introduction of new equipment and/or technologies). For practical reasons an evolutionary, continuity orientated, development is essential. OO-techniques promise to provide this.

II.A. The Object Based Model

Decreasing the complexity of large systems by structuring and decomposing them has already been investigated for a long time. The development of "*structured programming*", of which Modula-2 is a typical example, can be seen as a milestone on this way.

The central idea of this approach is to split large systems into smaller components that are responsible for only parts of the systems functionality. If these components are still quite large, they are split again, until a suitable component size is reached.

The central issue in this process, known as "stepwise refinement", is the independence of the components produced. If the programming language allows for access to internals of components and thus taking advantage of implementation details, software developers can seldom resist doing that. The result is that the obtained system is structured but not decomposed, because its components are not independent from each other and can only be understood by considering (and understanding) the entire system.

This lead to the idea of modelling components as "autonomous entities" (AE's) which hide all details about their internal structure and protect themselves against access from outside, except through a well defined (small) interface.

If this interface is based on sending and receiving messages, the AE is called an **object**. Systems build of objects are called **object based systems**. By correlating types to the objects and defining an inheritance relation for these types we obtain an **object oriented model**.

Object based systems have the advantage to encourage modularization and decomposition. This increases chances for software reusability and exchangeability of an object's implementation without affecting other parts of the system.

II.B. Objects as Service Providers

In object based systems the implementation details of a particular object become increasingly unimportant (perhaps this information is even not available any more). Instead the **service** provided by an object is emphasized. In this paper we see an object (-instance) essentially (only) as a service provider.

When specifying the appropriate object (that is needed for some purpose) there are basically two possibilities: Either we describe the object itself or the service it has to provide. The latter allows for dynamic selection of an appropriate implementation (i.e. object-type), while the former does not.

Objects can be composed of other objects which have to be specified using one of these methods. The former leads to a fixed design while the latter allows for flexible selection of appropriate sub-service providers. In this case we call the system **configurable**. Configuration can take place at compile time, at the time the application is started or immediately before the service is to be provided. Composing objects by service[4] is introduced in [7] as **Programming in Large Scale** whereas the implementation of the various objects is called **Programming in Small Scale**.

It should be stressed that it makes sense to imple-

[4] This is to determine (select) an object by its (abstract) service, not by its implementation.

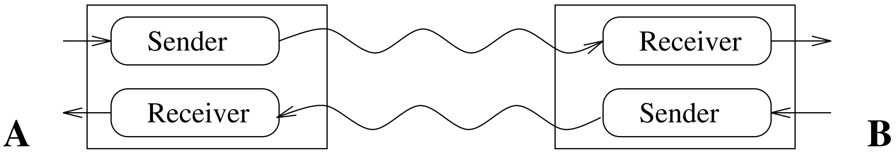

Figure 3: *In Version 1 there are two reliable unidirectional carrier services available. So we can construct the desired bidirectional service by simply combining them.*

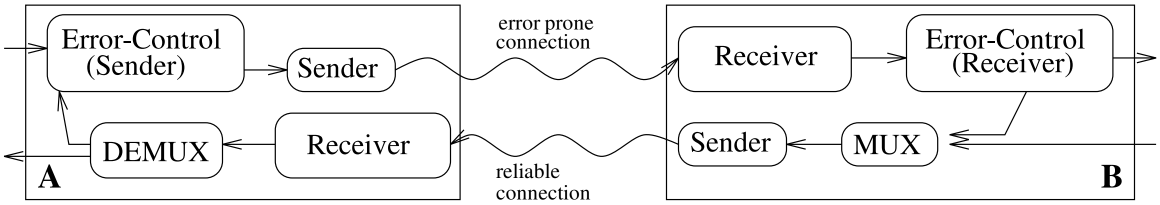

Figure 4: *In Version 2 only the unidirectional service from B to A is reliable, while the service from A to B is error prone (e.g. because of the quality of the chosen transmission media). In consequence its service has to be improved by adding components providing error-detection and error-correction functionality (to be placed on both sides). By using the reliable service and two additional MUX / DEMUX components we establish 2 reliable channels from B to A, one for transmission of the original user data, the other for transmission of error-protocol data (e.g. data for retransmission indications or confirmation of successfully received data blocks) which are to be send from B to A.*

ment many different service providers (object types) for one particular service. This is especially the case if only few of the implementations are applicable in a specific situation (e.g. because required resources are not available).

Figures 3 and 4 show 2 versions of a bidirectional carrier service implementation[5], using two unidirectional carrier services having opposite direction. Ignoring the contents of the boxes, which symbolize the service provider, both versions look the same. Hence, a service user (who doesn’t know anything about the box’s contents / the service providers implementation) can not distinguish between them.

Configuration allows to choose the version that is best suited for a specific environment (e.g. for which all required resources are available) and to integrate it into the application.

II.C. Service Market

If there are many different service providers available at the time a system is configured (i.e. selecting concrete objects and defining their interactional relations), a **service market** is established.

Configuration can be done statically (while compiling or linking the program) or dynamically (at program start or directly before usage of the service) or it may be modified dynamically (dynamic reconfiguration).

[5] A “carrier service” provides data transportation.

When applying dynamic configuration we need a mechanism allowing us to locate a certain service provider on the service market. It is clear that the application itself doesn’t know anything about the structure and maintenance of the service market.

Thus we assign a service identifier (**ServiceID**) to every service we use. A ServiceID looks like a term with the functor indicating the service class and the term’s arguments specifying the service more detailed (e.g. parameters or additional requirements).

A **broker** can then be entrusted to deliver an appropriate service provider (object) using the ServiceID as the service specification. Dynamically changing conditions in the environment (e.g. availability of resources) may affect the selection process.

It should be emphasized that there may exist many service providers for the same service (but usually differently implemented). On the other hand one single object may be able to provide various services (each having a different ServiceID associated).

III. FOCS

In section II we gave reasons for using an object-based approach when modelling and implementing complex communication systems. FOCS incorporates these concepts by means of a new, decoupled, inter-object communication mechanism that allows for:

- Temporal independencies of sender and receiver.
- Easy replacement of objects at runtime (dynamic (re-)configuration).
- Many (competing) receivers for one message.
- Explicit modelling of the flow of data and basic synchronization.
- Concurrent object activities.

III.A. Objects in FOCS

In FOCS, the terms *object* and *service provider* are used interchangeably. Virtually every object (that contributes to a piece of software) is a service provider. Exceptions are places and methods which are in some sense only auxiliary objects. Therefore we will go on and describe *service providers* having in mind that these are the fundamental *objects* in FOCS.

A service provider (SP) is a black box that provides a certain service and hides all details about its implementation. There are **channels** attached to an SP through which messages (i.e. message objects) can be received and send, respectively. For maintaining its (local) state it is equipped with local variables (which are neither visible nor accessible from outside the SP). A SP having no channels attached is called an **application**.

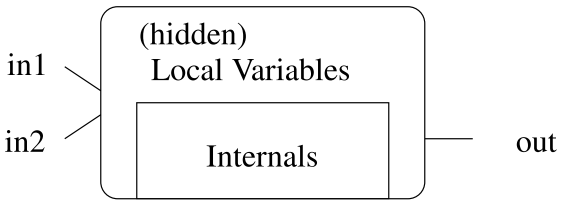

Figure 5: *This picture shows an SP with 3 channels: in1, in2 and out. The channel names are defined locally (where they are unique).*

We distinguish **basic** and **compound SP's**. A SP is called *basic* if it does not rely on any other SP. However, it can use services provided by non-FOCS systems (such as the underlying operating system). Because an SP hides its implementation anyhow and because its usage of non-FOCS systems has no impact on any other FOCS-object, there is no fundamental difference to an SP which provides its service completely by itself. This is an essential aspect of FOCS as it allows to incorporate many of todays already existing communication systems and by that, software reusability in a large scale.

In opposition to a basic SP a compound SP implements only part of its service. For the complete service it depends on sub-services provided by other SP's, which may themselves be compound or basic.

In figure 6 we see a CommandTool-SP that provides an interactive command driven tool (e.g. a shell or monitor). It depends on two sub-services: a text-window and a so called evaluator, each having two channels attached to them. The text-window-SP accepts (string-)objects on one channel and communicates them to the user (e.g. by displaying them in a text window on the screen), while words communicated from the user (e.g. typed in at the keyboard) will be sent as (string-) objects through the other channel. The evaluator SP on the other side accepts (string-)objects at its first channel, evaluates them according to the rules of a specific language (and producing the desired side effects), and sends the result through the other channel (as output). The CommandTool-SP only needs to connect these two sub-SP's. Doing so it establishes its own service, in this case some interactive CommandTool.

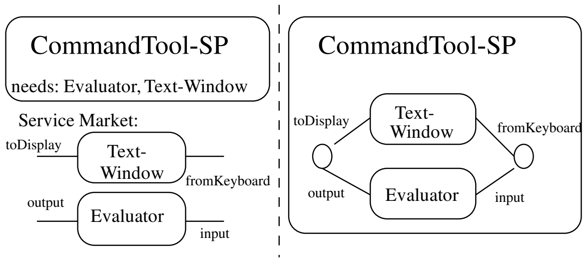

Figure 6: *The compound SP shown above consists of an "Evaluator-SP" and an SP providing facilities for communication between a human and the machine.*

A compound SP hides its sub-SP's like all other local details. During initialization it requests its sub-services to be established. The task of getting appropriate service providers for this is done by a broker.

It lies in the very nature of FOCS (and any other object-based system that adheres to hiding of implementation details) that it is not known *how* services are implemented. But in FOCS this means even more: One and the same service might end up being provided by totally different SP's. In our example, the service "text window" might be provided by a vt100 terminal, by an X-window, a NeXT-DPS window, or some other thing that can communicate strings to a user (i.e. display text somehow somewhere). Obviously, the choice of the "right" service provider depends on several factors, like availability of certain hardware or software (e.g. window servers), preferences of the user and so forth. The CommandTool-SP does not depend on the implementation of the text window service. Therefore one and the same (CommandTool-)SP works on different platforms (SUN or NeXT, with or without X server etc.) without the need of adapting it.

III.B. Inter-Object Communication

In the previous section one could get an idea of how service providers interact. This will be explained in more detail now:

Every service provider has a set of methods that act on its internal variables and may change its internal state just like in conventional object oriented systems. However, in those systems a method is usually invoked by sending a message to the object. Moreover, in many cases this is done synchronously, i.e. the sender has to wait until the receiver of the message has finished running the appropriate method. In FOCS, every method has some *input places* attached to it. A method is invoked when there is at least one (message-) object on each of the input places. The method then usually removes the message objects from its input places and eventually puts some (other) objects on its *output places* which are attached to it as well. Places are auxiliary objects that can store an unlimited amount of message objects.

Note that a method can only be invoked (can *fire*) if there are objects on each of its input places. If this is the case, a method is said to be *satisfied*. As a method may remove objects from its input places (which may in turn be input places of another method as well) other methods may *loose satisfaction* when this method fires. On the other hand a message may put objects on its output places which may cause other methods to be satisfied.

A method has to have at least one input place (otherwise it would be continuously firing), but does not necessarily have to have output places. This model has a number of features that can not be found in conventional object oriented systems:

- Inter-object communication is inherently decoupled and asynchronous. A method places a (message-) object on a place and goes ahead. It does not have to wait for the receiving service provider to process this message.

- Objects usually do not know their communication partner. All they see are some places on which messages appear and/or on which messages to put. That makes it easy to replace the actual object at runtime by a different object (that provides the same service and thus has the same interface). All we have to do is to disconnect the object (which is to be removed) from the places it is connected to, create the new object and connect it back to the same places. Because all other SP's reference only places (which haven't changed), they will not notice the *reconfiguration*.

- Many methods may "listen" to the same place(s). This may lead to one method not getting every message but only one in a while, thus having more time to process one message. This allows for easy load balancing.

- A method must not take objects from other places than its input places. Equally, no method may put objects on places other than its output places. This makes it quite easy to follow the flow of data.

- Having places and methods it is possible to achieve synchronization or to model capacities of places similarly to what can be modelled with Petri-Nets (although FOCS is by no means a Petri-Net implementation, in fact that is about the only similarity to Petri-Nets).

- The model implies concurrent object activities because method firing is done independently from other objects' activities.

At runtime the input and output places of the various methods of a service provider are assigned to existing places in the system. Now it is clear what the channels of the service providers meant to be: These are the places of some methods of a service provider (similar to "public" methods in other systems) and need to be assigned to existing places in the system in order to communicate with that object and (hence) benefit form the service provided by that object.

If we let circles denote places and bars denote methods we can "draw" a service provider in a manner that expresses some of the features explained above. Figure 7 gives an example.

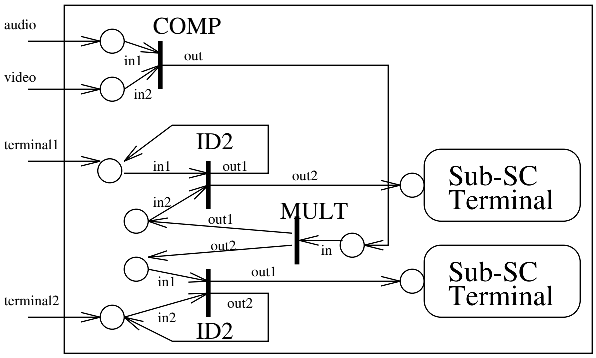

Figure 7: *This is a compound SP. It can be supplied with audio and video on two independent channels. After synchronization of the incoming streams, presentation is done at none, one or two terminals.*

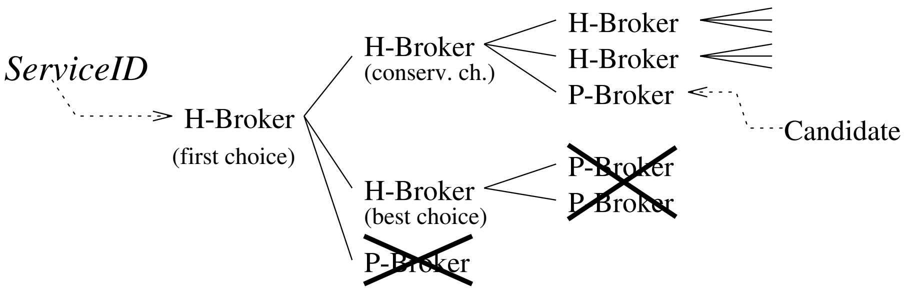

Figure 8: *Hierarchical brokers (H-Brokers) entrust sub-brokers (H-Brokers or P-Brokers) to look for service providers, using various strategies. The P-Broker marked "Candidate" is the first one found, which offers the desired service. All previously asked P-Brokers declined (crossed out in the picture).*

III.C. Brokers

Before SP's can be combined to higher level (compound) SP's, some service providers (sub-SP's) must be found. For this job FOCS integrates the concept of a **local service broker**.

Every object-type implemented in the local system provides one or more services (locally). It is possible that there are many providers for the same service which may be implemented quite differently.

By presenting a ServiceID[6] to the object-type, a broker can "ask" every (local) service provider whether it is able to provide the desired service. If not, the inquiry must be continued. This process either stops with an offer (of the service) or the service is not available at the local system. In the latter case the SP that asked for this service can now decide (autonomously) whether it is willing to lower its requirements (and restart the search, hoping that a lower quality service is available), or whether it can omit this particular service or whether the service was mandatory, so that it is not able to provide its own service (worst case).

For this reason a compound SP has to check for its resources (e.g. sub-SP's) before it commits itself to offer a certain service.

The ServiceID has the form of a term with the functor entitling the service class. The management of service class names (i.e. ServiceID functors) is described later. The arguments of the term specify the desired service more precisely. But there exist no general rules for how a ServiceID has to look like or how its arguments have to be interpreted (semantics of a ServiceID). Only objects really providing a particular service know the service and its description given by the ServiceID. All other service providers can easily detect that they don't know the service this ServiceID relates to (possibly by simply looking at the ServiceIDs functor). Because they will certainly not provide the desired service there is no need for them to understand the ServiceID.

Using ServiceID's it is possible to define service classes and to introduce concepts like inheritance and polymorphism. This leads us to an *object oriented model*, the details of which are beyond the scope of this paper.

Let us now discuss how a broker works:

We distinguish two types of brokers: **primitive brokers** and **hierarchical brokers**.

A *primitive broker* is responsible for only one single SP-implementation (but knows everything about it). It is able to decide whether or not the SP-implementation it manages is able to provide a service required by a given ServiceID. If yes, it returns an acceptance, otherwise it declines.

Hierarchical Brokers on the other hand don't know any SP-implementation. Instead, they know other brokers they can ask for service providers. There are several ways how to delegate the request to their sub-brokers: First, they can ask their sub-brokers sequentially (one after the other) or they can do it in parallel. Second, if there are more than one service-provider found, they have to select one. There are many strategies available how to do that: first choice, best choice (requiring an effective assessment-function), alternative choice, conservative choice, and many more.

If a hierarchical broker found some object providing the service it looked for, the state of the broker is usually saved. If it turns out (at a later time) that the selected object is not a good choice (e.g. because it blocks resources needed for other services), the broker can than be reactivated in order to provide an alternative object (backtracking).

[6] The ServiceID describes the service and the interface of an object which provides that service.

By combining brokers using different strategies it is possible to control how the broker observes the service market.

Note that in the paragraphs above we only discussed the local case which would hardly be sufficient for a telecommunication system. Brokers are services providers. So, if we want to search for a service in a distributed market[7] we simply ask a local broker to supply a service provider that offers the service of brokering across the net.

III.D. Composition of Objects (Phases of Object Creation)

From the above mentioned it is clear that the creation of new SP instances, especially if they are compound (i.e. they depend on services of other SP's), may be a complex process.

First of all a ServiceID specifying the desired service is presented to a broker. The broker now inquires other brokers until a primitive broker is found that manages an appropriate SP-implementation. This broker is capable to decide whether or not it can deliver an object that is able to provide the specified service (respecting all parameters and additional requirements as indicated in the ServiceID's arguments). The agreement notification can be subject to withdrawal, because it may turn out at a later time that certain resources which are necessary for the service are not available (e.g. the human communication partner on the other side of the line). A SP in this state is called **proposal**. If a proposal is a compound SP a list of sub-services needed must be built and brokers must be entrusted with finding the corresponding service providers. If it is possible to find an appropriate service-provider for every sub-service demanded, the (composed) SP is called **candidate**. If for some of the services specified in the demanded-subservices list, no service provider can be found the *proposal* has to decide whether to change the service requirements or to work without the unavailable services. If the latter is impossible, it has to declare itself unable to provide its service and the broker has to look for another service provider.

III.F. Call

If a user wants to use a certain telecommunication application, he (or she) starts a call. As a result an SP is created at every participating node. This SP's task is to control the call relation and to manage other SP's needed for this particular application (by managing SP's it manages real resources), including creation and termination of SP's.

[7] distributed across a network, that is.

The structure of a call providing traditional services for voice and/or data transfer is quite static. The termination of one component usually results in termination of the entire call. It is also unusual for traditional calls to involve more than 2 parties in one single call.

In FOCS it is an explicit goal to modify a calls structure after it has been established. This includes adding and removing parties to resp. from the call and the creation and termination of SP's on demand. Features allowing to modify an active SP should only be provided by the SP itself (encapsulation and autonomy of an SP).

FOCS distinguishes **mandatory** and **optional SP's**. Mandatory SP's are established and terminated together with the call, while optional SP's can be created and terminated dynamically on demand at any time during the call. They don't even have to be established at all. If an optional SP terminates because of errors, the SP that contains the terminating SP can decide whether to live without it, to restart, to replace it by an alternative service provider, or to terminate itself. Usually, this does not affect the existence of the call.

Because services usually act in a distributed manner, the existence of a powerful signalling system is tremendously important. It is hard to imagine how to provide the above mentioned dynamism of a call without separating call control and connection control (for further information refer to [6]). Signalling in FOCS is itself a service which can be provided by many (different) implementations.

III.G. Integration of Other Systems

In practice there exist already many service providers which can be used to construct new services. Typical examples are services providing data transport (e.g. TCP-IP, telephone), directory services (e.g. X500), but also services for human⇔computer communication (e.g. X-windows, multimedia toolkits from DEC and NeXT, ...). It must be possible to integrate these systems in FOCS, because otherwise FOCS would be just one more system (one of hundreds) to provide communication services.

To reach this goal SP's can be designed that reflect the functionality of the subsystem that is to be integrated in FOCS. Now it is possible for every FOCS-customer to access this object in order to use the subsystems service (via the newly designed FOCS interface). Moreover it is possible to select between subsystems by looking for appropriate SP's at the local service market.

It is understood that only SP's offering access to highly available services (these are right now especially services provided by non-FOCS systems) are likely to to reach each potential communication partner because that partner may not have the FOCS software available but may have access to the subsystem used (e.g. telephone, fax, internet).

III.H. Globally Unique ServiceID's

Selecting service providers satisfying various ServiceID's and combining them to a (distributed) application can work only if the naming convention of the ServiceID used to specify the cooperating services is common to all installed FOCS systems. Because FOCS intends to deal with global communication, this unique understanding of ServiceID's must be global too.

This does not mean that a particular implementation of a service must be globally available. It is even quite possible (and likely) that everybody owns his (or her) private service implementation of a service the ServiceID of which is one and the same. Hence it must be guaranteed that everybody associates the same service (including service semantics and service interface) with a given ServiceID.

For this reason everybody who wants to define a new service (for FOCS) has to provide a unique ServiceID for it (an additional detailed service description and a reference implementation would be appreciated).

Developers of (new) telecommunication applications can now decide whether they want to use this newly defined service. Providers of similar services may check whether their objects can provide the same service (in order to be selected by a broker if users start to use the new service).

To guarantee global unique ServiceID's a global authority is needed in order to administer the set of ServiceID's already in use. To avoid this it is proposed that every institution that intends to define new FOCS-services manages its own name space. The global unique ServiceID can then be yielded by appending a '@' and the internet address of the institution to the local name. Using this mechanism it is easy to know whom to ask for details or the implementation if only the name of a service is given. Names of new services (and a short abstract about what they do and how the interface is designed) may be distributed via *news*.

IV. Future Plans

The FOCS project, as presented in this paper, has just started. However, first prototype implementations of the framework system and a few services are available.

Future work is supposed to integrate many of todays (already existing) communication systems and will provide us with even new services and applications. Special interest will be given to issues of multimedia data transportation and processing in heterogeneous environments.

But for now the major task is to implement a set of basic services, which can be used for writing sample applications. These services will include various carrier services, signalling services, broker services for distributed service markets, services for code transfer and remote operation services as well (giving access to SP's available at remote hosts only).

For the future, access to "higher services" such as email, fax, telephone, video and processing of multimedia documents is planned.

A related project dealing with management of FOCS' service providers has just started.

Also, it is still under investigation what the concrete relationships between service providers and their brokers are. In our opinion it as an major advantage of FOCS that service providers (or their P-Brokers) are asked if they can establish a specific service instead of agreeing on a common (probably not optimal) notation for service description. However, this introduces the problem that there is possibly a large number of SP's (brokers) to be asked. It would be nice if the hierarchy that is provided by H-Brokers would logically correlate to a semantic structure of the service market which is yet to be established.

V. Conclusion

We stressed the necessity for a new, generic telecommunication application framework that allows to integrate many different services into one application and which benefits from the fact that some of these services can substitute each other.

We showed that OO-technologies are applicable in the field of telecommunication and that they lead to well structured applications. Late configuration allows for flexible services, i.e. services which are (mostly automatically) able to adapt to changing situations in the network or terminal equipment. This is achieved by utilization of brokers which locate appropriate service providers for a certain task on the (local) service market at runtime.

FOCS is a promising approach to implement these ideas in the form of a programming toolkit. It incorporates a new, powerful interobject communication model and allows to integrate existing (sub-) systems.

VI. References

[1] Bertrand Meyer, "*Objektorientierte Softwareentwicklung*", Hanser, München/Wien, 1990.

[2] N.Wirth, "*Programming in Modula-2*", Springer-Verlag, New York 1982.

[3] Andrew S. Tanenbaum. "*Computer-Netzwerke*", Wolframs Fachverlag, München 1991.

[4] Frank Bomarius, "*Ein System für die Programmierung verteilter objektorientierter Applikationen*", Dissertation am FB Informatik der Universität Kaiserslautern, 1990.

[5] Ludwig Keller, "*Vom Name-Server zum Trader - Ein Überblick über Trading in verteilten Systemen*", PIK 16, Heft 3/1993 Seite 122 ff.

[6] H. Ouibrahim, "*Access Signalling Architecture Based On An Object Oriented Service Description*", Workshop on Broadband Communications Estoril, 20-22. January 1992, Page 106-117.

[7] Jürgen Becher, "*Konfigurierung verteilter, heterogener Informationsverarbeitungssysteme*", Dissertation am Institut für Telematik, Universität Karlsruhe, 1993.

[8] Budd, "*Little Smalltalk*", Prentice Hall, 1987, ISBN 0-201-10698-1.

Author Information

Jens-Peter Redlich works as an assistant at the Dept. of Computer Science at Humboldt University Berlin / Germany, where he received a diploma in 1993.
His research currently focuses on practical design methods for and implementation of telecommunication and multimedia applications. His previous activities have been centered around the unix operating system, X-windows, and practical implementation of tools for machine ⇔ user communication.