FORHAN Pascal
Institut National des Télécommunications
IG3

LOTOS Specification of the Point to Multipoint Group Call Service from the General Packet Radio Service on Global System for Mobile Communications.

Supervisors: L. Logrippo for the University of Ottawa
A. Cavalli for the INT

University of Ottawa
Université d’Ottawa

February-June 1997

Abstract:

The purpose of this work is to design a specification in LOTOS of the Point to Multipoint Group Call (PTM-G) service of the General Packet Radio Service (GPRS) on Global System for Mobile communications (GSM).

In this work, we consider the software level; there is no mention of the physical architecture of the Public Land Mobile Network (PLMN) supporting GPRS, only an abstract architecture is given.

The reference document for this specification is "Digital cellular telecommunications system (Phase 2+); General Packet Radio Service (GPRS); Service description stage 1 (GSM 02.60)" version 2.0.0, November 1996, from the European Telecommunications Standards Institute (ETSI).

We used an approach based on scenarios, described as Use Case Maps, which capture the functionality of the PTM-G service.

This report presents the methodology used, the specification and its verification.

(Written in English)

Résumé:

Le but de ce travail est de réaliser une spécification en LOTOS du service appel de groupe multi-accès (PTM-G) du service de transfert de paquets (GPRS) défini pour le système global de communications avec des téléphones mobiles (GSM).

Dans ce travail, nous considérons le niveau applicatif; il n'est pas fait référence à l'architecture du réseau public terrestre (PLMN) proposant le service GPRS, seule une architecture abstraite est donnée.

Le document de référence de cette spécification est celui de l'Institut Européen de Normalisation des Télécommunications (ETSI): "Digital cellular telecommunications system (Phase 2+); General Packet Radio Service (GPRS); Service description stage 1 (GSM 02.60)" version 2.0.0, November 1996.

Ce travail est basé sur les scénarios d'usage decrivant les fonctionnalités du service PTM-G.

Ce rapport présente la méthodologie appliquée, la spécification et la vérification de la spécification.

(Écrit en anglais)

Table of Contents

Part 1: Introduction

1.1. Motivations and Objectives

The spirit of this work is to use functionality descriptions in order to achieve the realization of a specification. The example chosen was the *Point to Multipoint Group Call* (PTM-G) of the *General Packet Radio Service* (GPRS) on *Global System for Mobile communications* (GSM) to fit the current research and development at the University of Ottawa.

This work follows the research done by Daniel Amyot for his Ph. D thesis [ALB 97]. It uses *Use Case Maps* (UCM) defined by Buhr and Casselman [BC 95] to describe the functionality, and the *Language Of Temporal Ordering Specification* (LOTOS) [ISO 88] to formally describe the specification.

We validate the specification by using a test suite derived from the scenarios. We use two main tools to simulate and test the specification: ELUDO (Environnement LOTOS de l'Université d'Ottawa) and LOLA (LOtos LAboratory, developed at the University of Madrid) [PL 90].

The main interest of a scenario-based design is to detect, as soon as possible, the errors and the defaults in the system. Once the specification is validated against the scenarios (and thus, against the requirements), a prototype can be generated and the potential errors can be detected. The use of scenarios immediately allows the definition of a logical denomination for the most used terms. This familiarizes the reader with the terms used in the specification and with the principles of the specification. So, this simplifies the comprehension. The UCM also give a good documentation for the specification.

This work is divided in five parts. The first part introduces the subject, gives an brief overview of LOTOS, the main principles of GPRS, an overview of the UCM notation and a presentation of the methodology we used. The second part presents the Group Call service and the scenarios. The third part explains the specification. The fourth part gives the test cases and discusses the verification of the specification. The fifth part contains the conclusions of this work.

1.2. A Brief Overview of LOTOS

The goal of this section is not to make an exhaustive presentation of LOTOS [ISO 88], but only to give the principles of this language used in the work.

A. LOTOS is a FDT

LOTOS, *Language Of Temporal Ordering Specification* is a *Formal Description Technique* (FDT) developed in the eighties within the *International Organization for Standardization* (ISO). The FDT are used to design and test systems before getting to the real implementation. FDT also allow the rapid realization of prototypes.

LOTOS is designed for open distributed systems. The language is based on the idea that systems can be specified by defining the **temporal relation** among the interactions that constitute the externally observable behavior.

Specifications in LOTOS are, like with the other FDT, unambiguous, precise, complete and implementation independent descriptions.

In LOTOS, there is a clear separation between the data part and the control part. The control part is based on communicant and concurrent processes (*Calculus of Communicating Systems*, CCS [Mil 80], and *Communicating Sequential Processes*, CSP [Hoa 85]). The data part is based on the abstract data type language ACT ONE [EM 85].

B. LOTOS Data Structures

The *Abstract Data Types* (ADT) constitute the basis of the data structures in LOTOS. This method of description is consistent with the will of abstraction in LOTOS, but after less than ten years of use, some people grumble against ADT. In the forthcoming E-LOTOS, it seems that this method will be reviewed. In this work, we have used ADT as described in the 1988 standard.

In LOTOS, a **type** has a name, a list of inherited types, data carriers, called **sorts**, which describe the field of values of the type, a list of **operators** with their signature, and a list of **equations** which describe the behavior of the operators. The operators can be overloaded if their signature is different.
For example, here is the type C_ID:

```
type C_ID is NaturalNumber
 sorts C_ID
 opns
      NoC_ID: -> C_ID
      NatNum: C_ID -> Nat
      Succ: C_ID -> C_ID
      _eq_, _ne_: C_ID, C_ID -> Bool
 eqns
      forall cid, cid2: C_ID
      ofsort Nat
      NatNum(NoC_id) = 0;
      NatNum(Succ(cid)) = Succ(NatNum(cid));
      ofsort Bool
      cid eq cid2 = NatNum(cid) eq NatNum(cid2);
      cid ne cid2 = Not( cid eq cid2);
endtype
```

In this type we have:

- the name, **C_ID**, which means Call IDentifier
- the list of inherited types, **NaturalNumber** (from the International Standard library)
- the data carrier, **C_ID** (same as type name, to reduce the number of terms)
- a list of operators with their signature, **NoC_ID**, a nullary operator; **Succ** and **NatNum**, two unary operators; **eq** and **ne**, two binary operators
- a list of equations describing their behavior.

Equations can have a guard to be more specific about the domain of application of an equation:

```
x Notin s => Card(Insert(x,s)) = Succ(Card(s))
```

In this example, **x** is an element and **s** a list. This equation means that if **x** is not in **s**, then the number of elements of **s** <u>including</u> **x** is the number of element of **s** plus one.

The second possibility to describe a type is to rename another one. This feature is useful when we want to use an already defined data type, without any changes in the semantics. For example, it is more comprehensible to use renamed hexadecimal numbers than to use three or four times the **HexDigit** type for different data types.

```
type M_ID is HexDigit renamedby
 sortnames M_ID for Hexdigit
 opnnames
       NoM_ID for 0
       Adm1 for 1
       Init1 for 2
       Init2 for 3
       Mem1 for 4
       Mem2 for 5
       Mem3 for 6
       Mem4 for 7
       Adm2 for 8
       Init3 for 9
       Init4 for A
       Mem5 for B
       Mem6 for C
       Mem7 for D
       Mem8 for E
       Mem9 for F
endtype
```

This type, which describes the Member IDentifiers (**M_ID**), is a typical use of renaming. We use the type **HexDigit** from the International Standard library. We give new names for operations that interest us (in this example all the constant operations, i.e., nullary operations).

We can also describe generic types and instantiate them with new types that we want to define. This possibility is useful, for instance, to create lists of any types. These generic types are signaled by the term **formal**.

```
type Fboolean is
formalsorts FBool
formalopns    true :-> FBool
              not : FBool-> FBool
formaleqns
       forall x: FBool
       ofsort FBool
       not(not(x))=x;
endtype

type Statusinfolist0 is Set
actualizedby Statusinfotype using
```

```
sortnames
       Statusinfo for Element
       Bool for FBool
endtype
```

These two types present a use of the generic possibility. The type **Fboolean**, which represents the type **Boolean** in generic, is used in the definition of a type **Set**. The type **Statuslistinfolist0** represents the instantiation of the type **Set** with the formal types defined by **Statusinfo** and **Bool**.

We will explain in detail further in this work all the types used in the specification. Now we will present the control part of LOTOS.

C. LOTOS Control Structures

We present the constructors used in the specification. Basically, in LOTOS, a distributed and concurrent system is a *process,* possibly constituted of several sub-processes. Each sub-process is a process itself. Therefore, we

	Name	*Behavior Expression*	*Comment*
Basic Behavior Expressions	Inaction	**stop**	Cannot engage any action (deadlock).
	Successful Termination	**exit**	Indicates that a process has successfully performed its actions.
	Process Instantiation	$\texttt{ProcName}[g_1, \ldots, g_n]$	Creates a new instance of the process.
Basic Operators	Action Prefix	a;B	Used to prefix a behavior expression, B, with an action, a. There exists a special action, **i**, that a process can execute independently.
	Choice	B_1 **[]** B_2	Allows the user to define alternatives for a given process.
Enabling/ Disabling	Enabling	B_1 **>>** B_2	Used to sequence two behavior expressions. B_1 has to **exit** for B_2 to be executed.
	Disabling	B_1 **[>** B_2	Used to interrupt B_1 if B_2 has to be executed during normal functioning.
Composition	Parallel Composition	B_1 **|[**$g_1, \ldots, g_n$**]|** B_2	B_1 and B_2 behave independently except for the gates g_1, ..., g_n where B_1 and B_2 must synchronize.
	Interleaving	B_1 **|||** B_2	B_1 and B_2 behave independently.
	Full Synchronization	B_1 **||** B_2	B_1 and B_2 must synchronize on all their visible gates
Others Operators	Hiding	**Hide** $g_1, \ldots, g_n$ **in** B	Used to hide gates inside a behavior expression. The actions on these gates cannot synchronize with the environment
	Guarded Behavior	**[** P **] ->** B	B can be executed only if P is true.
	Local Definition	**Let** x:s = E **in** B	Substitutes a value expression (E) by an identifier (x) of sort s in B

Figure 1: Main LOTOS constructors

have a hierarchy of process definitions in a LOTOS specification. Processes have a *behavior expression* that describes the behavior of the process. Processes can communicate with their environment by the way of *gates*. Two, or more, processes can share information on a gate; they have to synchronize themselves to succeed in this action.

Such synchronization is called a *multiway rendezvous*; the processes cannot continue their behavior before the synchronization takes place. A rendezvous can have a lot of parameters (called *offers*), and two or more processes can synchronize themselves on them only when the parameters fit all the offers, and when the name of the gate is the same. For example:

```
ch_ID !Succ(NoC_ID) (* where Succ(NoC_ID) is of the type C_ID *)
```

and

```
ch_ID ?n:C_ID
```

can be synchronized, the name of the gate is **ch_ID** and we have the offer of one parameter. “!” means that the parameter must be a specific value for the synchronization to occur. “?” means that the parameter could be any value of the data carrier to synchronize.

There also exist internal gates hidden for the environment of the process.

The operators are recalled in figure 1, where **a** is an action, $\mathbf{B_i}$ are behavior expressions, $\mathbf{g_i}$ are gates and **P** is a predicate.

More operators exist, but they are not used in the specification.

The next section presents the approach chosen and specifically gives an idea of the Use Case Maps notation.

1.3. Use Case Maps, a Notation to Describe Functionalities

Recently, we have noticed the emergence of different approaches driven by use cases. This term, *use case*, covers different meaning according to the creators of these methods. The first step is to define the meaning of use cases. The second one is to present the notation used to describe the scenarios.

In this work, we will follow the definition of Buhr and Casselman [BC 95]. A scenario is a *sequence* of internal or external *responsibilities*. These responsibilities are linked by a causal relationship. With the chosen notation, we can describe functionalities at a high level. We do not have to commit too early on the architecture and the exchange of messages between the different entities. Moreover, this notation allows the definition of scenarios and their binding to different architectures without redefining the scenarios. In this way, we can define UCM for a complete sequence of activities of a system, from end to end. In this perspective, the UCM notation is not component-oriented.

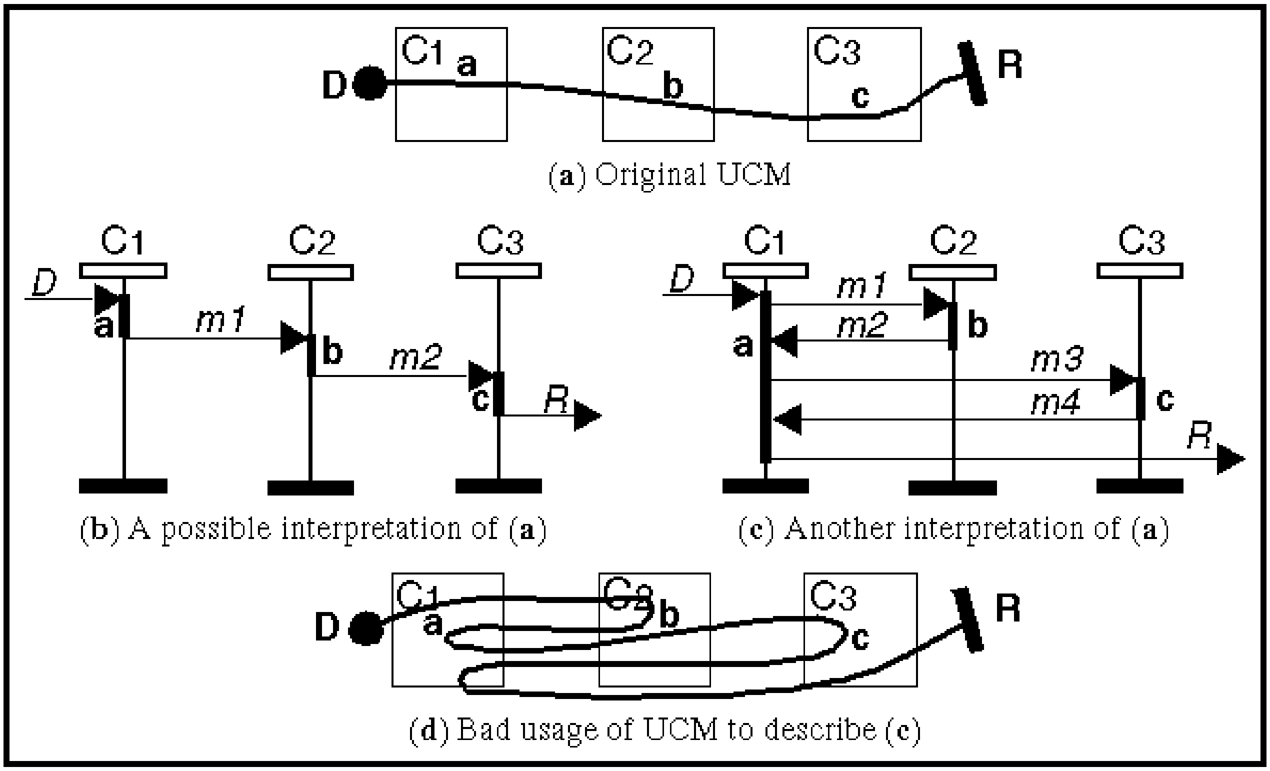

Figure 2: Use Case Maps: Causality and exchange of messages

Figure 2(a) presents a basic UCM: there is only one path that links a triggering event (**D**) to a resulting event (**R**). **D** launches the activity **a** in component $\mathbf{C}_1$. Then, this launches the activity **b** in component $\mathbf{C}_2$, then the activity **c** in component $\mathbf{C}_3$, and at last the resulting event **R**. The triggering events are represented by filled circles and the resulting events by bars, and the path is represented by a line. Of course, other possibilities are defined to allow almost all the actions possible; the same triggering event can have multiple resulting events, parallelism and choices are possible, we can describe waiting places and compose UCM to trigger other UCM. Figure 2(a) suggests that we can have another architecture behind the UCM: for example we could have decided that $\mathbf{C}_1$ and $\mathbf{C}_2$ were the same component; activity **a** and **b** would be in the same component, but the scenario would not have changed.

Figure 2 also presents interpretations of the UCM in terms of exchanges of messages. Figure 2(b) presents the most simple explanation of the UCM represented in 2(a): **D** launches **a** then a message **m1** is sent to $\mathbf{C}_2$ to trigger **b** off; at the end of activity **b** a second message, **m2**, is sent to $\mathbf{C}_3$ to launch **c**; at the end **R** is output. Figure 2(c) represents another totally different interpretation, in terms of exchanges of messages. The main control is done by component $\mathbf{C}_1$; at the end of each activity in $\mathbf{C}_2$ and $\mathbf{C}_3$, a message is sent back to $\mathbf{C}_1$ and the activity **a**. Figure 2(d) shows the straightforward interpretation of 2(c), but it is a bad use of UCM, because it commits too soon to the exchange of messages between the underlying components.

So this figure shows clearly the interest of this notation: we are not constrained by the messages, we can define the sequences of activities and then put these sequences on different architectures, and later choose the type and the number of messages needed (protocol).

This notation allows for a great reusability of the scenarios and a high level of description of the functionalities. Now we will present the GPRS.

1.4. The GPRS: A Service for Data with Mobile Communications

The GPRS allows the service subscriber to send and receive data in an end-to-end packet transfer mode, without using network resources in a switched mode. This service is a set of GSM bearer services that provides packet transfer in interworks with external networks and within the PLMN. This service is defined for applications with one or more of the following characteristics: intermittent, non periodic transmissions with a delay between two transmissions greatly superior to the average of the transfer delay; frequent transmissions of small volumes (typically smaller than 500 octets), with a rate of several transfers per minutes; infrequent transmissions of greater volumes, with a rate of several per hour.

The service is divided in two main branches: the *Point-To-Point* (PTP) and the *Point-To-Multipoint* (PTM) based on existing and standardized network protocols.

Typical applications for PTP services are: retrieval services, like *World Wide Web* (WWW); messaging services to individual users, like a mailbox; real time (no store and forward) conversations, like Internet’s Telnet application; short tele-action, like credit-card validation.

Typical applications for PTM services are: unidirectional distribution services, like news services; bi-directional dispatching services, like taxi fleet services; multi-directional conferencing services, without store and forward, between multiple users.

The PTM services have some specific capabilities including geographical routing to restrict distribution and scheduled delivery capability. The PTM service is divided in three different services:

- The *PTM-Multicast* (PTM-M) allows transmissions of data to specific users groups or to **all subscribers** within a geographical area. No knowledge of the number of participants is required at any point in time.
- The *PTM-Group Call* (PTM-G), purpose of this work, allows transmissions to specific groups of users. At any point in time, the network has the knowledge of the number of users and their location.
- The *IP Multicast* (IP-M), defined as a part of *Internet Protocol* (IP) suite. In this service, the messages are sent between participants of an IP-M group. This kind of group can be internal to the PLMN or distributed across the Internet.

We will present later with more detail the PTM-G service. The next section focuses on the methodology used in its specification.

1.5. Methodology: A Global Approach of the Development

The UCM, with the scenarios-based concept, seems to be a suitable bridge between the informal requirements and the formal design of the system. A good design approach takes care of the possible changes in the requirements, so the approach chosen is iterative, or more specifically in spiral with incremental refinements. It is an approach basically derived from the prototype development cycle.

Figure 3 presents this methodology. In a first step, we generate UCM from requirements, in parallel and *possibly independently* we select the underlying architecture. The second step consists in deriving test cases from the UCM, resulting in a test suite. In parallel, we superimpose the UCM and the architecture, thus we can synthesize a LOTOS prototype. We have to make some adjustments on the tests to fit the chosen architecture (the order of some exchanges of messages for example, typically to send data to a lot of members, in a test, we have to know the exact order of this operation).

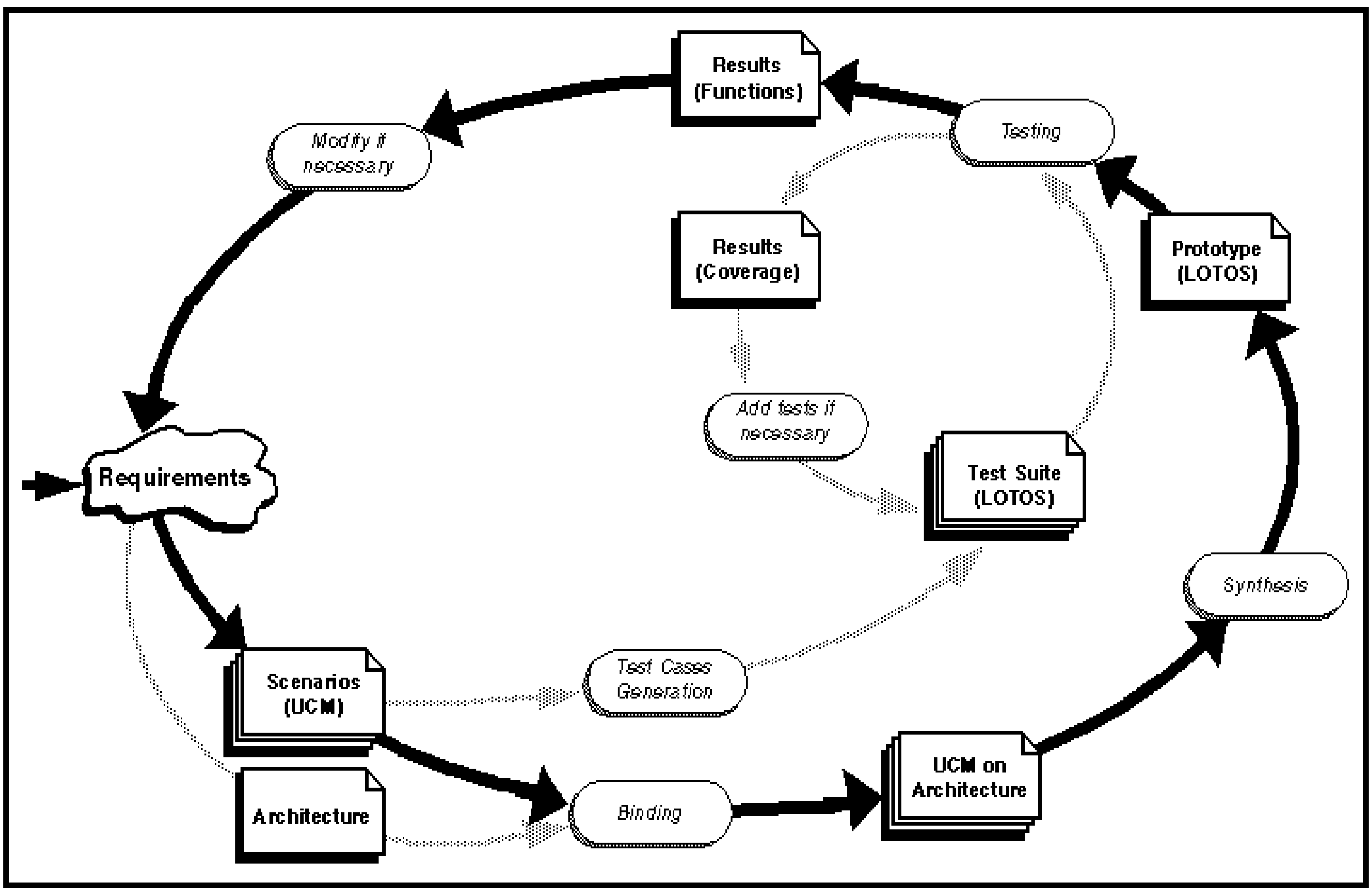

Figure 3: Methodology of conception

At this point, we can test the prototype, check the coverage of the tests, and eventually creates new tests to ensure the entire coverage of the prototype. At the end of the tests, we can take care of new requirements, we have completed the cycle.

The advantages of this method are:

- The separation of the architecture from the functionalities. The scenarios described by the UCM are highly reusable and we can change the architecture if this does not fit as well as possible.
- We can rapidly generate prototypes to verify the requirements. With the UCM and a documented architecture, a prototype can be easily and rapidly generated.
- We can almost mechanically generate test cases.
- The documentation is generated during the development. The UCM and the architecture contain documentation themselves. The comprehension and the explanation are simplified with this method.

Now, with this methodology in mind, we will discuss the first step. We will present the requirements of the GPRS PTM-G and the UCM created to fit them.

Part 2: Generation of the Specification

2.1. Requirements and Assumptions

All the requirements used in this work are described in [ETSI 96]. The PTM-G service is characterized by the transmission of messages between subsets of subscribers within a defined geographical area. More specifically, a message is sent throughout a geographical area defined by the sender. The messages are addressed to a specific group of users. The network may not have knowledge of which participants have received the message, but the network always knows which participants are present within the geographical area. The message delivery is in real-time; this means that the network transfers the messages as soon as possible following the receipt of a service request. The messages are sent to members who have **explicitly** joined the call by registering with the network. The messages are sent with confidentiality, provided by a ciphering of the transmissions.

The most important requirement defined above is the geographical area. In [ETSI 96], geographical areas are organized in hierarchical structures with areas, sub-areas and sets of areas. In this work, we assume that there are only two levels in the hierarchy: the level of the areas, and one general area that does not limit the geographical area (**NoGeozone**). This allows a distribution throughout the PLMN.

The groups of subscribers are addressed by an *International Mobile Group Identity* (IMGI). The subscribers have to register to the groups to which they want to connect. The registering operation is done towards the provider, or towards a specific type of subscriber called *controller*. A controller can create or delete groups and control the membership. These types of operation are not considered in the specification. The specification is more transmission-oriented. The two others types of subscribers are the *initiators*, who can create and delete calls, and the *participants*, who are only allowed to receive and eventually send messages. We assume that these three types of subscribers have basically more privileges at each level: a participant has basics privileges (receive and send messages, if the transfer mode allows it); the initiator has the privileges of the participants in addition with his specifics privileges (create and delete calls and choose the call parameters); the controller has the privilege of the initiator in addition to membership control and group administration.

A group can be *close* (with a pre-defined set of members) or *open* (anyone can attempt to connect), but we have not considered this possibility, because if we automatically register the open groups for each user, we have in fact open groups and closed ones.

The message transfer is by default unidirectional (*one-way*), only the initiator can send messages; but we can choose a bi-directional transmission (*two-way*), the initiator sends messages to all participants and each participant can send messages to the initiator; in fact, the initiator is the moderator of the call. We can also choose a multi-directional transmission (*N-way*), where anyone can send messages to the entire group without any restrictions. These three types of communication are fully implemented in the specification.

For each call, some attributes can be configured, the initiator is in charge of these arrangements. The attributes are:

- The group call notification, used to inform the participants of the creation of a new call.
- The group call notification repetition rate, used to repeat the group call notification. This attribute is not implemented because it depends on time and LOTOS cannot specify time.
- The join/leave call indication, used to inform initiator or participants to the connection or the disconnection of a participant.
- The geographical area, the quality of service.

Other restrictions could be defined, but they are not explicit in [ETSI 96]. We will only consider the restrictions from the standard.

Six operations are defined to implement the PTM-G service:

- The *call initiate* operation: to create a call.
- The *call terminate* operation: to delete a call.
- The *call status* operation: to get the attributes of a call or a group.
- The *join call* operation: to join an existing call.
- The *leave* operation: to quit a joined call
- The *data transfer* operation: to send messages and data.

The **terminate**, **leave** and **status** operations can be invoked by the network itself. For the **status** operation, the reasons for this invocation are not clear. To generate the **terminate** and **leave** operations invoked by the network, we defined three new operations. Two of them are located in the underlying services: the *attach GPRS* and *detach GPRS* operations; the third emulates the routing operation: the *change zone* operation.

The **attach GPRS** operation allows to a subscriber to report his presence to the network in order to use the GPRS resources. The **detach GPRS** operation is the opposite; the subscriber indicates that he no longer wants to use GPRS resources. When a participant leaves the geographical area defined for a call, he has to leave the call, so the network generates automatically a **leave call** operation. Eventually, if the participant is the initiator, the call is terminated.

We will return on each operation with more details when we will present the UCM. Now we will show the components of the architecture chosen to realize the network.

2.2. Architecture of the PLMN

Once the UCM are defined, we have to superimpose them on an architecture compliant with the requirements. The UCM are graphically presented and superimposed on an architecture, so it seems judicious to present the architecture first.

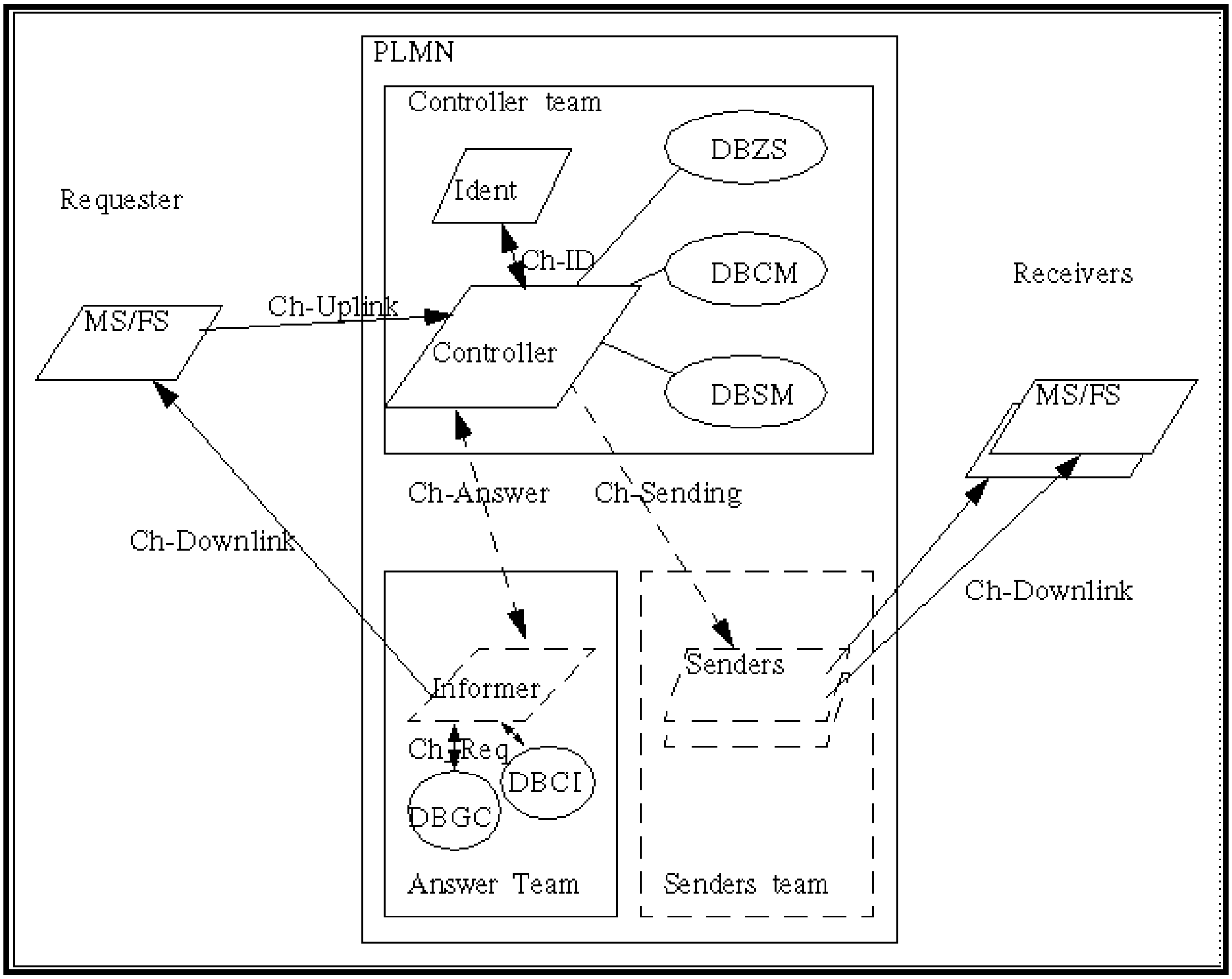

Figure 4: Architecture of the GPRS service

Figure 4 presents the chosen architecture for the PLMN. It shows several kinds of objects: *active processes* as parallelograms, *passive objects*, such as databases, as ellipses, and containers (*teams*) are represented by rectangles. The links between the components are represented by the arrows. The components represented by dotted lines are dynamically instantiated.

We have chosen an architecture slightly distributed but not excessively and not monolithic. We have defined three main teams in the PLMN, each with a specific role. The *controller team* contains the mechanisms of control and reception of requests. The *answer team* sends to the requester the answers to its requests. The *sender team* sends to the other participants, called *receivers* in this case, the indication generated by the requests (e.g., indication of leaving a call...).

The requester and the receivers are *Mobile Station* or eventually *Fixed Station* (MS/FS) because the GPRS can be supported for connections with non-mobile stations.

The passive components are databases. These databases store the useful information for the smooth running of the network.

- The first database stores all the information related to the users: the *Status-Member DataBase* (DBSM). This database has a key, which is the *Member IDentifier* (M_ID) of each member.
- The second one stores the lists of stations by geographical areas: the *geographical Zone-Stations DataBase* (DBZS). The key of this database is the *geographical zone* (GeoZone) number of each zone.
- The third one stores lists of members joined on each call: the *Call-Member DataBase* (DBCM). Its key is the *Call-IDentifier* (C_ID) of each call.
- The fourth database stores the information related to each call: the *Call-Information DataBase* (DBCI). Its key is the C_ID of each call.
- The fifth one stores the lists of calls for each group: the *Group-Call DataBase* (DBGC). The key is the IMGI of each group.

The three first databases are parts of the **Controller-team**. The two following databases are part of the **Answer-team**. The reason of this difference is the process that uses principally the databases: **Controller** for the three first ones and **Informer** for the two last ones.

The active objects are the processes. We distinguish two processes in **Controller-team**:

. The process *Ident*. This process supplies a unique C_ID at each request of the **Controller**

. The process *Controller*. This is the spinal column of the architecture, it manages the authentication, deals with the main part of the requests, and dispatches the appropriate answers to **Answer-team** and **Senders-teams**.

In the **Answer-team**, we have the process *Informer*. Its role is to send to the requester the answer needed to his request. Eventually, this process completes the actions started in the **Controller**.

In the **Senders-team**, we have a multiple instantiated process called *Senders*. This process sends to a receiver the indication provided by the **Controller**. It is only a transmission process; there is no processing done within it.

The last type of objects to study is the *channel*. We have defined two channels between the MS/FS and the PLMN. This is compliant with the GSM standard. In this standard, two different unidirectional channels are used, one for the communications from the Mobile Station to the PLMN (*Ch-Uplink*) and one from the PLMN to the MS (*Ch-Downlink*). We have defined four internal channels for the communications within the PLMN:

. *Ch_ID*: bi-directional channel between **Controller** and **Ident**.

. *Ch_Answer*: bi-directional channel to provide the communication between **Controller-team** and **Answer-team**.

. *Ch_Sending*: unidirectional channel to provide the transmission of information from **Controller** to the senders.

- *Ch_Req*: a multi-directional channel to ensure the communication between **Informer** and the two databases **DBGC** and **DBCI**.

Now, we can present the UCM.

2.3. The UCM of the Functionalities

In these figures, the asterisks point out the objects used in the PLMN.

The first UCM, in figure 5, describes the **Attach GPRS** request.

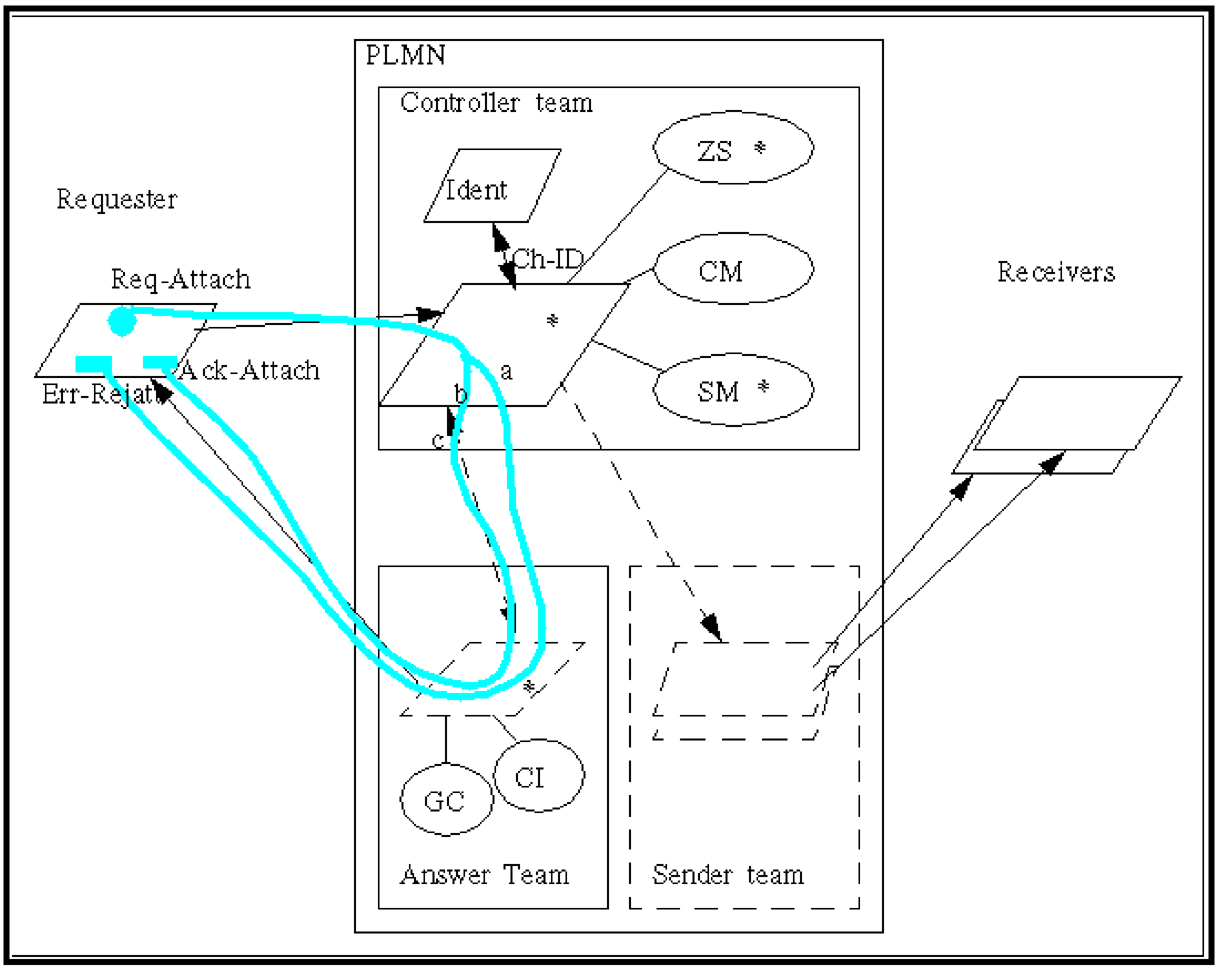

Figure 5: Attach GPRS request

This scenario provides the connection to the GPRS service. Hence, the PLMN has the knowledge of the requester's will to use the GPRS services.

There are only two possibilities of answer to this request, the acceptance and the rejection. We see that in this scenario, the senders-team is not involved. If the request is accepted, the requester is connected to the GPRS service and can use the GPRS resources. Its status is set up to ATTACHED and his current location is stored in DBZS.

Activity	Type	Information	Comments
Req_Attach	Request	M_ID, GeoZone	Request for attachment to GPRS service
Ack_Attach	Acknowledgment	M_ID	Request accepted, status ATTACHED
Err_Rejatt	Error	M_ID, Cause	Request rejected (requester does not have rights to connect, network overload...)
a	Condition		Network can accept the request? No
b	Condition		Network can accept the request? Yes
c	Internal		Updating BDZS, BDSM

The second scenario, shown in figure 6, is the **Detach GPRS** request.

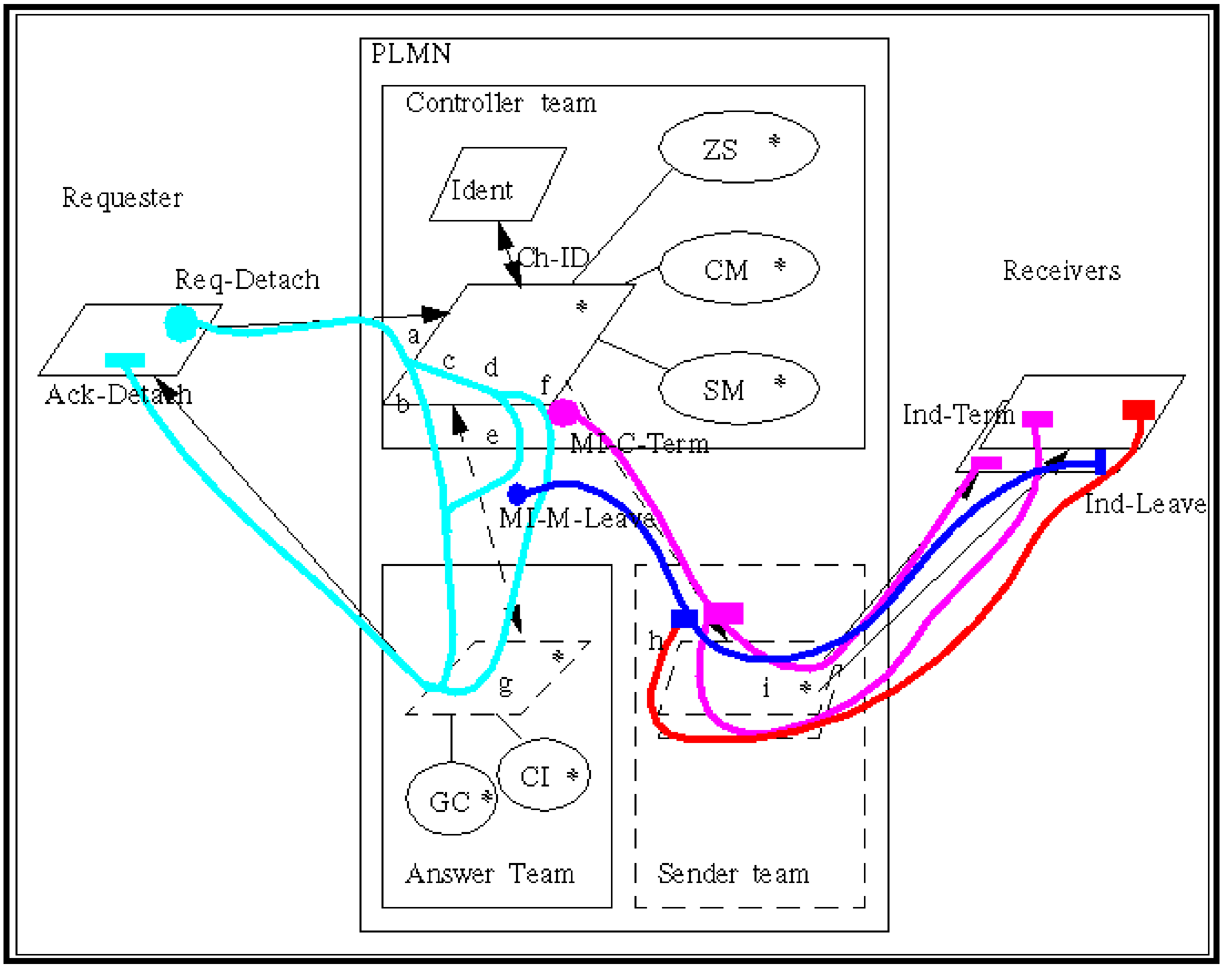

Figure 6: Detach GPRS request

This scenario is more complicated as it involves almost everything in the PLMN. This scenario corresponds to the disconnection from the GPRS service, this disconnection is seen as a physical disconnection (*off* button). so the network cannot reject the request. Moreover the request cannot be masked by the network. Following this request, the requester is DETACHED and he has to attach once again to use GPRS services.

Activity	Type	Information	Comments
Req_Detach	Request	M_ID	Request of detachment, requesters attached
Ack_Detach	Acknowledgment	M_ID	Indication of detachment, status DETACHED
MI_C_Term	Internal Message	C_ID,Cause	Internal indication of termination of a call
Ind_Term	Indication	C_ID, Cause	Indication of termination to the members
MI_M_Leave	Internal Message	C_ID, M_ID	Internal indication of leaving a call
Ind_Leave	Indication	C_ID, M_ID	Indication of leaving to the members
a	Internal		Updating DBZS and DBSM
b	Condition		Is requester participant of any call? No
c	Condition		Is requester participant of any call? Yes
d	Internal		Updating DBCM
e	Condition		Is requester initiator of any call? No
f	Condition		Is requester initiator of any call? Yes
g	Internal		Updating DBGC and DBCI
h	Condition		Has indication to be sent to all participants? yes
i	Internal		Sending indication

We have to notice that the internal message **MI_M_Leave** (member leave) is guarded by a condition. This message is set up only if the join/leave indication is needed by the call. Moreover, this message can be set up by the two branches **e** and **f** (requester is participant and requester is initiator) because

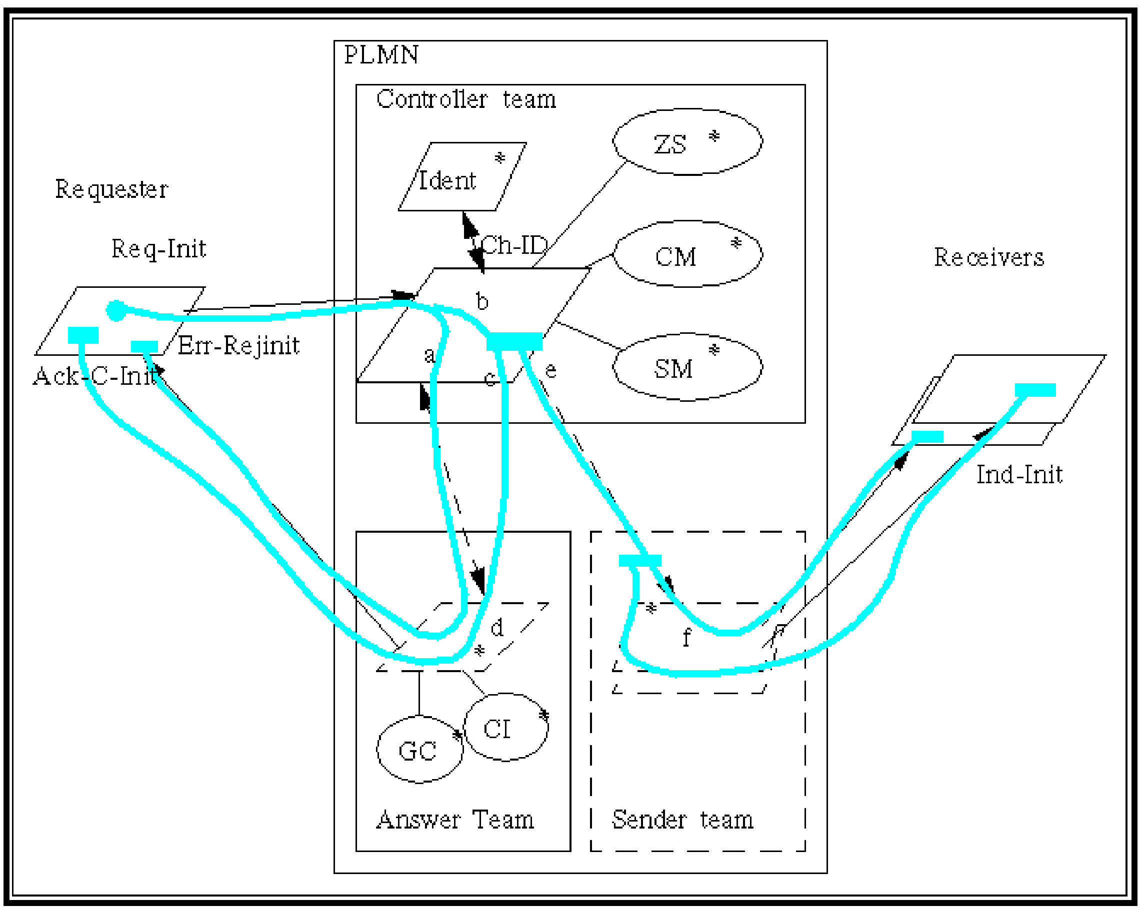

Figure 7: Initiate Call request

a call participant can be initiator of another call. The two internal messages **MI_C_Term** (call termination) and **MI_M_Leave** (member leave) can be set up more than once when needed, when the requester has to leave several calls, for example.

The third scenario is the **Initiate Call** request, shown in figure 7. This scenario is almost completely defined in [ETSI 96].

Activity	Type	Information	Comments
Req_Init	Request	M_ID, IMGI, DTM, QoS, GeoZone, join_leave, send_to_all, call_not	Request of creation of a new call, requester attached
Err_Rejinit	Error	M_ID, IMGI, Cause	Request rejected (wrong GeoZone, wrong IMGI,...)
Ack_C_Init	Acknowledgment	IMGI, C_ID, Cipher_Key	Acknowledgement of call creation
Ind_Init	Indication	IMGI, C_ID	indication of creation of a call
a	Condition		Request agreed? No
b	Condition		Request agreed? Yes
c	Internal		Updating DBCM, DBSM
d	Internal		Updating DBGC, DBCI
e	Condition		Call Notification needed? yes
f	Internal		Sending notification

This scenario provides the creation of a new call. Two answers are possible, the acceptance and the rejection. In case of acceptance, the indication **Ind_Init** (indication of Initiation of a call) is sent if the attributes defined for the call set the initiate call notification to yes. The different attributes negotiated during this request are: the *Data Transfer Mode* (DTM, one-way, two-way or N-way), the quality of service (QoS, three types defined in [ETSI 96]), the geographical area, the join/leave indication (to know if someone join or leave a call), and the initiate call notification.

A last attribute is negotiated; this one is not defined in [ETSI 96] but it seems natural to foresee it: the *send-to-all* possibility. This attribute allows the sending to all members of the join/leave indication, in [ETSI 96], the initiator is the only one who receives the join/leave indication.

The main reasons to reject an initiate call request are:

- The geographical zone defined does not correspond to the current location of the initiator, so the initiator cannot be a participant to hie own call, this is not allowed by [ETSI 96].
- The IMGI requested for the call creation is not valid with respect to the list of IMGI to which the initiator has rights to initiate call.
- The requester is not an initiator, he does not have the rights to initiate a call.

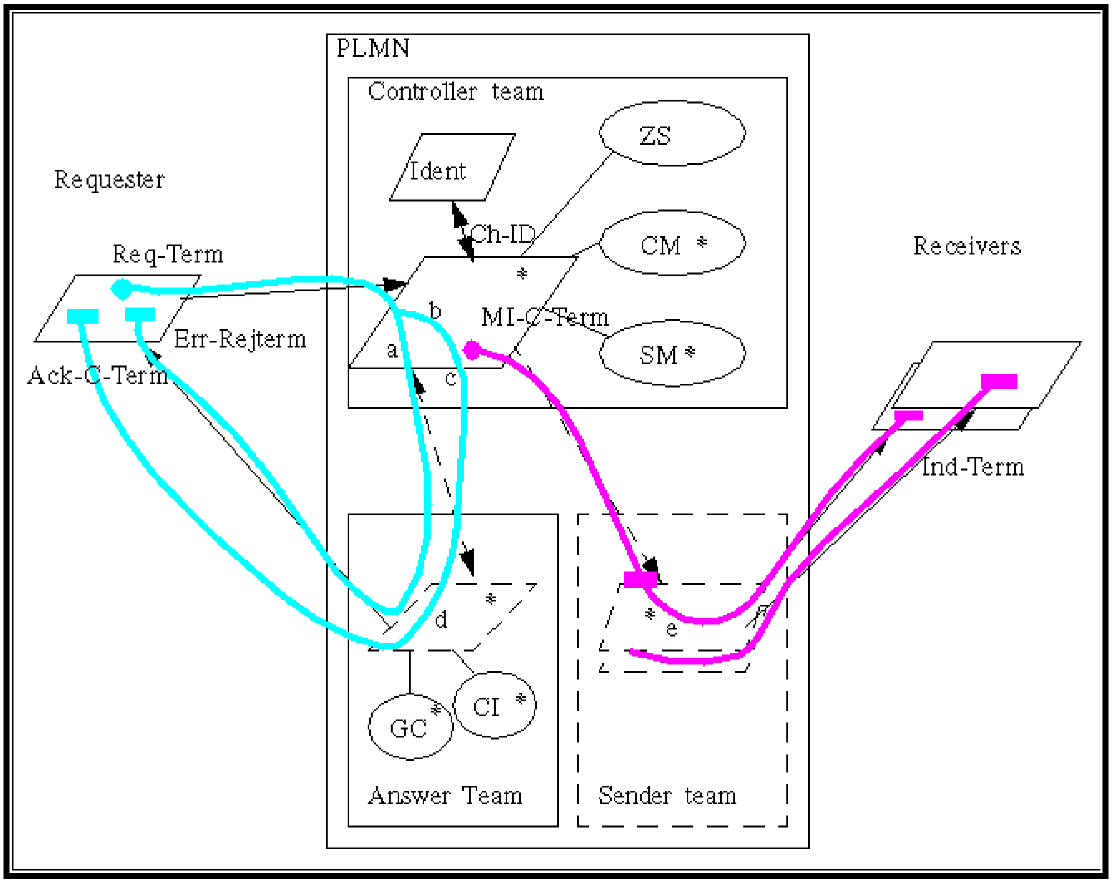

Figure 8: Terminate Call request

The fourth scenario, shown in figure 8, is the **Terminate call** request.

Activity	Type	Information	Comments
Req_Term	Request	M_ID, C_ID	Request of call termination
Err_Rejterm	Error	M_ID, C_ID, Cause	Request rejected (requester not initiator...)
Ack_C_Term	Acknowledgment	M_ID, C_ID	Acknowledgment of termination
MI_C_Term	Internal Message	C_ID, Cause	Internal indication of termination of the call
Ind_Term	Indication	C_ID, Cause	Indication to the participants of the termination of the call
a	Condition		Request agreed? No
b	Condition		Request agreed? Yes
c	Internal		Updating DBCM, DBSM
d	Internal		Updating DBGC, DBCI
e	Internal		Sending indication

This one provides the termination of a call, this possibility is only allowed to the initiator of the call. The two answers possible to this request are the acceptance, so the call is terminated and the rejection.

If the call is terminated, an indication of termination, **Ind_Term**, is sent to all participants.

The fifth scenario is the **Status** request (figure 9).

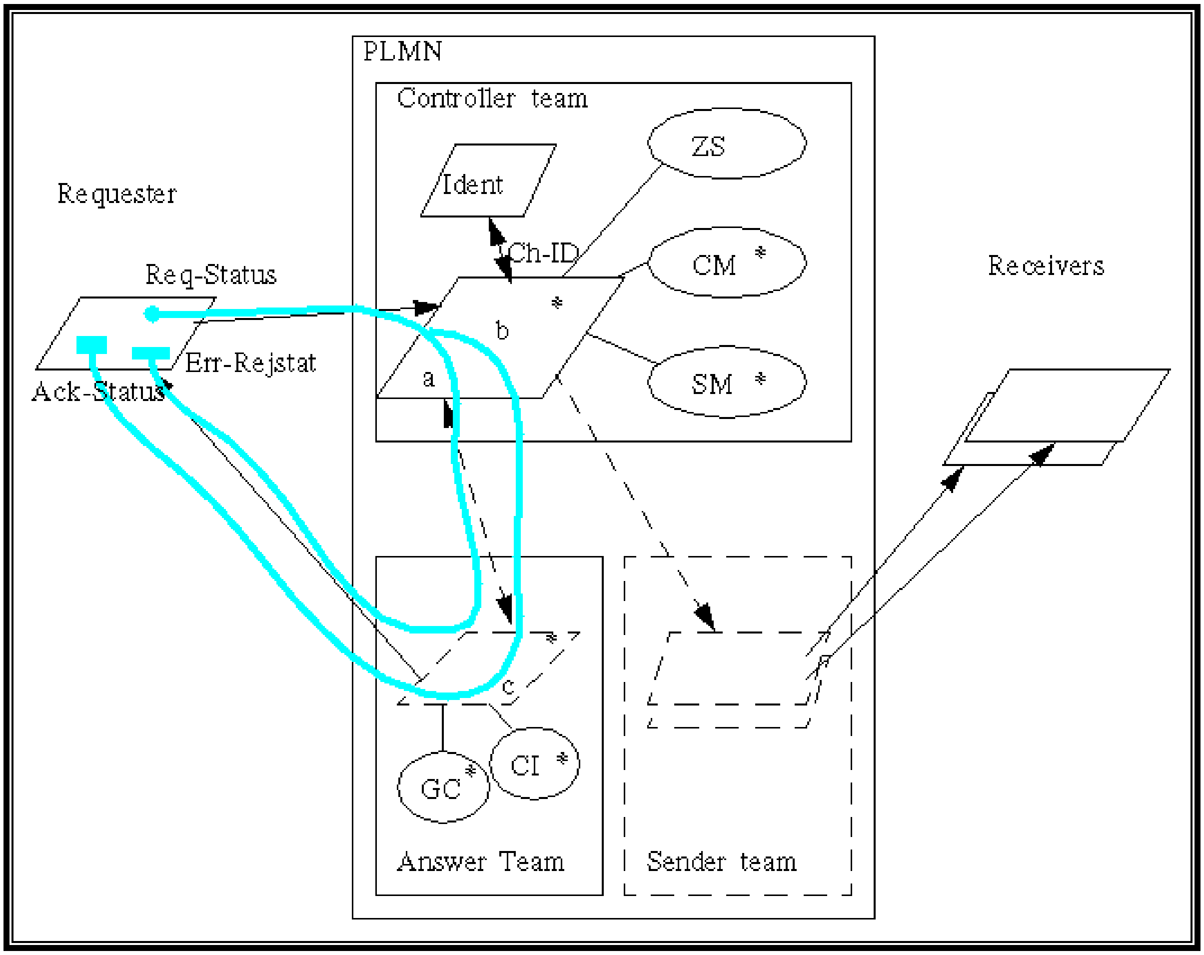

Figure 9: Status request

This request allows to the requester to ask for the attributes of a call, for the list of participants to a call, and for the list of the C_ID active for a given

Activity	Type	Information	Comments
Req_Status	Request	M_ID, IMGI, C_ID(opt), Add.Info.	Request of information, members attached
Err_Rejsta	Error	M_ID, IMGI, C_ID(opt), Cause	Request rejected (requester does not have rights on the group,...)
Ack_Status	Acknowledgment	M_ID, IMGI, C_ID(opt), Information	Answer to the request
a	Condition		Request agreed? No
b	Condition		Request agreed? Yes
c	Internal		Interrogate DBCM, DBCI, DBGC

IMGI. This third possibility is the most important option. Indeed, in the Join request, the requester has to know the C_ID of the call to whom he wants to connect. The only other way to know that, if he is not connected to GPRS when the initiate call notification is sent, is to make a Status request.

In the Status request the parameter C_ID is optional when the request is made. The Additional information (Add. Info.) are the list of the information required. They can be: the C_ID of a specific IMGI (this one exclude the others), the list of participants to a call, the numbers of participants to a call, the value of the join/leave attribute, the value of the QoS, the value of the DTM or the value of the GeoZone. Except for the first type of information, the others can be required separately or together.

The sixth scenario presents the **Join** operation (figure 10).

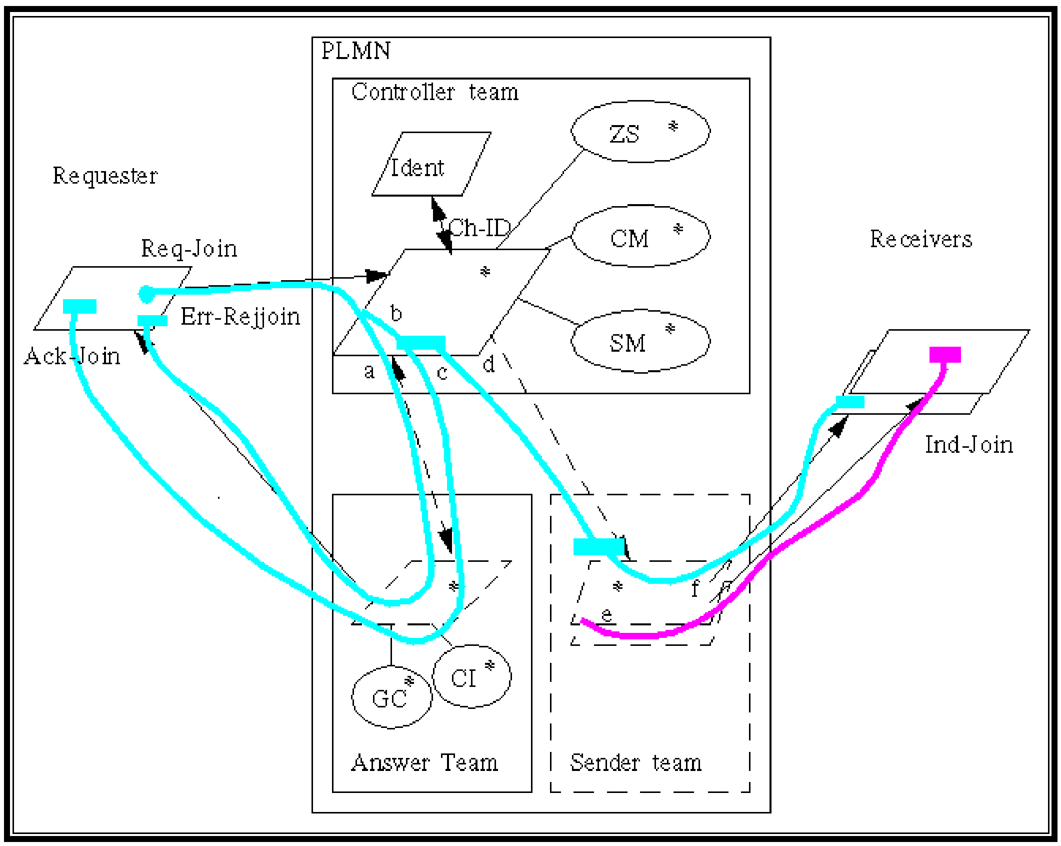

Figure 10: Join request

This request allows the connection to an existing call. If the request is accepted, the network answers with the cipher key used for this call.

An option is mentioned in [ETSI 96]. The requester can make a join operation with the parameter **any call** instead of parameter C_ID. In this case, the network should select a call or reject the request.

In this specification, we have considered that, in the case of **any call**, the network always rejects the request. This option seems a bit unsafe to be really implemented.

Activity	Type	Information	Comments
Req_Join	Request	M_ID, IMGI, C_ID	Request of junction
Err_Rejoin	Error	M_ID, IMGI, C_ID, Cause	Request rejected (wrong GeoZone,...)
Ack_Join	Acknowledgment	M_ID, IMGI, C_ID, Cipher_Key	Indication of junction to the requester
Ind_Join	Indication	M_ID, C_ID	Indication of junction to the participants
a	Condition		Requester can join the call? No
b	Condition		Requester can join the call? Yes
c	Internal		Updating DBCM
d	Condition		Send Join indication? Yes
e	Condition		Send indication to all participants? Yes
f	Internal		Sending indication

The seventh scenario presents the **Leave** request (figure 11).

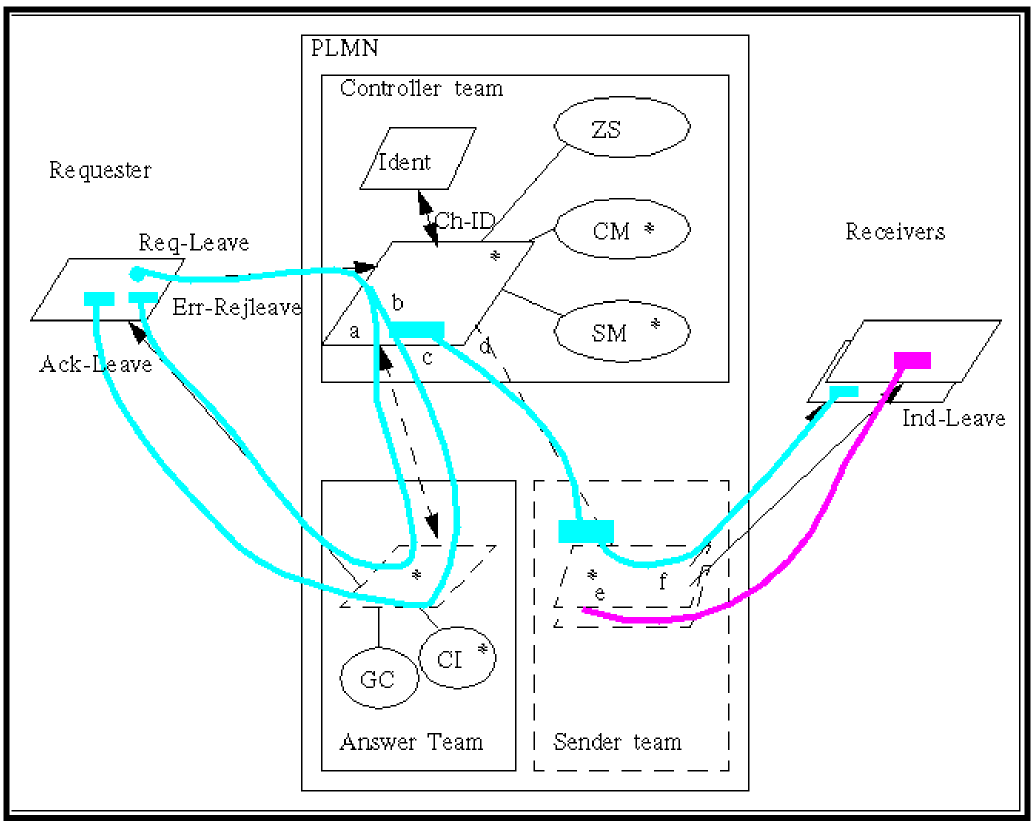

Figure 11: Leave Request

The leave operation is the opposite of the join operation, and its scenario is absolutely symmetrical.

We have to notice that the requester of the leave operation cannot be the initiator of the call. If the initiator wants to leave the call, he has to perform the **Terminate call** operation.

Activity	Type	Information	Comments
Req_Leave	Request	M_ID, IMGI, C_ID	Request of leaving call
Err_Rejleave	Error	M_ID, C_ID, Cause	Request rejected (requester initiator,...)
Ack_Leave	Acknowledgment	M_ID, C_ID	Notification of leaving for the requester
Ind_Leave	Indication	M_ID, C_ID	Indication of leaving for the participants
a	Condition		Request agreed? No
b	Condition		Request agreed? Yes
c	Internal		Updating DBCM
d	Condition		Send leave indication? Yes
e	Condition		Send indication to all participants? Yes
f	Internal		Sending leave indication

Figure 12 shows the eighth scenario: the **Data transfer** request.

The result of this request depends heavily of the DTM. We have a basic case of rejection, i.e., when the requester is not a call participant.

If the requester is the initiator, then there is no problem: the data are transferred to all participants.

If the requester is not the initiator and the mode is one-way, only the initiator can send data, so the answer is an error with the wrong mode of transmission (**Err_WMode**).

If the DTM is two-way, and the requester is not the initiator, then the initiator is a moderator, so the data are transferred only to the initiator (**Ack_SInit**).

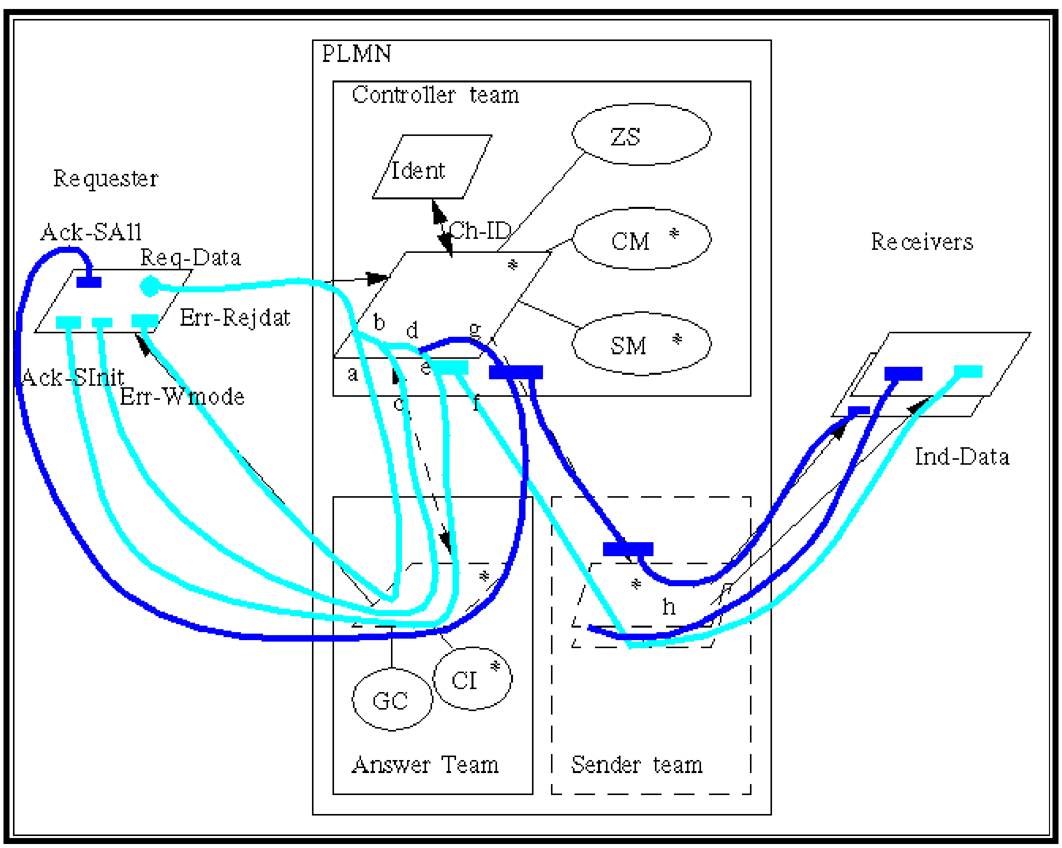

Figure 12: Data transfer request

Activity	**Type**	**Information**	**Comments**
Req_Data	Request	M_ID, IMGI, C_ID, Data	Request of sending data
Err_Rejdat	Error	M_ID, IMGI, C_ID, Cause	Request rejected
Err_WMode	Error	M_ID, IMGI, C_ID, Cause	Rejected, mode is one-way
Ack_SInit	Acknowledgment	M_ID, IMGI, C_ID	Data sent to the initiator
Ack_SAll	Acknowledgment	M_ID, IMGI, C_ID	Data sent to all participants
Ind_Data	indication	M_ID, C_ID, Data	Data transfer
a	Condition		Request agreed? No
b	Condition		Request is participant? Yes
c	Condition		DTM is one-way and requester is not initiator? Yes
d	Condition		DTM is not one-way or requester is initiator? Yes
e	Condition		DTM is two-way and requester is not initiator? Yes
f	Internal		Sending Data to initiator
g	Condition		DTM is N-way or requester is initiator? Yes
h	Internal		Sending data to the participants

If the DTM is N-way, the data are normally transmitted to all participants (**Ack_SAll**).

[ETSI 96] foresees the possibility of having an acknowledgment from the receivers to the PLMN; this option is not implemented in this specification.

The last scenario, shown in figure 13, presents the emulation of routing operation: the **Change zone** request.

This is an abstract request; in reality, the mobile station changes zone without requesting the authorization to the network. So following the example of the Detach scenario, this one cannot be rejected. The consequences are similar to the ones of the Detach request, with a little specificity: if a call is defined for a larger GeoZone than the one left by the requester, the network does not have to terminate the calls initiated by the requester or let the requester leave the calls to which he participates.

In our specification, two levels of GeoZone are defined: the basic level and a global GeoZone (**NoGeozone**). If a call is defined for the global zone, there is no change to do.

On the opposite, if the call is defined for a basic GeoZone, it is necessary to leave or eventually terminate the call, following the role of the requester.

With this scenario, we have finished the presentation of the functionalities. We can now synthesize the LOTOS specification and the implementation of these functionalities.

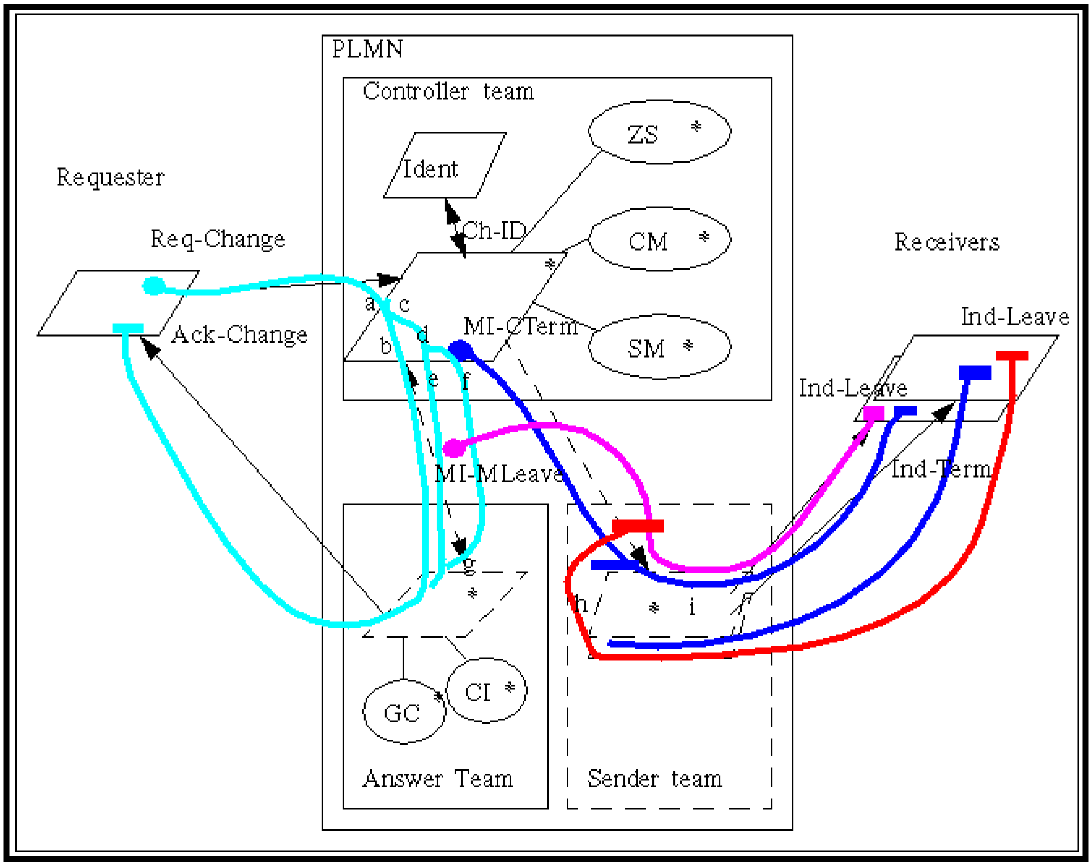

Figure 13: Change Zone request

Activity	Type	Information	Comments
Req_Change	Request	M_ID, GeoZone	Request of changing area
Ack_Change	Acknowledgment	M_ID	Notification of changing zone
MI_M_Leave	Internal Message	M_ID, C_ID	Participant leave
MI_C_Term	Internal Message	C_ID, Cause	Initiator leave
Ind_Leave	Indication	M_ID, C_ID	Indication to participants of leaving of athe requester
Ind_Term	Indication	C_ID, Cause	Indication to the particpants that a call is terminated
a	Internal		Updating DBZS
b	Condition		Is requester member of any call? No
c	Condition		Is requester member of any call? Yes
d	Internal		Updating DBCM
e	Condition		Is requester initiator of any call? No
f	Condition		Is requester initiator of any call? Yes
g	Internal		Updating DBGC, DBCI
h	Condition		Send leave indication to all participants? Yes
I	Internal		Sending leave indication

Part 3: The LOTOS Specification

3.1. Language-Oriented Constraints and Assumptions.

Each language has its own constraints. So each theoretical work has to be slightly modified to fit the constraints of the language. In LOTOS, the main constraint is the lack of databases. We have defined five databases, and we have to model them.

LOTOS does not allow the possibility to have variables: we can only create local renamings in order to simplify the writing.
Using `S` instead of: `RowSM(n, DBSM)` can greatly simplify the parenthesis count when we have to compose this with others operators. Unfortunately, these renamings are constants: we cannot change the value after their definition. In conclusion, the only possibility is the parameterization of processes. Therefore, the databases are defined as parameters of processes.

Now, we have two different cases. The databases are parameters but are they parameters of independent processes, which only manage one database, or parameters of an existing process, that has other purposes?

To answer this question, we have considered the minimization of the rendezvous. Each rendezvous is an action. We want to minimize the number of actions, and thus, accelerate the simulation. We have also considered the number of processes implicated in the use of these databases.

So, we have defined two types of databases: the *parameter* ones and the *independent processes* ones.

The databases located in the Answer-team are independent processes, because the Controller process and the Informer process have to use them.

The databases located in the Controller-team are used only by the Controller process, so they are parameters of the Controller process. All the updating operations are made by recursive instantiation of the processes.

The location of these databases is defined, but how do we define the rows of the database and how do we define the databases globally?

LOTOS does not allow many choices: we have to define abstract data types, one for each database, describing the rows. Then, we do a list with each type of row, and then we have a great emulation of static databases. We only have to define the operators to interrogate the database. The updating is very simple: we just have to delete, insert or update the row considered.

We use the keys of the databases to define an operator, which can extract the row corresponding to the key. Some operators, which extract each field of a row, are defined. With these operators, defined for each database, we can perform all the requests based on the key. Some additional operators will be presented with the description of the ADT.

Another modification of the theoretical work is the limited number of IMGI and M_ID. In theory, the number can be unlimited but for reasons of testing, we have limited them to a reasonable number. With the same idea, we have implemented a very limited database DBSM.

We have seen the modifications realized due to the language, now we will present the structure of the specification.

3.2. Organization and Roles of the Processes.

Basically, the behavior of the specification is the composition of the PLMN (process *PLMN*) and the subscribers (process *Subscriber*) synchronized on the two channels of communication, *Ch_Uplink* and *Ch_Downlink*.

Figure 14 shows the hierarchy of processes.

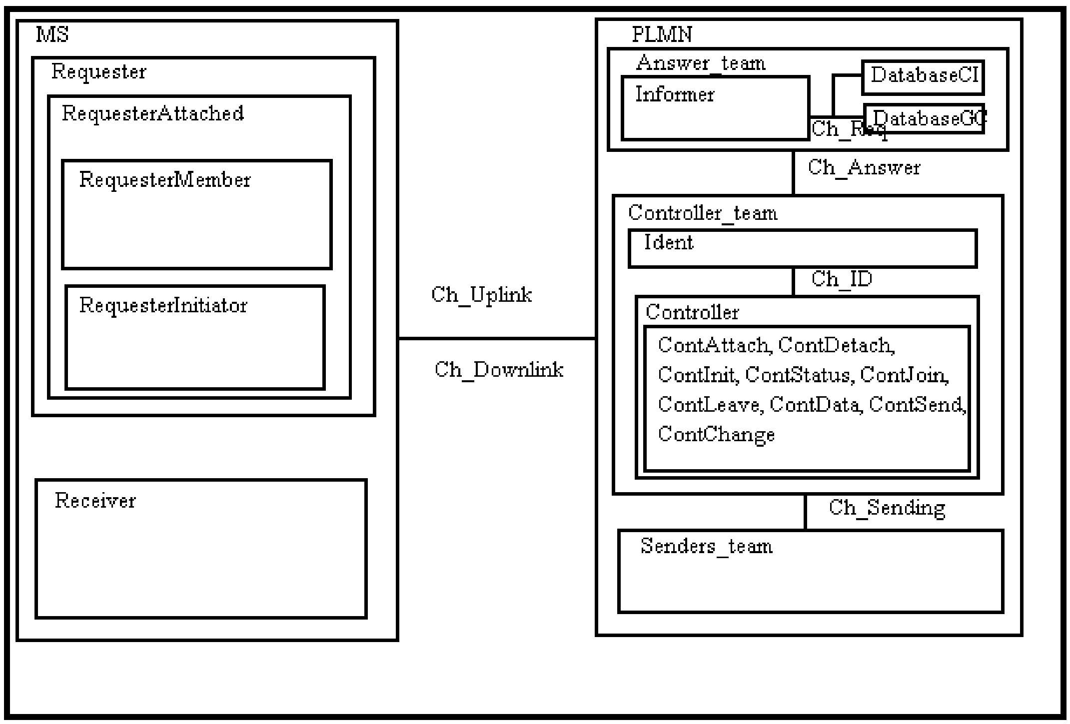

Figure 14: Hierarchy of processes

Process **Subscriber** contains an interleaving of processes *MS*, each one characterized by a different M_ID in parameter.

Each MS process is the interleaving between a process *Requester* and a process *Receiver*. The process Receiver is waiting for the indications from the PLMN. The process Requester sends the Attach request to the PLMN. In the process Requester we have a sub-process, *RequesterAttached*, which models the behavior of a member when he is attached to the GPRS.

Process **RequesterAttached** has two sub-processes: one to model the behavior of a member initiator of a call (*RequesterInitiator*) and one to model the behavior of a member participant to a call (*RequesterMember*).

Process **PLMN** is composed of the three teams, synchronized on the required channels (*Ch_Sending* between *Controller_team* and *Senders_team*, *Ch_Answer* between *Controller_team* and *Answer_team*).

Process **Senders_team** is waiting for an indication from the Controller to be forwarded to the appropriate receiver.

Process **Answer_team** contains three sub-processes. The two databases (DBGC: process *DatabaseGC* and DBCI: process *DatabaseCI*) and the process *Informer*. These three processes are synchronized on the channel *ch_Req*.

The behaviors of the databases are similar. they wait for a request, they answer it, and then reinstantiate themselves with, in parameter, the updated content of the database.

Process **Informer** is waiting for an acknowledgment from the Controller to the requester. In case of Status request, it has some processing to perform.

Process **Controller_team** contains two processes synchronized on *ch_ID*: the process *Controller* and the process *Ident*. Process Ident just provides a unique C_ID when it is required.

Process **Controller** is the most important one. It receives the requests and dispatches them to the right sub-process. The three databases contained in the Controller_team are parameters of this process. Due to the particular structure of the databases, each time we want to make an update on them, we have to reinstantiate a process with the database in parameters.

There are nine sub-processes, one for each type of request. All of them have their name prefixed by **Cont** and suffixed by the type of request. They all have the three databases in parameters in addition with the specific information provided by the request. At the end of processing, they also reinstantiate the process Controller with the updated databases.

- Process *ContAttach* manages the Attach request.
- Process *ContDetach* processes the Detach request. In case of leaving or terminating some call, it instantiates two sub-processes in charge of this work: *MICTerm* and *MIMLeave*. These two recursive processes empty the stack of calls to leave or to terminate, and then reinstantiates Controller.
- Process *ContInit* deals with the Initiate Call request.
- Process *ContTerm* performs the Terminate Call request.
- Process *ContStatus* performs the beginning of the Status request. It interrogates the database DBCM when needed and it sends all the results to the Informer, which performs the end of the processing.
- Process *ContJoin* processes the Join Call request.
- Process *ContLeave* manages the Leave Call request.
- Process *ContData* processes the Send Data request.
- Process *ContChange* processes the Change Zone request. It contains two sub-processes. The first one, *ContChangeLeave*, performs the leaving of calls. The second one, *ContChangeQuit*, terminates the calls. This two processes empty a stack recursively.
- A last process, *ContSend*, is a sub-process of Controller. It transmits to Senders_team an indication to be transferred to the receivers. This process is instantiated by almost every others sub-processes of Controller.

We have seen the hierarchy of the processes and their roles. We will now present the abstract data types needed to achieve the specification.

3.3. Abstracts Data Types: From Needs to Solutions

Often in LOTOS, problems distinct from the exchange of messages have a solution in the ADT domain. There is an operator that you need? Create a type and define this operator. We have used this idea for our ADT.

We need member identifiers, so we create the *M_ID* type based on hex-digit. We need IMGI, so we create the type *IMGI* based on hex-digit. We need call identifiers, so we create the type *C_ID* based on natural number, to have an unlimited number of call. We need geographical zones, so we create the *GeoZone* type, based on hex-digit. We need a quality of service, so we implement a *QoS* type, with the three qualities defined in [ETSI 96]. We need data transfer modes, so we define a *DTM* type, with three possibilities. We need privileges for users, so we define *Membertype*, with the three possibilities (cont, init, part).

We have nine types of requests, so we create a *Req* type based on the hex-digit with the nine possibilities implemented. We have ten possible acknowledgment, so we have a *Ack* type with the ten acknowledgment defined. The same idea is valid for the error (type *Err*), the indication (type *Ind*) and the information provided by the status request (type *Statusinfo*). A special type for the interrogation of the two database DBCI and DBGC is needed, so we have the type *DBReq* based on hex-digit.

We also need several kinds of lists. To achieve this goal, we have defined some formal types: the type *Element* and the type *Set*. These two types allow the definition of any type of list such as *M_IDlist*, *C_IDlist*, *IMGIlist*, *Statusinfolist*.

Some call attributes, set in Initiate call request, have only two values possible: yes or no. A mapping on boolean seems a good approach. These types are *Attach* (Attached/Detached) for the status of member, *Join_leave* (Join_leave_ind/No_join_leave_ind) for the join/leave notification, *Send_to_all* (Send_to_all_ind/No_send_to_all) for the notification to all participants and *Call_not* (Call_not_ind/No_call_not) for the initiate call notification.

Now, we will define the types of rows for the five databases and the corresponding lists:

- DBSM: type of row *Statusmember*, containing a M_ID, Membertype, a IMGIlist (group subscribed by the user), a C_IDlist (call initiated by the user) and an Attach. DBSM is implemented as a *Statusmemberlist*.
- DBCM: type of row *Callmember*, containing a C_ID, a M_ID (which one of the initiator), a M_IDlist (the list of the participants). DBCM is implemented as a *Callmemberlist*. An additional operator is defined to know, for a given M_ID, the list of participated calls.
- DBZS: type of row *Geomember*, containing a GeoZone and a M_IDlist (the list of the users currently in this GeoZone). DBZS is implemented as a *Geomemberlist*. An additional operator is defined to know, for a given M_ID, the current GeoZone.

- DBGC: type of row *GroupCall*, containing an IMGI and a C_IDlist (the list of the C_ID created for an IMGI). DBGC is implemented as a *GroupCalllist*. An additional operator is defined to get the IMGI when we know the C_ID.
- DBCI: type of row *CallInfo*, containing a C_ID, a DTM, a QoS, a GeoZone, a Join_Leave, a Send_to_all, a Call_not. DBCI is implemented as a *CallInfolist*.

We notice that we have a lot of different parameters to pass with each type of request. We simplify this in order to have only one type of message to pass after a request, an acknowledgment, an error or an indication. The type message (*Msg*) performs this simplification. In this type, we find two specific messages *Cause* and *Data*. These two messages emulate the unknown parameters. Cause emulates the error reports and Data emulates the users data sent in the messages. The main operator of this type is the overloaded operator *Encode*, which describe all the possible encodings. For each encoding, there exists the corresponding decoding. With this type, every communication has the pattern: M_ID, Operation, and Msg (where operation can be a request, an error, an acknowledgment or an indication).

With the structure of the processes and the ADT, we can achieve the realization of the specification.

3.4. Model Synthesis from Scenarios.

In Appendix 1, we can find the specification we generated from the scenarios. The processes follow exactly the scenarios in terms of behavior. We even have implemented some unreachable paths, shown in the scenarios, with the composition of process Subscriber and process PLMN. These paths result of some unexpected requests at some point of time. In theory, they should happen, with the complete specification, they should not.

We can yet reach these paths with some specific step by step simulation only on one part of the specification (Subscriber or PLMN).

Since the synthesis procedure is not automated, we followed three basic guiding rules to generate the specification from our scenarios:

- Each responsibility (activity) is allocated to a component in the architecture.
- A component implements the causal relationships of the paths that go through it.
- Causal relationships between components are refined as exchanges of messages: the protocols.

Now we can begin the verification of the specification

Part 4: Verification

4.1. LOTOS Testing Theory and Tools Used

We can test the specification by writing test cases and compose them with the behavior. This black box testing allows to test the behavior from the point of view of the environment, like the user. The explosion of the number of possible states and the dynamic behavior of the processes, due to the structure of the ADT, do not allow a verification based on model checking.

A test case is a succession of events (rendezvous). The test cases are executed with LOLA [PL 90], a tool developed at the University of Madrid. Each test case is composed with the main behavior. Then LOLA expands the new behavior obtained and generates all the paths possible. Three results are possible:

- **Reject**, the test fails
- **May pass**, there are some paths, but not all, leading to the success
- **Must pass**, all the paths analyzed are successful.

LOLA seeks a special event, in order to give the result of the test: a special gate that we called *success*. In the following example, we present a test generated for LOLA.

```
process Test_att2[ch_Uplink, ch_Downlink, Success]:noexit:=
(* Req_Attach, wrong requester Err_Rejatt expected *)
      Ch_Uplink!Mem3!Req_Attach!Encode(1 of GeoZone);
      ch_Downlink!Mem3!Err_Rejatt!Encode(Cause);
(* State checking *)
      ch_Uplink!Init1!Req_Attach!Encode(1 of GeoZone);
      ch_Downlink!Init1!Ack_Attach!Nomsg;
(* Event seeked by LOLA *)
      success;
      stop
endproc
```

This test case tries to connect the user Mem3 to the GPRS, a rejection of the connection is expected. The two lines with the user Init1 allow the test of the state of the PLMN. At the end of the test, there is a post-test sequence, previously validated, to verify the state obtained. For some test, the pre-test sequence is more complex than in this case (here, the pre-test sequence does not exist).

Some step by step tests were done with ELUDO, especially for exploring the specification. The main disadvantage of ELUDO is that it is somewhat slow.

There are two types of test cases: the tests that must pass and the tests that must be rejected. The first type seeks for a mandatory sequence of events. The second type verifies that a sequence of events is never seen with a specific preamble.

To test this specification we only have created success test cases, because of the very deterministic behavior of the specification. Choices are often mutually exclusive. So, even if the behavior is highly dynamic, it is highly deterministic.

4.2. Derivation of Test Cases from the UCM

The use of the UCM scenarios to generate test cases has two advantages. The first is the reusability of the tests cases in the hypothesis of modification of the architecture. The second is the strong correlation between the tests and the requirements. The scenarios were derived from the requirements, so the tests are closely related to the requirements.

The strategy used to generate test cases from the UCM is to explore each path described in the UCM. With this strategy, we think that we have a great consistency in the test suite.

Remember figure 5 and the Attach request where we have two path, one for the rejection, and one for the acknowledgment. So we define two test cases for this UCM.

The idea behind this strategy is to test the specification with respect to the informal requirements and not the conformance to another specification.

UCM led to the generation of 35 test cases corresponding to 42 different paths, including the different possibilities of the attributes for each UCM. These tests are written at the end of Appendix 1.

In the next section, we will discuss how to verify the coverage obtained from the test cases.

4.3. Coverage of the Specification

Creating test cases is good, but we do not know the percentage of the specification covered by these test cases. We know that, due to the method of derivation, the test cases provide a good functional coverage, but we do not know the portion of code visited by the test cases.

To solve this problem, we have inserted probes, with a simple tool, in the specification [Amy 97]. In general, before each *stop*, *exit* or *instantiation of process*, we have inserted a comment: `(*_PROBE_*)`. The tool modifies this comment by substituting an internal event with a unique identifier on a specific hidden gate (*Probe*). For example: `Probe!P_28;`

We have inserted 71 probes in the specification for the PLMN and 30 for the MS. So, we have divided the coverage testing into two parts, to minimize the time of testing. In the specification in Appendix 1, we can notice some comments `(*_PROBE_MS_*)` and other `(*_PROBE_PLMN_*)`. The first ones are defined to test the coverage of process MS and the second ones to test process PLMN. We have, of course, passed all the test cases to verify that we have not changed the behavior of the specification.

With the traces of the executions, we can notice whether or not these probes are visited by the test cases. We have also the number of times that the probes are visited. In theory, if the coverage is good, all probes should be visited.

We can now consider the results of these tests.

4.4. Experimentation

A. The Complete Specification

We put the specification under test. After some minor changes to the specification, the 35 test cases resulted in a *Must Pass*. Figure 15 recalls the results.

Test #	Process	# Successful executions	# States analyzed	# Transitions generated	Test #	Process	# Successful executions	# States analyzed	# Transitions generated
1	*Test_att1*	1	3	3	19	*Test_join2*	16	181	196
2	*Test_att2*	1	5	5	20	*Test_join3*	60	269	328
3	*Test_detach1*	1	7	7	21	*Test_join4*	3	29	31
4	*Test_detach2*	90	711	800	22	*Test_leave1*	9	54	62
5	*Test_detach3*	288	2673	2960	23	*Test_leave2*	120	591	710
6	*Test_detach4*	9	63	71	24	*Test_leave3*	96	435	530
7	*Test_detach5*	180	1191	1370	25	*Test_leave4*	6	51	56
8	*Test_init1*	4	29	32	26	*Test_data1*	6	63	68
9	*Test_init2*	1	7	7	27	*Test_data2*	6	63	68
10	*Test_init3*	3	15	17	28	*Test_data3*	6	57	62
11	*Test_init4*	1	7	7	29	*Test_data4*	3	39	41
12	*Test_init5*	1	7	7	30	*Test_change1*	1	7	7
13	*Test_term1*	240	1071	1310	31	*Test_change2*	162	1431	1592
14	*Test_term2*	27	90	116	32	*Test_change3*	288	2961	3248
15	*Test_status1*	3	27	29	33	*Test_change4*	9	63	71
16	*Test_status2*	3	27	29	34	*Test_change5*	144	1335	1478
17	*Test_status3*	3	33	35	35	*Test_change6*	126	651	776
18	*Test_Join1*	16	165	180		**Total**	**1933**	**14411**	**16309**

Figure 15: Test results for the complete specification

These 35 test cases have led to 1933 executions, and all executions were successful. These results allow us to be a little confident about the functionality compliance.

The number of states analyzed and the number of transitions generated do not take care of the internal events (3 minutes and 12 seconds on a Sparc Ultra, 96 Mb of RAM). The time needed to achieve testing taking care of the internal events is greatly increased. Therefore, the division of the specification into two parts (the MS and the PLMN) in order to test the effective coverage is an obligation. Due to this partition, the numbering of the probes is the same for the PLMN and for the MS.

Figure 16 shows the coverage obtained for the PLMN. After an attentive study of the specification, we expected to have seven probes unvisited, due to events that should not happen in the normal use of the specification. These probes are: 2, 17, 26, 35, 36, 37, and 40.

The first six probes are unreachable because they are not allowed to the user choices, and the probe 40 corresponds to an inconsistency of the databases within the network. This one is then unstable. The six first probes will be tested in specific step-by-step simulation of the PLMN alone in order to prove their utility and reachability. Probe 40 is unreachable in all cases (we hope so).

Probe #	# times visited	Probe #	# times visited	Probe #	# times visited	Probe #	# times visited	Probe #	# times visited
0	69	15	1	29	2	43	18	57	18
1	255	16	1	30	4	44	8	58	266
2	0	17	0	31	8	45	1	59	2
3	1	18	76	32	2	46	39	60	81
4	2	19	1	33	8	47	4	61	395
5	34	20	41	34	8	48	35	62	8
6	35	21	1	35	0	49	259	63	0
7	45	22	19	36	0	50	284	64	323
8	9	23	13	37	0	51	19	65	8
9	4	24	18	38	2	52	229	66	52
10	1	25	8	39	7	53	1	67	25
11	4	26	0	40	0	54	41	68	26
12	34	27	8	41	62	55	43	69	10
13	44	28	8	42	8	56	51	70	121
14	1								

Figure 16: Coverage of the PLMN

We notice a great coverage of the specification with the seven probes expected to be unreached. But we have one unexpected result with the probe 63.

This probe is located in the process DatabaseGC and corresponds to a unused portion of the code. This piece of code was removed from the specification. In part B of this section, we will trace the six reachable probes without using the process Subscriber.

The quality of coverage for the PLMN gives a great confidence to the written specification. Now we have to check the coverage of the Subscriber, and specifically, process MS.

Probe #	# time visited	Probe #	# time visited	Probe #	# time visited	Probe #	# time visited	Probe #	# time visited
0	106	6	161	12	160	18	0	24	2
1	23	7	119	13	311	19	2	25	2
2	1	8	2	14	32	20	2	26	2
3	3	9	2	15	102	21	8	27	6
4	55	10	120	16	13	22	14	28	245
5	146	11	138	17	0	23	0	29	1373

Figure 17: Coverage of the MS

Figure 17 shows the coverage of the MS by the test cases. We have 30 probes in this part. Like for the PLMN, we expect three unvisited probes due to forbidden choices (17, 18, 23). These three probes are reachable in simulation of the Subscriber alone.

For this part, we do not have any surprise. All the results are compliant with the forecasts. In part C of this section we will reach the unreached probes.

B. Process PLMN Alone

LOLA allows the testing of a part of the specification. The test suite on the PLMN only gives the same good results as the complete specification (all test cases have a *Must Pass* result). Now we can focus on the unreached probes.

In Appendix 2, we find the exhaustive traces realized step-by-step, which have led to the probe. The numbers refer to the probes of the PLMN.

- Probe 2 corresponds to the request Detach when the requester is not attached. This choice is not allowed by the specification, but it seems possible in the real implementation (request Attach aborted).
- Probe 17 corresponds to the request of Terminate call when the requester is not the initiator.
- Probe 26 corresponds to the request of Join call for a member who has no rights on the group of the call.
- Probe 35 corresponds to the request of Initiate call with a DTM equal to **noway** (no DTM chosen).
- Probe 36 corresponds to the request of Send Data in a call, to a user whom is not a participant.
- Probe 37 corresponds to the request Change zone for a user not attached.
- Probe 40 is not reachable. It corresponds to a database inconsistency.
- Probe 63 was removed with the useless code.

We have to notice that all these traces (in Appendix 2) were obtained in one time (no quit and load)

C. Process Subscriber Alone

We pass all the test cases on the Subscriber only with the same good results as previously. At the end of Appendix 2, we find the traces used to reach the unvisited probes. The number of probes corresponds to the probes of the MS.

- Probe 17 corresponds to the rejection of a Terminate call request for the initiator of the call.
- Probe 18 corresponds to the rejection of the request Send Data to the initiator of the call.
- Probe 23 corresponds to the rejection of the request Send Data for a participant with the **N-way** DTM.

All these paths, unreachable by the classic use of the specification, ensure the tolerance to errors (robustness). The specification can answer to some normally unexpected events (we cannot say that it answers to any unexpected events, the unexpected is never sure).

Part 5: Discussion and Conclusion

5.1. Methodology

The method used leads to great results. We have a prototype compliant with the requirements. We have a useful description of the functionalities reusable for other architectures. We have test principles (global ways to test the specification) reusable for improvements of the prototype, and a validated test suite for regression testing. We have a documentation generated during the design process (the UCM).

The only weakness about this method is the test cases generation. Basically, we derive test cases from the scenarios, but all the paths are not seen in the UCM due to some choices in the implementation. The total coverage is obtained only after taking care of some values of some attributes. For example, in the Status request, we have defined two types of requests, the first with all the information on a specific call, the second with the C_ID of a specific group. In the UCM we only have one path for these two possibilities, and in LOTOS code, these two paths are mutually exclusive.

This methodology will be greatly improved when we can take care of the implementation choices to describe, in one time, the test suite. In our case, the total coverage is obtained only because we have considered these choices before completing the test suite. The tests are implementation dependent, so we can generate some test cases in parallel with the development of the prototype and finish the test suite with the choices made in the implementation.

5.2. Specification

The specification describes the Group Call with the main restriction of the management of geographical areas, where only two levels are considered. This restriction provokes another one: the fixed users (users on fixed external network) are not fully considered (what is their geographical area?). The PLMN should have a solution for this problem.

Another restriction, more informal, is that in [ETSI 96], the network should invoke the Status request without external intervention. This case could not be implemented because we do not have found a logical explanation for this. We could not model a triggering event for this case.

The repetition rate defined in [ETSI 96] for the initiate call notification could not be implemented due to the use of LOTOS. We wanted to avoid a state explosion problem that such repetition could provoke.

We are confident about the compliance to the other requirements and about the tolerance to the errors of the requests, we have implemented for each predicate an complementary predicate. Hence, the specification should not deadlock. Of course, we have not implemented the loss of radio transmission. In most cases, there is a solution to request which are not in the right sequence (this is the reason for the pieces of code that should not happen).

The two operators Data and Cause have to be refined as operators with the real value as parameters. This can slightly change the specification.

We have found some weaknesses in [ETSI 96] and some lack of information. We noticed that when a rejection is provided, [ETSI 96] does not expect a reason (we have implemented the parameter Cause for this problem). When a Join or a Leave operation is provided and when the attribute of call requires a notification, this notification is sent to the initiator. However, [ETSI 96] does not expect it to be sent to all participants. In N-way mode call, we could want to know who is present without performing a Status operation, especially if there is no change. We have defined the **Send_to_all** attribute to solve this problem. In a Join operation, the requestor can ask to be joined on any call. We have decided that this possibility is not very secure and we have decided not to implement it. Finally, [ETSI 96] does not foresee the sending of a reason when the network disagrees with a request. We have modeled this possibility with the parameter **Cause**.

5.3 Experience with LOTOS

This work was my first real development in LOTOS. So, I have experienced some interesting error to avoid in the future.

Firstly, with the ADT. The renaming seems to be an interesting feature, but its use is very delicate. The types that rename booleans cannot be used in predicate: LOTOS accepts only a boolean. The renaming provokes a total lack of compatibility between the original type and the renamed type. The most awful error to do is to rename the natural numbers for any reason. We should better be defining a totally new type based on natural numbers and create a conversion operator that translates a value of the new type into a number.

Secondly, with the control structure.

- The operator “;” is definitively **not** a command separator. The most common error is to put a “;” after each command. Of course, these error are immediately found during the compilation, but it is still boring to correct the specification due to this error.
- The multiway rendezvous can provoke some mistakes. For instance, in the specification, Controller_team and Answer_team are synchronized on ch_Answer. In the first development, DatabaseGC, DatabaseCI and Informer were also synchronized on this channel. The specification was blocked because the simulation could not find a synchronization with all these processes at the same time. This problem led to the implementation of a specific channel in the Answer_team: ch_Req.
- Another mistake, more subtle, is due to the interleaving operator “|||”. This operator means that the two behaviors on each side are not synchronized on any gate. **False**: they are synchronized on the internal gate that carries the *exit* message (δ). Therefore, a process that wants to provide an exit operation, composed with a non exiting interleaved process, cannot exit normally. This can be annoying with a disabling behavior: the process never exits, and then the disabling behavior is always accessible.

I also used the LOTOS tools for the first time. There is not a lot to say about them. LOLA is more useful than ELUDO to test the specification (the testing can be automatically done), but ELUDO can better help clarifying the eventual bugs that block a specification. For the step by step execution, ELUDO is more convenient with its help to the decision in the building of messages. The probe utilities developed by Daniel Amyot are very useful for the coverage of the testing and they are very simple to use. I could not use Caesar due to the structure of the ADT.

5.4. Conclusion

This work presents the creation of a specification in LOTOS with a scenario-oriented method. The notation chosen for the scenario is the UCM created by Buhr and Casselman. The object of the specification is the GPRS Group Call defined on GSM. We have seen the interest of the methodology to create a specification and the good results obtained in this way.

We have obtained a specification for the Group Call with only few restrictions. This specification seems to be operational and compliant to the requirements considered. We also have generated a reusable test suite.

In future work, we can refine the specification by the full implementation of the geographical area. A solution could be to describe the hierarchy in a database with a link between each area and the basic one that constitutes it. This solution has the great advantage of preserving the key for the GeoZone. Therefore, the main operators and the main part of the behavior would remain unchanged. With this implementation, we would also fix the problem of the fixed users.

References:

[ALB 97] D. AMYOT, L. LOGRIPPO, and R.J.A. BUHR *Spécification et conception de systèmes communicants : une approche rigoureuse basée sur des scénarios d'usage*, Colloque Francophone sur l'Ingénierie des Protocoles 1997, Liège, Belgique, September 1997.

[Amy 97] D. AMYOT, *Group Communication Server: A Scenario-Based Design Exercise*, technical report, Department of Computer Science, University of Ottawa, 1997.

[BC 95] R.J.A BUHR and R.S. CASSELMAN *Use Case Maps for Object-Oriented Systems,* Prentice-Hall, USA, 1995.

[ETSI 96] ETSI *Digital Cellular Telecommunications system (Phase 2+); General Packet Radio Service (GPRS); Service Description Stage 1 (GEM 02.60), Version 2.0.0,* November 1996

[ISO 88] ISO, Information Processing Systems, Open Systems Interconnection, *LOTOS- A Formal Description Technique Based on the Temporal Ordering of Observational Behavior,*IS 8807,1988

[EM 85] H.EHRIG and B.MAHR *Fundamentals of Algebraic Specification 1,* Springer-Verlag, Berlin, 1985.

[Hoa 85] C.A.R. HOARE *Communicating Sequential Processes*, Prentice-Hall intl., UK, 1985.

[Mil 80] R.MILNER *A Calculus of Communicating Systems* In Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, Berlin, 1980

[PL 90] S. PAVON and M. LANAS *The Testing Functionalities of LOLA*, In: FORTE III, 3rd International Conference on Formal Description Techniques, Madrid, November 1990.

Abbreviations:

ADT: Abstract Data Type
CCS: Calculus of Communicating Systems
CSP: Communicating Sequential Processes
DBCI: DataBase Call-Information
DBCM: DataBase Call-Member
DBGC: DataBase Group-Call
DBSM: DataBase Status-Member
DBZS: DataBase geographical Zone-Stations
E-LOTOS: Extended LOTOS
ELUDO: Environnement Lotos de l'Université D'Ottawa
ETSI: European Telecommunications Standards Institute
FDT: Formal Description Technique
FS: Fixed Station
GPRS: General Packet Radio Service
GSM: Global System for Mobile communication
IMGI: International Mobile Group Identity
INT: Institut National des Télécommunications
IP: Internet Protocol
IP-M: Internet Protocol Multicast
ISO: International Organization for Standardization
LOLA: LOtos LAboratory
LOTOS: Language Of Temporal Ordering Specification
MS: Mobile Station
PLMN: Public land Mobile Network
PTP: Point To Point
PTM: Point To Multipoint
PTM-G: Point To Multipoint Group call
PTM-M: Point To Multipoint Multicast
QoS: Quality of Service
UCM: Use Case Maps
WWW: World Wide WEB