Signalling in ATM Networks: Experiences With an Object-Oriented Solution

Robert Engel, Toni Bieri, Beat Keller

ASCOM TECH
Morgenstr. 129, CH-3018 Bern, Switzerland
++ 41 31 999 39 47, engel@tech.ascom.ch

ABSTRACT Today the Asynchronous Transfer Mode (ATM) is extensively used on the level of public pilot networks. In the first stages of introduction these networks do not yet support signalling.
However, the standardisation bodies ITU-TS (former CCITT) and ATM Forum have already produced first specifications for signalling in ATM networks. These specifications define the basic procedures for point-to-point signalling. Further specifications, that are now being defined, will enlarge this basic set to support fully signalling for multimedia applications between several users (point-to-multipoint).
This signifies that developers of signalling software have to cope with two demands: Until a Recommendation for a specification is approved it undergoes constant change. The introduction of a new specification is associated with an enormous increase in software functionality.
In this paper we show how an object-oriented solution allows the demands of flexibility and extendability to be satisified. This solution consists of a set of classes that are put together to form a very flexible, extendable framework. This framework can be refined to realise any signalling protocol.
We show how this framework was used to build an implementation of the ATM signalling protocol Q.2931 defined by ITU-TS. The reusability and flexibility of it was verified when the framework classes were integrated in the signalling architecture of an ATM switch.
The framework was realised with object-oriented techniques and implemented in C++.

1 Introduction

1.1 ATM Signalling Protocols

The Asynchronous Transfer Mode (ATM) is the selected transfer mode for Broadband Integrated Services Digital Networks (B-ISDN). It is connection-oriented and will optimally use available bandwidth (with statistical multiplexing), whilst also providing a negotiated Quality of Service.
This is only possible due to a new Layer 3 Signalling Protocol (see Figure 1-1) that can enable source traffic characteristics to be given to the network. Two standardisation bodies each define a slightly different version of this User Network Interface Protocol: The ATM Forum calls it “Q.93B” [1], the ITU-TS calls it “Q.2931” [2]. The Protocol is defined in three steps, called “Capability Sets” by ITU-TS and “Phases” in ATM Forum. Each new specification will be a super set of the preceding one and increase the functionality. The third Capability Set / Phase will finally define a signalling protocol that *fully supports point-to-multipoint Multimedia Applications.* In this paper we refer to ITU-TS Capability Set One, but the ATM Forum version is essentially the same except that it already supports point-to-multipoint signalling.

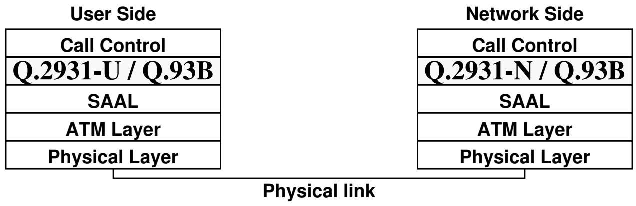

Figure 1-1 *Protocol Stack of a UNI showing the ITU-TS and the ATM-Forum Layer 3 protocol*

Capability Set One has reached its definite form: The Recommendation Q.2931 that defines the Layer 3 of the User Network Interface is now in State APPROVED. Capability Set One defines only the basic procedures for signalling between two endpoints and does not fully support the capabilities ATM will offer. This means that the increasing functionality of the two following Sets will be considerable.

1.2 Possibilities to implement a Protocol

Today’s technology allows three main possible realisations of protocols:

- *Classical*: With a classical design method and a language like ‘C’ software is developed “like it has always been done”. This method is relatively quick and well known. However, reusability and flexibility are poor.
- *Code generation*: CASE-tools like “SDT” [3] enable protocols to be defined with a specification language such as ITU-TS’s Specification and Description Language SDL [4]. The CASE-tool generates code of this detailed description. Though this method seems to be elegant, a specification language like SDL is not well suited to describe the details of a Protocol. In addition, the set of target platforms is restricted.
- *Object-oriented*: The object-oriented approach provides flexibility and extendability due to new design concepts. Compilers for OO-languages are available for almost every platform.

1.3 Demands for ATM Signalling Software

The final goal for ATM networks is to fully support multimedia applications between several users (point-to-multipoint). Such applications use a lot of bandwidth that can change dynamically and very quickly. The new business of telecommunication vendors will be to sell a *certain amount of bandwidth* in a *certain time interval*. ATM signalling software differs from existing signalling software in so far that it has to be able to *manage bandwidth* in order to use it optimally. The requirements for such software are the following (see Figure 1-2):

- Signalling protocols are becoming increasingly complex.

Ultimately ATM signalling will have to handle multimedia sessions between several people, during which connections can be added and dropped, users can be added and dropped or traffic parameters (e.g. bandwith) can be renegotiated. Software that can handle these requirements will be *very complex*.

- The time-to-market for every type of software is getting shorter and shorter, regardless of the complexity. Companies cannot afford to wait until a definite standard is defined to start implementing software since then the first products will already be appearing on the market. This means that software has to be developed in *parallel with the emerging standards*.
- The defining of ATM signalling software in steps with increasing functionality requires software that is *extendable*. The cost of developing completely new software every time a new Capability Set is defined, is too high. The software architecture has to be sufficiently open and must allow the integration of additional functionality.
- Both ITU-TS and ATM Forum standards have to be respected. It must be possible to *reuse* software for both standards.

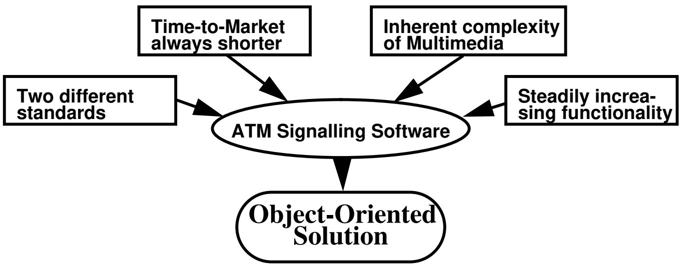

Figure 1-2 *Requirements for ATM Signalling Software*

Only with object-oriented concepts different requirements like complexity, extendability and reusability of a software can be handled.

2 Signalling Protocols

What are the components that build a signalling system and how can they be matched with Software elements?

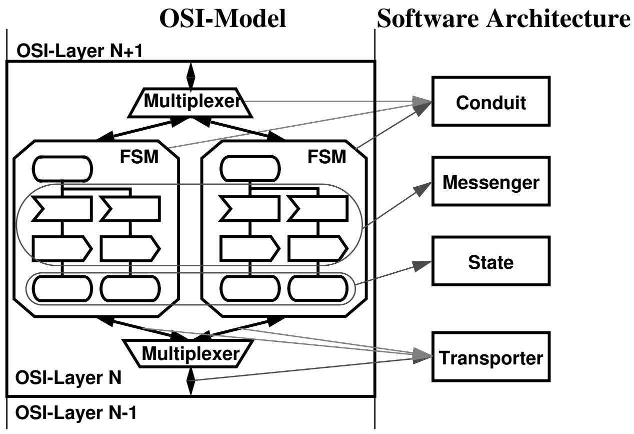

Figure 2-1 *Mapping Components of a Signalling Protocol to Software Classes*

Signalling protocols are always alike: They are connected to the upper and lower OSI-layer by a bidirectional channel. Within the OSI-layer there can exist several Finite State Machines in parallel that are mutually independent. Primitives from the upper and lower OSI-layer have to be directed to the correct instance of a FSM where a transition is executed.

We defined the following software classes that execute the behaviour of these components of a signalling protocol (see Figure 2-1):

- The "container" that encapsulates the FSM and the components that split up the incoming primitives and direct them to the FSMs they belong to are represented by the `Conduit` class.
- The messages that stimulate the FSM are represented by the `Messenger` class.
- The States that actually define the different transitions are represented by the `State` class.
- The messages have to be transported to the correct FSM instance. It is the responsibility of the `MessengerTransporter` class to do this. This class is derived from a general `Transporter` class (see section 4.4).

Of course all of the above classes are top classes of an entire class hierarchy. Classes with the specific behaviour (e.g. the different messages, the different States etc.) have to be derived from this base classes. In section 4 the different base and derived classes and their responsibilities are explained in detail.

3 Building the Q.2931 Protocol

The following example shows how we built an application realising the Q.2931 protocol. This application was required to be an independent module, that anyone realising ATM signalling software could use. For communication with the upper and lower OSI-Layer, we offered a API.

3.1 The Q.2931 Protocol

The Q.2931 Recommendation specifies the signalling protocol at the User Network Interface in an ATM network. It is situated at the Layer 3 of the OSI-model. The protocol has two specified interfaces: An interface to Call Control (CC) on the upper side and an interface to the Signalling ATM Adaption Layer (SAAL) on the lower side (see Figure 1-1).

The Q.2931 protocol consists of two FSMs:

- The Co-ord FSM assures that downgoing data is only sent to the SAAL when the SAAL is in a state which allows an assured transmission.
- The Q.2931 FSM is instanciated for every call in progress. Every call has its own instance of an FSM to which all signalling messages have to be guided. Here the actual signalling takes place.

3.2 Layout of the Conduits

In Figure 3-1 we see how different `Conduits` have to be connected to build the Q.2931 protocol.

What happens when a primitive from SAAL carrying e.g. an ALERTING message enters the system?

a) The `Adaptor` converts the Layer 3 Message in a `Mes-`

senger object that a MessengerTransporter transports to the first Protocol.

b) The Protocol passes the contents of the MessengerTransporter (the Messenger) to its State where the Messenger and the State execute a transition. Let us suppose that during the transition the Messenger is encapsulated in another MessengerTransporter and sent to the Mux. If another state has to be generated at the end of the transition, a new object State replaces the old one.

c) With its dispatchKey, the MessengerTransporter demultiplexes at the Mux and finds the correct Protocol instance to which it delivers its Messenger. If no Protocol exists yet because this is the first message of a new call the MessengerTransporter is sent to the ConduitFactory. The ConduitFactory creates a Protocol-State pair and connects it to the upper and lower Mux. The MessengerTransporter is then sent to the Protocol to which it delivers its Messenger.

d) Again the State that is attached to the Protocol executes a transition when stimulated by the Messenger. Let us assume that a new Messenger is created which is meant for CallControl. It is enveloped with a MessengerTransporter and sent to the upper Mux.

e) The upper Mux muliplexes (gives the MessengerTransporter a dispatchKey in function of the Protocol it comes from) the MessengerTransporter and passes it to the upper Adaptor.

f) The upper Adaptor converts the Messenger into the format needed by the software implementing the upper layer (Call Control).

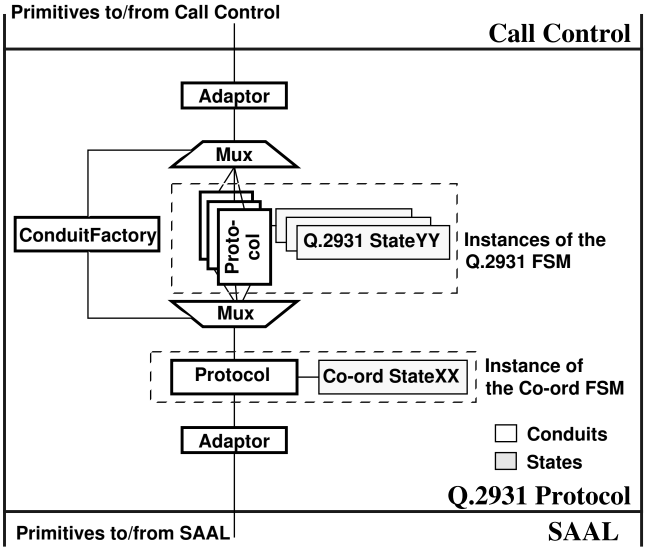

Figure 3-1 *Conduits building the Q.2931 Protocol*

3.3 Implementing a Finite State Machine

In our application we have two FSMs. In the following we will show how the Q.2931 FSM was realised. The realisation of the Co-ord FSM is analogous.

Three types of classes are involved realising an FSM: Protocol, State, Messenger.

3.3.1 Protocol and States

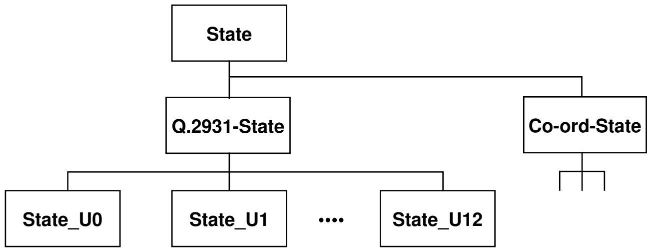

Figure 3-2 *State Hierarchy*

The Protocol class is an abstract concept that knows nothing about messages and transitions. It offers all the mechanisms any FSM might need like setting/resetting timers etc. These mechanisms are implemented as class methods a State object can call in a transition.

The changing of State at the end of a transition is represented by exchanging the State that is attached to the Protocol with the next State that is needed. This is possible with the dynamic binding that object-oriented languages offer: The Protocol knows only that it points to a *kind of* State. This kind of State is in fact the base class of an entire State hierarchy (see Figure 3-2). Each State defines a specific transition for each message. As in fact in most States only a few messages lead to a valid transition and for all other messages a Null transition or a “Error” transition happens this can be delegated to the base class. This is an effective way to implement “default transitions”: In the base class Q.2931-State a default transition is defined for every possible stimulus (every possible message/primitive). If this default transition is not correct for a specific (derived) state this state can overwrite (redefine) this transition according to its needs.

For example a SETUP-Messenger coming from SAAL will be ignored (lead to a Null-transition) by all states except State_U0. So we can implement the Null-transition in the base class Q.2931-State. All derived states will inherit this transition/method. As State_U0 should execute a valid transition when it is stimulated by an incoming SETUP-Messenger we overwrite the inherited method.

3.4 Messenger

As we have made a class for every message/primitive we have the class hierarchy represented in Figure 3-3.

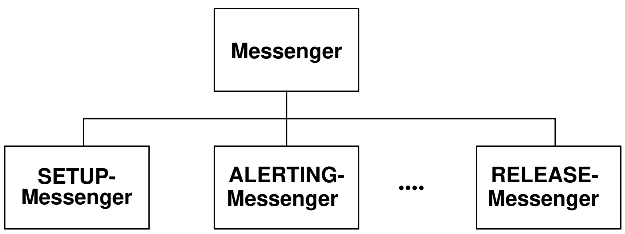

Figure 3-3 *Messenger Hierarchy*

It is the Messenger’s responsibility to stimulate a transition

when it encounters a State by calling the State's specific method.
However, this is just one responsibility among many. A Messenger also has to be able to provide certain information it carries (a RELEASE-Messenger has to be able to provide the Cause-value that tells why it has been emitted). Another duty that is quite complex consists in the verification procedure: When a Messenger from SAAL stimulates a transition in a Q.2931-State the State has to know whether the message is encoded correctly (since the peer entity could violate the encoding rules) and if not, which kind of error there is. This leads to a ping-pong between State and Messenger, each calling a certain method of the other until the transition is terminated.

3.5 Interaction of States and Messages

3.5.1 Classical Approach

Classically (non object-oriented) a FSM is encoded by interleaving two SWITCH/CASE statements. In the first SWITCH the state is determined, in the second one the message type. As a transition is a function of state and message type this allows to define the transition in the second SWITCH statement. This is shown in the following pseudo-code:

```
switch(state)// determine which is the
{             //    actual state
   case STATE_XX:
      switch(message_type)
      {     // determine which message
            // has been received
         case SETUP:
                  ....(transition XX)
                  ......
         case ALERTING:
                  ....(transition XY)
                  ....
         case.....
      } // end switch

   case STATE_YY:
         ........
   case STATE_ZZ:
         ........

}  // end switch
```

There is an essential drawback in this coding: Whenever a change has to be made (e.g. a message is removed) the code has to be *edited in every CASE* statement. Changing code in a lot of places always bears a certain danger of missing a change.
By the way, this is not a theoretical change but a real one. During the evolution of the Recommendation Q.2931, the message DISCONNECT was removed!

3.5.2 Object-Oriented Approach (Double Dynamic Binding)

Our object-oriented approach replaces the two interleaved SWITCH statements by double dynamic binding (see Figure 3-4): At the first dynamic binding the specific Messenger is determined, at the second dynamic binding the dispatching of the State is done.
Instead of having a centralised entity (the main program) check which message type is stimulating the state, this work is delegated to the Messenger (since a Messenger knows exactly of its message type, this seems to be quite a natural way). So the Messenger is told it has arrived at a Protocol (it does not need to know whether it is the Q.2931-FSM or the Co-ord-FSM) and within this method-call the State, that the Protocol is pointing at, is passed as parameter:

1)Messenger -> upProtocol(State)

When this method of the Messenger is called, by dynamic binding (first dynamic binding) the method of the derived class is called. In our example the method upProtocol is defined in the Messenger class and overwritten in the derived class SETUP-Messenger. As the object is an instance of the class SETUP-Messenger its specific method is activated and it calls now the following method of the received State

2)State -> upSetup(SETUP-Messenger)

Due to dynamic binding (second dynamic binding) not the method upSetup of the Q.2931-State is called, but the overwritten method in the State_XX class. In this method the whole transition is defined which can now be executed.
The dynamic binding frees us from the need to use CASE-statements which are difficult to update. In addition if changes have to be made they are locally restricted to a class and there is no need to scan a lot of CASE statements.

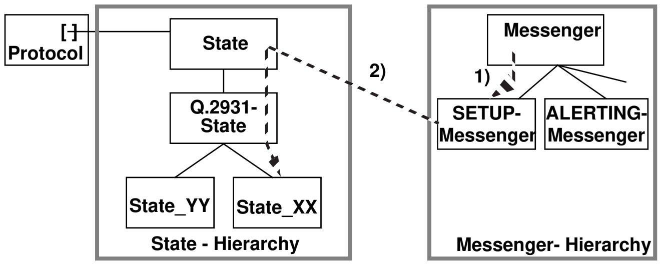

Figure 3-4 *Mixture of Object-Scenario and Hierarchy graph explaining double dynamic binding*

At the end of most transitions a new State replaces the old one. We did this using the concept described in [5]. With this method we have a choice if we want to create and destroy States dynamically (and save memory by deleting them when they are not used anymore) or we can create them at the starting up of the software (this uses more memory, but is faster, because no dynamic memory allocation is necessary).

3.5.3 Examining a Message on its correctness

As Protocols grow more and more complex the messages that are exchanged between them also get more complex: A lot of effort has to be invested in error procedures. Error procedures apply when a message arrives from SAAL that is not correctly (according to the Recommendation Q.2931) encoded.
These error procedures are more complicated than in existing protocols: To guarantee compatibility with the future Capability Sets, a Layer 3 Message not only carries signalling data but also encodes which error procedures happen if the message is not recognised/expected by a protocol or if it is erronous. In this case we have different possibilites of handling such an error depending on *error type and State and message type:*

a) If the message is correct but unexpected in a certain state a specific error transition happens. This means that the transition depends also on the state!

b) The message contains a flag that, if set, references another part of the message where special error handling is defined [2]. If it is not set, the default error transition happens.

c) If the mentioned flag is set, a specific error transitions happens according to the referenced part of the message.

d) Depending on the `kind` of error a specific error transition happens.

All these error transitions can be solved elegantly by using our concept with double dynamic binding. In the case of an erronous message we do not execute the transition as in 2) of Figure 3-4 but continue with the "ping-pong" between `State` and `Messenger`:

For example, a third step (3) could consist in a call to `Messenger`

```
3) returnValue = Messenger -> YouAreUnexpected()
```

This solves the problem for a).

For b),c) and d) we have to define an additional call (4) that returns the error of the message, respectively which error transition should happen:

```
4) errorTransition = Messenger -> verify()
```

This assures us, that our concept with very active and intelligent `Messenger` classes, that can interact instead of simply being manipulated, simplifies implementations.

4 An Object-Oriented Solution

4.1 Gaining Abstraction through a Framework

Our solution for implementing signalling software consists of two parts: Firstly we have a protocol-independent set of classes; a so-called framework [6]. Secondly we have protocol-dependent classes (mostly derived from the independent ones), with which the concrete application (the protocol Q.2931) is built.

In this context, a framework represents a skeletal structure of protocols. To build a specific protocol, this structure has to be refined. The primary goal of our framework was therefore to make it refinable. It had to specify how classes communicate together on an abstract level; a level that is valid for any specific protocol. This is the reason why we chose to make an object-oriented design and implementation. Object-oriented techniques allow interaction between system components (classes) to be specified on an abstract (and thus application-independent) level. The refinement for a specific protocol can then easily be done *without changing existing code or design*, simply by adding specific functionality to derived classes.

This leads to the question: What aspects are essential in all signalling protocols? These terms have to be modelled in the framework. A protocol always consists of one or several Finite State Machines (FSM) of which several independent instances can exist at the same time. Primitives from the lower and upper OSI-level will stimulate these FSM instances. When considering independent instances of FSMs that react on messages/primitives, it is necessary to define how a message is guided to the instance it belongs to. These two concepts: FSMs that can be instanciated, and an architecture that guides a message to an FSM instance, are modelled in our framework.

In our framework, we used concepts of other communication frameworks [7], [8]. These frameworks model how different protocol layers in the OSI-model can interact together. However, we concentrated on the modelling *within* an OSI-Layer. In our view, this is the more critical point where we have to find an abstraction. The connection of different OSI-Layers can be very application-specific and depends strongly on the environment. For example, it is an implementation decision, if each OSI-Layer is implemented as a single process, or there are modules for each Layer that are linked together to form one single process.

4.2 Interaction between classes in the Framework

We have implemented four class hierarchies to build our framework. The abstract classes on top of the hierarchies are called `Conduit`, `Transporter`, `Messenger` and `States`. To avoid a strong coupling between all four classes we designed our framework so that `Conduit` and `Transporter` communicate together, as do `Messenger` and `State`. However, there is hardly any cross-communication. The goal of the design is that `Messenger` can find the correct instance of a FSM (i.e. a specific `State`) and can select a transition. This choice of the transition and its execution represents the actual signalling. The `Conduit` and `Transporter` classes are necessary to guide the `Messenger` to this `State`.

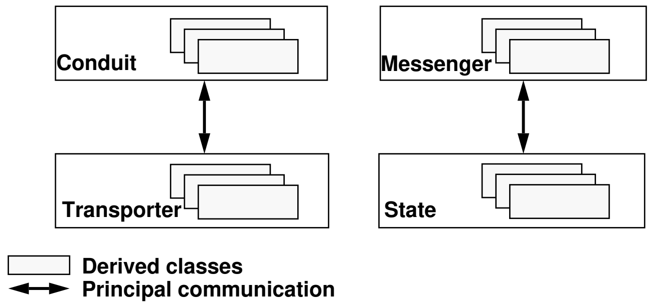

Figure 4-1 *Principal Communication between the different Class Hierarchies*

There is also another difference between the `Conduit-Transporter` pair and the `Messenger-State` pair:

The `Conduit-Transporter` pair is basically application-independent. This means that its classes are implemented in the framework. Their code does not need to be adapted when implementing a specific application (like the Q.2931 protocol).

However the `Messenger-State` pair is strongly application-dependent. As the interaction between `Messenger` and `State` determines the FSM (which is of course application-dependent) these classes only exist in a very abstract form within the framework. To build an application they have to be refined by derivation (inheritance) and the adding of application-dependent code.

4.3 The Conduit Hierarchy

The easiest way to understand the `Conduit` hierarchy is to imagine `Conduits` as kind of "hardware components" or physical conduits/pipes. The abstract[1] class `Conduit` implements the following behaviour:

1. An abstract class is a class that can not be instanciated

- It has two sides. Each side can be connected to another Conduit.
- It accepts objects of type Transporter and puts them into an input queue.
- When it is activated, it activates the Transporters in its input queue in the order in which they have arrived, by telling them where they are (calling the respective methods atConduitFactory, atMux etc.).

The derived classes (ConduitFactory, Mux, etc) add more specific behaviour. These different classes allow an object of the Transporter hierarchy to find its way to the instance of FSM it has to reach. All these classes implement a specific (abstract) behaviour:

- Mux: A Mux allows an object of the Transporter hierarchy to demultiplex with a dispatchKey to a specific Conduit in one direction. If no specific Conduit is found, the Transporter has to go to the default Conduit. In the other direction, a Mux provides a multiplex function (receive a dispatch key) when coming from a specific Conduit.

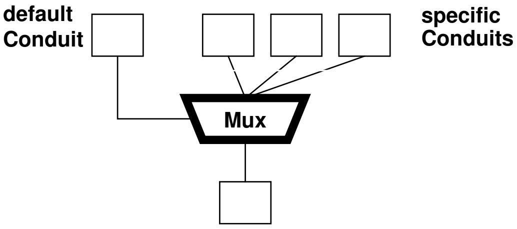

Figure 4-2 *Multiplexer/Demultiplexer connected to other Conduits*

- Adaptor: An Adaptor allows the framework to communicate with the outside world. As we cannot make any assumptions about what is outside our framework, we have to provide this class. An Adaptor can convert “something” from the outside world (for example a message in String format) into an object of the Messenger hierarchy and vice versa.
- ConduitFactory: The ConduitFactory creates new Protocols (together with the initial States) and connects them at the corresponding Muxes.
- Protocol: The Protocol is the handle to a specific instance of a FSM. As shown in Figure 4-3 it is attached to an element of the State class. Together they build an instance of a FSM. The Protocol offers all the functions that an FSM has to perform such as setting/resetting timers, saving messages for later treatment etc.

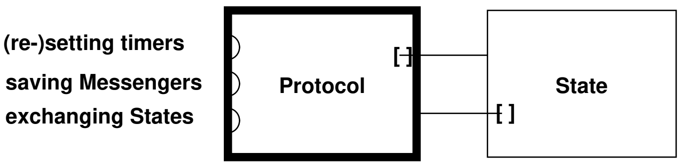

Figure 4-3 *An FSM consists of a Protocol to which a State is attached*

4.4 The Transporter Hierarchy

The responsibility of the abstract class Transporter is to find its way through the system consisting of connected Conduits to the Conduit where it has to perform its special task. This way it implements the following behaviour:

- It knows the recent Conduit it has come from.
- It knows how to behave at each Conduit. (For example at a Mux it knows if it has to dispatch itself to one certain specific Conduit (demultiplexing by using its dispatch key), or if it has to broadcast itself (sending a copy of itself and an eventual carried item) to all specific Conduits etc).

Three classes are derived from Transporter:
The MessengerTransporter is responsible to guide a Messenger object through the system until it has reached the FSM instance it is meant for. After having performed this task, the MessengerTransporter can be deleted.
There are also “administrative” Transporters. The Connector has to connect aProtocol at a Mux, i.e. add a new relation Conduit-dispatchKey. The Disconnector does the inverse.

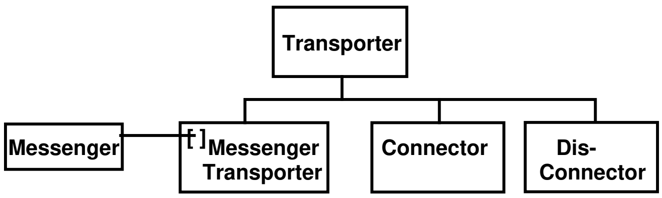

Figure 4-4 *Transporter hierarchy*

4.5 The State Hierarchy

The State classes are responsible for executing the correct transition when they are “triggered” by a Messenger. The States are attached to a Protocol. The changing of a state in a protocol after a transition, corresponds to the exchange of a State object in our framework with another State. As the functionality of the States depends strongly on the FSM they implement, they are not further encoded within the framework.
This means that the responsibilities of the State classes are well defined in the framework, but the effective implementation can only occur when the application for which the FSM is needed is fully known.

4.6 The Messenger Hierarchy

The Messenger class hierarchy is the most important one of our framework.
In other communication frameworks, the messages/primitives that stimulate the FSM are usually no more than mere data containers. That is, they do not have any functionality or intelligence but just consist of data. Differences only exist in so far as how easy or how difficult it is to access the data.
In our implementation we explored a new way by using the object-oriented features provided by a language like C++ to encapsulate the messages/primitives not only in an Abstract Data Type but in an intelligent class to which a lot of functionality is added. In this way, for each message/primitive there exists an own class. Instead of a main program that executes a transition in the

FSM (after having determined what the type of message/primitive is), in our framework the specific `Messenger` class (that naturally knows of which type it is) stimulates the `State` class with a class-specific method. So the creation of a transition happens like a "dialogue" between `State` class and `Messenger` class until the transition is finished, whereas in other communication frameworks a transition happens as a "monologue" which means that a state simply extracts the data it wants of the passive message container.
The advantage of having intelligent `Messengers` lies in having intelligence more evenly distributed over the system, which is one of the goals of the object-oriented technology. This also guarantees that functionality and data that belong together are not separated.

5 Experiences

5.1 Developing the Framework

The developing of this framework was clearly a process with many iterations. When we started developing this software our goals were to have both an application realising the Q.2931 Protocol and a framework that represents an abstraction of Protocols and that can be reused in other contexts.
However we realised soon, that *it is not possible to develop separately* a framework and an application. On the contrary: It is simply not possible to develop only a framework without having a concrete application in mind. We noticed that it is rather the other way round: When designing a concrete application you have to ask yourself over and over again, whether a class, that is being designed, is application-specific, application-independent or a mixture of both. Application-independent classes are then defined as belonging to the framework.
But mostly the latter case will occur that a class has application-independent elements and application-dependent elements. Then the complicated process begins of splitting such a class in a dependent and an independent part.
Unfortunately the term application-independent is not invariant with the time. What seems to be independent at the beginning of a development can change, when going deeper into details. So with every iteration, with every refinement of design you have to critically review existing parts concerning their dependencies.
We even believe, that a framework cannot be judged if it has been used in only one application. Every design of a framework suddenly looks differently, when you want to reuse it for another application than it was originally used. In the point of view of such a new application the term application-independent changes again. This is a normal effect, as it is not possible to foresee every application in which a framework could be used and thus seperate rigorously application-dependent and application-independent features.

5.2 Long Term View

A typical management question is whether to develop an application with object-oriented techniques or not. Besides the point that today you cannot sell a software product if you cannot claim it is object-oriented (no matter, whether it uses an object-oriented design or only an object-oriented language), it is important to know what is the *life cycle* of the product.
If you want to implement a protocol which is fully specified and of whom you are certain that no major changes will ever happen, an object-oriented design might not be the best one. This is due to the fact that a good encapsulation of functionality and data in classes needs *several design iterations* and *more time* than the modular decomposition used in classic software development.
But in a new protocol like the Q.2931 where you are aware in advance that changes have to be made a, very *flexible software structure is crucial*. In addition the increase of functionality with the further Capability Sets make it necessary that existing code is *reused*. Otherwise, if you make a new implementation each time a new Capablity Set is defined you have no possibility to compete on the market.
So an object-oriented solution using a framework is clearly an *investment for the future*. Though it takes longer to implement the first Capability Set than in a classic implementation, the advantages will clearly show when the next Capability Set is ready, or when the framework is used for another protocol (e.g. the ATM Forum version).

5.3 Experiences with the Framework in Another Application

Besides the Q.2931 protocol we used our framework also in another context:

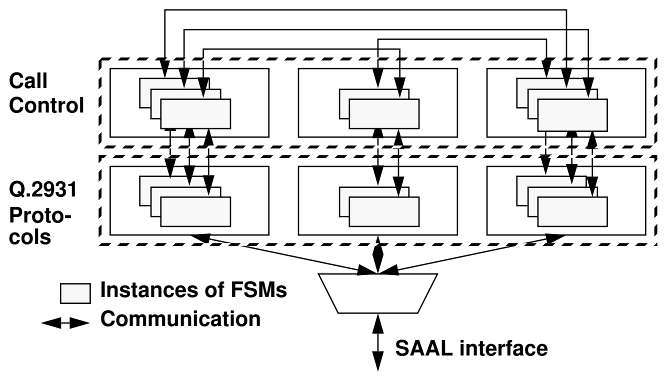

Figure 5-1 *Simplified signalling architecture in a switch*

This context is the signalling within an ATM switch (see Figure 5-1): A switch has several lines. By simplifying somewhat we could say that each line of the switch needs its own instance of a Q.2931-Protocol. So on each line we have an instance of a Q.2931-Protocol which itself has an instance of a Q.2931 FSM for every call that is processed on this line.
On top of the Q.2931-Protocol there is the Call Control that connects calls between different lines (also simplified!). We divide the Call Control that has to provide such a connection in two seperate FSM. As soon as a call SETUP arrives on a line, a new instance of a Q.2931 FSM and a Call Control FSM has to be created. After having made the necessary reservations and after routing, the Call Control FSM on the departing line is created. The two Call Control FSMs are then connected. This connection is done exactly the same way as a Layer 3 Protocol is connected with a Mux. From then on messages can easily be transported through the switch and be processed.To integrate the Call Control into our new architecture we had to develop a new FSM. This was done by simply deriving a new set of classes of our State, defining all the transitions necessary for the Call Control

FSM as depicted in Figure 5-2.
As the Call Control connects "lines" using the same Protocols (Q.2931 on Layer 3) no new `Messenger` hierarchy had to be defined.

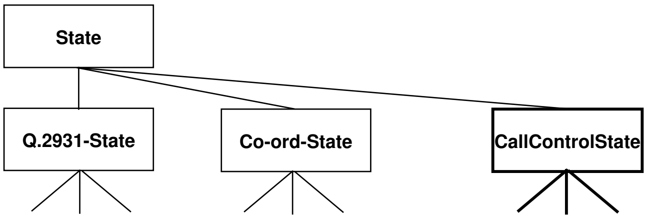

Figure 5-2 *State Hierarchy in the Switch Architecture*

A great success is that the existing `Transporter` and `Conduit` class and their derived classes have remained unchanged. By using the framework in an environment with extended functionality (the Call Control block increases functionality considerably) we could test its extensibility. In fact we managed to integrate the existing classes of the framework with some minor changes.
However, it is vital to execute these changes as a framework itself is not a rigid construction but will evolve with every time it is reused in an application. Of course the necessery changes added to the framework should decrease with every reuse, as the framework is getting better and better.

6 Conclusions

ATM has been accepted as the transfer mode of the future. Signalling for ATM networks is already becoming a necessity as more and more users want to establish and release connections as in today's telephone network.
With the complexity inherent in managing such sophisticated features as Multimedia applications, optimising the valuable bandwidth etc., the introduction of object-oriented technologies for signalling software is mandatory. In ASCOM TECH we have developed a framework that was used to build an application implementing the Capability Set One of the Layer 3 protocol.
We are convinced that our method of giving a lot of intelligence to the `Messenger` class will provide flexibility to the requirements for the further Capability Sets, that will add new messages to the existing ones. As our framework has already been used for the Q.2931 protocol and in the Call Control of an ATM switch, we believe that it has reached a stable and application-independent state that can be reused and extended very easily.

7 Acronyms

AAL: ATM Adaptation Layer
API: Application Programmer Interface
ATM: Asynchronous Transfer Mode
B-ISDN: Broadband - Integrated Services Digital Network
CC: Call Control
FSM: Finite State Machine
IBC: Integrated Broadband Communication
ITU-TS: International Telecommunication Union - Telecommunication Standardization Sector
L3: Layer - 3
OSI: Open Systems Interconnection
OO: Object Oriented
SAAL: Signalling ATM Adaptation Layer
SDL: Specification Description Language
UNI: User Network Interface

8 References

[1] ATM Forum; *ATM UNI specification Q.93B* Prentice Hall, 1993
[2] ITU-TS Recommendation *Q.2931*
[3] Telelogic; *SDT, Specification and Description Language Tool* Malmo, Sweden
[4] ITU-TS; *Specification and Description Language* Recommendation Z.100
[5] Aamod Sane; *Subclassing and Composition of State Machines,* University of Illinois at Urbana-Champaign, February 1 1994
[6] Rebecca Wirfs-Brock; Designing Object-Oriented Software Prentice Hall 1990, ISBN 0-13-629825-7
[7] J.M. Zweig; An object-oriented Framework for implementing network protocols University of Illinois at U-C 1991
[8] N. C. Hutchinson and L.L. Peterson; *The x-kernel: An Architecture for Implementing Network Protocols* IEEE Transactions on Software Engineering, vol. 17, January 1991, pp. 64-76,