Using Object-Oriented Programming Techniques for Implementing ISDN Supplementary Services

Bent Gabelgaard,
DMT Dansk MobilTelefon I/S*,
Network Planning Division,
Skelagervej 1,
DK-9000 Aalborg C.

Abstract

This paper describes the use of the object-oriented programming language BETA for modelling and implementing a simple prototype of an ISDN D-channel layer 3 (I.451) entity. In particular, the implementation of ISDN supplementary services on top of the I.451 entity is considered using an object-oriented language, whereby several advantages are gained in comparison with other types of implementation languages. Aquaintance with the BETA programming language is assumed although the most vital constructs for use in this paper are explained.

Organization of the paper

The first section describes the background for the implementation project, i.e. the ISDN concept, the ISDN D-channel and the philosophy behind ISDN software.

The second section describes present-day implementation strategies for ISDN services and supplementary services and the problems encountered are highlighted.

The third and fourth section then describes the modelling and implementation work done in BETA. The reader is assumed to be reasonably familiar with the BETA programming language. For a detailed description of the BETA language and the inherent capabilities of the BETA system for code fragmentation and separate compilation, the reader is referred to [Madsen91].

Some familiarity with the OSI reference model for interconnection is desired, but not a necessity for understanding the issues discussed in this paper.

*Dansk MobilTelefon is a private consortium consisting of GN Store Nord A/S (DK), BellSouth Inc. (USA) and NordicTel AB (S). Dansk MobilTelefon is a carrier for the pan-European GSM digital mobile network in Denmark. DMT Dansk MobilTelefon is marketing its GSM service under the reg. trademark SONOFON.

1 ISDN

ISDN (Integrated Services Digital Network) is a network concept with three premium purposes:

- Creating an all-digital system for telephony.
- Integrating telephony and data transmission on a single network and in particular on a single subscriber access line.
- Allowing flexible introduction of new services and/or supplementary services. The latter could be Call Hold (CH), Call Forwarding Busy (CFB), Call Transfer (CT), CCBS (Call Completion to Busy Subscriber), Calling Line Identification Presentation (CLIP) etc.

A good introduction to ISDN in general is [Bocker87] although it emphasizes problems concerning the co-existence of ISDN and ordinary equipment for telephony and data (packet) transmission. GSM[1] may be regarded as the extention of ISDN into cellular mobile communication.

A characteristic of ISDN is the emphasis on services. Examples of ISDN services are telephony, packet-switched data transmission and image transmission (Fax-4). These services may be supplemented by supplementary services. A supplementary service may be regarded as an extension of the functionality of the basic service. The telephony service may be extended with a CLIP[2] supplementary service, for example.

ISDN supplementary services generally come in two types:

- on request (type 1)
- on provision (type 2)

The type 1 supplementary services are requested by an ISDN subscriber by signalling the ISDN local exchange (LE) from the user terminal equipment (TE) when the corresponding service is in use. See also figure 1.

The type 2 supplementary services are allocated statically, i.e. the supplementary service is activated automatically on each call to and/or from an ISDN subscriber.

An example (type 1): If the telephony service is in use by a subscriber, a supplementary service like CALL HOLD[3] is requested by sending a CALL HOLD-signal on the ISDN D-channel.

Another example (type 2): If an ISDN subscriber is subscribing to CLIP, he/she will automatically be notified about the calling party's ISDN-number on incoming calls.

If it is recognized by the LE that the requesting ISDN subscriber has access to the requested supplementary service, the logic for the supplementary service is executed in the LE. Virtually all logic for ISDN supplementary services is implemented in the LE, thus making the LE the most crucial network component in an ISDN.

[1]GSM is an abbreviation of Global System for Mobile communications (originally Groupe Spécial Mobile).

[2]CLIP=Calling Line Identification Presentation. This supplementary service makes it possible for a subscriber to identify the calling party's ISDN number before accepting a call.

[3]Setting a call on hold gives the subscriber the opportunity to answer another registered call and, if so desired, retrieve the held call later on.

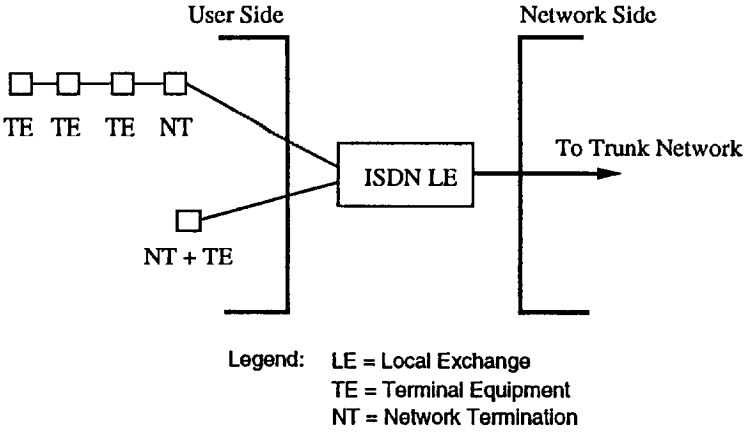

Figure 1: *The ISDN LE is a vital component in an ISDN.*

The ISDN TE is requesting ISDN services and supplementary services by signalling to the ISDN LE using the D-channel layer-3 protocol. The basic user access line to the ISDN LE is logically split into 3 channels, 2B + D, of which the two B-channels are meant for carrying traffic using any protocol the two ISDN subscribers can agree upon. The D-channel is a signalling channel with a standardized protocol profile. As such, the ISDN basic access is a common channel signalling system. See figure 2 for an overview of the D-channel protocol profile.

The ISDN services, supplementary services, the ISDN protocols etc. are all standardized by the CCITT[4] for making them comprehensible by all ISDN terminals.

2 Present-day implementation strategy

The main issue of this section is to show that present-day implementation strategies for implementing ISDN supplementary services create an integration between the D-channel layer-3 entity and the ISDN supplementary services. The integration found in a typical implementation makes it difficult to introduce new supplementary services without actually changing or duplicating a substantial portion of the existing code.

We will begin by considering the implementation of the D-channel layer 3 protocol block: The implementation module for the D-channel layer 3 protocol is considered an implementation of an I.451 state machine. This basic state machine is in the ISDN standards characterized by an SDL-diagram[5] describing the Basic Call Process (BCP). See [ETSI90] for the SDL diagrams describing the BCP. The implementation of these SDL-diagrams is referred to as the I.451 state machine.

[4]CCITT is an abbreviation of Comité Consultatif Internationale de Télégraphique et Télélephonique. CCITT is the organization under the UN responsible for recommending world standards for telecommunication purposes.

[5]SDL=Structured Description Language. An SDL diagram is a kind of flow chart. SDL is the preferred protocol description language within the CCITT.

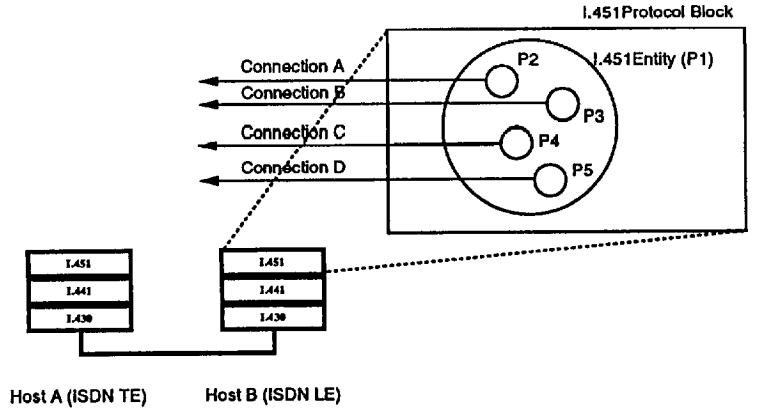

Figure 2: *The ISDN Basic Access D-channel protocol profile. The magnification of the I.451 Protocol Block shows individual processes (P1-P5) of which P1 is the entity itself and P2-P5 are processes denoting active connections.*

The implementation of the I.451 state machine is a process receiving I.451 signals from User Side as well as Network Side. See also figure 1. Each active call (connection) has a BCP attached to it, i.e. all calls are set up and taken down using the structure of the BCP. Depending on which signals have been received at a given point in time it is possible to conclude which signals are anticipated. This is basically the notion of a state machine.

Pay attention to the fact that in [ETSI90] all standardized states of the BCP are denoted by integers. For a Network Side entity the states are denoted Nx, where x is an integer. For instance, N0=null (=idle), N10=active etc. For a User Side entity the states are denoted Ux, where x is an integer.

Occasionally the ISDN implementor will introduce additional states to the BCP, i.e. states that are not standardized. A very common use of such states is inquiries in a subscriber database for checking the validity of the subscriber. The signalling necessary for getting out of such additional states is purely *internal* signalling in the exchange (if a request for process status is received, the i451process will answer the immediately following standardized state). Obviously such subscriber database inquiries will have to be made before any further call processing is initiated. Therefore the states denoting database inquiry are typically inserted in the normal BCP call processing just after leaving the initial (null) state. Additional states of this kind will of course have to be known by the implementor of the BCP.

A supplementary service is implemented as an extension to the I.451 state machine in a way that makes the new supplementary service available to subscribers. This means that supplementary services create an extension to the number of signals the D-channel layer 3 entity shall deal with. The I.451 state machine is conceptually extended with new states denoting the extra functionality. As always, the states are used for stating precisely what signals may be expected next.

Thus the implementation of the supplementary services will of course assume the existence of the BCP. Since they are logically independent however, the code for the BCP

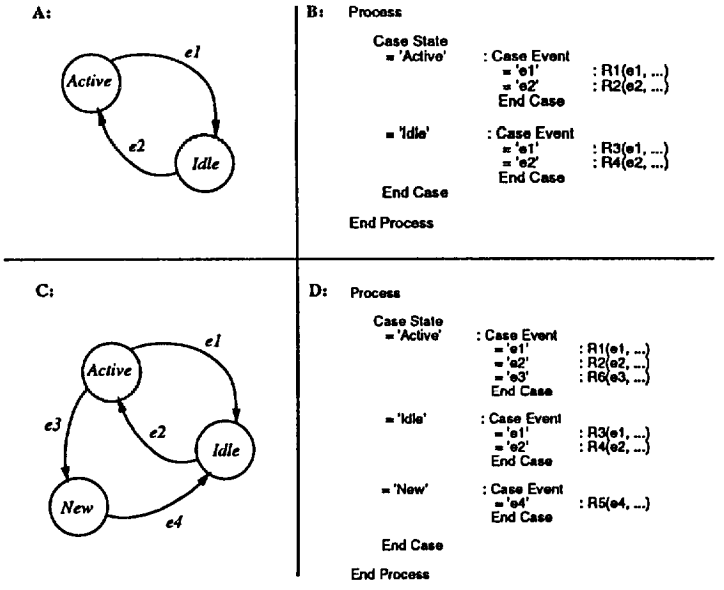

Figure 3: *The implementation of a state machine and an extension thereof.*

should not be mixed with the code for the supplementary services.

Consider the state machine of part A in figure 3. This could be the BCP; the implementation thereof is shown in part B. The extensions shown in part C may denote a supplementary service requested by sending the signal 'e3'; part D is the implementation of the extended state machine.

We see that for extending the state machine we need to be able to extend all CASE-statements in the table-driven implementation shown in part B in figure 3. Since the typical implementation languages are standard procedural languages such as C or Modula-2, the extentions are usually done by calling a procedure immediately after each CASE statement[6]. Such procedures, typically called software "hooks", will then implement a new CASE statement testing for the new signals that can be expected from the particular state.

To sum up, the basic construct for extending a program written in C, for example, is the procedure. Extensions are done by calling a procedure, possibly with parameters, and after the execution of the procedure body the execution of the main program is resumed from the point of call.

However, this procedure call scheme does not bring the kind of program extension that is important in our context. This is due to the following:

- All symbols in a C program must be fully specified before executable code can be

[6]SDL compilers for compilation from an SDL diagram to high-level source code are also available. Recent research in Petri Nets have integrated specification and executability in a formal description of the ISDN Basic Rate Interface [Huber91].

produced.

- The construction needed for extending code resembles a textual macro expansion substantially more than a procedure call.

- The philosophy in using procedures is creating code factorization but does not support the idea that the responsibility for development of the code *extensions* will be left for someone else.

During the discussion of our prototype implementation these issues will reappear.

Using C or Modula-2 or similar languages for implementation of supplementary services, extending a process through procedure calls inhibits the possibility of deferring the specification of the software hooks. As noted above, all symbols in a C program, for instance, must be fully specified before executable code can be produced. This in turn will have the effect that the procedures implementing the supplementary services will be fully integrated with the BCP code.

For the ISDN implementor this situation is unfortunate. It would be highly desirable to separate the code for the BCP and the ISDN supplementary services. Some of the reasons are outlined below:

- The BCP implementation ought to be completed and tested without having to pay attention to the extra complexity caused by the supplementary services.

- Having completed (and tested) the BCP implementation, it becomes possible to experiment with implementations of supplementary services without having to consider the functionality of the BCP.

- It is regarded by analysts that quite some part of the revenue of operating an ISDN will come from charging subscribers for the use of supplementary services. Therefore it becomes essential for the operating company to be able to experiment with and introduce new supplementary services quickly. The integration of the BCP and the supplementary services is an obstacle to this strategy.

- The network surveillance center should be able to down-line load code for new supplementary services by just down-line loading a service code module by itself. No down-line load of further copies of the BCP code should be necessary.

  Furthermore, it should be possible to execute the down-line load without interrupting traffic at a given network node. Whether this should be done by allowing multi-version execution [Hedin86] or not, this dynamic linking process is non-trivial.

  It should be remembered however, that a dynamic linking scheme is not only an application language issue. It is equally important that the operating system/run-time system on the node machine will support linking of fragmented code. For instance, the UNIX operating system supports fragmentation in terms of processes but not (at kernel level) objects.

Implementation engineers within the telecom industry are emphasizing the need for completion and test of the BCP process without any attention to the supplementary services. However, due to this demand present implementations quite commonly duplicate

a lot of code, i.e. a process with the ability to handle supplementary services will be a rewrite of the ordinary BCP. For comparison, the process shown in part D in figure 3 will not reuse the code from part B in a typical present-day implementation. Experience has shown that the programming languages and programming tools typically used in the telecom industry do *not* support extendibility or reuse of software to a widely influential degree.

Furthermore, experience show that this lack of support very often force implementation engineers to do the implementation of supplementary services in an inconsistent way. Some quite uncomplicated supplementary services are implemented by changing the source code of the BCP while other more sophisticated services are implemented by a process containing a duplication of the existing BCP code and the additional necessary code for the supplementary service. A general structure for adding code segments to existing code while preserving strong type checking during the development phase and not inhibiting code reuse is therefore sought by telecom engineers.

Before describing an object-oriented implementation of the scenario in figure 3, there are a few points that need to be emphasized.

First, any routine implementing a supplementary service will have to obey the general rule of leaving the I.451 state machine in a well-defined state. However, it is essential to grasp the idea that the code for a supplementary service not always leaves the state machine in exactly the same state as by entry. If the conditions for the call handling have been changed during execution of a routine, i.e. the I.451 process has received further signals from either User or Network Side, the state machine will have to specify what "suitable" state to return to under the present conditions. The definition of what is suitable is determined by what is thought to be a reasonably logical behaviour of the subscribers ISDN terminal. Thus a subroutine call is not - neither conceptually nor in terms of implementation - an elegant way of describing the execution of supplementary service code.

Second, supplementary services are not necessarily independent. We will return to this issue after discussing an implementation of an isolated supplementary service.

3 Implementation project using BETA

The main issue in this section is to show that the use of an object-oriented implementation language like BETA will - in comparison with conventional languages - give the ISDN implementor a better tool for implementing ISDN supplementary services by supporting a natural separation between the BCP code and the code for the supplementary services.

The prototype implementation is simple in the sense that it does not contain the full functionality of a "real-life" I.451 entity. Any details not essential for doing a feasibility study on separating the I.451 BCP and the ISDN supplementary services have been omitted. This includes timer control, non-normal call setup and clearing etc.

3.1 Object-oriented design and the I.451 BCP

When doing an object-oriented design, it is essential to create a model of the real-world scenery. For this reason one would expect to see concepts like I.451 Entity, I.451 Basic

Call Process, I.441 Service Access Point etc. in the sense described in the ISDN standards being modelled in the implementation.

Basically, the I.451 Entity is modelled as a class with the I.451 Basic Call Process as an attribute. There are two important things to consider about the I.451 BCP attribute:

- It should be possible to model the fact that an I.451 BCP is an independent thread within an I.451 Entity. It should be possible *dynamically* to create multiple instances of the I.451 BCP within a single instance of the I.451 Entity for modelling the fact that multiple calls can be handled simultaneously by a single I.451 Entity.

  Thereby we see a strong need for the possibility of having parallel or quasi-parallel execution *within* an object. Language support for this kind of multi-sequential execution becomes a must in this context.

- The I.451 BCP is a process that may be extended to handle ISDN supplementary services. The supplementary services supported by a particular ISDN network node may be different from other ISDN network nodes. Therefore the specification of the I.451 process to handle calls in a particular ISDN node is not completely known at the time of designing the I.451 BCP.

  This is modelled in BETA as declaring the I.451 BCP *virtual*. The virtual mechanism in BETA is a very strong tool for modelling an attribute that may be extended in a specialization of the surrounding class. A virtual attribute may be extended in specializations of the surrounding class in order to include new aspects of the concept modelled by the class.

BETA is a language that provides support for the basic features mentioned above. On top of this, the syntax for dynamically generating ordinary objects or co-routine objects (for quasi-parallel execution) is highly uniform.

One aspect of this is particularly important for our purpose. Consider a virtual attribute modelling the I.451 BCP. This attribute may be used for generating co-routines at run-time in exactly the same manner as when generating new static objects at run-time. The BETA model of the I.451 Entity class has the structure shown in figure 4.

This combination of mechanisms - the virtual mechanism and language support for multi-sequential execution within an object - provides the very basis for creating a good model of the real-world facts that:

- The I.451 BCP may be extended for handling ISDN supplementary services.
- I.451 BCP (and extensions) may be generated at run-time, i.e. each new call will force the generation of a new process.

To the best of my knowledge, an accurate modelling of this scenery cannot be accomplished in any but a few object-oriented languages. BETA is one such example - the programming language Emerald is an interesting alternative.

One very important issue should be emphasized; by modelling the I.451 BCP as virtual and thereby allowing for future extensions, the very problem of separating the I.451 BCP and the ISDN supplementary services has found an adequate solution. We will return to this when demonstrating how to incorporate ISDN supplementary services in the model of the I.451 BCP structurally shown in figure 3.

```
i451Entity : Process
    (#
       i451Process : < Process (# .. #);      (*i451Process declared virtual*)

       newProcess : < (# .. #);               (*procedure for creating and initializing
                                                 new instances of the virtual i451Process*)

       ENTER i451Signal                       (*i451Signal is input-parameter*)

       DO                                     (*action part*)
           (IF <connection exists>
             //TRUE THEN <pass on i451Signal to active connection>
             //FALSE THEN <generate new i451Process>
            IF);
    #);
```

Figure 4: *The structure of the BETA model of I.451 Entity class. The prefix-notation "i451Entity: Process" specifies that Process is the prefix (superclass) of i451Entity.*

A: (# <declarations> DO I_1 ; .. ; I_j ; INNER; I_m; .. ; I_n #);

B: A (# <declarations> DO I_k ; .. ; I_l #);

Figure 5: *Typical use of the BETA Inner-construct.*

3.2 Extending the BCP

BETA includes a language construct for specializing the action part of a class (in the literature an action part of a class is often referred to as support for active objects). The BETA construct is called "Inner" and is typically used as shown in figure 5.

When creating objects of class B in figure 5, the Inner-imperative present in the description of class A (the superclass of B) will cause the action-part of B-objects to be the imperatives $I_1..I_j; I_k..I_l; I_m..I_n$. Thus, the Inner construct is a macro for textual substitution of action parts.

As noted earlier, incorporating supplementary services is done by extending the I.451 BCP. A state machine can be extended in two ways - by extending the number of states and extending the number of relations between states. In any way we would like to preserve the structure of the BCP shown in part B of figure 3. As in an implementation using a procedural language we need to be able to extend all CASE statements in the state machine implementation. Thus we need several different Inner's. The structure of the extendible state machine is then similar to the one shown in figure 6.

The reader may find it worthwhile to compare this kind of extendibility with a procedure call. The latter seems unnatural because we will never need any parameter passing. All what is needed is a kind of macro expansion of a code segment. Software hooks using the Inner construct truly allow deferring the specification of the code extensions.

BETA does not contain separate constructs for several different Inner's, but this pro-

```
Process

  CASE State
    = "Active"     : CASE Event
                       = x -> R1(x, ...)
                       = y -> R2(y, ...)
                       = z -> R3(z, ...)
                     END CASE;
                     InnerEventActive

    = "Idle"       : CASE Event
                       = x -> R4(x, ...)
                       = y -> R5(y, ...)
                       = z -> R6(z, ...)
                     END CASE;
                     InnerEventIdle;

  END CASE;
  InnerState;

End Process
```

Figure 6: *The structure of a BETA-implementation of an extendible state machine. All InnerXX are virtuals.*

```
A:
 (# inner1: < (# DO l1 ; .. ; lj ; INNER; lm; .. ; ln #);

    DO inner1
 #);

B: A
 (# inner1: : < (# DO lk ; .. ;l| #); #);
```

Figure 7: *Using a virtual binding and Inner.*

blem can be circumvented by exploiting the fact that the Inner construct is a special kind of a virtual binding. The functionality shown in figure 5 may be achieved by creating virtuals as shown in figure 7. The point to be made about figure 7 is that it becomes possible to extend the construction into having several different Inner's by having virtuals Inner1, Inner2, ... InnerN.

Designing extendibility in the way shown in figure 6 does put a constraint on the kind of supplementary services that can be supported. For instance, it would not be possible to support a hypothetical service that would actually *change* the way the BCP or another supplementary service behaves. In practice this constraint is not a problem at all for the ISDN implementor because the situation will never occur. The reason being that the SDL specification of the BCP and the supplementary services is standardized.

Incorporating an isolated supplementary service now becomes a relatively easy task, as we will show in the next section.

3.3 Incorporating supplementary services

We will take a closer look at three ISDN telephony supplementary services:

- Call Hold (CH)

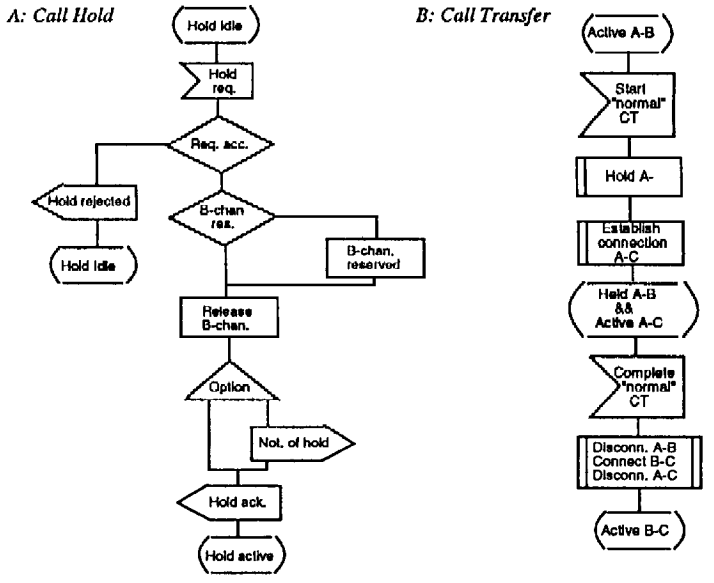

Figure 8: *The overall SDL specification of Call Hold (Hold operation - diagram A) and the overall SDL specification of Call Transfer (diagram B).*

- Call Waiting (CW)
- Call Transfer (CT)

The supplementary services are standardized using SDL diagrams accompanied by explanatory text [I.250][7]. Supplementary services may imply an extension of the I.451 state machine, in terms of signals as well as the number of states.

Do note that we in our system have named the BCP "i451process". This is due to the fact that we are going to implement multi-level extensions and thereby may extend a process that is more than just a model of the BCP.

3.3.1 Call Hold

Let us start out by considering the CH supplementary service. This has been standardized by the overall SDL diagram in figure 8.

CH allows a user to interrupt an existing connection and, if so desired, re-establish (retrieve) this connection.

[7]The specification of an ISDN supplementary service is actually divided into three parts: A stage 1 description which is defining the overall service description, seen from the user's standpoint. A stage 2 description defining the information flows needed to support the stage 1 description and finally a stage 3 description defining the signalling system protocols and switching functions needed to support the stage 1 description. The stage 3 descriptions of CLIP, CLIR and CW can be found in [ETSI91]. Stage 3 descriptions of CT and CCBS are not yet finalized.

```
callHoldEntity: I451Entity
  (#
      I451process : : <
          (#   innerState : : <
                   (# innerEvent501 : < (# DO inner #);

                       DO
                           (if currentState
                           // 501                          (*state 501 = holdActive*)
                               then (if s.signal            (*signal 51 = call retrieve*)
                                        // 51 then s[ ] -> &callRetrieve -> currentState
                                        if);
                                        innerEvent501
                           if);
                           inner
                    #);    (* end innerState *)

                  innerEvent10 : : <            (* state 10 = active *)
                       (# DO
                            (if s.signal         (* signal 50 = call hold *)
                             // 50 then s[ ] -> &callHold -> currentState
                            if);
                            inner;
                       #);  (* end innerEvent10 *)

                #);  (* end further binding of i451process*)

      #);  (* end callHoldEntity *)
```

Figure 9: *The BETA-implementation of Call Hold.*

The point to be made about CH is that it is relatively simple. CH extends the BCP with one state (holdActive) and two signals (hold, retrieve). There are no other connections involved when activating CH than the one to be held and/or retrieved.

The extension of the BCP (in the implementation called i451process) in BETA is shown in figure 9. Let us take a closer look at this code.

The code in figure 9 is in the prototype implementation separated completely from any other code fragments for the BCP, the multiprocess-scheduler running the co-routines etc. We see that the code can be developed without changing the BCP-code. The code design can shortly be described as follows: In state 10 (active) the i451process should be able to handle one more signal (hold). This is accomplished by further binding of the virtual 'innerEvent10', which is the event-handler for state 10. The callHold routine sets the connection in a new state (holdActive) which we have chosen arbitrarily to give the number 501. Therefore the virtual 'innerState' is extended with state 501 and the "intelligence" about what to do when receiving signals in state 501 (holdActive). The 'innerState' extention includes the declaration of a new event-handler, for the ability to extend the extension. The callHold routine which contains the actual code for the SDL-diagram shown in figure 8, is separately compiled in another code fragment and linked into the code shown in figure 9.

Do note that it is the supplementary service code developer who controls the extensions of the i451process. The code developer of the BCP can rest assured that his/her code is "complete" in the sense that no changes need to be made *whatever* supplementary services may be implemented afterwards (the superclasses may be used in any conceptually sound context).

```
callTransferEntity: callHoldEntity
    (#
       i451process : : <
            (#   innerState : : <
                    (# innerEvent801 : < (# DO Inner #);
                       innerEvent802 : < (# DO Inner #);

                       DO
                          (if currentState              (* state 801 = held AB&&activeAC*)
                           // 801
                              then (if s.signal         (* signal 81 = complete call transfer*)
                                    // 81 then s[ ] -> &completeCallTransfer -> currentState
                                    if);
                                    innerEvent801
                           // 802
                              then (if s.signal         (* signal 51 = call retrieve, signal 82 = new attempt to complete*)
                                    // 51 then s[ ] -> &callRetrieve -> currentState
                                    // 82 then s[ ] -> &completeCallTransfer -> currentState
                                    if)
                           if);
                           Inner
                     #);   (* end innerState *)

                     innerEvent10 : : <          (* state 10 = active *)
                        (# DO
                             (if s.signal         (* signal 80 = call transfer *)
                              // 80 then s[ ] -> &callTransfer -> currentState
                              if);
                              Inner;
                         #);  (* end innerEvent10 *)

                  #);  (* end further binding of i451process*)

          #);  (* end callTransferEntity *)
```

Figure 10: *The BETA-implementation of Call Transfer.*

3.3.2 Call Transfer

The CT supplementary service is more complicated than CH. The reason being that CT involves multiple connections. Since all connection setup and control should be carried out by the I.451 Entity, an instance of i451process must have some way of communicating and/or synchronizing with the surrounding entity.

CT enables a user to transfer an established call to a third party. That is, if A and B are subscribers with a presently active connection and A is subscribing to the CT service, it is possible for A to transfer the connection to C, resulting in an active connection between B and C.

The structure of the extentions is the same as shown in figure 9, though. CT extends the BCP with two states and three signals, *as well as* making use of CH. The overall SDL-diagram for CT looks like part B in figure 8. In this diagram one of the CT states is not shown.

In figure 8 only the "normal" call transfer has been included; there are two other ways to do a call transfer - single step and explicit call transfer - that have not been considered. The exclusion of these is without loss of generality.

As for CH, we see in figure 10 that the creation of the CT service can be implemented without any concern for the BCP implementation. Again, the virtual event-handler for state 10 (active) is further bound in order to include handling of the call transfer-signal. The virtual 'innerState' is further bound for defining the code describing what signals to expect in the CT-related states of the i451process.

The most important thing for the service code developer remains the fact that it is

possible to design the code for new services (almost) independently of the superclasses (abstractions) designed for handling the basic functionality, e.g. the I.451 BCP.

3.3.3 Call Waiting

CW enables a subscriber to have a queue of calls waiting at the ISDN terminal. The subscriber will be notified each time the queue is updated. When combined with CLIP and CH, the subscriber is able to intelligently select the calls that he/she wants to accept in a busy situation.

CW is relatively uncomplicated to implement. In fact, the CW is an example of a supplementary service that is simple enough for an ISDN implementor to be tempted to implement it directly by rewriting the source code for the BCP. The object-oriented approach taken in this paper allows us to use a general structure for code extension, yet obtaining an efficient solution. The CW service has been implemented by specializing the BCP in much the same manner as was done for the CH supplementary service.

For an implementation of CH *and* CW at a given ISDN network node, we would then need to inherit from both the CH- and the CW-extensions but only one copy of the common superclass (the BCP code). Multiple inheritance is presently not supported in BETA, however. In general, we realize a need for a multiple inheritance construct if the conceptual structure devised in this paper for implementing ISDN supplementary services should not be jeopardized.

3.3.4 Calling Line Identification Presentation

CH, CT and CW are all type 1 supplementary services, i.e. they are explicitly requested by an ISDN subscriber. CLIP is a type 2 supplementary service available on provision.

An incoming call from an ISDN subscriber (the A-subscriber) will be presented on the receiving ISDN subscriber's (the B-subscriber) terminal with the A-number displayed, provided the B-subscriber is subscribing to CLIP.

The check for CLIP subscription is merely a simple inquiry in the subscriber database in the B-subscriber's ISDN LE. In general, inquiries in the subscriber database are not part of the standardized SDL description of the BCP [ETSI90]. As mentioned earlier, such inquiries are necessary for other vital purposes. Therefore the i451process will contain a virtual routine "checkSubscriberData", which obviously may be extended in order to check for CLIP subscription.

3.4 Comments on the modelling of the entity

The callTransferEntity is in our implementation modelled as a specialization of the callHoldEntity. In other words we claim that the entity capable of doing Call Transfer, is a special kind of an entity capable of delivering a Call Hold supplementary service.

This claim can be justified if holding a call (i.e. putting it in a suspended state) is given the meaning that it is suspended forever. The transferred call is then regarded as a new connection. Furthermore, the standardization of the Call Transfer supplementary service presupposes the existence of the Call Hold supplementary service. Even so, a better way to look at things would perhaps be to view the Call Hold- as well as the Call Transfer supplementary service as aggregate components of the I.451 entity.

The inheritance hierarchy used in our prototype system is showed in figure 11.

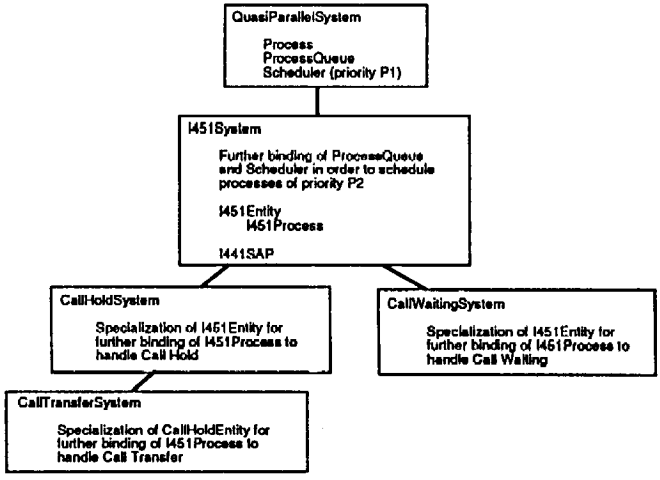

Figure 11: *The inheritance-relations found in the prototype simulation system. Thick lines are denoting the inheritance relations. Containment is shown by boxes.*

3.5 Comments on the implementation

We stated earlier that complications may arise when two supplementary services are not independent. The invocation and execution of CT and CH services are not independent as can be seen in the ISDN standards [I.250]. For example, if subscriber B places an active connection with subscriber A on hold during subscriber A's transfer of the same connection to subscriber C, the resulting connection between subscribers B and C shall remain held by subscriber B until it is retrieved by subscriber B. Another example: If an ISDN subscriber is engaged in a conference call, it should be determined whether the subscriber is able to receive a call waiting notification from a subscriber who is outside the conference.

Such dependencies must be dealt with explicitly by the SDL specifications of the supplementary services. Otherwise the resulting state of the i451process after a specific signal permutation may be undefined or deadlocked. In theory, this could bring an entire ISDN network to a screeching halt.

There are other situations where the ISDN standards will show an implicit prioritization of the supplementary services, e.g. if a call setup is arriving at an ISDN LE and the user for whom the call is determined is subscribing to CFU[8] as well as CFB[9], the CFU takes precedence over CFB.

In practice, the dependencies between supplementary services do not present any problems for the ISDN software implementation. On the other hand, the supplementary service specification is made substantially more difficult.

In implementation terms, the example outlined above with CFB and CFU simply translates into a test for whether the user is subscribing to the services in question and then

[8]CFU=Call Forwarding Unconditional.

[9]CFB=Call Forwarding Busy.

invoke the one that takes precedence. In our prototype implementation all "subscribers" are for simplicity granted access to all supplementary services. Therefore the code - without lack of generality - does not contain any tests for service availability for an individual "subscriber".

Another limitation of our implementation is that only one connection may exist between any two subscribers. It is perfectly possible though, for one subscriber to have multiple active connections.

One other mentionable detail is the uniformity of the evaluations in the i451process. All routines must leave the i451process in a well-defined state. Due to this, the i451process contains evaluations like the following only:

$$s[\,] \rightarrow \&proc \rightarrow currentState$$

This reads as: The signal-reference 's' is given as input to the procedure-pattern 'proc'. The output from this procedure is given as input to 'currentState'.

Readers familiar with the content of [ETSI90] will know that input to the I.451 Network Side Protocol Block (the entity) is denoted differently whether it is I.451 protocol input (input from user side) or input originating from SS7 protocol signals (input from network side). In our prototype implementation, all signals look alike.

The user of the prototype will have to be aware of which connections are active at a given moment. The implementation of the I.451 entity therefore contains a reasonably detailed printout mechanism for giving an overview of active connections.

4 Further aspects of object-orientation

The reader may have noted that any implementation of an I.451 entity, using object-oriented techniques or not, will suffer when a machine on which the implementation of the entity is executing goes down. The prototype implementation described in this paper does not attempt to do anything to compensate for unwanted effects of such a machine failure.

However, the object-oriented paradigm contains the notion of persistent objects that may be applicable in this context. Persistence may be defined as the ability to share data among several program executions [Agesen89] and the object encapsulation is creating a reasonable unit for persistence.

Consider the possibility of making the "currentState" in each BCP a persistent object in the sense that all changes in the state of object "currentState" will be reflected in a mirror object stored on permanent storage. This is essentially the functionality of the checkpoint-construct in the programming language Emerald.

It can be argued that persistence is not wanted in this context. A successful restore of all active connections registered by the time of a machine failure, will possibly result in a lot of subscribers regaining open connections. By the time the restore is complete, however, most people will probably have given up their connections (they have "hung up"). A better way would be to accept the restore only if it can be done within a few seconds - otherwise it makes more sense to see to that all resources are halted gracefully in order to make new connections available as quickly as possible. Persistence will in this respect have limited applicability.

For charging purposes, however, the situation may be a little different. In theory, it would be possible to protect a subscriber's account when a node is going down by charging only up to the last (periodically activated) checkpoint.

5 Perspective

In recent years the concept of IN (Intelligent Network) has gained significant interest in the telecom network community. The main idea behind an IN architecture is that the service execution logic (the information services in IN terminology) is strictly separated from the network protocols. The flexibility gained by doing so is perhaps best illustrated by the very interest in the concept of IN. Large-scale distributed databases and data processing will probably emerge as one of the main trends in commercial IN implementations. Very fast service deployment will also be a major feature of the IN. Recent experience in Scandinavia points towards a few hours for deploying simple IN services country-wide. In comparison, network operators operating a conventional, digital PSTN[10] or even an ISDN face a deployment time span of several months, if not years, for implementing new supplementary services.

The IN concepts will be standardized according to the pending CCITT Q.1200-series standards. The very interesting point about IN is that the IN "Basic Call Model" (BCM) has a structure that is almost identical to the structure devised in this paper for the ISDN BCP and extensions.

The semantics of the IN BCM components is different however, than the ISDN BCP and extensions. The latter, i.e. in this paper the "InnerXX" virtuals, are meant for describing code executing on the same physical exchange. In the IN BCM the components that structurally correspond to the extensions of the ISDN BCP (i.e. the IN "triggering points") are meant for identifying where, in the processing of a call, service execution logic possibly residing on *another machine* may receive notification (stimulus) in order to influence subsequent processing of the call.

To sum up - although developed for an ISDN-related feasibility study, the structuring techniques discussed in this paper are in spirit very similar to the IN code concepts. The advantages that have been identified will certainly hold true for implementation of IN software as well. In fact, they will probably be of even greater value in an IN software architecture.

See [Söderberg88] for a good introduction to the concept of IN. A more recent publication is [Berman92].

When considering future standards on Distributed Processing, i.e. ODP[11], the focus is on creating transparency in software, e.g. service code location transparency. This is a very similar problem to that found when describing IN architecture and service logic because of the strict separation between service logic and network protocols. The object paradigm is very well suited for use in an ODP context because of the message passing concept, i.e. message passing is conceptually the same mechanism whether objects are located in the same address space or not. See for instance [Blair91] for a good introduction to ODP.

The intricate links between IN software, distributed processing, operating system and/or hardware capabilities for large-scale process management will be a highly interest-

[10]PSTN=Public Switched Telephone Network.

[11]ODP is an ISO acronym for Open Distributed Processing.

ing field of study for telecom engineers in the years to come. Object-oriented programming techniques as presented in this paper may be a good way to proceed.

6 Conclusion

It seems that a powerful object-oriented programming language is presenting an alternative to conventional implementation languages, that has certain advantages in the context of ISDN and ISDN service software. The advantages are numerous, but have mostly to do with the basic notions of modelling, instantiation of code and extending existing code rather than changing existing code. These aspects are true for most software designs using the notion of object/class and inheritance.

The prototype implementation of the I.451 system is done using BETA, which is an especially powerful language with respect to multi-sequential execution in an object-oriented context. However, an equivalent implementation would certainly be possible using C++ or another object-oriented language - the support for certain structures already found in BETA (e.g. multi-sequential execution) would then be left to the programmer as an additional task, though.

Critics may point out that the Inner-construct for combining action parts can just as well be achieved by calling C-routines (if C is the implementation language) in the sense that all invocations of the virtuals innerXX in figure 6 should be replaced by a call to a C-routine innerXX. We have argued that the C alternative is not as elegant a solution as the object-oriented approach demonstrated in this paper.

One of the reasons to the inferiority of the C alternative is that all symbols in a C-program must be fully specified before linked, i.e. executable code can be produced. The same statement holds for any other procedural language. Using the BETA-system, it is perfectly possible to execute code with an unbound Inner-imperative. However, if a network operator would like to link new code into existing, *executing* code, this kind of dynamic linking remains a problem no matter if the code is designed using object-oriented techniques or not. See [Hedin86] and [Hedin88] for an introduction to incremental compilation and linking for procedural languages and object-oriented languages, respectively.

In spite of the problems related to dynamic linking, the separation (factorization) of code which is necessary for building large software systems, is in my opinion supported to a substantially greater extent in object-oriented languages than procedural languages. The Inner construct found in BETA is but one way of combining action parts in a more natural way than through standard procedure calls.

Similar features like virtuals and the Inner-construct can certainly be programmed in C or in assembler (and it would be rather simple, too) but the point is that when a certain programming philosophy is not explicitly supported in a given language, it becomes doubtful whether the philosophy will be used at all. If used, it will probably not be used in a consistent way by the large number of programmers involved in a large-scale software project such as an implementation of ISDN software.

For use in a telecom environment where large code segments are down-line loaded from a central node to the traffic-carrying nodes, the code factorization achieved using an object-oriented language as demonstrated in this paper seems like a very competitive solution.

Finally, it should be emphasized that the structure for extending a process as presented

in this paper seems general in the sense that it is feasible to implement any of the presently standardized ISDN supplementary services, including those that have not been discussed in this paper. The project has not revealed any evidence that this statement does not hold true.

7 Comparison with previous work

Not much work in the field of object-oriented design on network concepts has been published. Only very few of these are dealing specifically with ISDN. Two articles can be mentioned, however: [Arnouat91] and [Maruyama91].

The former, [Arnouat91], is not recommendable because the concepts of ISDN services and supplementary services are slightly misinterpreted, unfortunately. Among other issues the paper mentions that supplementary services are independent of the services. It is true that [I.250] lists a number of supplementary services irrespective of its application. However, the same standard also specifies the applicable associations between services and supplementary services. The modelling of supplementary services described in [Arnouat91] thereby becomes inappropriate.

The second paper, [Maruyama91], is concentrating on a different issue - the modelling of a switching program for digital exchanges.

Recently the CCITT has adopted OSDL (Object-Oriented SDL) as a description language. OSDL contains much the same descriptive power as BETA. The ISDN standards, however, have not yet been rewritten in terms of OSDL and it is doubtful if they ever will be.

8 Acknowledgments

Thanks are due to Professor Ole Lehrmann Madsen for his neverending encouragement to pursue problems related to object-orientation in general and to BETA in particular. Thanks are also due to Mr. Henrik Dyrholm at DIAX Telecommunications A/S for many fruitful discussions on telecommunication aspects as well as programming languages and to Mr. Asger Hansen at the ISDN laboratory, Jydsk Telefon A/S (Jutland Telecom Corporation), for sharing his insight on ISDN implementations.

References

[Bocker87] Peter Bocker: *ISDN - The Integrated Services Digital Network.* Springer-Verlag, 1987.

[ETSI90] ETSI *Document No. ETS T/S 46-30, ETS T/S 46-31.* ETSI Telecom Standards, 1990.

[ETSI91] ETSI *Document No. ETS 300 058, ETS 300 092, ETS 300 093.* ETSI Telecom Standards, 1991.

[I.250] CCITT *Recommendation I.250: Definition of supplementary services.* CCITT BLUE Book, Fascicle III.7, ITU, Geneva, 1989.

[Madsen91] Ole Lehrmann Madsen et.al: *Object-Oriented Programming in the BETA Programming Language.* Draft (January 1991), Aarhus University, Computer Science Department, 1991.

[Hedin86] Görel Hedin et.al: *Incremental Execution in a Programming Environment based on Compilation.* In: Proceedings of the 19th Hawaii International Conference on System Sciences, Vol. IIA Software, January 1986.

[Hedin88] Görel Hedin et.al: *Supporting exploratory programming in Simula.* LU-CS-TR: 88-31, Lund University, 1988.

[Söderberg88] Lennart Söderberg: *Architectures for Intelligent Networks.* In: Ericsson Review, 1/1989.

[Berman92] Roger K. Berman et.al.: *Perspectives on the AIN Architecture.* In: IEEE Communications Magazine, Vol.1, No.2 (Feb.92).

[Blair91] Gordon Blair et.al. (ed.): *Object-Oriented Languages, Systems and Applications.* Pitman Publishing, 1991.

[Agesen89] Ole Agesen et.al.: *Persistent and Shared Objects in Beta.* DAIMI IR-89, Aarhus University 1989.

[Huber91] Peter Huber et.al.: *A Formal, Executable Specification of the ISDN Basic Rate Interface.* In: Proceedings of the 12th International Conference on Application and theory of Petri Nets.

[Maruyama91] K.Maruyama et.al: *A Concurrent Object-Oriented Switching Program in Chill.* In: IEEE Communications Magazine, January 1991, Vol.29, No.1.

[Arnouat91] C.Arnouat et.al: *An Application of the Object Oriented Paradigm to the Modelization of Telecommunication Services.* In: Tools 1991 Conference Proceedings, Prentice-Hall 1991.