FROM SIBS TO DISTRIBUTED OBJECTS: A TRANSFORMATION APPROACH FOR SERVICE CREATION[1]

Elie Najm, Frank Olsen, Sylvie Vignes
École Nationale Supérieure des Télécomunications
Elie.Najm@enst.fr,olsen@acm.org

Abstract This paper describes how to apply correctness-preserving transformations to the problem of service creation using the IN CS-1 SIB (Service Independent Block) concept. We show how SIBs can be represented by a UML class diagram augmented with constructs from an action language and then automatically translated into a Java like language. We also show how this transformation can be implemented using a UML-based CASE-tool.

1 INTRODUCTION

This paper presents our work on service creation for telecommunications networks. We position ourselves in a world where the predominant network architecture is the Intelligent Network (IN). However, the IN has many well-known limitations. In this paper we consider a major one: The use of a completely algorithmic approach to service creation and reuse of "components". This problem has been acknowledged by most of the actors in the telecommunications business and has led to interest in new and improved architectures based on objects and distributed computing technologies. The major example of these new architectures is TINA.

The starting point for this paper is our previous work on correctness-preserving transformations for open object-based distributed systems [1]. There we defined an automatic transformation between two languages: a *source language* (*SL*) and a *target language* (*TL*). *SL* is based on the assumption of operation on a global state while *TL* is a representative of distributed object-based languages.

Our main contribution in the present paper is to apply our transformation for IN Service Features (SIB chains). In other words, we are interested in

[1]This work has been partially supported by RNRT project PILOTE

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35581-8_35

a top-down approach aided by automatic transformations. Since the result of the transformation is an object-based language, our work also provides a possible evolution path from IN-style service creation towards open object-based distributed systems as exemplified by CORBA or TINA.

1.1 Outline of the paper

In section 2 we give an overview of the IN architecture and outline the current process used for creating IN services. Next we introduce the main results from our previous paper mentioned above that are used as a basis for the current paper in section 3. Then comes the central part of the paper where we provide a mapping of SIBs (and Service Feature) into distributed objects in a Java-like language. Finally, we conclude in section 5 with a brief summary of this contribution, a comparison with existing work, and an outlook for further work.

2 IN SERVICE CREATION

In this section we give an overview of the Intelligent Network architecture and with emphasis on the service creation aspect.

2.1 The Intelligent Network architecture

Work on the IN architecture (ITU-T Recommendation Series Q.12xy [3]) started in 1988 with the aim of making service creation more rapid than for the existing telecommunications network. Before the IN, all the intelligence was in the switches and service creation was performed by the switch (hardware) providers (on demand from the telecommunications operators). The main solution proposed by the IN was to separate the service logic from the call and connection control. The latter were still performed in the switches, but the former was separated out into a new piece of equipment called a Service Control Point (SCP). The different functions of the network are performed by Functional Entities (FEs); the main FEs defined by the IN are the CCF (Call Control Function) which performs call and connection control, the SSF (Service Switching Function), and the SCF (Service Control Function) which contains the service logic. Typically, the SCF is located in the SCP and the CCF and SSF in the switches (also called Service Switching Points (SSP)). The role of the SSF is to hand over control from the SSP to the SCP at specific points in a call where service logic can be invoked. A typical example is that a user dials a number which the SSF hands over to the SCF: the SCF can now invoke a feature like Originating Call Screening (OCS) which verifies that the dialed number is not on a list of screened numbers.

Acknowledging that the telecommunications world is in constant evolution with new technologies constantly appearing, the IN is defined in phases called

Capability Sets (CS). As of mid 1999, two CSs have been released and a third one is due at the end of 1999. Simultaneously, work has already started on a fourth capability set. In this paper we only take into account IN CS-1 (Q.121x) [5].

2.2 Intelligent Network Conceptual Model

The Intelligent Network Conceptual Model (INCM) provides a conceptual framework for the provisioning of telecommunications services and service features. It defines four different planes: The Service Plane (SP), the Global Functional Plane (GFP), the Distributed Functional Plane (DFP), and the Physical Plane (PP). These planes define different levels of abstraction of the telecommunications network and its services. In this paper we will not discuss the Service Plane, since it is not defined for IN CS-1; suffice to say that the SP deals with Services and its constituent Service Features.

In relation to service creation the main idea behind the INCM is to describe Services and Service Features at different levels of abstraction thus enabling people with different skills and detail of knowledge to be actors in the service creation process. The skills range from business-related skills at the SP to detailed knowledge of the network and its architecture at the PP where new network functionality may need to be implemented to create a new service.

Global Functional Plane. The GFP provides a set of Service Independent Building Blocks (SIB) which are used to create Service Features. A SIB is a procedural building block that can be reused in several Service Features.

A Service Feature is a flow graph of SIBs (also called a SIB chain) with the outputs from one SIB connected to the input of another. The starting point for a service feature is a POI (Point Of Invocation) which corresponds to a specific point in the BCM (Basic Call Model). The BCM is a state machine representation of a telephone call. A service feature has one or more PORs (Points Of Return) which return control to specific points in the BCM.

Distributed Functional Plane. In the Distributed Functional Plane (DFP), the SIBs at the GFP are decomposed into Functional Entity Actions (FEA). Each FEA belongs to a single Functional Entity (FE). A single SIB can be distributed over several FEs: In this case there will be several FEAs for the SIB and they will communicate using Information Flows (IF). The parameters passed with the IFs are called Information Elements (IE).

Physical Plane. The basis for the Physical Plane (PP) of the IN architecture is the *Signalling System no. 7* (SS7) which is the protocol stack (up to the Application Layer of the OSI seven-layer model) that supports the *Intelligent*

Network Application Protocol (INAP). INAP uses the services provided by *Transaction Capabilities Application Part* (TCAP).

At the center of of TCAP (or more precisely of the *Component Sub-layer*) lies the ISO *Remote Operations Service* (ROS) that provides support for distributed objects. ROS describes interactions between distributed objects using the *Abstract Syntax Notation One* (ASN.1) *information object class* (or ASN.1 macros in older versions of the ITU-T recommendations). The *Transaction Sub-layer* is then used to provide a "skinny" end-to-end connection for transporting operations between Functional Entities.

3 TRANSFORMATIONS FROM ACTIONS TO DISTRIBUTED OBJECTS

As mentioned in the introduction, the present paper extends previous work [1]. In this section we give a very brief overview of the previous paper in order to make this contribution more or less self-contained[1].

Thus, in [1] we presented a correctness-preserving transformation from a source language (*SL*) - capturing a centralised view of services - to a target language (*TL*) where distribution is taken into account. Here we provide a reminder of the two languages through an example.

The source language for our transformation is a simple imperative class-based language with actions operating on a global state. The following simple bank account example shows the main features of the language:

```
Account = class owner: string;
                balance: integer;
          endclass

(* the following action transfers a sum equal to amount from the *)
(* provider account to the recipient account *)

transfer(provider:Account; recipient:Account; amount:positive) =
action
case provider.balance < amount: return NACK;
case provider.balance >= amount:
     provider.balance := provider.balance - amount;
     recipient.balance := recipient.balance + amount; return ACK;
endaction
```

[1]Please refer to the previous paper if more detail is needed: Notably there is an emphasis on the formal aspects of the transformation, including the operational semantics of both the source and target languages.

The main features of *SL* is that there are two kinds of classes: *data classes* containing only attributes (represented by the Account class in the example, and *action classes* providing the behaviour (represented by the transfer action). The most prominent feature of the transformation itself is that assignments are mapped into remote method invocations thereby introducing a distribution aspect. The target language is a distributed object-based language which can be compared to Corba compliant languages or to Java enhanced with RMI. To understand the target language and the transformation we now return to the example given in *SL* above after having run it through the transformation. This gives the following program in *TL:*

```
Act_Account(owner, balance) =
    [get_owner(ret) ->
         ret.value(owner);
         become Act_Account(owner, balance),
     get_balance(ret) ->
         ret.value(balance);
         become Act_Account(owner, balance),
     set_owner(new_owner) ->
         become Act_Account(new_owner, balance),
     set_balance(new_balance) ->
         become Act_Account(owner, new_balance)
    ]
Act_Transfer() =
    [transfer(provider, recipient, amount, return) ->
      provider.get_balance(self);
      [value(p_balance) ->
         if p_balance < amount then return.value(NACK)
         else recipient.get_balance(self);
              [value(r_balance) ->
                 provider.set_balance(p_balance-amount);
                 recipient.set_balance(r_balance+amount);
                 return.value(ACK)
              ]
      ]
    ]
```

The target language considers active objects (called *actors* - thus the prefix Act_). The data class has become an object Act_Account where the attributes have been transformed into accessor functions prefixed by get_ and set_. The *action class* has been transformed into another object Act_Transfer. We will not go into any more detail about *TL*, since the language used in this paper, although inspired from *TL*, has a syntax closer to Java.

4 TRANSFORMING SIBS INTO DISTRIBUTED OBJECTS

This section forms the central part the paper. It presents a translation from IN Service Features and SIBs to distributed objects.

4.1 Translating individual SIBs

We have already given an overview of SIBs in section 2.2 where we discussed the GFP of the INCM. Although SIBs appear on several of the INCM planes with differing levels of abstraction we concentrate on the GFP since our contribution is on a *top-down approach* to service creation where distribution aspects is provided by the transformation. So, let us now remind ourselves of the SIB model as presented in the ITU-T CS-1 Recommendations [5] (Q.1213)[2].

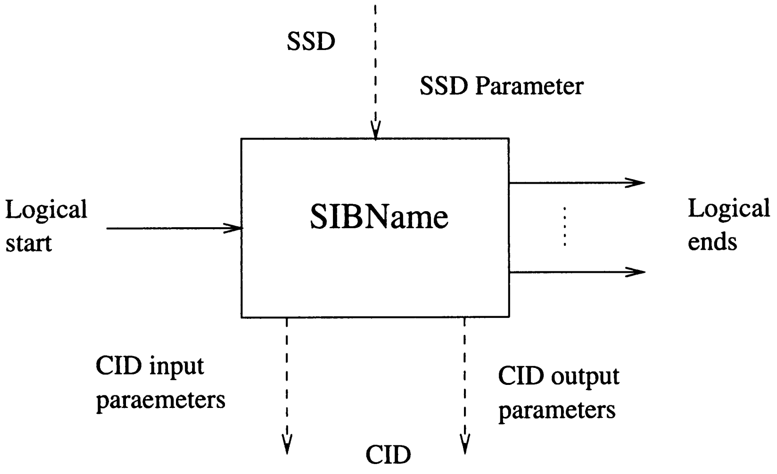


Figure 1 SIB model for the GFP.

Figure 1 shows the graphical representation of CS-1 SIBs. The *CID* (Call Instance Data) parameters describe the dynamic input and output of values to and from the SIB at run-time. To make SIBs usable in several contexts (i.e., in several Service Features) the *SSD* (Service Support Data) are used to specialise a SIB for use in a specific Service Feature.

[2]Later we may extend our work to take into account CS-2 and later capability sets.

So far we have given the impression that only the GFP is used for modelling SIBs. However, to implement SIBs we also need information from the DFP (see section 2.2). The reason is that the SIB concept has been conceived by studying the existing telecommunications network; i.e., it has been created in a bottom-up fashion. It thus is dependent on the underlying network architecture provided by the DFP and PP, notably in that it is implemented using INAP operations between Functional Entities (see section 2.2). Note that so far we have only taken into account the SCF, SDF, and SRF. Also we do not model the BCSM in any way.

Although SIBs could be modeled directly in *SL*, we prefer instead to capture them using UML. As we explain before, UML allows us to easily implement the transformation using the *Objecteering* UML CASE-tool. However, the UML is augmented with constructs from *SL* to provide an *implementation* for the SIB functionality using the *action classes* described in an earlier section.

The capture of SIBs is into two classes: One action class containing the SSD together with a method that implements the SIB from the SCF point of view, and another data class that contains the CID. A relation between these two classes links the static part of the SIB with the dynamic (CID) part. Thus a SIB is represented by a UML class diagram as shown in figure 2.

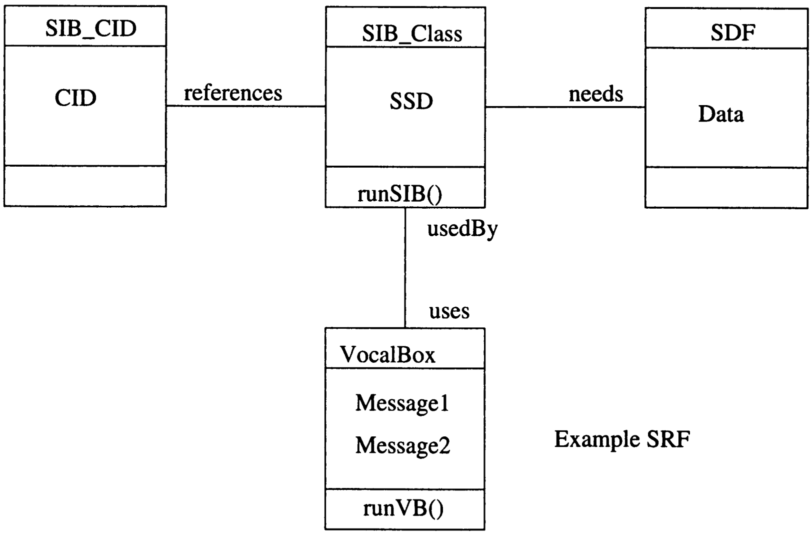
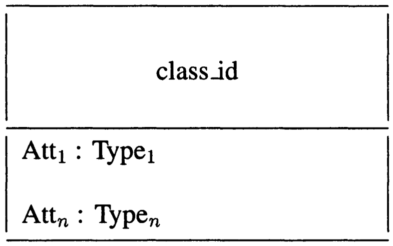


Figure 2 SIB mapped into a UML class diagram.

Figure 2 only shows the pure UML part of our mapping: For the implementation of the SIB functionality (the `runSIB` method in the `SIB_Class`) we use *SL* as comments attached to the `SIB_Class`. We model Service Features as a UML class with a single method called `main` whose implementation is given in *SL*. Since Service Features are modelled as SIB chains, there is a need for a SIB to return a value corresponding to its *logical end.* We do this by adding an attribute called `returnSIB` (of an enumerated type) to the `SIB_Class`.

4.2 Transformation to distributed objects

So far we have shown how SIBs and Service Features are represented in UML class diagrams annotated with *SL* constructs in comments. In this section we show how this new source representation (UML + *SL*) is transformed into a variant of *TL* based on the transformation from [1]. The transformation has been slightly modified to add a `private` section to the transformed class: All the attributes of the UML class are placed in the private section to preserve encapsulation. The accessor methods for these attributes are retained from the original transformation. We also add a constructor with no arguments (for classes without any attributes), as well as a constructor with all attributes as parameters for classes containing data. Thus, the follwing data class containing attributes:


class_id
Att_1 : Type_1
Att_n : Type_n

is transformed into the class below:

```
class class_id
{
 //--------------
 //ATTRIBUTES
 //--------------
 private T[Type_i] att_i;

 //--------------
 //CONSTRUCTORS
 //--------------
 public class_id () {};
 public class_id ( ..., T[Type_i] att_i, ... )
 {
    ...
    this.att_i = att_i;
    ...
  }

  //--------------
  //ACCESSORS
  //--------------
   public T[Type_i] get_att_i ()
   {
     return att_i;
   }
   public set_att_i ( T[Type_i] att_i )
   {
     this.att_i = att_i;
   }
}
```

It is important to note that the notation `T[Type_i]` does *not* represent a construct in the target language: It is the notation we use to show the mapping of types. The same is true for all the transformation rules. It is due to the fact that the transformation is done in stages.

For the remainder of the transformation rules we refer to the appendix. Instead we now go on to show a practical example of what we can do.

4.3 An example: TeleVote

We now turn to an example to give a more intuitive impression of what we can do with our transformations. We use the *TeleVote* Service Feature given in [6] (see the conclusion for a comparison with this work).

Definition. After dialing the service access number for `TeleVote` the voting user receives a vocal message telling him/her to choose between two or more possible choices, each corresponding to a number on the telephone handset. Finally, a new message confirms that the choice has been taken into account.

GFP description. Figure 3 shows how `TeleVote` can be modelled on the GFP of IN CS-1.

User Interaction SIB. Figure 4 shows the UML model for the SIB `User-Interaction` according to our previous description in section 4.1. We do not show the other two SIBs involved in the `TeleVote` Service Feature since they are simpler than the `UserInteraction` SIB (they are purely algorithmic and do not communicate with any other Functional Entitie). For the original

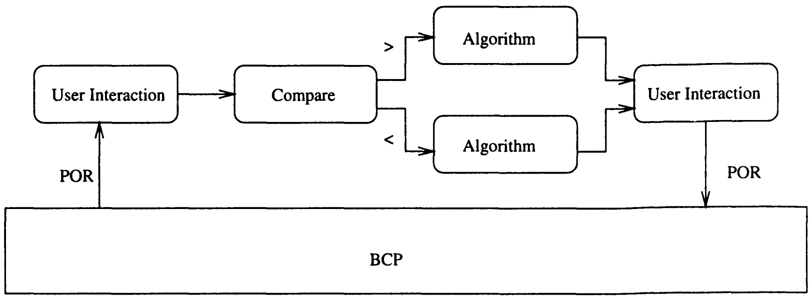


Figure 3 GFP model of TeleVote Service Feature.

description of the SIB we refer to the relevant ITU-T recommendation Q.1213 (the GFP) and Q.1214 (the DFP).

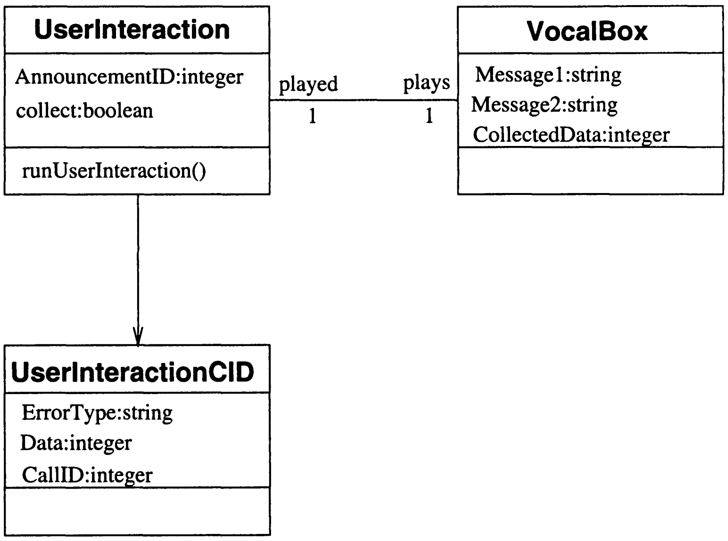


Figure 4 UML class diagram for User Interaction SIB.

The most interesting part of this diagram is the `VocalBox` which is part of the SRF interface. Note that due to limitations in the datatypes provided by *SL*, we do not model all the SSD of the SIB, since we do now have types to handle for instance dates and times used in the INAP operations that implement the communications part of this SIB.

4.4 Implementing the transformation

The translation has been implemented using a UML-tool called *Objecteering*[3]. The tool, in its current version, provides a language (called H[4]) that allows UML-models to be easily mapped into an implementation language.

As shown above, SIBs and Service Features are modelled in UML augmented with structures from the *SL* language. In fact, *SL* constructs appear as comments attached to the UML classes representing SIBs in UML class diagrams. These comments are taken into account by H to allow our previously defined transformation to be used to map SIB classes into *TL*. A trial implementation was conducted by a student project at ENST [4].

5 CONCLUSION

We have shown one possible way to evolve from the world of IN service creation to the one of open object-based distributed computing. This was done by mapping IN CS-1 Service Features and SIBs into a language resembling Java. The transformation is based on previous work on correctness-preserving transformations [1].

The work most closely related to ours is that of Nasreddine et al. [6]. They also map SIBs into classes, but instead of making the SSD a class attribute they provide it as a parameter to each call of the method implementing the SIB. Also, at least in the paper, they do not show how the SIBs are implemented. The biggest advantage of our work is that we provide a transformation from Service Features and SIBs to a language based on distributed objects that have been implemented. Unlike us, they have implemented their work using CORBA (although without taking into account the existing SS7-based network). Finally, a very notable feature of their work is that they aim for methodology for evolving existing algorithmic-based service creation towards object-oriented service creation, taking into account the various actors (with differing skills) that are involved in the service creation process. Instead, we have concentrated on a key issue: how to automate the passage from distribution transparent service specifications to distribution aware implementations.

Clearly, the work presented here is still in its early stages. Our current research focuses on comparing and integrating it with other research efforts on the evolution of the IN architecture. Notably, we want to make it possible for our SIBs to communicate with existing SS7-based Functional Entities. It is important to preserve the existing investment in SSPs; thus, we need a way for call events in the SSPs to trigger Service Features located in a CORBA

[3]Objecteering is a trademark from Softeam.

[4]The name of this language will soon be changed into J because it is moving towards something close to Java.

environment. There are many open questions in this exiting area of research. Both EURESCOM, the OMG and TINA-C have effected interesting work in this area, either through projects (like EURESCOM P508 and P847-GI) or RFIs and RFPs by OMG and TINA-C. Clearly, a pure CORBA environment is not currently suited to the real-time, high-reliability demands of telephony services, although recent extensions to CORBA will alleviate these problems: In particular, *CORBA 3* supports *Asynchronous Messaging* (all INAP operations are asynchronous), *Real-Time*, *Components*, and *Firewalls* (to allow callbacks through firewalls).

Appendix

Transformation rules from SIBs to distributed objects

Relations between classes are transformed as follows (showing only the mapping for one of the two classes in the relation).

```
class Class1
{
  //-----------
  //ATTRIBUTES
  //-----------
  private Class2 role2Class2;

  //-----------
  //ACCESSORS
  //-----------
  public get_role2Class2 ()
  {
    return role2Class2;
  }
  public set_role2Class2 ( Class2 role2Class2 )
  {
    att_i = new_value;
  }
}
```

Note that we use the names of roles played by each class in the relation together with the class name to construct a unique name for the reference to the other class.

Finally, the transformation of actions given below closely follows the one in [1]. (We do not give an explanation for the cases where the transformation is straightforward.)

Action header.

```
T [ Action ( ..., Parameter_i : Type_i ,...) : Type ]
  ->
public T [ Type ]
     Action ( ... , T [ Type_i ] Parameter_i , ...) ]
```

Unlike our original transformation we do not create a new thread to handle message execution. The reason is that Service Features are based on the sequential execution of SIBs. Synchronising the resulting system if threads were used would be very complex indeed.

Action body.

```
T [ Declaration   -> T [ Declaration ]
    beginaction      T [ Statement ]
    Statements
    endaction ]
```

The transformation is direct.

Variable declarations.

```
T [ Decl1Decl2 ]  -> T [ Decl1 ]
                     T [ Decl2 ]

T[ var : Type ]   -> T [ Type ] var;
```

The transformation is direct.

Statements. For statements we consider the cases separately.

```
T [ var:= Exp; ]        -> var = T [ Exp ];
T [ var.att := Exp; ] -> var.set_att( T [ Exp ] );
T [ if Exp then         -> if T [ Exp ]
        S1                 {
    else S2                   T [ S1 ]
    endif; ]               }
                           else
                           {
                              T [ S2 ]
                           }
T [ return Exp; ]       -> returnClassName = T [ Exp ];
T [ S1S2 ]              -> T [ S1 ] T [ S2 ]
```

We note that for assignments to an attribute we must use the generated accessor methods.

In the case of `FunctionCall` we get the following transformation.

```
T [ Object.Method ( ..., Exp, ...); ]
  -> Object.Method ( ..., T [ Exp ], ...) ;
T [ print(..., Exp, ...); ]
  -> System.out.println ( ... + T [ Exp], ... );
T [ var := read() ]
  ->
var = Integer.parseInt((new java.io.BufferedReader
(new java.io.InputStreamReader(System.in))).readLine());
```

Expressions.

```
T[c]        -> c;
T[var]      -> var;
T[var.att] -> var.get_att;
T[nil]      -> null;

T [ new class_id ( ..., Exp_simple, ...) ]
  -> new class_id( ..., T [ Exp_simple ], ... );
T [ operation_id ( ..., Exp_simple, ...) ]
  -> operation_id( ..., T [ Exp_simple ], ... );
T [ action_id ( ..., Exp_simple, ...) ]
  -> action_id( ..., T [ Exp_simple ], ... );
```

Types. Class names are conserved.

```
T[class_id] -> class_id;
```

For the base types we obtain the following mapping.

SL	***TL***
integer	long
real	double
boolean	boolean
string	String

References

[1] Cinzia Bernardeschi, Joubine Dustzadeh, Alessandro Fantechi, Elie Najm, Abdelkrim Nimour, and Frank Olsen. Consistent semantics and correct

transformations for the ODP information and computational models. In *Proceedings of Second IFIP conference on Formal Methods for Open Object-based Distributed Systems - FMOODS'97*. Canterbury UK, Chapman & Hall, July 1997.

[2] I. G. Dufour, editor. *Network Intelligence*, volume 10 of *BT Telecommunications Series*. Chapman & Hall, first edition, 1997.

[3] ITU-T. *Intelligent Network: Q.1200-Series Intelligent Network Recommendation*. Number Q.1200. International Telecommunication Union Standardization Sector, Geneva, October 1995.

[4] J. Gastaud. *Conception de SIB (Service Independent Blocks) pour les Réseaux Intelligents*. Mémoire de fin d'études - ENST - Juillet 1999.

[5] ITU-T. *Intelligent Network: Q.1210-Series Intelligent Network Recommendation Structure*. Number Q.1210. International Telecommunication Union Standardization Sector, Geneva, October 1995.

[6] Hassan Nasreddine, Anisse Idir, and Simon Znaty. Moore: Méthode et outils orienté-objet pour la création de service reseau intelligent. In *GRES'99*, 1999.