Introduction of Gate Types into SDL

E. Holz, J. Fischer
Humboldt-Universität zu Berlin, Dept. of Computer Science
A.-Springer-Str. 54a, 10117 Berlin
{holz|fischer}@informatik.hu-berlin.de

1. Rationale

This paper proposes two sets of extensions to SDL [8]:

- Gate Types and dynamic Gate Instances to make SDL more suitable for the specification and description of open distributed systems.
- Concepts for the dynamic creation of blocks and services to harmonize these constructs with processes and to enable a more flexible specification design.

The decomposition of a system into objects which interact at their interfaces is a common baseline for all technologies for the design and implementation of open distributed systems. This holds as well for the abstract framework, the Reference Model of Open Distributed Systems (RM-ODP, [2]-[5]) as for concrete architectures as CORBA [1], COM/DCOM or TINA [11].

Objects are dually characterized by their state and their behaviour. They are always encapsulated, that is, changes in an object‘s state can only occur as a result of an internal action or an interaction with its environment. Interfaces again are abstractions of the behaviour of an object consisting of a subset of the object‘s interactions and constraints on their occurence. An object may support multiple occurences of interfaces (of the same or different types). In order for two objects to interact, they have first to obtain references to the respective interface instances, what is supported by the infrastructure.

These concepts are also reflected in the different languages for the computational specification of distributed systems, e.g. CORBA IDL or TINA ODL [12]. The computational languages provide means for the specification of interface types/templates (*interface* definition in ODL, IDL) and for object types/templates (*object* definition in ODL). On the other side, the concrete computational languages are missing features for the descrition of behaviour and the actual configuration of the system to be developed. To overcome this shortcome different approaches to combine SDL and IDL or ODL have been made (cf. [5], [15], Y.SCE tool of GMD and other). These techniques start with a computational specification in IDL or ODL, map them to SDL-92 and add the behaviour specification at the SDL level. A major obstacle is however SDL‘s lack of concepts directly reflecting the concept of objects and interfaces. This implies that the ODP concepts themselves have to be modelled with SDL. Different combinations of blocks containing processes and being connected by channels are used instead of a single ODP construct, what implies additional specification overhead.

2. Definition of Gate Types

Gate Types are defined by a gate type definition similar to the other structural type definitions:

```
<gate type> ::=[<virtuality>]
        gate type <gate name>
            [<formal context parameters>]
            [<specialisation>] <end>
            [adding]{<entity in gate type>}*
            <gate constraints> <end>
        endgate type [<gate name>] <end>
```

```
<entity in gate type> ::=
                                 <signal definition>
                               | <signal list definition>
                               | <remote variable definition>
                               | <remote procedure definition>
<gate constraints ::=          <gate constraint> <end> [<gate constraint> <end> ]
```

A <gate type> can be defined in those scope units where <service type>s can be defined. A <formal context parameters> can be a <signal context parameter>, a <remote procedure context parameter>, a <remote variable context parameter> or a <sort context parameter>. It is assumed, that the declaration of remote procedure or remote variable identifiers in a gate constraint implies the existence of the appropriate implicit xCallA and XReply signals in the respective gate constraints.

Gate types reflect the concept of computational interface templates. The signals, remote variables and remote procedures of the **in** gate constraint denote that an object supporting an interface of that type will accept these stimuli in a server role, whereas having the same elements in an **out** constraint denotes that the supporting objects uses them in a client role.

The following example shows a definition of an interface type in CORBA IDL and the corresponding definition of a gate type in SDL. The definition of the data types as well as the use of exceptions have been omitted to keep the example small. The interface specifies a simple broadcast service, where chat clients can register or unregister themselves and can broadcast messages to all registered chat clients. The derived interface NickBroadcaster adds two operations to change the nickname and to receive a description of the receiver object belonging to a nickname. Additionally an attribute has been introduced containing the number of currently registered chat clients. The example will also be used in the subsequent chapters to visualize the application of gate types and type based gate definitions.

```
interface Broadcast
{
void register(
          in Receiver rcvr,
          in string id,
          in in string nick);
void registerWCheck
          (in Receiver rcvr,
          in string id, in string host, in string nick);
void unregister(in Receiver rcvr);
StringSeq getRecvrNames();
oneway say(in string text);
};
interfaceNickBroadcast : Broadcast
{
readonly attribute long NoOfClients;
void setNickName(in Receiver rcvr,
          in string nick);
void getReceiverByNick(in string nick,
          out ReceiverDesc desc);
}
```

```
gate type Broadcast;
          remote procedure register
          fpar in rcvr Receiver, id, nick charstring;
          remote procedure registerWCheck
          fpar in rcvr Receiver,id, host, nick charstring;
          remote procedure getRecvrNames
          returns StringSeq;
          signal say (text) ;
in with register,registerWCheck,getRecvrNames,say;
endgate type;
gate type NickBroadcast;inherits Broadcast
adding
          remote NoOfClients integer;
          remote procedure setNickName
          fpar in rcvr Receiver, nick charstring;
          remote procedure getReceiverByNick
          fpar  in nick charstring;
               in/out desc ReceiverDesc;
in with setNickName,getReceiverByNick, NoOfClients;
endgate type;
```

3. Type Based Gate Definition

Gates may be defined directly or based on a gate type definition. Therefore the following changes have been made to in Z.100 Chapter 6.1.4:

<gate definition ::= <type based gate definition>
| <direct gate definition>

<direct gate definition> ::=**gate** <gate>
[**adding**] <gate constraint> <end>
[<gate constraint> <end>]

<type based gate definition> ::=**gate** <type based gate heading> <end>

<type based gate heading> ::=<gate name> [<number of gate instances>] : [**reverse**] <gate type expression>

A type based gate definition defines a set of gates. Several instances may exist at the same time. The initial number of type based gate instances which exist when the system is created, is zero. This is due to the fact that a gate instance is always related to a behaviour unit (i.e. its creator). However, to bootstrap the communication at most one type based gate definition of a <block type definition>, a <process type definition> or a <service type definition> may be preceded with the keyword **initial**. Number of instances represents the maximum number of simultaneous instances of the gate, if it is omitted the number of simultaneous instances is unbounded. The keyword **reverse** indicates that the **in** and **out** gate constraints of the gate instance set are in reverse direction to the constraints of the gate type.

4. GId sort

The GId sort is a specialization of the PId sort or syntype to PId. GId values denote unambiguously gate instances and will be referred to as gate references. GId expressions can be used at the same places as PId expressions.

5. Creation and Deletion of Gate instances

The creation and deletion of instances of type based gate definitions is part of the behaviour. A behaviour instance (i.e. process or service instance) must only create gate instances which are directly attached to itself or to one of its surrounding units (i.e. process or block). To create gate instances the create request has been changed as follows:

<create request> ::= **create** <create body>

<create body> ::= {{ process identifier> | **this**} [<actual parameters>]}
| <gate identifier>

The create action causes the creation of a new gate instance in the gate instance set denoted by <gate identifier>. The creating process **offspring** expression has the same unique new GId value as has been assigned to the new gate instance (i.e. offspring is a reference to the new gate instance).

If an attempt is made to create more gate instances than specified by the maximum number of instances in the type based gate definition, then no new instance is created, the offspring expression of the creating process has the value Null and interpretation continues. As soon as a new gate instance has been created, all signals, remote procedures and remote variables routed inward through that gate instance are further routed directly to its creator. Only the creator itself may use the gate instance for outward directed communication. (cf. Sect. 6.). Dynamically created gate instances must only be deleted by its creator. Deletion of a gate instance is done by a delete action:

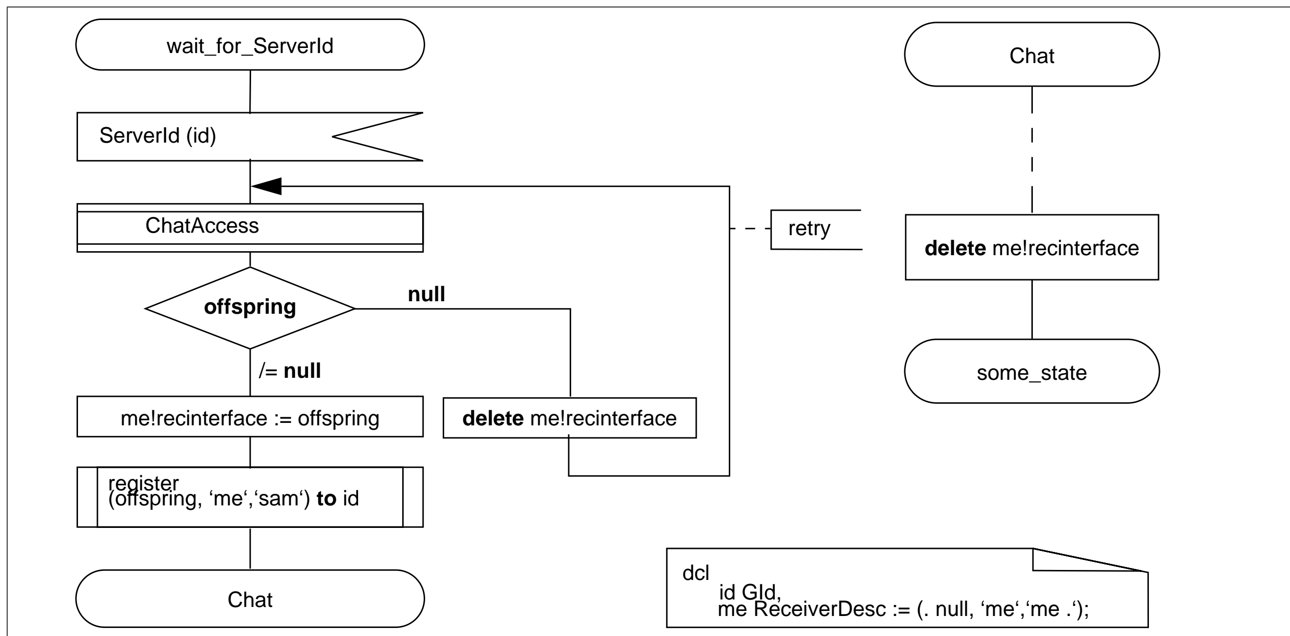
<action> ::= ...
| <delete action>

<delete action> ::= **delete** <GId expression>

A delete causes the immediate removal of the referenced gate instance. All further attempts to communicate via this gate instance lead to a discard of the signal resp. remote procedure. If a behaviour unit ceases to exist (i.e. it interprets a stop), all existing gate instances it has created cease to exist too.

The next example shows a fragment of the chat client process. After receiving a reference to a broad-

cast server the process creates a gate instance for chat communication and registers itself with this gate instance at the chat server. This leads to a binding between a gate of the broadcast server and the new created gate. After participating for some time in the chat session the process deletes its gate and leaves therefore the chat session



6. Communication using type based gates

Gates based on gate types can be used in <output>s, <import expression>s and <remote procedure call>s. This may be done using the <gate identifier> in a <via path> or using a reference to a gate in a <via path> or as <destination> (cf. Sect. 4.). This implies the following changes to the syntax:

```
<via path element> ::=   <signalroute identifier>
                         |<channel identifier>
                         |<gate identifier>
                         |<GId expression>
```

If a <GId expression> is used in a <destination> the <signal>, <remote procedure call> or the <remote variable> is delivered to the behaviour instance supporting the reference gate instance. Explicit or implicit channels and signalroutes will be used. If the gate instance does not exist, the <signal>, <remote procedure call> or the <remote variable> is discarded. The same holds in case that the gate type of the referenced gate does not contain the <signal>, <remote procedure call> or the <remote variable> in a <gate constraint> with direction **in**.

If a <GId expression> is mentioned as <via path element> it must reference a gate instance supported by the sending behaviour unit or by the receiving behaviour unit. If no **via all** is stated at most two <GId expression>s must occur within a <via path>. Where two such <GId expression>s are used one must reference a gate in the reverse direction of the other. If these conditions do not hold the <signal>, <remote procedure call> or the <remote variable> will be discarded. The same holds in case that the gate type of the referenced gate does not contain the <signal>, <remote procedure call> or the <remote variable> in a <gate constraint> with direction **out** in case of a sender gate. Otherwise the <signal>, <remote procedure call> or the <remote variable> will be delivered to the behaviour instance supporting the referenced receiver gate or will be transmitted via a communication path connected to the referenced sending gate (in case no receiving gate is mentioned).

If a <gate identifier> of a type based gate definition is mentioned explicitly in a <via path element> or if in the communication path to be used exists a <channel endpoint> or a <signalroute endpoint> denoting a type based gate definition, an arbitrary selection between the delivery of the <signal>, <remote procedure call> or the <remote variable> to one of the behaviour instances supporting a gate instance within the gate instance set <gate identifier> or the discarding of the <signal>, <remote procedure call> or the <remote variable> in case of an empty gate instance set will be done. The delivery can be made deterministic when additionally a <GId expression> is mentioned in the <via path> or as <destination> or a <PId expression> is used as <destination>.

If <destination> is given by a <PId expression> or a <GId expression> and a <GId expression> is mentioned in a <via path> the <destination> must identify that behaviour instance which supports the gate instance referenced in the <via path> or the gate instance must be supported by the sending behaviour instance.

The following additional semantics applies to the consumption of <signal>, <remote procedure call> or the <remote variable>: The **sender** expression of the receiving behaviour instance is given the GId value referencing the outgoing gate instance of the originating process in case such a gate was mentioned in the <via path> of the <output> <import expression> or the <remote procedure call> and the PId value of the originating process otherwise. These values will be implicitly assigned by the originating process and carried by the <signal>, <remote procedure call> or the <remote variable>.

7. Additional Concepts

The concepts explained below are not necessary for the introduction of gate types. However, together with the gate type concepts these concepts ease the modelling of open distributed systems using SDL.

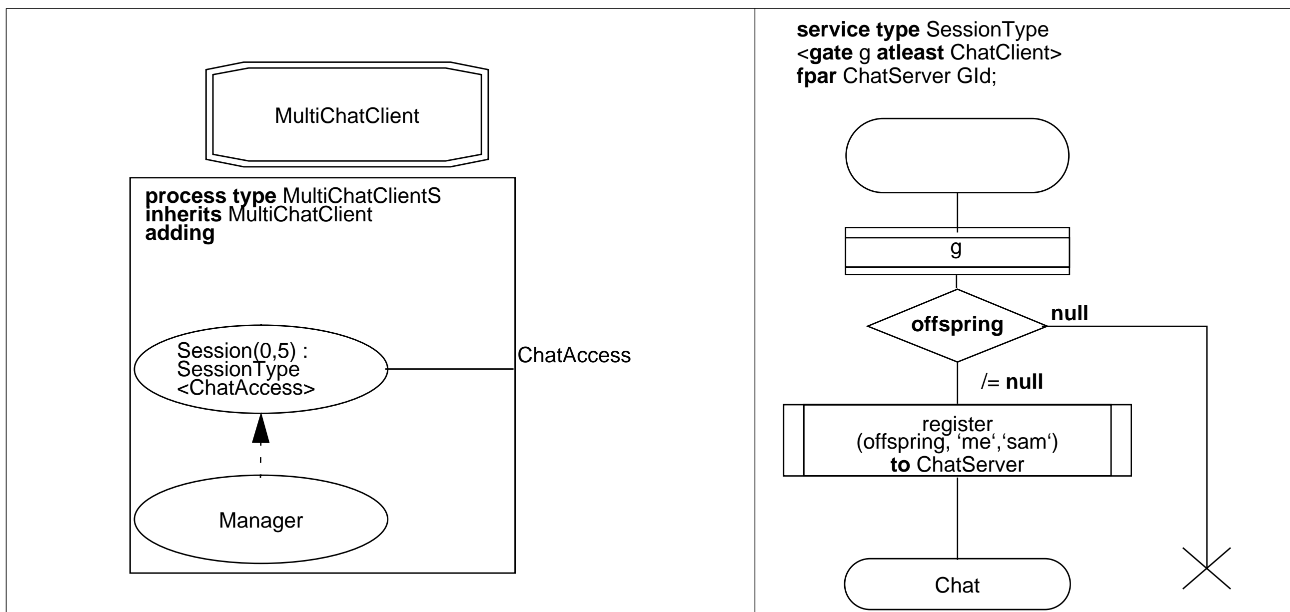
Dynamic Services

Services can be created dynamically similar to processes. The creation rules, the number of initial instances and the maximum number of instances follow semantically and syntactically the rules for processes. A service instance may only be created by another service instance of the same process. The newly created service instance executes its start transition when its creator reaches a state. At least one service instance set of a process must have an initial number of instances greater than zero. At most one of the services with an initial number of instances greater than zero may be preceded with the keyword **initial**. The complete valid input signal sets of two service instances within one process may overlap.

The concept of dynamic gates allows a server process to support multiple gate instances at the same time in an interleaved fashion. Although all services have their one state body, they do share the variables of the process. The figure shows a service decomposition for the MultiChatClient process type, where each single chat session is controlled by a separate service instance. The service instances are created on request by the Manager. In order to communicate with the Broadcaster the service instances do create in their start transition a new gate instance of type ChatClient.

Dynamic Blocks

Blocks can be created dynamically similar to processes. Again the creation rules, the number of initial instances and the maximum number of instances follow semantically and syntactically the rules for processes and at least one process instance set of a block must have an initial number of instances greater than zero. The **offspring** expression of the creator references an arbitrary initial process instance within the created block instance. The **parent** expressions of all initial process instances within the new block instance have the same PId value as the creating process **self**. At most one of the proc-



esses with an initial number of instances greater than zero may be preceded with the keyword **initial**. In this case the **offspring** expression of the creator references an arbitrary instance of the initial process.

The rules may be applied recursive in case of a dynamic block being decomposed into blocks. In this case at most one of the blocks with an initial number of instances greater than zero may be preceded with the keyword **initial**. A dynamic block instance ceases to exist if all its contained process instances or block instances cease to exist.

Initial Gate instances

At most one type based gate definition of a <block type definition>, a <process type definition> or a <service type definition> may be preceded with the keyword **initial**. This implies that after the creation of an instance of a type based definition of such a type implicitly also an instance of the initial gate will be created. The **offspring** expression of the creator as well as the **offspring** expression of the behaviour instance supporting the gate instance both have the same value referencing the initial gate instance. If the initial gate is not directly attached to a behaviour instance, the defining unit of the gate must contain an initial instance. This rule may be applied recursive until the initial instance is a behaviour instance. This instance implicitly creates the initial gate and supports it.

With the concept of dynamic blocks the creation of configurations of processes is possible. The example shows, how a relay broadcaster can be build by using the existing BroadcastServer.

Dynamic Binding

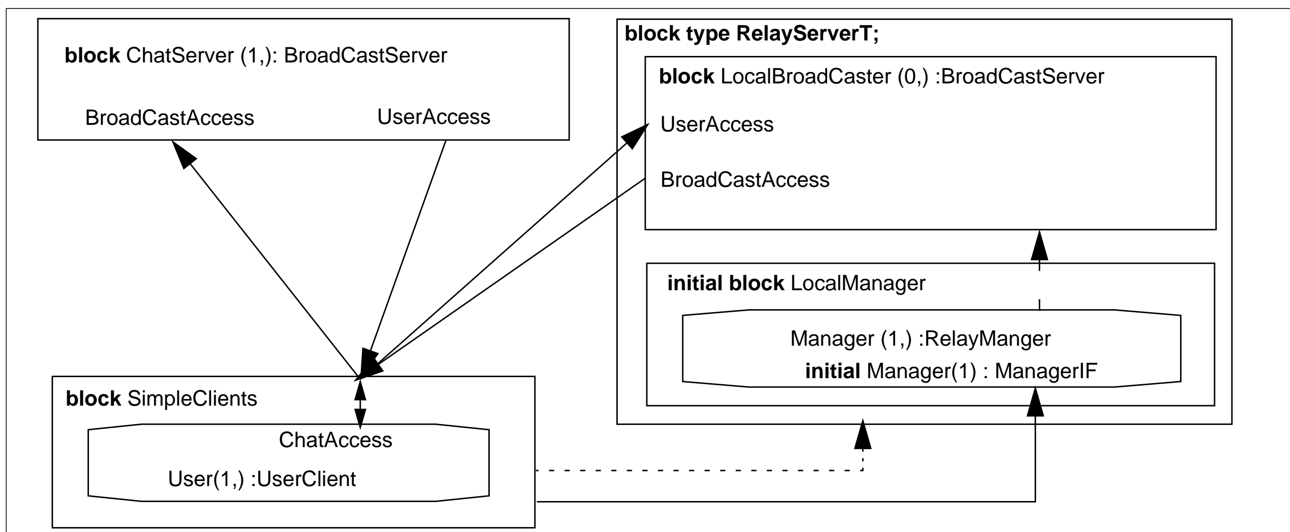
Dynamic binding allows the configuration of the communication infrastructure (channels, signalroutes) at interpretation time. The bind action sets up a communication path between two interfaces:

<imperative operator> ::= ...
|<bind expression>
|<unbind expression>

<bind expression> ::= **bind** (<GId expression> , <GId expression>)

<unbind expression> ::= **unbind** (GId expression [, <GId expression>])

Both expressions are of the predefined Boolean sort. The <bind expression> sets up a dynamic com-



munication path between the gates referenced by the two <GId expressions>. The following conditions apply:

1. both referenced gate instances must exist.
2. at least one of the referenced gate instances must be supported by the calling behaviour unit
3. the types of the two gates must be compatible, i.e. all <signal list items> within the **out** gate constraints of one gate type must be contained in the <signal list> of the **in** gate constraint of the respective other gate type and vice versa.

If these conditions are fulfilled, the <bind expression> yields the value true otherwise false. In the successful case a dynamic communication path is set up. As long as the conditions are fulfilled multiple gate instances can be bound to a gate instance. Unbind deletes the dynamic communication path between the two referenced gate instances. The caller of unbind must support at least one of the gates referenced in the unbind expression. If only one <GId expression> is given, all bindings to the referenced gate are deleted. Unbind yields true in case where the deletion is performed and otherwise false.

An <output>, <remote procedure call> or an <import expression> having a bound gate supported by the sending unit in its <via path> will always be transmitted to the behaviour unit which supports the gate instance the sending gate is bound to. If multiple gates are bound to the sending gate, the **via all** semantics applies.

8. Conclusion

The introduction of gate types and the dynamic creation of blocks and services enhance SDL for the computational specification of distributed systems.

The concept of gate types is currently under consideration at the ITU-T SG10/Q6. An extended version of this paper has been presented at the Q6 Expert meeting in Erlangen and at the Study Group Meeting in Geneva (March/April 1998).

We would especially like to thank Mr. Anders Olsen from Cinderella I/S / TeleDenmark, whose paper on gate types [14] was a starting point for our work, for the helpful and intensive discussions which have significantly contributed to this paper.

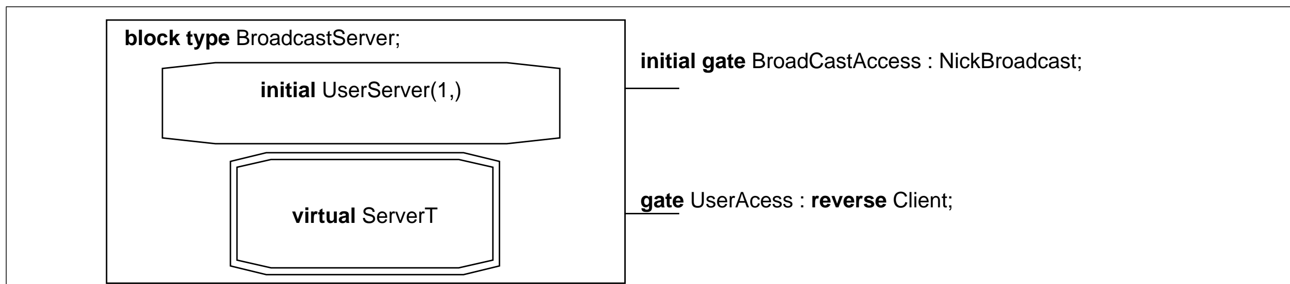
9. References

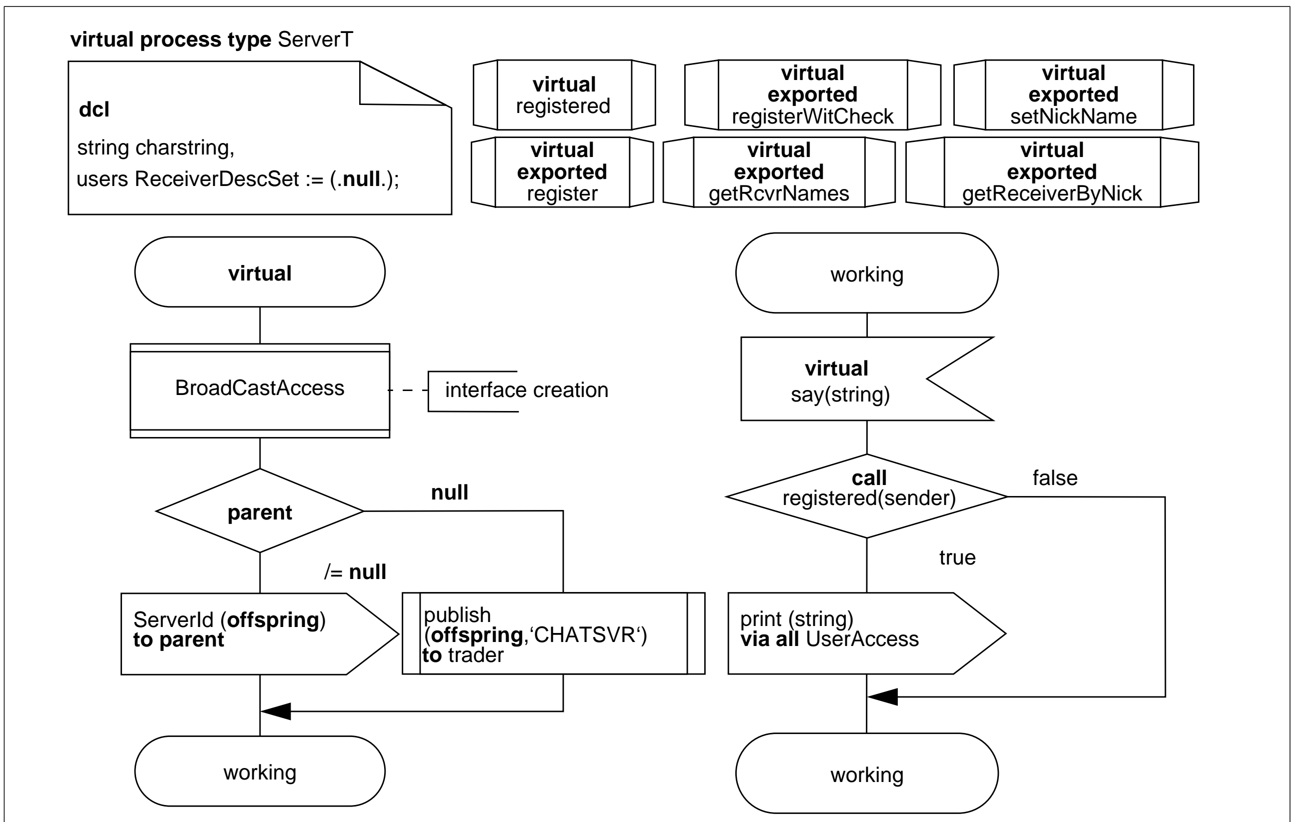
1 OMG: “The Common Object Request Broker Architecture CORBA 2.1”, OMG, 1997
2 ITU-T Rec. X.901 | ISO/IEC 10746-1: „Open Distributed Processing - Reference Model Part 1: Overview“, ITU-T ‘95
3 ITU-T Rec. X.902 | ISO/IEC 10746-2: „Open Distributed Processing - Reference Model Part 2: Descriptive Model“, ITU-T ‘95
4 ITU-T Rec. X.903 | ISO/IEC 10746-3: „Open Distributed Processing - Reference Model Part 3: Prescriptive Model“ ITU-T ‘95
5 ITU-T Rec. X.904 | ISO/IEC 10746-4: „Open Distributed Processing - Reference Model Part 2: Architectural Semantics“, ITU-T ‘95
6 ISO/IEC CD 14750„Open Distributed Processing - Interface Definition Language“, ISO/ITU-T DIS ‘96
7 ITU-T Rec. X.950 | ISO/IEC 13235: „Open Distributed Processing - Reference Model ODP Trading Function“ ITU-T ‘95
8 ITU-T Rec. Z.100: „SDL ITU-T Specification and Description Language“, ITU-T, 1992
9 Rational: „UML Semantics V1.1“, 1997
10 Rational: „UML Notation Guide V1.1“, 1997
11 TINA-C „Overall Concepts and Principles of TINA“, TINA-C 1994
12 TINA-C:“Object Definition Language - Manual Version 2.3“, TINA-C, 1996
13 TINA-C:“Computational Modeling Concepts“, TINA-C, 1996
14 A. Olsen:“Introduction of Gate Types in SDL„, ITU-T Temprary Document, Tele Danmark, 1997
15 E. Holz:“ODP Architectural Semantics in SDL“, ISO JTC1 SC21 Input Document, 1996
16 E. Holz: „Application of UML within the Scope of new Telecommunication Architectures“, GROOM Workshop on UML, Mannheim, Physica-Verlag, 1997

10. Annex

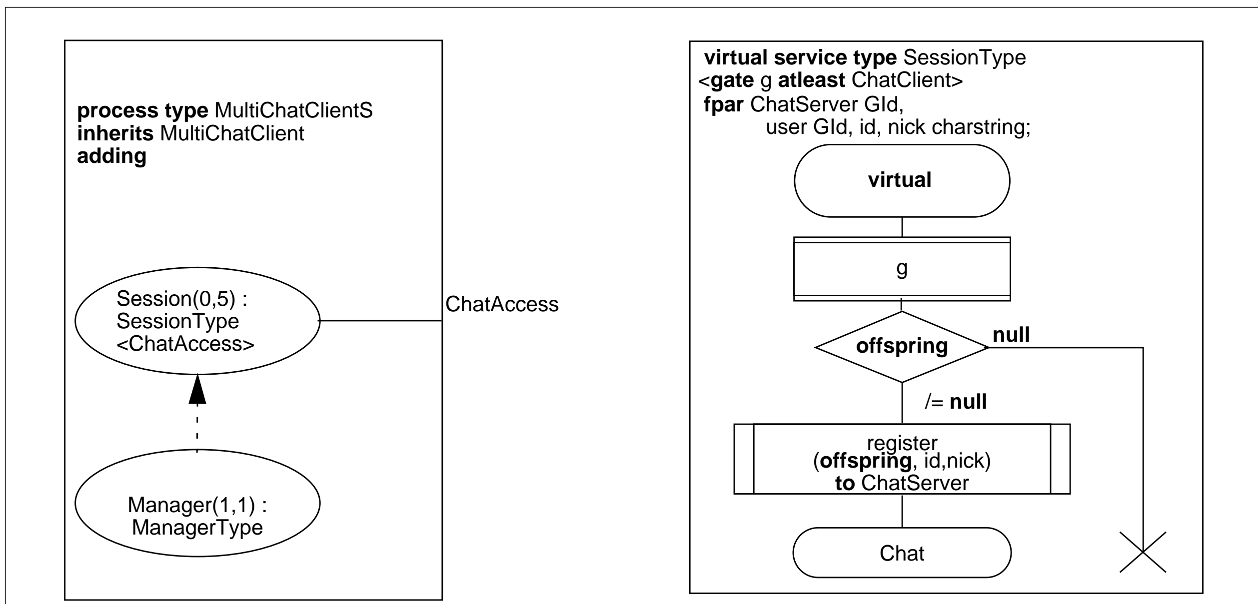
```
package ChatInterfaces;
gate type Broadcast;          /* Instances of this gate allow a user to register at a broadcaster, to get the list of
                                 registered users and to submitt messages to be broadcastet */
                remote procedure register fpar in rcvr Receiver, id, nick charstring;
                remote procedure registerWithCheck fpar in rcvr Receiver, id, host, nick charstring;
                remote procedure getRecvrNames returns StringSeq;
                signal say (charstring) ;
        in with register,registerWithCheck,getRecvrNames,say;
endgate type;
gate type NickBroadcast;      /* Users of instances of this type may use additional nicknames, ask for receivers by
                                 their nickname and  can obtain the number of registered clients */
        inherits Broadcast
        adding
                remote NoOfClients integer;
                remote procedure setNickName fpar in rcvr Receiver, nick charstring;
                remote procedure getReceiverByNick            fpar  in nick charstring;
                                                              in/out desc ReceiverDesc;
        in with setNickName,getReceiverByNick, NoOfClients;
endgate type;
gate type NickBroadcast2;     /* instances of this type will be used by to broadcast messages to its clients */
        inherits NickBroadcast
        adding
                out with <<gate type ChatClient >> print ;
endgate type;
gate type Client /* chat users can use this interface to receive messages */
                signal print(charstring);
        in with print;
endgate type;
gate type ChatClient /* chat users use this interface to contact the broadcaster and to receive messages */
        inherits Client;
                out with        register, registerWithCheck, getRecvrNames, say, setNickName,
                                getReceiverByNick, NoOfClients;
endgate type;
syntype Receiver = GId endsyntype;
newtype ReceiverDesc;
        struct  recinterface Receiver;                 /* references the gate instance of a client */
                id, nickname charstring;               /* name and nickname of the client */
endnewtype;
newtype
        ReceiverDescSet array(GId, ReceiverDesc)
endnewtype;
endpackage;
```

```
use ChatInterfaces;
use Infrastructure / gate type TraderInterface;
package ChatObjects;
```

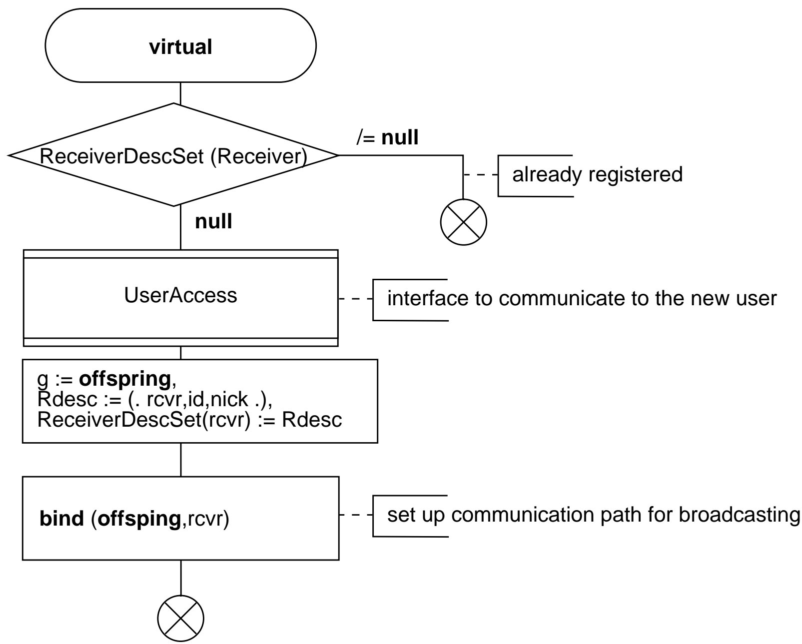




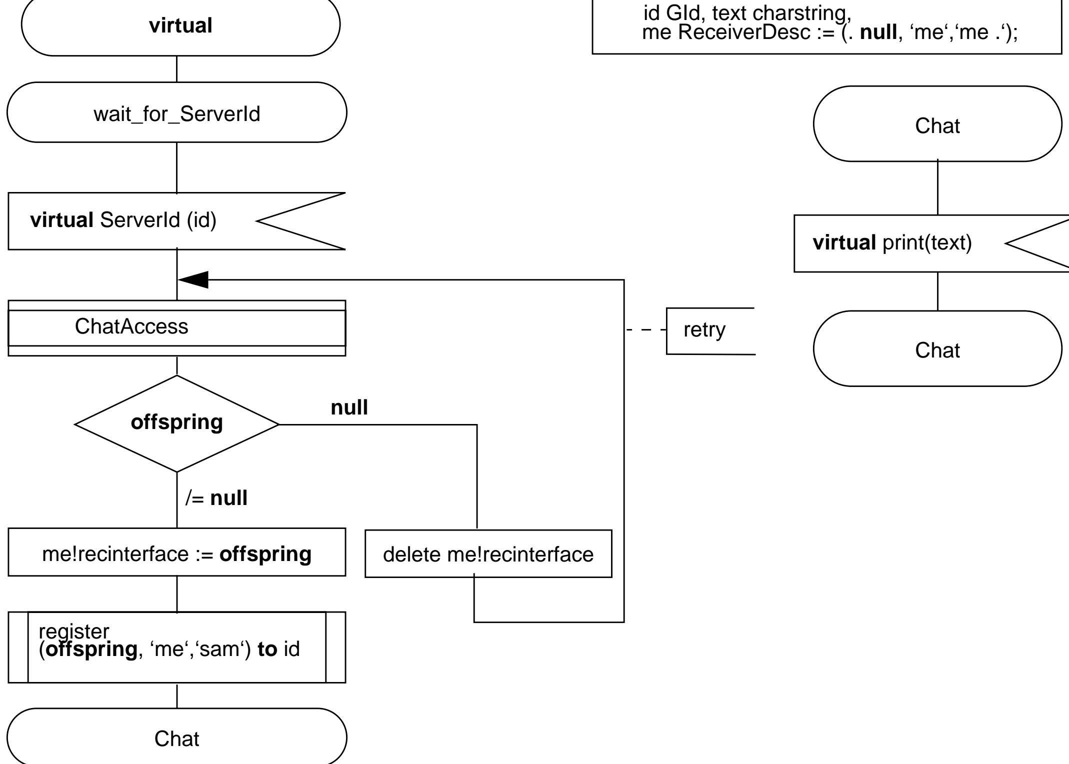
virtual process type Client;
gate ChatAccess (1) : ChatClient;
endprocess type;
virtual process type MultiChatClient;
gate ChatAccess : ChatClient;
endprocess type;



virtual exported procedure register /* other procedure definitions have been omitted */
fpar in rcvr Receiver, id, nick charstring;
dcl g GId, Rdesc ReceiverDesc := (. null, ‘ ‘,‘ ‘ .);



virtual process type Client;



endpackage;

use ChatObjects;
use ChatInterfaces;
use Infrastructure;