A LOTOS Specification of the MAP/E Protocol

a first step in conformance testing the MAP/E protocol
within the framework of GSM-testing

Rinke Terpstra

Preface

On the occasion of the 50th anniversary of the liberation of The Netherlands, a number of Canadian companies awarded several Dutch students with a special scholarship. Northern Telecom awarded three students with the ***"Nortel Netherlands - Canada 50th Anniversary Commemoration Scholarship"***. It enabled these students to carry out a research project at a Canadian University and to establish contacts with Canada and explore relations with Canadian colleagues. This opportunity was also given to me.

Within the framework of the ongoing scientific collaboration between the TIOS Formal Methods group of the computer science department of the University of Twente and the LOTOS group of the computer science department of the University of Ottawa, I was invited to carry out a research project within the LOTOS group of the computer science department of the University of Ottawa.

I would like to thank the members of the LOTOS group for a stimulating environment. In particular Jalel Kamoun and Gaurav Rampal for being helpful colleagues at the LOTOS lab and for helping me to settle down. They and Hans van der Schoot, Matthias Falkner and Riverson Rios are thanked for support, enlightening discussions and nerve-racking activities.

I would like to thank prof. Luigi Logrippo, Chair of the department and Professor of the LOTOS group, for inviting me and for the supervision of my work. In the same light I would like to thank dr. Jan Tretmans for the supervision on behalf of the University of Twente and for his comments on my report. Also ir. Lex Heerink extensively commented my report and I am very thankful for that.

Rinke Terpstra

feb. 14, 1996
Enschede, The Netherlands

Abstract

The Global System for Mobile Communications (GSM) is a digital cellular communication network covering an international area. Such an area is divided into a great number of so called cells, each with its own radio interface. The GSM system consists of a number of PLMNs (Public Land Mobile Networks) each possibly covering a whole country. A PLMN again can be divided into a number of subsystems, among which the Network and Switching Subsystem (NSS). The NSS includes the main switching functions of GSM. The NSS-entities are attached to a signalling network for the exchange of control data (signalling) in order to fulfil some functional requirements, for instance the required mobility of GSM-users. The Mobile Application Part (MAP) is the top protocol layer of this signalling network. The MAP protocol layer actually consist of a number of protocols each providing the communication means between two NSS-entities. These protocols are not stacked up but operate independently next to each other. One of these protocols is the MAP/E protocol for the purpose of signalling between two MSCs (Mobile Switching Centre). The main provided service is the so called *handover procedure*. The handover procedure is a means to switch automatically from a communication path, say path A, to another, say path B, when a GSM-user moves from a cell attached to path A into another cell attached to path B during a call.

By the development of a LOTOS specification of a MAP/E protocol a first step is made in conformance testing a realistic MAP/E protocol. LOTOS is an ISO standard formal specification language. The LOTOS specification captures the bahaviour of a MAP/E protocol which is able to support one dialogue between two MSCs in order to handover a call. Besides two MAP/E protocol entities, the specification contains an underlying service. The specification can be used to derive test cases, although the test puposes are yet limited. The requirements to support the normal course of one dialogue for the purpose of handing over a call can be tested. The specified underlying service makes it possible to consider different test architectures. The LOTOS specification is structured in a way such that it can easily be extended and enhanced. It also serves well as a basis for the development of other MAP/x protocols for conformance testing within the framework of GSM-testing.

Contents

Preface 3

Abstract 5

Contents 7

1. Introduction 9

Part I: Introduction to MAP and GSM 11

2. Overview of GSM 11
    2.1. Introduction to GSM 11
    2.2. Network architecture 11

3. Mobile Application Part 14

4. Handover procedure 16

Part II: MAP/E protocol 21

5. Service and protocol descriptions. 21
    5.1. Service description of MAP/x 22
        5.1.1. Modelling concepts 22
        5.1.2. Service Sequencing 24
    5.2. TCAP, the underlying service of MAP 27
        5.2.1. Modelling concepts 27
        5.2.2. Service sequencing 28
    5.3. Mapping MAP and TCAP service-providers 29

6. MAP/E protocol description 33

7. Remarks 39

Part III: LOTOS specification of MAP/E protocol & service-provider 41

8. Service decomposition 41
    8.1. LOTOS spec of TCAP service-provider 42
    8.2. LOTOS specification of MAP/E protocol entity 44
        8.2.1. Local constraints 44
        8.2.2. Remote constraints 46

9. Conformance testing 48

Conclusion 53

Literature 55

Appendix I: Overview of specific MAP services 57

Appendix II: LOTOS spec of the MAP/E service-provider (ADT) 61

Appendix III: LOTOS spec of the MAP/E service-provider (behaviour) 75

1. Introduction

The cellular phone has clearly proven itself and has justified its usefulness in a world of alternative means of wireless communication. The cellular's success may even be compared to the introduction of the radio, the television, and the telephone. One of the current operating cellular systems is GSM (Global System for Mobile communications). GSM distinguished itself from the (other systems for mobile communication by being the first *digital* cellular telecommunications system. ETSI (the European Telecommunication Standard Institute) took account for the standardization of the GSM system. The standardization work is not yet finished, and it will be continued for years to come. However, nowadays GSM networks are already operational, but to what extend does the equipment, subsystems and the provision of cellular services comply with the standard? Testing is one of the means to answer this question. At the department of computer science of the university of Ottawa, the LOTOS group started an ongoing project GSM-testing.

This report is established within the scope of this project. It presents a first step in conformance testing one of the protocols used by a GSM subsystem. The particular protocol is called MAP/E. MAP stands for Mobile Application Part, which consists of a set of protocols, and MAP/E is one from this set. Conformance testing in short, deals with checking the correctness of protocol implementations with respect to their formal specification by means of testing. A first step in conformance testing the MAP/E protocol is the development of a formal behaviour specification of this protocol. In order to describe the protocol in a formal way, the standard formal specification language LOTOS is used. The resulting specification will be used for the derivation of abstract test cases in a next step.

This report is divided in three parts. In order to indicate where the MAP/E protocol is situated within the whole complex of GSM-subsystems and protocols, an overview of GSM is given in the first part. The main service provided by the MAP/E protocol is the *handover service*, which deals with the mobility of a GSM user during the performance of a call. The provision of this service by the GSM-system in its entirety as well as by the MAP/E protocol is presented. In the second part an overall description of the MAP/x protocols and the underlying service-provider is given. From this a MAP/E protocol description is derived, which provides the handover service to entities within one of the GSM subsystems. The third part presents the LOTOS specification which captures the mentioned protocol description. Besides its usefulness for conformance testing a realistic MAP/E protocol implementation is considered in this part.

Part I: Introduction to MAP and GSM

Chapter 2 gives and overview of GSM. Chapter 3 presents the *Mobile Application Part* (MAP), and chapter 4 gives a description of the so called *handover procedure*.

2. Overview of GSM

Section 2.1 gives an short introduction to GSM, and section 2.2 presents its network architecture.

2.1. Introduction to GSM

Mobile communications is not a very recent technology, but it is a rapidly evolving one. A major step was made at the beginning of the 1980's, when the first analog cellular systems were introduced. In the US the first operating cellular system was the AMPS (Advanced Mobile Phone Service). In Europe, the NMT (Nordic Mobile Telephone) system covered the Scandinavian countries, and TACS (Total Access Communications System) came into use in the UK.

Soon it became obvious to several European countries that existing analogue systems had limitations. The different systems in operation did not offer compatibility for mobile users and the increasing demand for mobile services became larger than the capacity of the existing analogue systems. This has led to a new standard.

This standard is well known as the Global System for Mobile communications (GSM), which was drafted by the European Telecommunications Standard Institute (ETSI). The system is based on digital transmission for speech as well as data, and its capacity should be better than existing analogue networks. GSM allows free roaming of the subscribers within Europe. Since then GSM-systems came also into use in other continents. Practically speaking, the same GSM mobile station, e.g., a small wireless telephone set, must enable its user to call or to be called anywhere within the international coverage area.

2.2. Network architecture

The European GSM system is divided into a number of separate operational networks, each being operated to a large extent independently from the others. Each of these networks is called a PLMN (Public Land Mobile Network). The GSM PLMN is considered as an access network to existing telecommunication networks for users with mobile stations. From a distributed perspective the PLMN system is composed of 4 interacting subsystems:

- the mobile station (MS);
- the base station sub-system (BSS);
- the network and switching sub-system (NSS);
- the operation sub-system (OSS).

Subscribers are GSM customers which are subscribed to a certain PLMN, called the home-PLMN. The **MS** is the apparatus which connect these subscribers to the GSM network. A subscriber is uniquely identified by an International Mobile Subscriber Identity (IMSI). This identity is stored in a Subscriber Identity Module (SIM). This is a small card the user has to insert into a mobile set before it gets access to the GSM services. This card is of importance with respect to mobility, security, and billing.

Since the number of available radio channels is much smaller than the total number of potential users, communication channels are only assigned at need. When an MS is not engaged in a call it learns about an incoming call by listening to a specific broadcast channel. This channel carries messages to

indicate that a given MS is being called. Besides this channel, an MS can use another specific channel to set up a call itself. An MS not engaged in a call is in **idle mode**. It listens to the broadcast channels and has no channel of its own. An MS switches to **dedicated mode** when it gets engaged in a call. Then a bi-directional channel is allocated to the MS for its communication needs.

Best known examples of MSs are wire-less telephone sets such as car-phones, and handhold telephone sets. Besides this, also other terminal equipment exist for transmission and reception other than speech, such as text, images, and facsimile.

Basically, the **BSS** groups the infrastructure machines which are specific to the radio aspects of GSM. A PLMN contains several BSSs, each consisting of one **BSC** (Base Station Controller), and several **BTSs** (Base Transceiver Station). The BTS comprises devices for radio transmission and reception, including the antennas, in order to interact with the MS on the radio interface. The radio coverage of a BTS is called a cell. The cell range is limited to a few tens of kilometres in rural areas, and in urban areas, this limit comes down to 5 to 10 kilometres. The BSC is connected, on one side, to the BTSs and on the other side, to the NSS. Its main roles are the management of the channels on the radio interface, and the performance of handover procedures. A handover procedure is performed when an MS moves from one cell to another, during a call.

A PLMN contains one **NSS** which includes the main switching functions of GSM, as well as databases. The MSCs (Mobiles services Switching Centre), and GMSCs (Gateway MSC) contain the switching functions. The VLRs (Visitor Location Register), and the HLRs (Home Location Register) are the databases. The databases contain information relevant to telecommunications services to which the subscriber has access, and information related to the current location of the subscriber. The NSS-entities are described in the next chapter.

The NSS is at one side connected to several BSSs, and at the other side to existing telecommunication networks, such as PSTN (Public Switched Telecommunication Network) and ISDN (Integrated Services Digital Network). The switching functions and databases are linked by the signalling network SS7 (Signalling System No. 7, standardized by CCITT) for the purpose of signalling. Signalling is referred to the exchange of control-data. The user-data only flows through the switches.

The MS, BSS, and NSS form the operational part, whereas the **OSS** provides means for the operator to control them. The operator observes and modifies the operational part to maintain or improve its functioning.

Figure 1 shows the architecture of a PLMN. The OSS is not included in this figure.

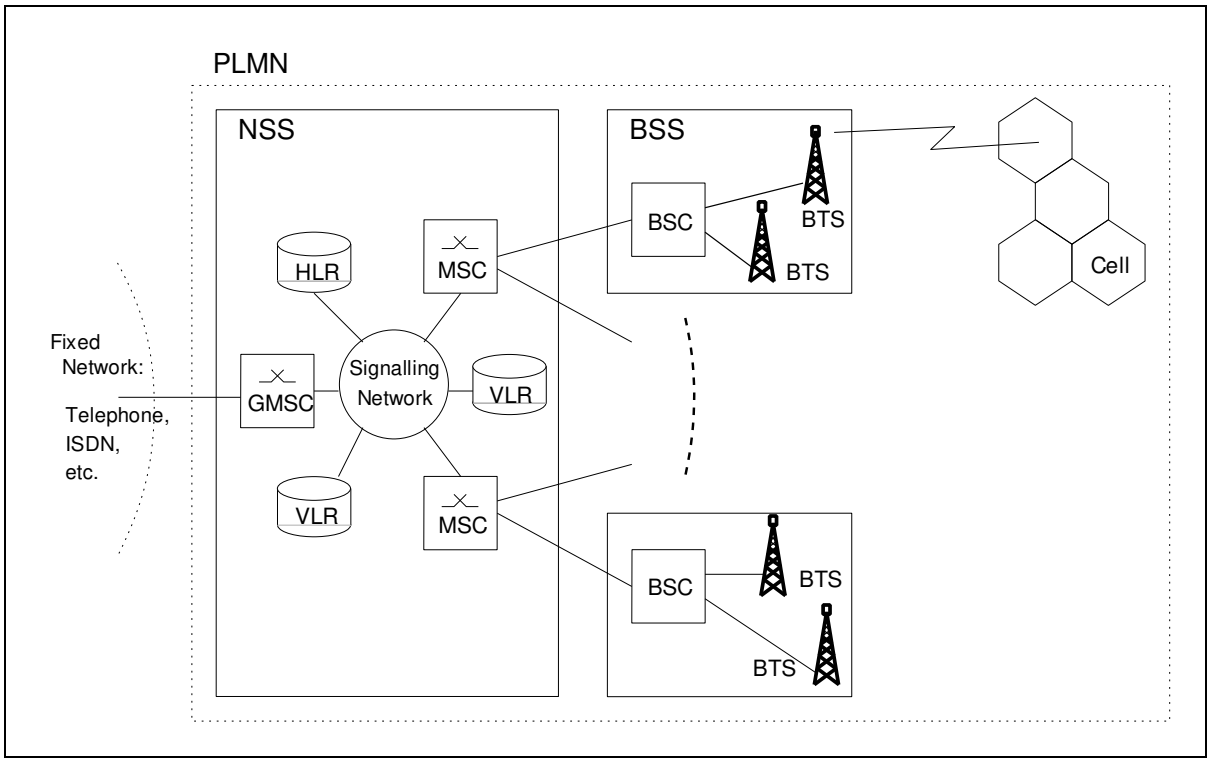

Figure 1: GSM network-architecture.

3. Mobile Application Part

As mentioned before, a GSM PLMN is considered as an access network to existing fixed telecommunication networks for users with mobile stations. An important node in this is the NSS. Its main role is to manage the communication between fixed network users and GSM users, but also between mutual GSM users.

The basic switching function is performed by the MSC, whose main function is to coordinate the setting-up of calls to and from GSM users. An MSC is at one side connected to several BSSs, through which it is in contact with GSM users. Some MSCs are additionally connected at the other side to other telecommunication networks. These MSCs require a gateway, also called the Interworking Functions (IWF).

The other type of switch contained in the NSS is the Gateway MSC or GMSC. Its main function is to route calls received from other networks towards GSM users. In order to set up such a call towards a GSM user, this call is first routed to a GMSC. The GMSC is, instead of an MSC, in the position to find out about the whereabouts of the subscriber to which the call eventually has to be routed to. After it is aware of this it routes the call towards the MSC serving the geographical area in which the subscriber is situated. This area is called the MSC-area.

The term GMSC is somewhat misleading. According to the ETSI standard, an GMSC has no relation to MSC functions, and could be thought of as an independent equipment. However, the GMSC function is usually integrated in an MSC. Accordingly, because of the integrated GMSC, this MSC contains the IWF, by which it has an interconnection to external fixed networks. Usually in literature, the faulty assumption is made that an GMSC is nothing more than an MSC additionally containing the IWF.

Besides switching functions, the NSS includes databases. Subscriber information relevant to the provision of telecommunications services is stored in the HLR, as well as some information related to the current location of the subscriber. As a physical machine, an HLR is typically a stand-alone computer able to handle hundreds of thousands of subscribers. Accordingly, an NSS usually contains one HLR. An other database function identified in GSM is the VLR. It is linked to one or more MSCs, and in charge of temporarily storing subscription data only for those subscribers currently situated in the service area of the corresponding MSC(s), as well as holding data on their location at a more precise level than the HLR. The data stored in a VLR is of more temporary nature compared to HLR data. Locating data in the VLR, as well as in the HLR, is a means to reduce the data traffic to the HLR.

Two other databases included in an NSS, are the AuC (Authentication Centre), and the EIR (Equipment Identity Register). These databases are used to check the identity of callers and of mobile phones whenever call attempts are made, in order to prevent fraud.

The NSS makes use of the so-called Signalling System no. 7 (SS7) for the purpose of signalling between the NSS-entities. SS7 has been drafted by CCITT, which originally defined it for fixed network signalling. For use in the NSS, it had to be expanded such that it could be adapted to the NSS-entities. For this purpose an extra protocol layer has been designed, which nowadays is drafted by ETSI. This layer is called Mobile Application Part (MAP). The resulting network is not only used for signalling between the NSS-entities within one PLMN, but also between NSS-entities within different PLMNs. It connects all NSSs of all PLMNs within a GSM system. This enables the routing of calls between users of different PLMNs.

As a service, MAP groups seven distinct service-providers, each providing the service for signalling between particular NSS-entities. MAP service-providers are distinguished using names, such as MAP/A, MAP/B, and so on to MAP/G. The MAP/x service-providers are listed below.

- MAP/A: although it is one of the MAP service-providers, it is not used for the signalling between NSS-entities, but between an MSC and a BSC, which is a BSS-entity. In literature this service-provider is usually called BSSMAP;
- MAP/B: link between a VLR and its associated MSC(s);
- MAP/C: link between an HLR and a GMSC;
- MAP/D: link between an HLR and a VLR;
- MAP/E: link between MSCs. The main provided service is called *handover control*. It provides the means to exchange control data between the MSCs involved in a handover procedure, in the case when an MS moves from a cell within an MSC-area into a cell within another MSC-area;
- MAP/F: link between an MSC and an EIR;
- MAP/G: link between VLRs;

Figure 2 presents a distributed view of the NSS network-architecture used for the exchange of control data. Also presented in this figure is the underlying service, provided by Transaction Capability Application Part (TCAP). TCAP is the top protocol layer of SS7. The figure is composed of three layers of squares. The top layer represents the NSS-entities, the bottom layer represents the TCAP service-provider, and the layer in between represents the MAP protocol layer. The squares in the latter layer contain letters which refer to a particular protocol within MAP, and the arrows present the possible control data flows. Not presented in this figure is the NSS-entity AuC, because it is always integrated in the HLR. According to [ETSI] an AuC could be thought of as an independent equipment within an NSS, but nevertheless a MAP service-provider between an HLR and AuC is not drafted yet.

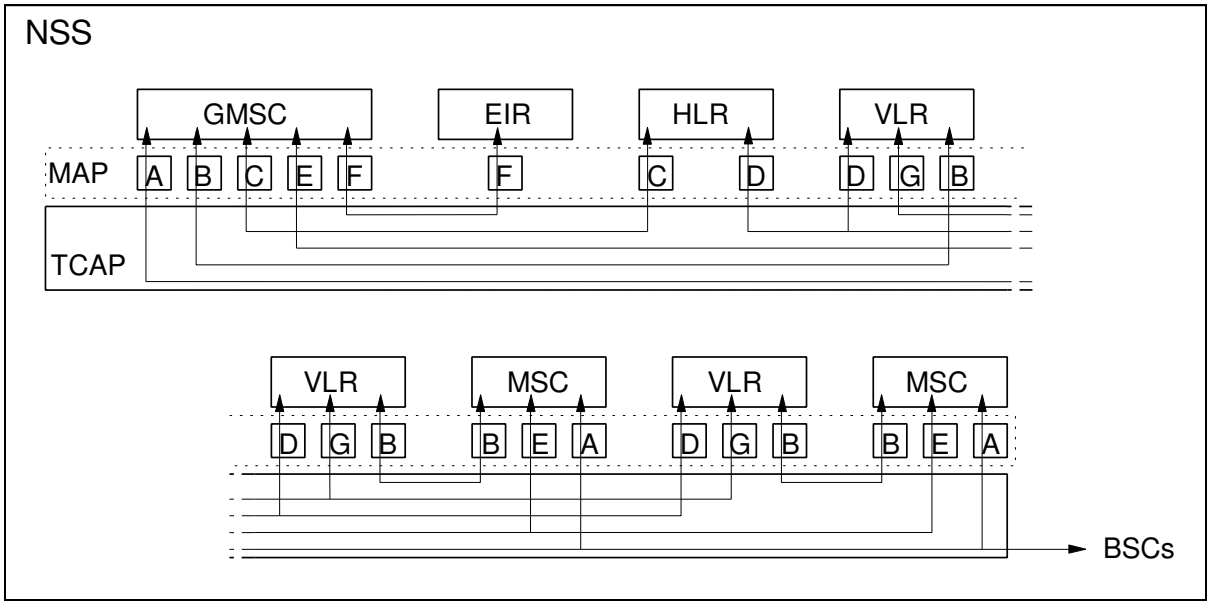

Figure 2: NSS network-architecture.

4. Handover procedure

A PLMN covers a geographical area with adjacent cells. Within the borders of a cell, an user of an MS is during a call dedicated to a communication channel. Because of the mobility of the user it can cross the border of one cell into another cell. The handover procedure is a means to continue the call when this happens. The handover procedure is performed automatically, and ideally in a way not noticeable by the user of an MS. During a handover the communication is switched from a channel in a given cell to another channel in another cell. An MS continuously performs measurements on the signal quality of the serving cell, and of its adjacent cells, and the results are periodically sent to the BSC. When the BSC detects that the signal quality is below a certain minimum threshold it starts the handover procedure.

The GSM system distinguishes different types of handovers. In order to describe the different cases, the suffix *old* shall be used to refer to all functional entities along the communication path between an MS and its peer before the handover. The suffix *new* shall be used for the entities after handover. BTS-old, BSC-old and MSC-old represent the machines in charge of the old cell, and BTS-new, BSC-new, and MSC-new represent the machines in charge of the new cell. The communication path is referred to the route the user-data flows along, between two callers of which at least one is a GSM user.

Every type of handover is composed of two phases:

- in a first phase, BSC-old triggers a set of events (control-data) with the purpose of establishing the future communication path (for user-data). Once this is done, this phase terminates with sending a command to the MS that it can change to a new channel.
- in a second phase, the MS accesses the new channel. This access results in the use of the new communication path, and the release of the old path.

The different types of handover are presented below.

Intra-BSC handover.

In this case BSC-old is the same as BSC-new. This BSC possesses all relevant information to manage the handover by itself. After the BSC indicates that a handover is needed it allocates the future communication channel for the MS in BTS-new (figure 3.a: message RSM CHANNEL ACTUATION). The BSC is able to send the identities of potential new cells because it receives these identities together with the signal measurement results periodically sent by the MS. After the allocation of the new channel it passes this information to the MS, together with the command to change to the new channel (figure 3.b: messages RSM CHANNEL ACTUATION ACK and HANDOVER COMMAND). Finally the MS accesses the new channel, and the old communication path will be released (see figure 3.c,d).

The MSC in charge of the BSC is not involved in the execution of the handover, but the BSC has to inform the NSS to which cell the MS moved. For that purpose the BSC informs the MSC, which in turn updates the databases of the NSS. Figure 3.a,b,c,d shows the functional entities involved in this handover, and the sequence of the message exchange.

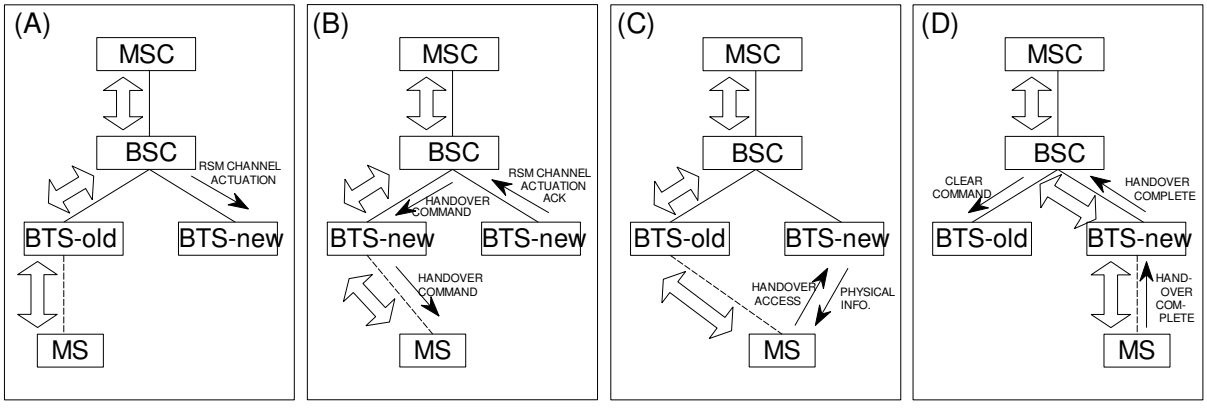

Figure 3: Intra-BSC handover.

Inter-BSS, intra-MSC handover.

In this case MSC-old is the same as MSC-new. When BSC-old indicates that a handover is needed, it notifies the MSC by sending a message (HANDOVER REQUIRED) containing the identity of the current cell, and the identities of potential new cells.

After the MSC received this message it establishes a new communication path towards BSC-new, and also sends it the information about the cells received from the MS (HANDOVER REQUEST).

BSC-new requests BTS-new to allocate a communication channel (for the incoming MS), and waits until BTS-new is ready (RSM CHANNEL ACTUATION and its acknowledgement). Then it sends a message to the MS, via MSC, BSC-old, and BTS-old, to command it to change to the given channel included in this message (figure 4.b: HANDOVER REQUEST ACK to HANDOVER COMMAND). On reception of this message by the MS, the first phase of the handover is completed. See (a), and (b) of figure 4.

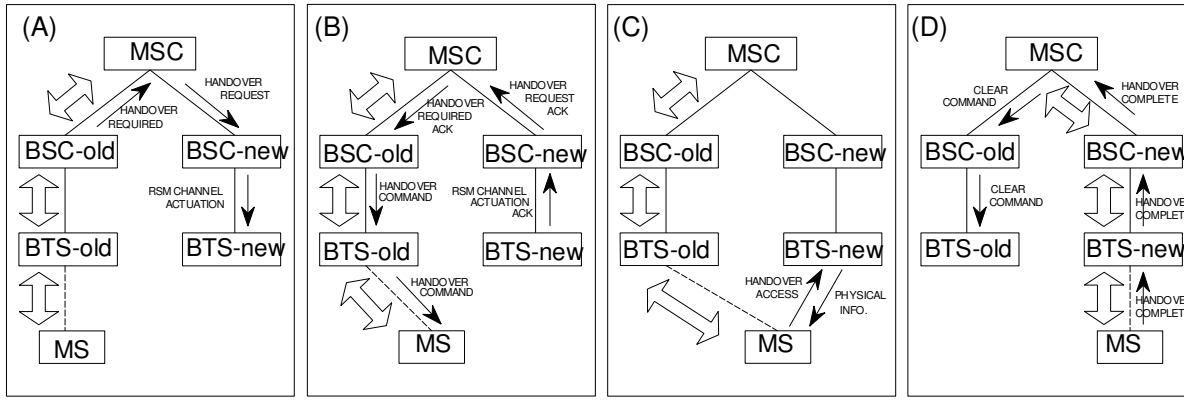

Figure 4: Inter-BSS, intra-MSC handover.

The second phase starts with switching to the new channel by the MS. This involves the exchange of a number of control messages first (HANDOVER ACCESS and PHYSICAL INFO.). During this exchange the user-data still flows through the old communication path. After successful completion of this procedure, the MS informs BTS-new about this (HANDOVER COMPLETE). The new communication path is accessed, and the old path can be released. To accomplish this, BSC-new informs the MSC about the successful handover (HANDOVER COMPLETE), which in turn informs BSC-old (CLEAR COMMAND), whereafter they release the old communication path. See (c), and (d) of figure 4.

Inter-MSC handover, and subsequent inter-MSC handover.

Before describing these cases the terms anchor MSC, relay MSC, and MSC-area will be introduced.

An MSC-area is a geographical area covered by the set of cells connected to one MSC. An MSC gets involved in the establishment of a call towards a GSM user, when this user is roaming in its MSC-area. This MSC is called anchor MSC. An MS could move to a cell within an MSC-area not under control of the anchor MSC, but another MSC. This MSC is called relay MSC. The anchor MSC is in charge of the connection towards the GSM user, throughout the duration of the call. Besides, the user data shall always be routed via the anchor MSC.

The MSCs involved in these handover cases communicate with each other using the MAP/E service-provider. This service-provider is modeled in LOTOS, and its implementation will be subjected to conformance testing using this model.

- In the case of inter-MSC handover, the anchor MSC is MSC-old, and is not the same as MSC-new.

  The behaviour of BSC-old is the same as in the previous case, but MSC-old will behave different. As in the previous case, after BSC-old indicates that a handover is needed, it notifies MSC-old by sending a message (HANDOVER REQUIRED) containing the identity of the current cell, and the identities of the adjacent cells.

  After the MSC received this message it wants to establish a new communication path towards BSC-new, and it wants BSC-new to allocate a new communication channel at a new BTS. To accomplish this it sends a similar message as it received (PERFORM HANDOVER) towards MSC-new, which will deal with it further (HANDOVER REQUEST). After BSC-new is informed it requests BTS-new to allocate the communication channel (CHANNEL ACTUATION). The message sequence is depicted in figure 5.a.

  BSC-new waits until BTS-new is ready (CHANNEL ACTUATION ACK). Then it sends a message to the MS to command it to change to the allocated channel. This message travels via MSC-new, MSC-old, BSC-old, and BTS-old. On reception of this message (HANDOVER COMMAND) by the MS, the first phase of the handover is completed. This means that the future communication path is established, which will be accessed in the second phase. The sequence of messages resulting in commanding the MS to change to the new channel is depicted in figure 5.b.
  The second phase starts with the exchange of messages between the MS and BTS-new in order to be able to access the new communication path smoothly in a next step (HANDOVER ACCESS and PHYSICAL INFORMATION). The user data still flows along the old communication path. See figure 5.c.

  After successful completion of this procedure the MS informs BTS-new about this (HANDOVER COMPLETE). The new communication path is accessed, and the old path can be released. To accomplish this, BSC-new informs the anchor MSC, which is MSC-old, about the successful handover (HANDOVER COMPLETE). The anchor MSC will then release the old communication path by sending a message to the functional entities along this path (SEND END SIGNAL and CLEAR COMMAND). MSC-old will remain in the new communication path because it is the anchor MSC, which stays in control of the connection throughout the duration of a call. See figure 5.d.

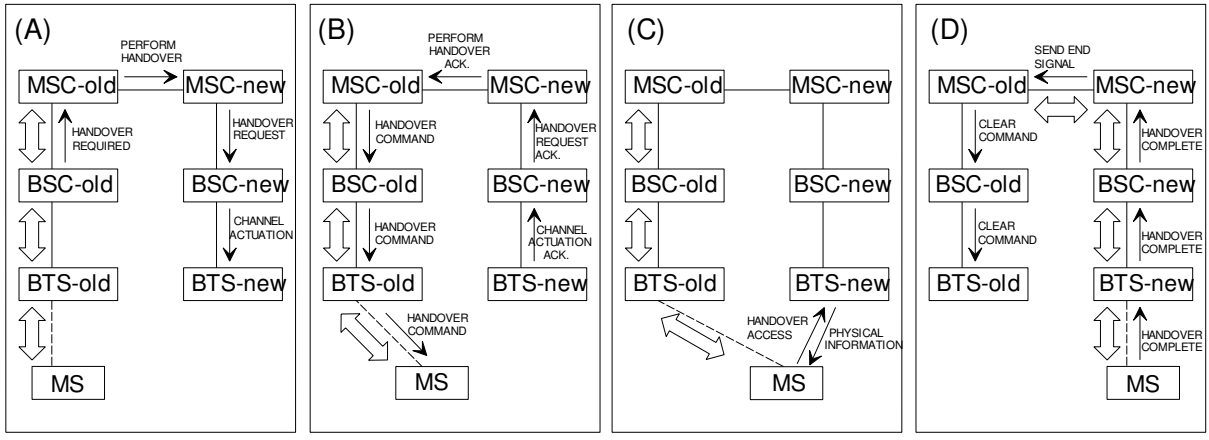

Figure 5: Inter-MSC handover.

- In the case of subsequent inter-MSC handover, the anchor MSC, MSC-old, and MSC-new are different MSCs.

  This case is almost the same as the case of inter-MSC handover. The difference is that MSC-old and MSC-new are not communicating with each other directly, but via the anchor MSC.

  When MSC-old indicates that a handover has to be performed it informs the anchor MSC about this (figure 6.a: PERFORM SUBSEQUENT HANDOVER). The anchor MSC in turn informs MSC-new in order to establish a new communication path (figure 6.a: PERFORM HANDOVER). After establishing the new path, MSC-new informs the anchor MSC about this, and subsequently the anchor MSC informs MSC-old (figure 6.b: PERFORM HANDOVER ACK. and PERFORM SUBSEQUENT HANDOVER ACK.). Accessing the new path proceeds exactly as in the former cases (figure 6.c.). In order to command the MS to switch to the new channel, MSC-new sends the appropriate message to the anchor MSC (SEND END SIGNAL), which sends this message to the MS, via MSC-old, BSC-old, and BTS-old (figure 6.d: SEND END SIGNAL RESULT and CLEAR COMMAND respectively). The new path is accessed, and the old one is released.

- An other case of subsequent inter-MSC handover appears when an MS moves from an area served by MSC-old back to an area served by the anchor MSC (MSC-new). This handover is almost executed in the same way as the inter-MSC handover presented in figure 5. The only differences are the names of the messages exchanged between the MSCs. Instead of PERFORM HANDOVER and PERFORM HANDOVER ACKNOWLEDGE, the message names are PERFORM SUBSEQUENT HANDOVER and PERFORM SUBSEQUENT HANDOVER ACKNOWLEDGE, respectively.

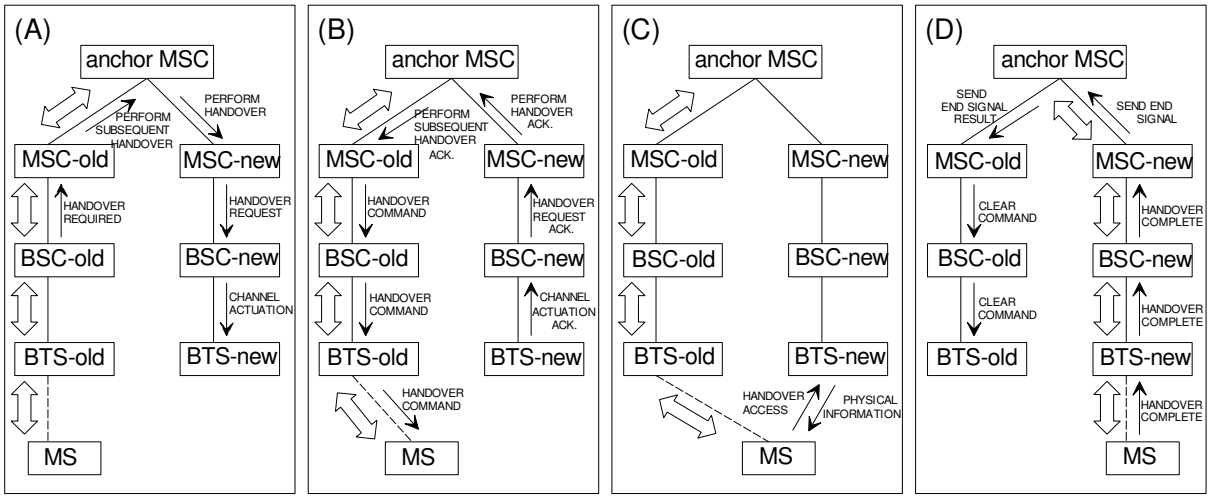

Figure 6: Subsequent inter-MSC handover.

Part II: MAP/E protocol.

In this part a description is given of the MAP/x service-providers, and protocols, and of the underlying service. The approach is to first give the overall description of the MAP/x service-providers, followed by the description of the underlying service, and subsequently the mapping between these service-providers resulting in the overall description of MAP/x protocols. The eventual goal of this part is to give a protocol description of MAP/E.

The descriptions presented in this part are used for the derivation of a LOTOS specification of the MAP/E protocol, but can also be used to derive LOTOS specifications of other MAP/x protocols. These specifications will be used for the derivation of test cases for conformance testing MAP/x protocol implementations. With the description given in this section it was sufficient to derive a first LOTOS specification which could be extended, and enhanced easily in a later stage when more detail is desired. With the LOTOS specification derived from the presented descriptions, test cases can already be derived.

Chapter 5 presents the description of MAP/x as a service-provider and as a protocol, and the description of the underlying service. Chapter 6 presents the protocol description of MAP/E, and chapter 7 gives some remarks.

5. Service and protocol descriptions.

Service users are engaged with each other via a service-provider during a certain period of time. This common engagement is well known as association, or interaction [OTS, p.102]. In [ETSI], and in this report, the term **dialogue** is used for this engagement.

Two service users engaged in a dialogue, interact with a service-provider for the transfer of their messages. The point at which an interaction takes place is called **service access point** (SAP) located at the boundary of the service-provider, and the exchange of messages at the SAPs is accomplished by the performance of **service primitives** (SPs).

To distinguish between the user who starts a dialogue, and the remote user that consequently is engaged in it, respectively the terms **calling** user and **called** user are used. These roles are not fixed. A service user which is *calling* user in some interaction may be *called* user in another, and vice versa.

A service can be defined by its observable behaviour. The complete observable behaviour of a service-provider will be defined by describing all possible SP occurrences, and the possible sequences in which the SPs can be performed, called **Temporal Orderings** (TO) of SPs [OTS, p.85]. A MAP/x protocol will be described by the mapping between MAP SPs and TCAP SPs, and TOs of these SPs.

Section 5.1. presents an overall service description of MAP/x service-providers, and section 5.2 presents a service description of the TCAP service-provider. Section 5.3. presents a mapping between MAP SPs and TCAP SPs.

5.1. Service description of MAP/x

The distinct MAP/x service-providers are all modelled in the same way, no matter which NSS-entities are engaged in a dialogue. This is presented in the next sections, and is used to describe MAP/E as a service-provider. In section 5.1.1. the organization of the MAP/x service-providers, and their SPs are given. Section 5.1.2. presents a TO of SPs which is generic for all MAP/x service-providers.

5.1.1. Modelling concepts

The total services provided by all MAP/x service-providers together can be classified in a number of ways.

First, the services are classified as **user-specific** and **common** MAP services. The common MAP services are available to all MAP service-users. The main purpose of these services is for opening, continuing, closing, and aborting a dialogue between two MAP service-users. The number of user-specific services counts 16, and are divided among the distinct MAP/x service-providers. By this division a particular service-user, i.e., an NSS-entity, has only access to the services accommodated in the MAP/x service-provider to which it is attached.

A MAP service is build from service elements that are classified as unconfirmed, confirmed, and provider initiated services. Each service element carries a specific name, and refers to a specific subset of SPs. The unconfirmed service is based on the pair request/indication SP types. The confirmed service is based on the quadruple request/indication/response/confirm SP types, and the provider initiated service is based on SP type indication. The following denotation is used for MAP SPs belonging to a certain service element:

map_service-element-name_type.

Service-element-name is the name of a specific service element, and *type* can be any of the SP-types, req (request), ind (indication), rsp (response), or cnf (confirm).

For clarity reasons an example is given. The user-specific MAP service *handover control* is accommodated in service-providers MAP/E and in MAP/B. More specific, the service *handover control* is build from a number of service elements, which are divided into two distinct subsets each accommodated in a separate service-provider. One of the service elements is called *perform_-handover* and is accommodated in service-provider MAP/E. The type of this service element is the confirmed service type. This yields the following SPs.

map_perform_handover_req,
map_perform_handover_ind,
map_perform_handover_rsp,
map_perform_handover_cnf. □

The total amount of the service elements within the user-specific MAP services counts 73. It is obvious that this is not the place to describe all these service elements and the related SPs. For detailed descriptions, the reader is referred to [ETSI, section 5.3.] and [ETSI, chapter 6 to 10]. However, all user-specific service elements are presented in appendix I. The number of common MAP service elements counts 6, and are listed below.

map_open : this service element is used for establishing a MAP dialogue between two MAP service users. The service is a confirmed service, thus its SPs are of type

	request, indication, response, and confirm.
map_close :	this service element is used for normal release of a dialogue. With normal is meant that not errors, failures, or exceptions possibly occurred in service-users or service-provider engaged in a dialogue, triggered the release. The service is an unconfirmed service.
map_delimiter :	this service element is used to delimit a sequence of user-specific MAP SPs. The service is an unconfirmed service.
map_u_abort :	this service element enables a service user to abort a dialogue in which it is engaged. The abortion can be initiated at any time during a dialogue, and is triggered by an error, a failure, or an exception occurred at the user side. The event is known as user initiated abort. The service is an unconfirmed service.
map_p_abort :	this service element enables the MAP service-provider to abort the normal progress of a dialogue (provider initiated abort). The abortion can be initiated at any time during a dialogue, and is triggered by an error, a failure, or an exception occurred within the MAP service-provider.The service is an provider initiated service.
map_notice :	this service is used to notify the MAP service-user about protocol problems related to a dialogue. This service is an provider initiated service.

Naturally, all MAP SPs contain parameters, and [ETSI] dedicates a whole section of more then 10 pages to present these parameters and their purposes. It would be too much to present this here, and the reader is referred to [ETSI, section 5.3.] for a detailed description of the parameters. However, a few parameters have to be noted. These parameters are necessary to maintain one dialogue between two service-users. The other parameters contain information which have to be processed by an NSS-entity, e.g., cell identities. Besides, there will be parameters necessary to handle multiple dialogues, but unfortunately [ETSI] is unclear at describing this.

The SPs of the common MAP service elements contain a parameter, called **application context name**. The value of this parameter refers to an user-specific MAP service, e.g., *handover control*. It is used to indicate the purpose of a dialogue. Besides it could partly be used as an identifier for a certain dialogue, by which corresponding independent message flows (of parallel dialogues) can be distinguished. Only dialogues set up for different purposes could be distinguished by the *application context names*. The SPs of the user-specific service elements all contain a parameter, called **invoke identifier**. The value of this parameter refers to a specific service element to which the corresponding SP belongs. In [ETSI] these values are called **operation codes**, and are presented by tables in which they are related to the specific service elements. The usage of the *invoke identifier* within the specific MAP SPs is not so clear, and seems like doing the same thing twice because the name of an SP is already unique, which should be sufficient to relate the SP to a specific service element. In chapter 7 this subject will be re-entered. The presented information is already sufficient to give an abstract, and generic description of the MAP/x service-providers, presented in the next section.

5.1.2. Service Sequencing

A generic TO of SPs, of a typical dialogue between two service-users using a certain MAP/x service-provider, is depicted in figure 7. The left vertical line indicates the SAP between calling user and MAP/x service-provider, the right vertical line the SAP between provider and called user. Horizontal arrows indicate SPs. The direction of the arrows indicates the main flow of information. Global time is assumed to progress along the vertical lines showing the SAPs. The TO of SPs is generic, because it present the way a dialogue is performed, independent from the invoked specific service.

[ETSI] also gives a written description of a typical dialogue. However, this description appeared to be not complete. Therefore an attempt is made to improve this description which is given below.

A dialogue can be divided into 3 phases: **opening** phase, **continuing** phase, and **closing** phase. The ordinary course of phases is opening/continuing/closing, or opening/closing, the choice depending on the invoked specific service. In the opening phase a service-user initiates a dialogue with another user, and in the closing phase one of the service-users terminates it. The users exchange messages in the continuing phase, but they can do this also in the opening and closing phase. Besides normal termination in the closing phase, the ordinary course of a dialogue can also be terminated as a result of errors, failures, or exceptions, occurred in one of the service-users, or in the service-provider. This kind of termination of a dialogue is called **aborting**.

In the opening and in the continuing phase sequences of a number of specific service SPs are performed, and each sequence is delimited by an SP of service element *map_delimiter*. The number of specific service SPs in a sequence can be none, or more. When a service-user sends data such a sequence is delimited by SP *map_delimiter_req*, and when it receives data, SP *map_delimiter_ind* is used. Only in the closing phase such a sequence is delimited by the SPs belonging to service element *map_close*. Below the phases of a dialogue are described in terms of the performance of SPs.

Opening phase.

The calling service-user starts a dialogue by first performing the common SP *map_open_req*. The common SP contains the parameter *application context name* which indicates which user-specific service has to be invoked.

Subsequently it can perform none, or more specific service SPs, the number depending on the specific service it has invoked. As mentioned above, this sequence must be delimited by the common SP *map_-delimiter_req*. The three arrows, top left in figure 7, show the opening of dialogue as described above.

The called user receives the data by first performing the common SP *map_open_ind*, followed by a sequence of specific service SPs related to the SPs performed at the calling user side. The end of such a sequence is indicated by the performance of common SP *map_delimiter_ind*. The three arrows, top right in figure 7, show the opening of a dialogue at the called side.

Service element *map_open* is a confirmed service, and accordingly the called service-user has to respond by performing the common SP *map_open_rsp*. Subsequently it performs none, or more specific service SPs, depending on which specific service is invoked. This sequence has to be delimited again by common SP *map_delimiter_req*. See figure 7, at the called side, the last three arrows of the opening phase.

Note that the invoked specific service could be a confirmed, or unconfirmed service, resulting in the performance of SPs of type response or request, respectively.

The calling service-user receives the data by performing the SP *map_open_cnf*, possibly followed by specific service SPs. See figure 7, at the calling side, the last three arrows of the opening phase.

Continuing phase.

If the users entered the continuing phase, they perform one, or more specific SPs, each sequence delimited by a SP of service element *map_delimiter*. Which SPs, and in what sequence they are performed, is determined by the specific service invoked during the opening phase.

In figure 7, only the transfer of data from calling user to called user is sketched, but the other direction is also possible. The service-provider allows bi-directional communication, and concurrent exchange of messages between the users.

Besides, it is possible that multiple dialogues exist between two users. These dialogues have to be opened separately. How the continuing phase evolves depends on the invoked (not necessarily different) specific services, and on the service user.

See figure 7, the continuing phase.

Closing phase.

The calling user as well as the called user can initiate the closing phase. This user (the initiator) first performs none, or more specific service SPs, but this time the sequence is delimited by common SP *map_close_req*. After performance of this SP, the dialogue is terminated concerning the initiator. Data related to the terminated dialogue will not be excepted any more.

The peer user then, first performs the specific service SPs which are related to the SPs performed by the initiator of the closing phase, and delimits this sequence by performing SP *map_close_ind*.

After SP *map_close_ind* is performed, the dialogue between the service-users is terminated. See figure 7, the closing phase.

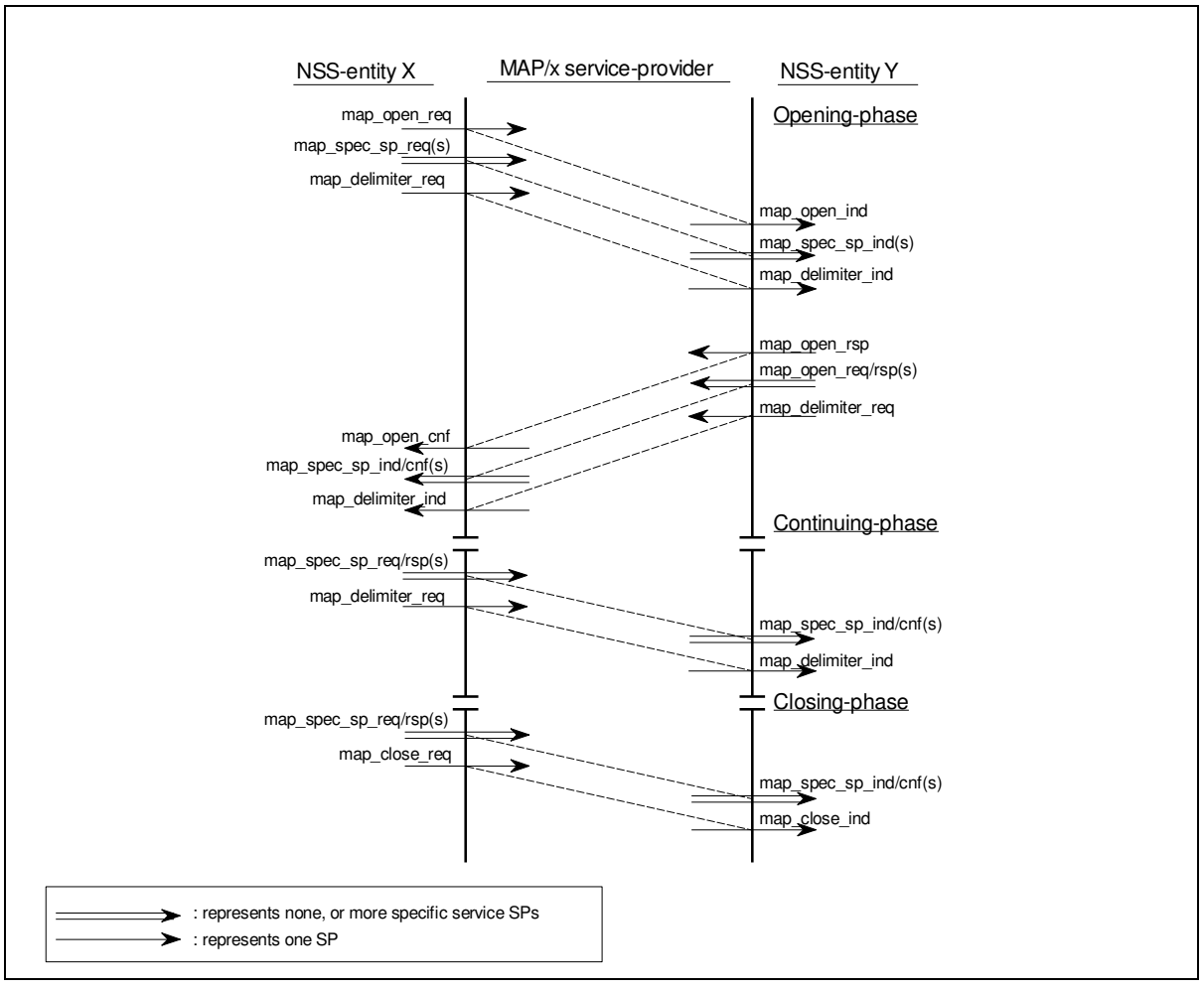

Figure 7: generic TO of MAP SPs.

It must be noted that not sooner then after the performance of SP *map_delimiter_req*, shall data related to the SPs performed before, be sent to the peer user. In order to indicate this way of exchange, figure 8 shows a part of the TO presented in figure 7. In this figure the progress of global time along the vertical lines is indicated by time units ($t_1..t_n$).

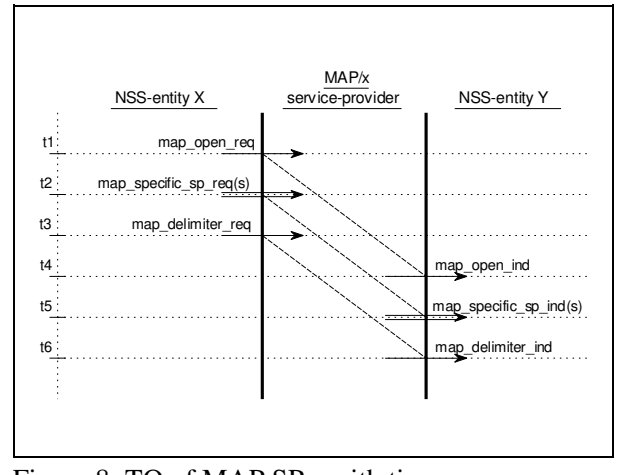

Figure 8: TO of MAP SPs, with time axes.

Aborting.

A dialogue can be aborted by the service-user as well by the service-provider. Aborting of a dialogue is the result of the occurrence of certain errors, failures, or exceptions. The reasons on which the decision is made to abort a dialogue is captured in [ETSI]. Aborting is not presented in figure 7, but some timing aspects are described below.

A service-user, calling as well as called, can abort the dialogue at any time after the dialogue has been opened. A called service-user can also abort the dialogue as a response to an attempt to open a dialogue.

Aborting a dialogue starts with the performance of common SP *map_u_abort_req*, and as a result the peer user performs common SP *map_u_abort_ind*. Precisely, the calling user can abort the dialogue at any time after it performed SP *map_open_cnf*, and the called user can abort by performing SP *map_u_abort_req* instead of *map_open_rsp*, and at any time after SP *map_open_rsp* is performed.

The MAP service-provider may issue at any time common SP *map_p_abort_ind* towards MAP service-users for which a dialogue exists.

Note: TOs are very useful for specifying protocols and systems in LOTOS. The thick vertical lines in figure 7 represent the SAPs of the MAP service-provider. The arrows represent the SPs which are performed at the SAPs. Using the constraint-oriented style, the SPs, and the sequence in which they are performed at a certain SAP can be captured in processes containing the local constraints in a constraint-oriented LOTOS specification, and the relation between the SPs at different SAPs can be captured in a process containing the remote constraints.

5.2. TCAP, the underlying service of MAP

All MAP/x protocols make use of the services provided by TCAP. The description of this service-provider is based on [ETSI]. However, [ETSI] is not very clear in describing this.
Some comments about this are given chapter 7. Some details can be found in [ETSI, chapter 12], but for useful information one should consult the CCITT recommendations Q.771 to Q.775 which contain a full description of TCAP and SS7.

Section 5.2.1. presents the TCAP SPs. These SPs are described as far as that at least a service is provided to the MAP/x protocols to support a dialogue, and such that a LOTOS specification could be derived of TCAP as an underlying service. Section 5.2.2. presents the sequencing of TCAP SPs.

5.2.1. Modelling concepts

Below the service elements, and their SPs are presented below. Each service element is based on the connection-less service type. Thus, based on SP types request and indication.

tc_invoke: SPs belonging to this service element contain the information related to SPs of specific services offered by MAP. Every single SP of a MAP specific service is related to a single *tc_invoke* SP. They relation is established by the parameter *invoke identifier* of a *tc_invoke* SP. The value of this parameter refers to the name of a MAP specific service element.

tc_begin: The SPs belonging to this service element are performed by the MAP PEs during the attempt to open a dialogue. It contains the parameter *application context name* which corresponds to a specific MAP service.

tc_end: The purpose of the SPs belonging to this service element is for normal termination of a dialogue between TCAP service-users. With *normal* termination is meant the termination at the end of a successful dialogue, and not due to the occurrence of errors, failures, or exceptions.

tc_continue: The *tc_continue* SPs are performed during the continuing phase of a dialogue, and are

used to delimit a sequence of *tc_invoke* SPs. *Tc_continue* SPs contain a parameter called *application context name*. This parameter indicates to which MAP specific service, data is related, contained in *tc_invoke* SPs.

Besides the SPs given above, there exist a number of TCAP SPs, for the purpose of handling exceptions and failures. For reasons mentioned in the introduction of this chapter these SPs are not presented.

[ETSI] mentions that all TCAP SPs contain parameters, but remains unclear. For instance, of a lot of parameters it is not mentioned to which TCAP SP(s) they belong. This subject will be addressed again in chapter 7. However a few parameters are stressed out a little more detailed. Particularly two parameters which are directly mapped between MAP SPs and TCAP SPs, namely **invoke identifier** belonging to *tc_invoke* SPs, and **application context name** belonging to *tc_begin*, and *tc_continue* SPs. These parameters are already described in section 5.1.1. Naturally this looks strangely enough, because now the MAP protocol layer seems transparent. This of course is not the case. Among others, the MAP/x service providers provide means to distinguish between message flows belonging to different dialogues in which the particular MAP service-user is engaged. Chapter 7 comes back on this subject.

5.2.2. Service sequencing

A TO of TCAP SPs is depicted in figure 9. The left vertical line indicates the SAP between calling user, which is a MAP/x PE, and TCAP service-provider, the right vertical line the SAP between provider and called user, which is another MAP/x PE. Horizontal arrows indicate SPs. Global time is assumed to progress along the vertical lines showing the SAPs. The presented TO is divided into three phases, corresponding to the phases of a dialogue between MAP service-users.

In every phase a sequence of a number of none, or more *tc_invoke* SPs are performed. At the sending side such a sequence is delimited with one of the request SPs belonging to service elements *tc_begin*, *tc_continue*, or *tc_end*. Not until after one of the latter mentioned SPs is performed, data related to *tc_invoke* SPs is send. This is presented by dashed lines in figure 9. At the receiving side an indication SP belonging to service elements *tc_begin*, *tc_continue*, or *tc_end* is performed first, and then followed by *tc_invoke_ind* SPs. These *tc_invoke_ind* SPs are related to the *tc_invoke_req* performed at the sending side.

The allowed courses of phases are the same as for a MAP dialogue presented in section 5.1.2. In the **opening** phase the calling user performs SP *tc_begin_req*, to indicate the start of a dialogue. During the **continuing** phase the transfer of data from calling user to called user and vice versa is sketched. The service-provider allows bi-directional communication, and the parallel exchange of messages between the users. The **closing** phase is entered when one of the service-users performs SP *tc_end_req*. The normal execution of an association can be aborted by the service-user as well by the service-provider, at any time after the first SP is performed in this association. Aborting an association is the result of the occurrence of certain errors, failures, or exceptions.

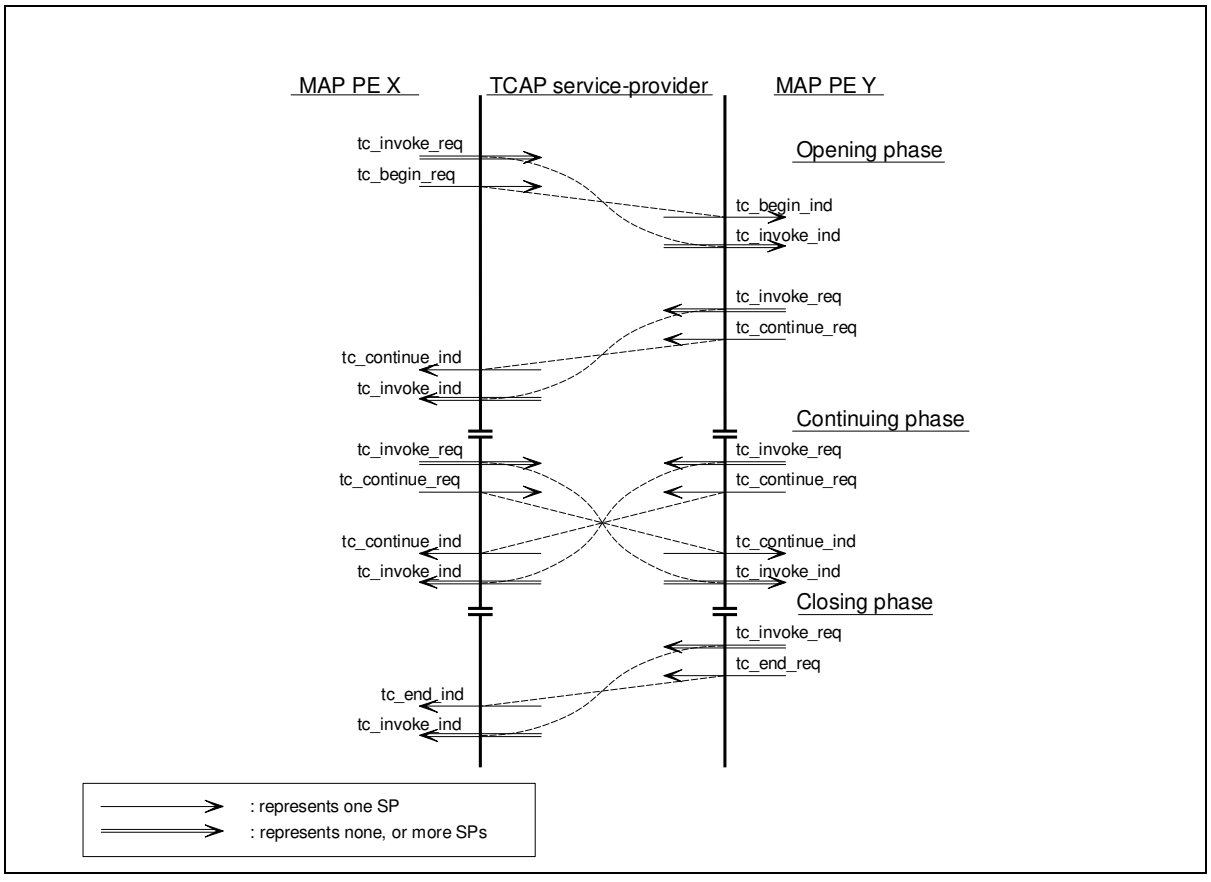

Figure 9: TO of TCAP SPs.

5.3. Mapping MAP and TCAP service-providers

[ETSI] also gives a mapping between MAP SPs and TCAP SPs. However, this mapping appeared to be not entirely correct. Therefore an attempt is made to improve this mapping. Like in [ETSI], this mapping is presented using tables.

The tables are divided into three parts which correspond to the three phases of a dialogue. The rows contain the SPs, and parameters are given between parentheses. SPs which have to be performed in a particular order are grouped in a single column. In this column, the performance of a SP in row *n* is only allowed after the performance of SP in row *n-1*. Each row is divided into two columns of which at least one contains a SP. The SP contained in the left column of this row is mapped onto the SP contained in the right column.

Mapping between MAP SPs and TCAP SPs presented by tables is given below.

Opening phase

MAP SPs	TCAP SPs
map_open_req(ApplContext)*	-
specific service SP (InvokeId)	tc_invoke_req(InvokeId)
map_delimiter_req()	tc_begin_req(ApplContext)*

* parameter *ApplContext* of SP *map_open_req* is directly mapped onto parameter *ApplContext* of SP *tc_begin_req*.

TCAP SPs	MAP SPs
tc_begin_ind(ApplContext)	map_open_ind(ApplContext)
tc_invoke_ind(InvokeId)	specific service SP (InvokeId)
-	map_delimiter_ind()

MAP SPs	TCAP SPs
map_open_rsp(ApplContext)*	-
specific service SP (InvokeId)	tc_invoke_req(InvokeId)
map_delimiter_req()	tc_continue_req(ApplContext)*

* parameter *ApplContext* of SP *map_open_rsp* is directly mapped onto parameter *ApplContext* of SP *tc_continue_req*.

TCAP SPs	MAP SPs
tc_continue_ind(ApplContext)	map_open_cnf(ApplContext)
tc_invoke_ind(InvokeId)	specific service SP (InvokeId)
-	map_delimiter_ind()

Continuing phase

MAP SPs	TCAP SPs
specific service SP (InvokeId)	tc_invoke_req(InvokeId)
map_delimiter_req()	tc_continue_req(ApplContext)*

* parameter *ApplContext* contains a value which refers to the invoked specific MAP service.

TCAP SPs	MAP SPs
tc_continue_ind(ApplContext)	-
tc_invoke_ind(InvokeId)*	specific service SP (InvokeId)
-	map_delimiter_ind()

* parameter *InvokeId* of SP *tc_invoke_ind* refers to a specific MAP service element.

Closing phase

MAP SPs	TCAP SPs
specific service SP (InvokeId)	tc_invoke_req(InvokeId)
map_close_req()	tc_end_req()

TCAP SPs	MAP SPs
tc_end_ind()	-
tc_invoke_ind(InvokeId)	specific service SP (InvokeId)
-	map_close_ind()

The mapping presented by these tables could lead to fault interpretations. For example, from the first table the interpretation could be made that a *map_delimiter_req* SP is directly mapped onto a *tc_begin_req* SP.

The tables could better be seen as groups of SPs which are strongly related to each other. For example the first table should be interpretated as follows. A *map_open_req* SP is directly mapped onto a *tc_begin_req* SP, but a *tc_begin_req* SP will not be performed sooner then after the performance of a *map_delimiter_req* SP.

Figure 10 presents a clearer mapping between MAP SPs and TCAP SPs and besides it presents the progress of a typical dialogue, by means of a TO of Sps.

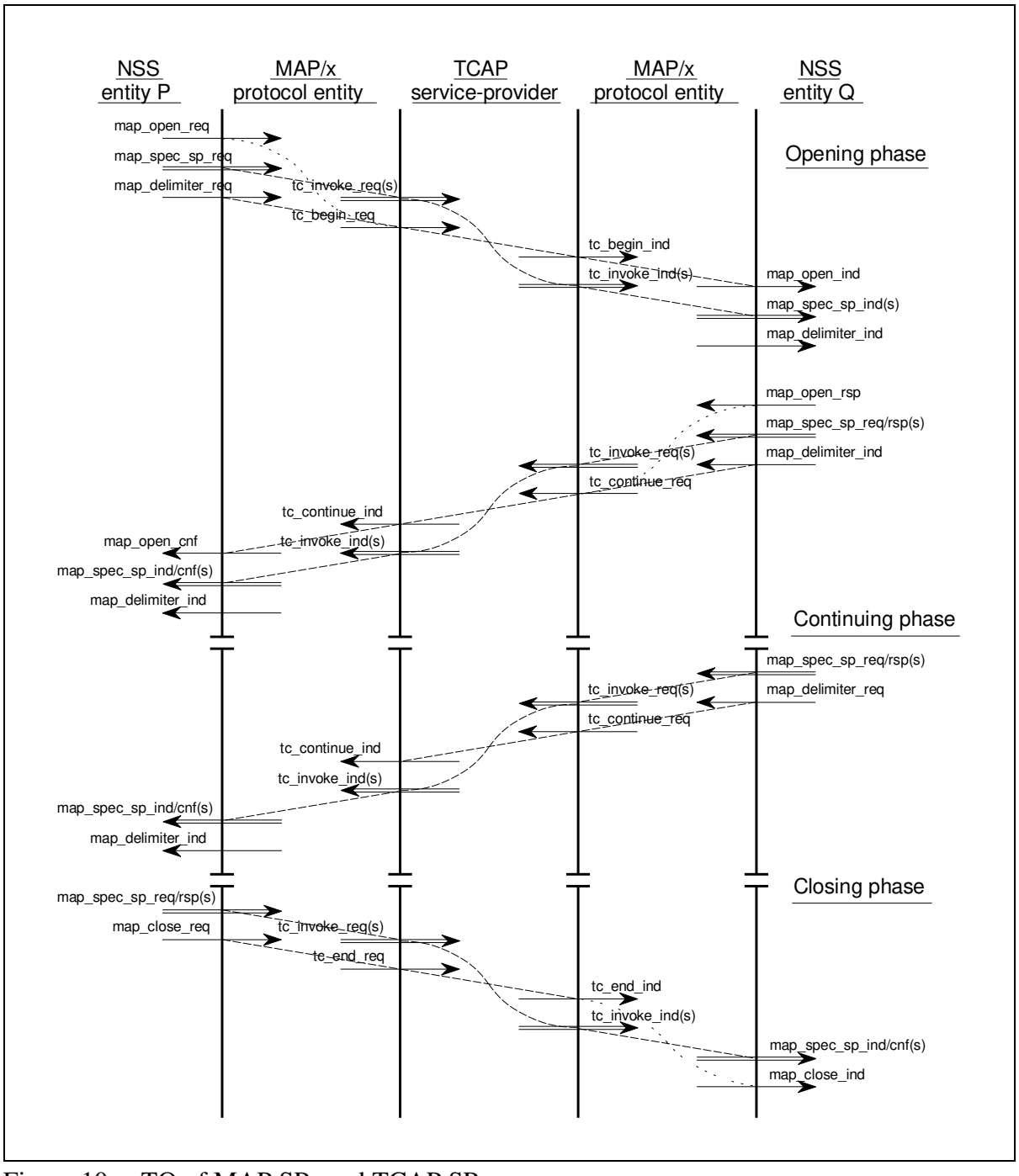

Figure 10: a TO of MAP SPs and TCAP SPs.

Note: this TO is very useful for specifying the MAP protocol(s), and service, using the resource-oriented and constraint-oriented styles. The thick vertical lines represent the SAPs. The resources can be identified by the space between those lines. The resources depicted in this figure are the MAP protocol entities and the underlying service provider TCAP. Every resource could be specified in a constraint-oriented way. The sequence of SPs at a SAP represent the local constraints, and the dashed lines represent the remote constraints.

6. MAP/E protocol description

In order to give a MAP/E protocol description, a service description is given first and subsequently the mapping between MAP/E SPs and TCAP SPs.

As mentioned the MAP/E service provides the means to exchange control data between MSCs in order to handover a call of a GSM user. The type of handovers which the MAP/E service-provider supports are *inter-MSC* and *subsequent inter-MSC handover*. The overall procedures of these handovers were described in chapter 4. In this chapter these procedures are described in terms of the performance of SPs for the exchange of control data between two or three MSCs.

The handover service provided by the MAP/E protocols is built from service elements, *map_perform_handover*, *map_perform_subsequent_handover*, and *map_send_end_signal*. These service elements are all of the confirmed service type. The purpose of these service elements (and their related SPs) will be clarified by the description of TOs of the related SPs. The common MAP SPs are not presented in these TOs. TOs including common MAP SPs can be derived by combining these TOs with the generic TO presented in figure 10.

Figure 11 presents a TO of *handover service* SPs of a successful inter-MSC handover. In this handover the call is handed over from the MSC-old (the anchor MSC), to MSC-new.

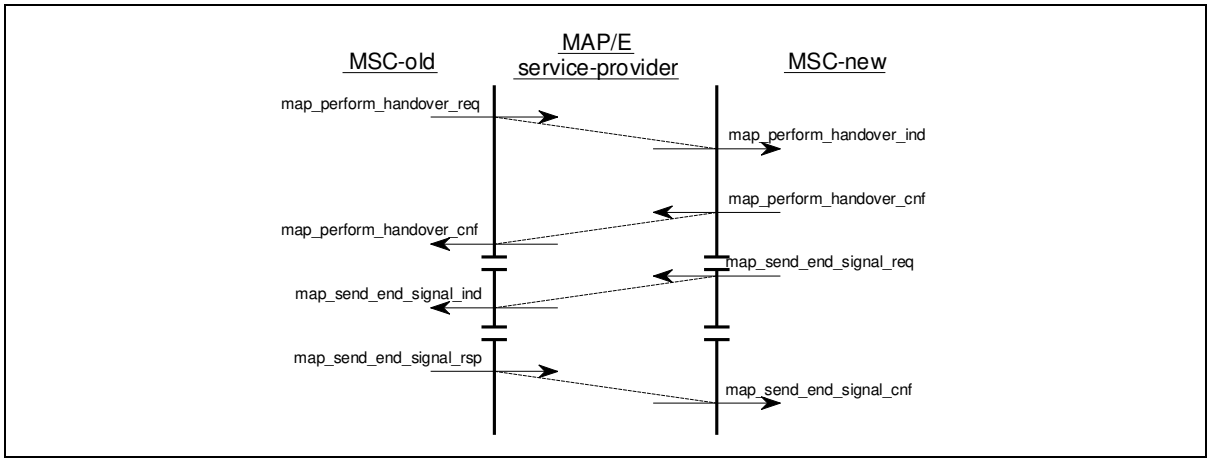

Figure 10: basic handover procedure.

When MSC-old has signalled that a call has to be handed over to MSC-new, it performs the *map_perform_handover_req* SP. This SP will contain all information required by MSC-new to establish a new communication path, and to allocate necessary radio resources.

After MSC-new received this request it performs the *map_perform_handover_rsp* SP to confirm that it is establishing the new communication path. When the new path is established, MSC-new indicates this event to MSC-old by performing the *map_send_end_signal_req* SP.

When a call is released MSC-old (the anchor MSC) will inform MSC-new about this by performing the *map_send_end_signal_rsp* SP. Subsequently the communication path will be released and the dialogue between the involved MSCs terminates.

Figures 12.a, b, and c show TOs of *handover service* SPs of a successful inter-MSC handover procedure followed by two successful *subsequent* inter-MSC handover procedures. In this case a GSM user engaged in a call wanders in a circlet by which a number of cells are passed served by three

different MSCs. Figure 12.a shows a TO in the case the GSM user wanders from a cell served by the anchor MSC into a cell served by MSC-B (inter-MSC handover). Figure 12.b shows the case when the GSM user wanders from a cell served by MSC-B into a cell served by MSC-B'. Figure 12.c shows the case when the GSM user wanders from the latter cell into a cell served by the anchor MSC again.

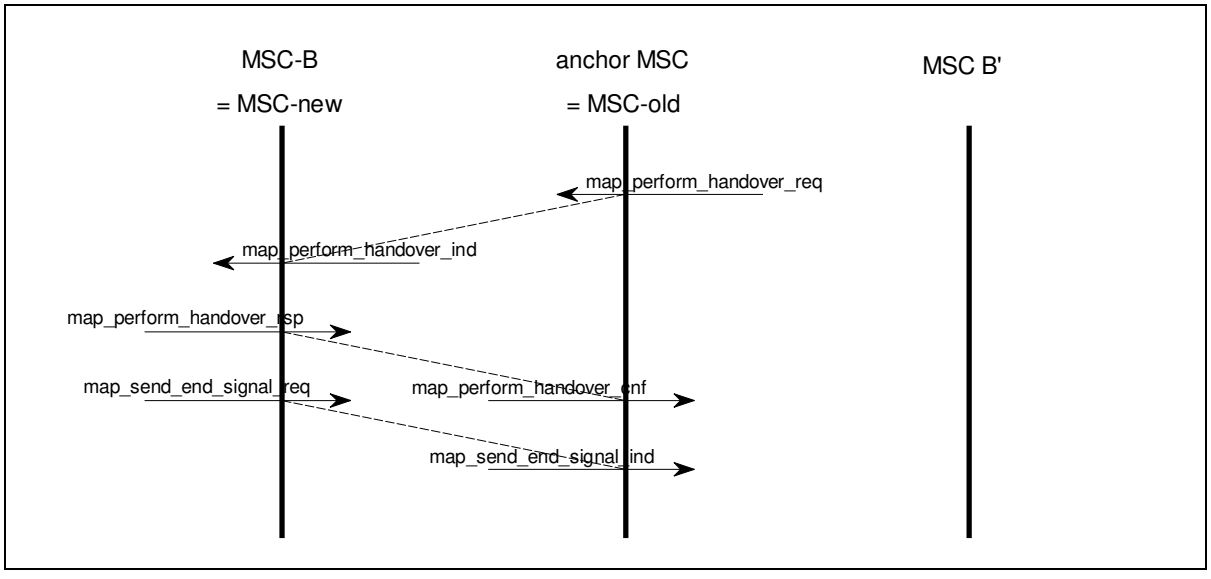

Figure 12.a: inter-MSC handover from the anchor MSC to MSC-B.

In figure 12.a the call is handed over from MSC-A to MSC-B. This is the same as presented in figure 11, except in this case the call is not terminated, or the GSM user did not move into a cell within an other MSC-area yet. Thus, it is not necessary yet to perform SPs *map_send_end_signal_rsp* and *map_send_end_signal_cnf*.

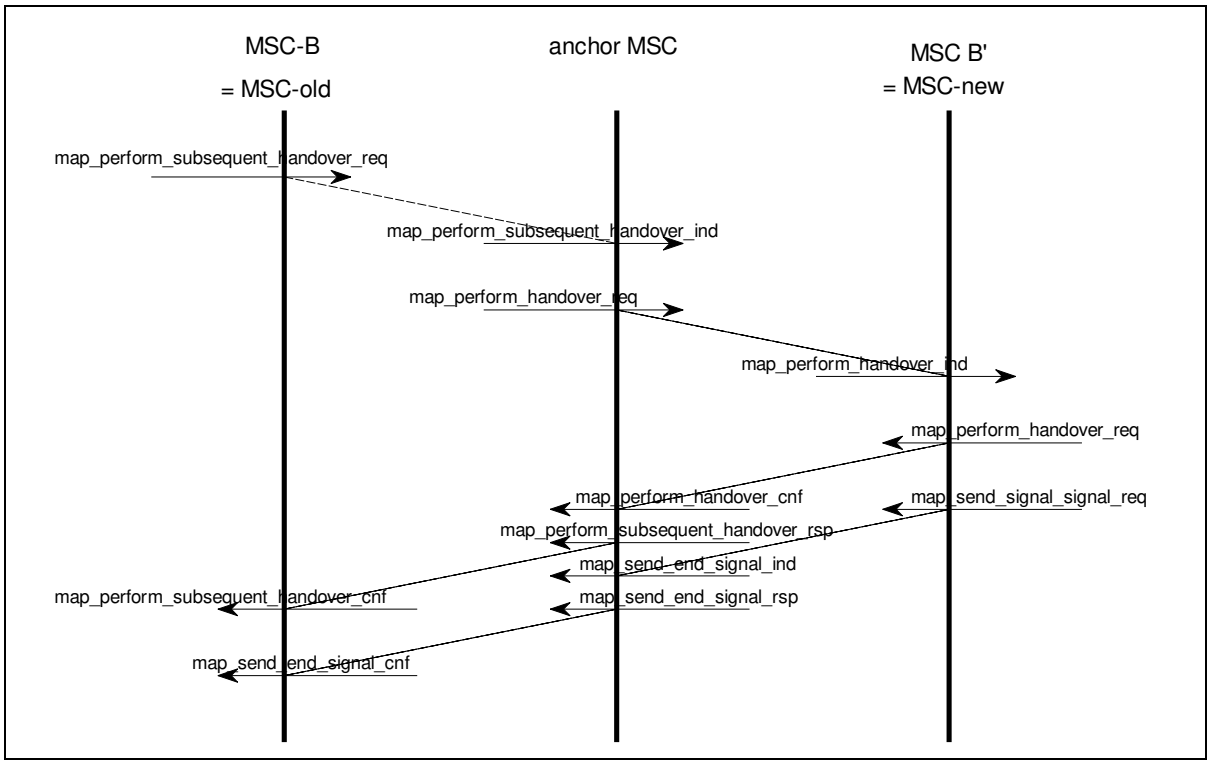

Figure 12.b: subsequent inter-MSC handover from MSC-B to MSC-B'.

In figure 12.b the call is handed over from MSC-B (MSC-old) to MSC-B' (MSC-new). When the need for an handover is signalled, MSC-old informs the anchor MSC by performing SP *map_perform_subsequent_handover_req*, and subsequently it waits for SP *map_send_ind_signal_cnf* which results in the termination of the dialogue between MSC-old and the anchor MSC. In between the anchor MSC sets up a new dialogue with MSC-new to which the call has to be handed over. The setting up of this dialogue is executed in the same as presented in figure 12.a. First, anchor MSC performs SP *map_perform_handover_req* to inform MSC-new about the incoming GSM user. After MSC-new is informed (by *map_perform_handover_ind*) it confirms this and established a new communication path. After the new path is established MSC-old performs SP *map_send_end_signal_req* to inform the anchor MSC about this. In the mean time the anchor MSC waited for the related SP *map_send_end_signal_ind*, after which it performs SP *map_send_end_signal_rsp* to inform MSC-old that the old communication path can be released and the dialogue between the anchor MSC and MSC-old can be terminated.

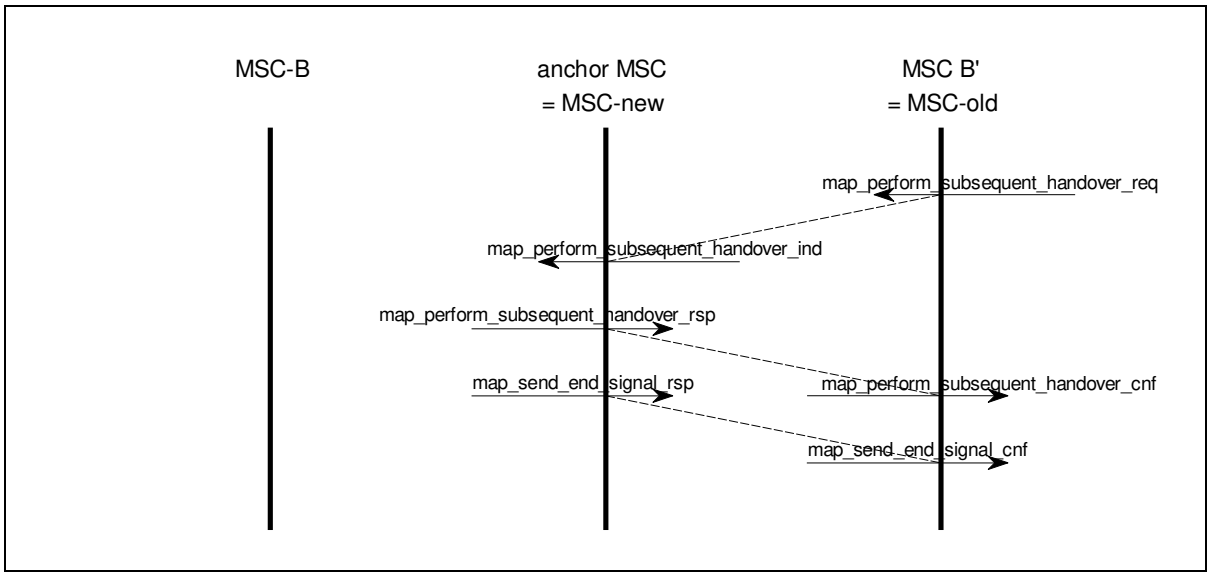

Figure 12.c: subsequent inter-MSC handover from MSC-B' to the anchor MSC.

In figure 12.c the call is handed over from MSC-B' (MSC-old) back to the anchor MSC (MSC-new). In the former case of a subsequent inter-MSC handover three MSCs were involved, but in this case of a subsequent inter-MSC handover only two MSCs are involved. A handover could be called *subsequent* inter-MSC handover in the case the anchor MSC is not the initiator. In the other case it is called inter-MSC handover (without the suffix subsequent). MSC-old initiates the handover by performing SP *map_subsequent_handover_req* in order to inform MSC-new about this. Subsequently it will wait for the confirmation (*map_perform_subsequent_handover_cnf*) and for SP *map_send_end_signal_cnf* after which it can release the old communication path and the dialogue with MSC-new. This confirmation is the result of the request it has sent when it was MSC-new (see figure 12.b). MSC-new (anchor MSC) acts exactly as the former MSC-new (MSC-B') in the former case (see figure 12.b).

The *handover service* SPs are mapped onto TCAP SPs according to the mapping given in section 5.3. TOs of SPs, with common and specific MAP SPs and TCAP SPs, of a successful inter-MSC handover procedure is given in figures 13, 14 and 15. These figures are an example of combining the generic TO of SPs presented in figure 7 with the TO of SPs of a successful inter-MSC handover presented in figure 11.

Concerning the parameters, parameter *application context name* is instantiated with a value called *handover control*. The SPs containing this parameter are the SPs related to the common MAP service elements *map_open* and *map_close*. The TCAP SPs which are mapped onto these common MAP SPs will also contain this value. See also the mapping presented in section 5.3.

The TCAP SPs related to service element *tc_invoke* contain the parameter *invoke identifier*. This parameter is instantiated with a value called *perform_handover* for those *tc_invoke* SPs which are mapped onto the MAP SPs related to service element *map_perform_handover*. For service element *map_perform_subsequent_handover* this is *perform_subsequent_handover*, and for service element *map_send_end_signal* this is *send_end_signal*.

Figure 13 presents a TO of SPs when a MAP/E protocol entered the opening phase. In figure 14 a MAP/E protocol entered the continuing phase and in figure 15 the closing phase.

Opening phase

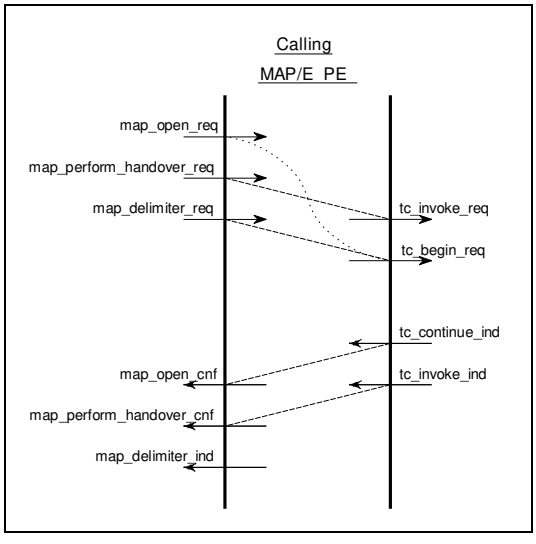

Figure 13.a: a TO of SPs performed by a calling MAP/E PE during the opening phase.

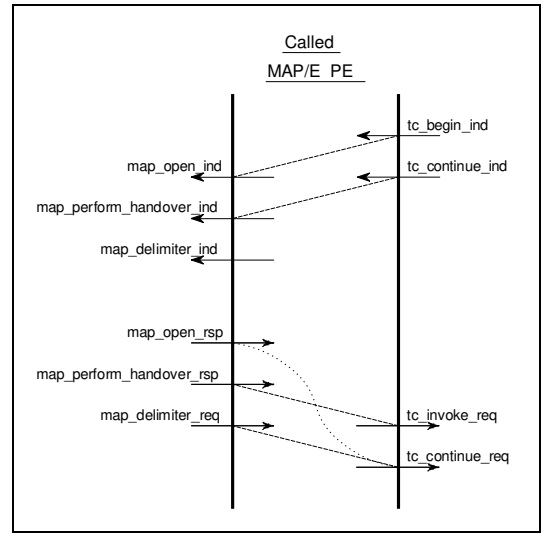

Figure 13.b: a TO of SPs performed by a called MAP/E PE during the opening phase.

Continuing phase

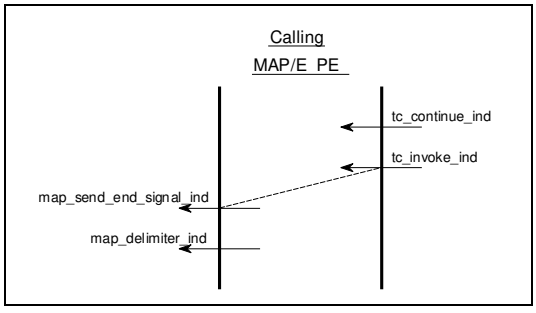

Figure 14.a: a TO of SPs performed by a calling MAP/E PE during the continuing phase.

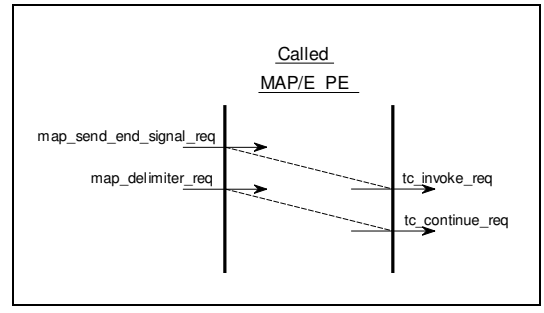

Figure 14.b: a TO of SPs performed by a called MAP/E PE during the continuing phase.

Closing phase

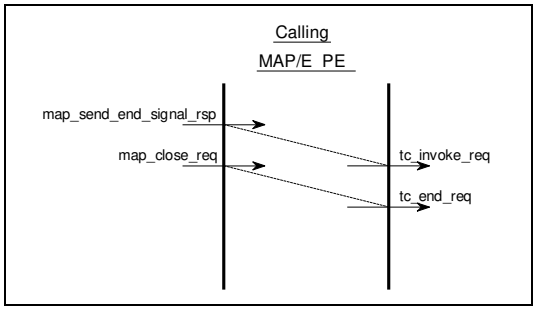

Figure 15.a: a TO of SPs performed by a MAP/E PE during the closing phase.

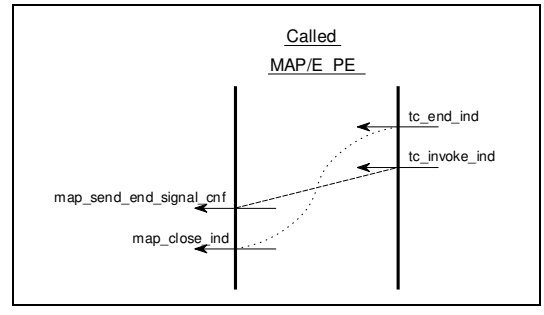

Figure 15.b: a TO of SPs performed by a MAP/E PE during the closing phase.

7. Remarks

This chapter gives some remarks with respect to MAP/x protocol descriptions. These remarks could be considered in order to obtain more complete descriptions of MAP/x protocols. It would lead to more complete LOTOS specifications compared to the one presented in the next part, and accordingly more and better test cases for conformance testing MAP/x protocol implementations.

In chapter 5 it is mentioned that more parameters exist besides the parameters *application context name* and *invoke identifier*. With these parameters it was already possible to handle one dialogue. However in order to handle multiple dialogues these parameters are not enough. To adopt this requirement the descriptions given in the former chapters have to extended (orthogonal).

For example one could consider source and destination addresses. Nevertheless, the SPs *map_open-_req* and *map_open_ind* are the only SPs which have such parameters. However this could be used to set up a *connection* between two MAP/x service user. Subsequently SPs can be performed and data can be sent regardless (of) the addresses. With respect to this, [MP] mentions that MAP/x service users communicate by means of virtual circuits, and [ETSI] mentions that the MAP protocol layer makes use of the connection-less services provided by TCAP. A MAP/x protocol handling multiple dialogues could be visualized as in figure 16 below.

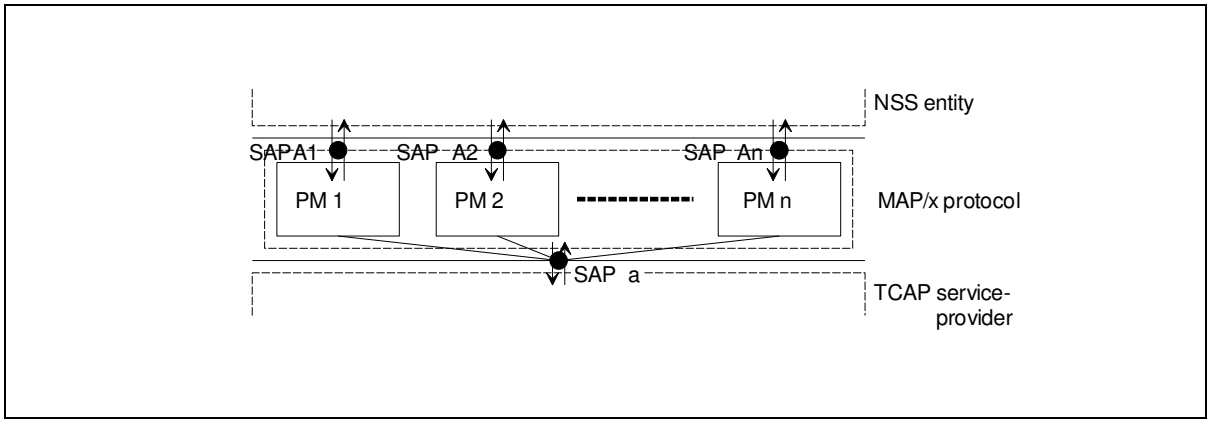

Figure 16: MAP/x protocol machines.

[ETSI] also mentions that for each dialogue a MAP/x protocol starts up a new so called *protocol machine* (PM). In figure 16 the squares could be thought of such PMs. It is also mentioned that each PM is mapped onto a *dialogue identifier*. However [ETSI] is not clear at describing in what way this identifier is used. It is mentioned in a part which describes TCAP; TCAP uses this in order to distinguish between independent message flows, each belonging to a separate dialogue. So it could be assumed that it is used as a parameter of TCAP SPs, but does not mention which SPs. [CCITT] is probably clearer at this point.

Points of considerations for future enhancement and extension of MAP/x protocol descriptions should be on the area of identification of SAPs between MAP/x as a service-provider and service user and how this is mapped onto TCAP SPs.

Besides it should be considered if MAP/x protocols make use of PDUs. It probably does, but [ETSI] does not mention PDUs and PDU formats. However it is mentioned that some MAP SP-parameters are put together by MAP/x protocols after which they are mapped onto a single parameter of one or more TCAP SPs. In conformance testing MAP/x protocols it is of importance that this knowledge is present.

Part III: LOTOS specification of MAP/E protocol & service-provider

The inter-MSC handover procedure in its entirety is briefly described in chapter 4. A typical dialogue between two MSCs involved in a handover procedure has been stressed out further in terms of the performance of SPs in chapter 6. The MAP/E service-provider supporting such a dialogue is modelled in LOTOS and presented in this part. The decomposition of this service specification consists of an underlying service and two MAP/E protocol entities.

The LOTOS specification is validated by observation. The LOTOS simulators SMILE and SELA are used to derive symbolic simulation trees which are unfolded step by step. After each step it is checked whether the LOTOS specification showed the required behaviour or not.

Chapter 8 presents the LOTOS specification of the MAP/E service-provider. Chapter 9 describes the usefulness of the LOTOS specification for conformance testing a MAP/E protocol implementation.

8. Service decomposition

The service-provider is modelled using the resource-oriented specification style. Two MAP/E protocol entities and the TCAP service provider are the resources, and each is captured in a separate process. Process *TCAPServices* models the TCAP service-provider and serves as the underlying service. Process *MAPE_PE* models a MAP/E protocol entity. Each process possesses two gates which present the SAPs. Gate *AA* of the one protocol entity and gate *BB* of the other represent the SAPs for the interaction between MAP/E service-user and service-provider. Gate *a* and *b* represent the SAPs for the interaction between MAP/E protocol entities and the underlying service.

The decomposition of the MAP/E service-provider is sketched in figure 17 Each square represents a resource. Also in this figure presented are MAP/E service users MSC-A and MSC-B. The bullets in this figure present the SAPs, and the arrows indicate the SPs. The LOTOS code presenting this decomposition is given below.

```
behaviour
   MAP_E_Service[AA, BB]

where
   process MAP_E_Service[AA, BB] : noexit :=
   hide a, b in
      (MAPE_PE[AA, a] ||| MAPE_PE[BB, b])
       |[a, b]|
      TCAPServices[a, b]
   endproc
endspec
```

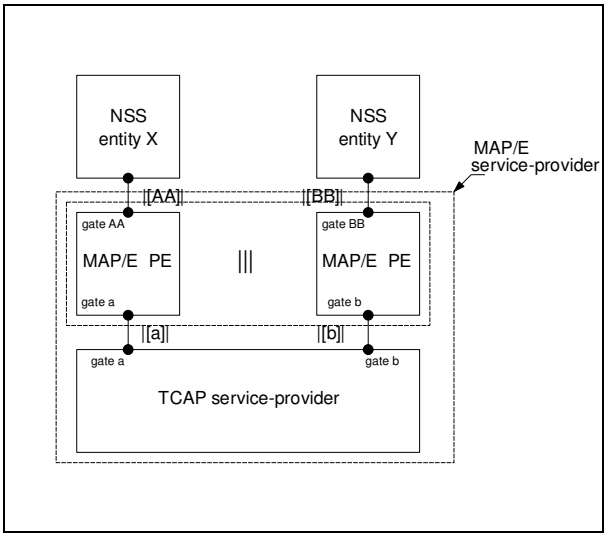

Figure 17: MAP/E service decomposition.

Processes which may operate independently, and concurrently are composed using the parallel composition operator '|||'. When such processes need to communicate they are composed using the parallel composition operator '|[..]|'. The exchange of data is established by the synchronisation on the gates listed between the brackets.

Interprocess communication is established as follows. Processes *A* and *B* are able to synchronize on gate *x*. In the case process *A* wants to pass a value to process *B*, *A* contains a line such as: **x ! value**, and *B* contains a line such as **x ? v:ValueType**. After synchronisation **v** shall posses **value**. The condition is that **value** must be of type **ValueType**.

Values and value types are described in the Abstract Data Type part (ADT part) of the LOTOS specification. As mentioned in the former part the exchange of data is established by the performance of SPs. The exchange of data by the performance of SPs is modelled in LOTOS as described in the former paragraph. In this case **value** will be an SP, and is described in the ADT part. A gate in a receiving process additionally has a guard. This guard only allows the performance of a particular SP. A sending process will have a typical line such as **gate ! service_primitive**, and a receiving process will have a typical line such as **gate ? sp:SPType [ConstraintsOn(sp)]**. A function such as **ConstraintsOn** is also described in the ADT part.

The concepts described above are used in the LOTOS descriptions presented in the following sections. Section 8.1. presents the LOTOS specification of the TCAP service-provider, and section 8.2. presents the LOTOS specification of a MAP/E protocol entity.

8.1. LOTOS spec of TCAP service-provider

Figure 9 in section 5.2.2. presented a typical TO of TCAP SPs in the case two service-users exchange data using the TCAP service-provider. In summary, a sending user performs a number of *tc_invoke_req* SPs, followed by one of *tc_begin_req*, *tc_continue*, or *tc_end_req* SPs. After performance of one of the latter SPs data related to *tc_invoke_req* SPs is transferred to the peer user. This transfer results in the performance of indication SPs at the receiving side. First either one of *tc_begin_ind*, *tc_continue_ind*, or *tc_end_ind* SPs, followed by a number of *tc_invoke_ind* SPs.

In order to describe how such a transfer is captured in LOTOS, it is divided in two phases.

- In a first phase, a number of *tc_invoke_req* SPs is performed, and stored in a FIFO queue, called *inqueue*.

  A sequence of *tc_invoke_req* SPs is captured in the following LOTOS description.

```
TCAPService[a,b](inqueue:TCAPQ) := a ? sp:TCAP [IsTcInvokeReq(sp)];
                                        TCAPService[a,b] (add(sp, inqueue))
```

  The meaning of rule: *a ? sp:TCAP [IsTcInvokeReq(sp)]*, is already described in the beginning of chapter 8. In a nutshell, it means that only value *sp* (an SP) of type *TCAP* can be generated at gate *a* (SAP a). Besides, this value can only be generated if the guard yields true. The only value yielding true is *tc_invoke_req*, which represents SP *tc_invoke_req*. After the performance of an SP the process is re-entered recursive. Every SP is added on top of *inqueue*. In principle, the recursion allows the performance of an infinite number of SPs.

  Besides *inqueue*, there is another queue called *outqueue*. This queue is used in the second phase to perform indication SPs. As mentioned, a sequence of *tc_invoke_req* SPs is delimited by one of the SPs *tc_begin_req*, *tc_continue_req*, or *tc_end_req*. When such an SP is performed it is not added on top of *inqueue*, but on the bottom. The performance of such an SP is followed by a new process invocation. This time, the new value of *outqueue* is the concatenation of *inqueue* and the old value of *outqueue*, and the new value of *inqueue* is empty. Subsequently the second phase is entered.

  The possible performance of TCAP request SPs is captured in the following LOTOS description.

```
TCAPService[a,b](inqueue:TCAPQ, outqueue:TCAPQ) :=
                        a ? sp:TCAP [IsTcInvokeReq(sp)];
                               TCAPService[a,b] (add(sp, inqueue), outqueue)
                      [] a ? sp:TCAP [IsTcBeginReq(sp)];
                               TCAPService[a,b] (empty, concat(dda(sp, inqueue), outqueue))
                      [] a ? sp:TCAP [IsTcContinueReq(sp)];
                               TCAPService[a,b] (empty, concat(dda(sp, inqueue), outqueue))
                      [] a ? sp:TCAP [IsTcEndReq(sp)];
```

```
                                    TCAPService[a,b] (empty, concat(dda(sp, inqueue), outqueue))
```

- In a second phase *outqueue* will be dequeued one by one until it is empty. Every dequeued request SP results in the performance of the related indication SP at gate *b*. After the performance of one indication SP the process will be invoked recursive, with a new *outqueue* containing one request SP less then the old *outqueue*.

The possible performance of TCAP indication SPs is captured in the following LOTOS description.

```
TCAPService[a,b](inqueue:TCAPQ, outqueue:TCAPQ) :=
                     [NotEmpty(outqueue)] ->
                     (let sp:TCAP = last(outqueue) in
                      b ! tc_invoke_ind(invokeIdOf(sp), ArgumentOf(sp)) [IsTcInvokeReq(sp)];
                              TCAPService[a,b] (inqueue, init(outqueue))
                   [] b ! tc_begin_ind(ACOf(sp), UserInfoOf(sp)) [IsTcBeginReq(sp)];
                              TCAPService[a,b] (inqueue, init(outqueue))
                   [] b ! tc_continue_ind(ACOf(sp), UserInfoOf(sp)) [IsTcContinueReq(sp)];
                              TCAPService[a,b] (inqueue, init(outqueue))
                   [] b ! tc_end_ind [IsTcEndReq(sp)];
                              TCAPService[a,b] (inqueue, init(outqueue))
```

The TCAP service-provider must provide the ability for the service-users to perform SPs indepently and concurrently. Therefore, the possibility exist that during the performance of indication SPs at gate *b*, new request SPs can be performed at gate *a*. This is accomplished by putting together the above presented LOTOS descriptions in a single process (process *UniTCAPService*). Besides, the service must provide bi-directional and concurrent message flows. This is accomplished by putting two of those processes in parallel, of which one provides the exchange of data from SAP *a* to SAP *b*, and the other from SAP *b* to SAP *a* (process *TCAPService*). The complete LOTOS specification of the TCAP service provider is given below.

```
process TCAPService[a, b] : noexit :=
   UniTCAPService[a, b](empty, empty)
   |||
   UniTCAPService[a, b](empty, empty)

where
   process UniTCAPService[a,b](inqueue:TCAPQ, outqueue:TCAPQ) : noexit :=
         a ? sp:TCAP [IsTcInvokeReq(sp)];
                     UniTCAPService[a,b] (add(sp, inqueue), outqueue)
      [] a ? sp:TCAP [IsTcBeginReq(sp)];
                     UniTCAPService[a,b] (empty, concat(dda(sp, inqueue), outqueue))
      [] a ? sp:TCAP [IsTcContinueReq(sp)];
                     UniTCAPService[a,b] (empty, concat(dda(sp, inqueue), outqueue))
      [] a ? sp:TCAP [IsTcEndReq(sp)];
                     UniTCAPService[a,b] (empty, concat(dda(sp, inqueue), outqueue))
      [] [NotEmpty(outqueue)] ->
      (  let sp:TCAP = last(outqueue) in
         b ! tc_invoke_ind(invokeIdOf(sp), ArgumentOf(sp)) [IsTcInvokeReq(sp)];
                     UniTCAPService[a,b] (inqueue,  init(outqueue))
      [] b ! tc_begin_ind(ACOf(sp), UserInfoOf(sp)) [IsTcBeginReq(sp)];
                     UniTCAPService[a,b] (inqueue,  init(outqueue))
      [] b ! tc_continue_ind(ACOf(sp), UserInfoOf(sp)) [IsTcContinueReq(sp)];
                     UniTCAPService[a,b] (inqueue,  init(outqueue))
      [] b ! tc_end_ind [IsTcEndReq(sp)];
                     UniTCAPService[a,b] (inqueue,  init(outqueue))
      )
   endproc (* UniTCAPService *)
endproc (* TCAPService *)
```

8.2. LOTOS specification of MAP/E protocol entity

A MAP/E protocol entity is specified in a constraint-oriented style. The local constraints for the upper SAP (gate X) are captured in process *UpperLSI*, and the local constraints for the lower SAP (gate y) are captured in process *LowerLSI*. *LSI*, used in the process names, stands for *Local Service Interface*. The remote constraints are captured in process *PFs*. *PFs* stands for *Protocol Functions*. The LOTOS description presenting the decomposition of a MAP/E PE is given below.

```
process PE[X, y] : noexit :=
   (UpperLSI[X] ||| LowerLSI[y])
   ||
   PFs[X, y]
endproc
```

Besides *handover control*, the MAP/E protocol provides a few other services. Every constraints part of every single service could be captured in separate processes. For example, process *UpperLSI* is decomposed in several processes each containing the upper local constraints related to a specific service. In this way the LOTOS specification can be extended systematically. The LOTOS code which presents this idea is given below. The italicized lines are not used in the current LOTOS specification.

```
process UpperLSI[X] : noexit :=
       HandoverULSI[X]
   ||| OtherServiceULSI[X]
endproc

process PFs[X, y] : noexit :=
       HandoverPFs[X, y]
   ||| OtherServicePFs[X, y]
endproc

process LowerLSI[y] : noexit :=
       HandoverLLSI[y]
   ||| OtherServiceLLSI[y]
endproc
```

8.2.1. Local constraints

The role of a MAP/E PE in a dialogue can be *calling* as well as *called*. Therefore the processes containing the constraints are decomposed further into two concurrent processes. One process contains the constraints for the *calling* role, and the other process contains the constraints for the *called* role. The name of a process containing the calling constraints has suffix *Calling*, and the other process' name has a suffix *Called*. Such a decomposition for the local constraints (upper and lower) is given below. The remote constraints are decomposed the same way. In figure 18 a visualization of this decomposition is given.

```
process UpperLSI[X] : noexit :=
   HandoverULSI[X]
where
   process HandoverULSI[X] : noexit :=
      CallingHOULSI[X] ||| CalledHOULSI[X]
   endproc
endproc


process LowerLSI[y] : noexit :=
   HandoverLLSI[y]
where
   process HandoverLLSI[y] : noexit :=
      CallingHOLLSI[y] ||| CalledHOLLSI[y]
   endproc
endproc
```

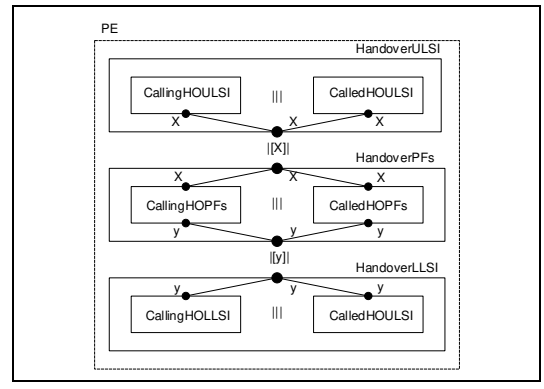

Figure 18: decomposition of MAP/E PE.

In chapter 6 a protocol description of MAP/E is given. The presented TOs are used to derive a LOTOS specification of this protocol. The vertical lines indicate SAPs and the arrows crossing the vertical lines are the SPs performed at those SAPs. These particular SPs and the sequences in which they are performed at particular SAPs are the local constraints. In the LOTOS specification of the MAP/E PE each particular sequence is captured in one of the processes presented above. The contents of these processes are given below.

```
process CallingHOULSI[X] : noexit :=
   X ? sp:MAPCS ! Calling [IsMapOpenReq(sp) and (ACOf(sp) eq Handover_Control)];
   X ? sp:MAPSS ! Calling [IsMapPerformHandoverReq(sp)];
   X ? sp:MAPCS ! Calling [IsMapDelimiterReq(sp)];
   X ? sp:MAPCS ! Calling [IsMapOpenCnf(sp) and (ACOf(sp) eq Handover_Control)];
   X ? sp:MAPSS ! Calling [IsMapPerformHandoverCnf(sp)];
   X ? sp:MAPCS ! Calling [IsMapDelimiterInd(sp)];
   X ? sp:MAPSS ! Calling [IsMapSendEndSignalInd(sp)];
   X ? sp:MAPCS ! Calling [IsMapDelimiterInd(sp)];
   X ? sp:MAPSS ! Calling [IsMapSendEndSignalRsp(sp)];
   X ? sp:MAPCS ! Calling [IsMapCloseReq(sp)]; CallingHOULSI[X]
endproc (*  CallingULSI *)

process CalledHOULSI[X] : noexit :=
   X ? sp:MAPCS ! Called [IsMapOpenInd(sp) and (ACOf(sp) eq Handover_Control)];
   X ? sp:MAPSS ! Called [IsMapPerformHandoverInd(sp)];
   X ? sp:MAPCS ! Called [IsMapDelimiterInd(sp)];
   X ? sp:MAPCS ! Called [IsMapOpenRsp(sp) and (ACOf(sp) eq Handover_Control)];
   X ? sp:MAPSS ! Called [IsMapPerformHandoverRsp(sp)];
   X ? sp:MAPCS ! Called [IsMapDelimiterReq(sp)];
   X ? sp:MAPSS ! Called [IsMapSendEndSignalReq(sp)];
   X ? sp:MAPCS ! Called [IsMapDelimiterReq(sp)];
   X ? sp:MAPSS ! Called [IsMapSendEndSignalCnf(sp)];
   X ? sp:MAPCS ! Called [IsMapCloseInd(sp)]; CalledHOULSI[X]
endproc (* CalledHOULSI *)



process CalledHOLLSI[y] : noexit :=
   y ? sp:TCAP ! Called [IsTcBeginInd(sp) and (ACOf(sp) eq Handover_Control)];
   y ? sp:TCAP ! Called [ IsTcInvokeInd(sp) and
                          (InvokeidToOpcode(InvokeIdOf(sp)) eq PerformHandover)];
   y ? sp:TCAP ! Called [ IsTcInvokeReq(sp) and
                          (InvokeidToOpcode(InvokeIdOf(sp)) eq PerformHandover)];
   y ? sp:TCAP ! Called [IsTcContinueReq(sp) and (ACOf(sp) eq Handover_Control)];
   y ? sp:TCAP ! Called [ IsTcInvokeReq(sp) and
                          (InvokeidToOpcode(InvokeIdOf(sp)) eq SendEndSignal)];
   y ? sp:TCAP ! Called [IsTcContinueReq(sp) and (ACOf(sp) eq Handover_Control)];
   y ? sp:TCAP ! Called [IsTcEndInd(sp)];
   y ? sp:TCAP ! Called [ IsTcInvokeInd(sp) and
                          (InvokeidToOpcode(InvokeIdOf(sp)) eq SendEndSignal)];
   CalledHOLLSI[y]
endproc (* CalledHOLLSI *)

process CallingHOLLSI[y] : noexit :=
   y ? sp:TCAP ! Calling [ IsTcInvokeReq(sp) and
                          (InvokeidToOpcode(InvokeIdOf(sp)) eq PerformHandover)];
   y ? sp:TCAP ! Calling [IsTcBeginReq(sp) and (ACOf(sp) eq Handover_Control)];
   y ? sp:TCAP ! Calling [IsTcContinueInd(sp) and (ACOf(sp) eq Handover_Control)];
   y ? sp:TCAP ! Calling [ IsTcInvokeInd(sp) and
                          (InvokeidToOpcode(InvokeIdOf(sp)) eq PerformHandover)];
   y ? sp:TCAP ! Calling [IsTcContinueInd(sp) and (ACOf(sp eq Handover_Control)];
```

```
      y ? sp:TCAP ! Calling [ IsTcInvokeInd(sp) and
                            (InvokeidToOpcode(InvokeIdOf(sp)) eq SendEndSignal)];
      y ? sp:TCAP ! Calling [ IsTcInvokeReq(sp) and
                            (InvokeidToOpcode(InvokeIdOf(sp)) eq SendEndSignal)];
      y ? sp:TCAP ! Calling [IsTcEndReq(sp)];
      CallingHOLLSI[y]
   endproc (* CallingHOLLSI *)
```

The rule describing the possible occurrence of an SP is the same as given in the beginning of chapter 8. However, in the LOTOS descriptions of the constraints, such a rule has an additional value, namely *Calling* and *Called*. This value has to be used because calling constraints processes, and called constraints processes may contain the same SP descriptions. If no care is taken, gates with such SP descriptions which are not supposed to synchronize could synchronize although. For example, *X ? sp:MAPCS [IsMapDelimiterReq(sp)]* in process *CallingHOULSI*, and *X ? sp:MAPCS [IsMapDelimiterReq(sp)]* in process *CalledPFs*, could synchronize if values *Calling* and *Called* were not used.

8.2.2. Remote constraints

The remote constraints (contained in process *PFs*) describe the relation between MAP SPs performed at the upper SAP and TCAP SPs performed at the lower SAP of the MAP/E PE. The relation between these SPs is described in chapter 6. In TOs in this chapter these relations are presented by dashed lines. For example, the performance of *map_spec_sp_req* SP at the upper SAP, results in the performance of *tc_invoke_req* SP at the lower SAP.

A typical remote constraint is captured in LOTOS by a rule like:

```
uppergate ? sp:SPType [ConstraintsOn(sp)] ; lowergate ! service_primitive
```

The LOTOS processes containing the remote constraints are presented below.

```
process PFs[X, y] : noexit :=
   HandoverPFs[X, y]

where
   process HandoverPFs[X, y] : noexit :=
      CallingHOPFs[X, y] ||| CalledHOPFs[X, y]

   where
      process CallingHOPFs[X, y] : noexit :=
         X ? sp1:MAPCS ! Calling [IsMapOpenReq(sp1) and (ACOf(sp1) eq Handover_Control)];
         X ? sp2:MAPSS ! Calling [IsMapPerformHandoverReq(sp2)];
         y ! tc_invoke_req(InvokeIdOf(sp2), arg_1) ! Calling; stop
         |||
         X ? sp3:MAPCS ! Calling [IsMapDelimiterReq(sp3)];
         y ! tc_begin_req(Handover_Control, ui_1 ) ! Calling; stop
         |||
         y ? sp1:TCAP ! Calling [IsTcContinueInd(sp1)];       (* opening phase *)
         X ! map_open_cnf(ACOf(sp1), op_1) ! Calling;
         y ? sp1:TCAP ! Calling [IsTcContinueInd(sp1)]; stop  (* continuing phase *)
         |||
         y ? sp2:TCAP ! Calling [ IsTcInvokeInd(sp2) and
                                (InvokeidToOpcode(InvokeIdOf(sp2)) eq PerformHandover)];
         X ! map_perform_handover_cnf(InvokeIdOf(sp2 of TCAP), op_1) ! Calling;
         X ! map_delimiter_ind ! Calling; stop
         |||
         y ? sp1:TCAP ! Calling [ IsTcInvokeInd(sp1) and
                                (InvokeidToOpcode(InvokeIdOf(sp1)) eq SendEndSignal)];
         X ! map_send_end_signal_ind(InvokeIdOf(sp1)) ! Calling;
         X ! map_delimiter_ind ! Calling; stop
         |||
         X ? sp1:MAPSS ! Calling [IsMapSendEndSignalRsp(sp1)];
         y ! tc_invoke_req(InvokeIdOf(sp1), arg_1) ! Calling; stop
         |||
         X ? sp2:MAPCS ! Calling [IsMapCloseReq(sp2)];
         y ! tc_end_req ! Calling;  CallingHOPFs[X, y]
      endproc (* CallingHOPFs *)

      process CalledHOPFs[X, y] : noexit :=
         y ? sp1:TCAP ! Called [IsTcBeginInd(sp1) and (ACOf(sp1) eq Handover_Control)];
         X ! map_open_ind(ACOf(sp1), op_1) ! Called; stop
```

```
        |||
        y ? sp2:TCAP ! Called [ IsTcInvokeInd(sp2) and
                              (InvokeidToOpcode(InvokeIdOf(sp2)) eq PerformHandover)];
        X ! map_perform_handover_ind(InvokeIdOf(sp2), op_1) ! Called;
        X ! map_delimiter_ind ! Called; stop
        |||
        X ? sp1:MAPCS ! Called [IsMapOpenRsp(sp1) and (ACOf(sp1) eq Handover_Control)];
        X ? sp2:MAPSS ! Called [IsMapPerformHandoverRsp(sp2)];
        y ! tc_invoke_req(InvokeIdOf(sp2), arg_1) ! Called; stop
        |||
        X ? sp3:MAPCS ! Called [IsMapDelimiterReq(sp3)];           (* Opening phase *)
        y ! tc_continue_req(Handover_Control, ui_1) ! Called; stop
        |||
        X ? sp3:MAPCS ! Called [IsMapDelimiterReq(sp3)];           (* Continuing phase *)
        y ! tc_continue_req(Handover_Control, ui_1) ! Called; stop
        |||
        X ? sp1:MAPSS ! Called [IsMapSendEndSignalReq(sp1)];
        y ! tc_invoke_req(InvokeIdOf(sp1), arg_1) ! Called; stop
        |||
        y ? sp1:TCAP ! Called [IsTcEndInd(sp1)];
        y ? sp2:TCAP ! Called [ IsTcInvokeInd(sp2) and
                              (InvokeidToOpcode(InvokeIdOf(sp2)) eq SendEndSignal)];
        X ! map_send_end_signal_cnf(InvokeIdOf(sp2), op_1) ! Called;
        X ! map_close_ind ! Called; CalledHOPFs[X, y]
      endproc (* CalledHOPFs *)
   endproc (* HandoverPFs *)
endproc (* PFs *)
```

9. Conformance testing

Conformance testing involves verifying by means of systematic testing whether the observable behaviour of the *implementation under test* (IUT) conforms to the requirements contained in its protocol description. The aim can be formulated as enlarging the probability (and confidence) that an implementation satisfies its specification, and consequently, enlarging the probability that the implementation interworks with other implementations.

In order to derive abstract test cases the symbolic simulation trees could be used which are generated by a LOTOS simulator (SMILE, SELA). Such trees present the possible sequences of SPs.

Although the presented LOTOS specification is yet not complete it can already be used to derive test cases. However the test purposes are limited. For example only requirements to support the normal course of one dialogue for the purpose of handing over a call can be tested.

After othogonal extension of the current LOTOS specification the normal course of a dialogue does not change substantially. Thus, simulation trees which express the observable behaviour of a MAP/E protocol engaged in such a dialogue will not have substantial differences either. At most the SPs will contain more parameters with other information than information related to the maintenance of one dialogue. The other parameters could contain information related to e.g., cell identities. Since test cases are derived from these simulation trees, test cases could also stay the same. Even after extending the LOTOS specification, the test cases will contain the same required and/or allowed sequences of SPs. They will only contain some more information.

In order to use abstract test cases to test a protocol implementation the abstract SPs of the these test cases have to be translated to physical messages useable for the tester. This requires knowledge about message formats, described in the recommendations (in [ETSI] and [CCITT]), and their implementations (data structures and system calls). It can be assumed that such a message is related to an abstract SP by means of an appropriate message field. Besides, there will exist fields containing SP-parameters. Specific to MAP SPs and TCAP SPs datafields should exist which contain the values of parameters *invoke identifier* and *application context name*, which are relevant to the maintenance of one dialogue.

As mentioned, there exist more parameters but these are not captured in the current specification yet. Independently from information contained by these parameters (the other data fields), every normal dialogue will always exhibit the same progress. Consequently, as mentioned above, the involved MAP/E protocol will exhibit the same observable behaviour too. So by only looking at the message fields containing the SP types, and the parameters relevant to the maintenance of a dialogue, the abstract test cases could be used. Although the current LOTOS specification can already be used to derive such test cases, in a later stage the mapping of abstract SPs onto physical messages will have to be described, before test cases actually could be executed. Also what have to be taken into account are *source* and *destination addresses*. This can be added to the abstract test cases in a later stage too.

However new test cases are needed when the MAP/E protocol specification is extended in a way that it can support multiple dialogues, each invoking a particular service. For instance, two dialogues each for the execution of a single handover procedure. In chapter 7 considerations are given which could be used to accomplish this.

The LOTOS specification presented in chapter 8 not only contains the specification of a MAP/E protocol entity but also of the TCAP service-provider which serves as an underlying service. This enables the derivation of test cases with respect to *the distributed single layer testing architecture*. This test architecture is presented in figure 20, and in figure 19 *the local testing architecture* is given.

Although the specification of the TCAP service-provider is rather an abstract one, its typical exchange of messages presented by TOs in section 4.3. is captured. To recall, this exchange prescribes a particular order, which however is not a simple FIFO order. Consequently, test cases derived with respect to *the local testing architecture* are not as easily adjusted as most people think such that test cases are derived with respect to *the distributed single layer testing architecture*.

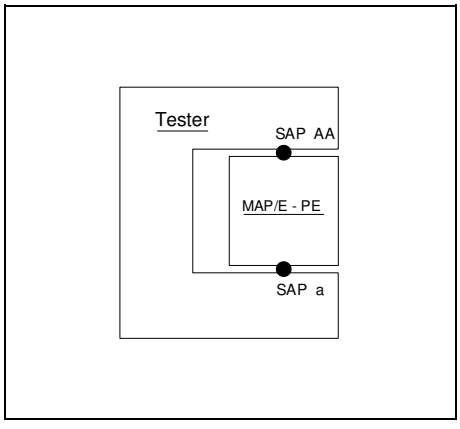

Figure 19: the local testing architecture.

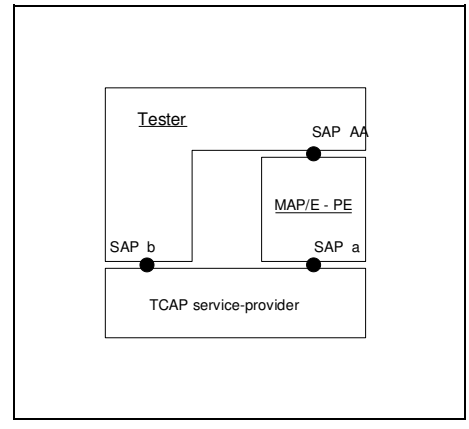

Figure 20: the distributed single layer testing architecture.

Below an example is given which presents two parts of two simulation trees which could be used to derive a test case. Both simulation trees are derived with SMILE. The one is derived from the LOTOS specification of a MAP/E protocol entity using the TCAP service-provider, and the other from the specification of a single MAP/E protocol entity. The purpose is to show the difference between the simulation trees resulting in different test cases.

For example, it could be desired to test the requirement that after the performance of SP *map_perform_handover_req* a MAP/E PE should respond by performing a *tc_invoke_req* SP. This could be visualized by:

X ? map_perform_handover_req(...) --> y ! tc_invoke_req(...)

Simulation trees which show this behaviour are presented below. However in the simulation tree on the right no *tc_invoke_req* SPs are present.

Single MAP/E protocol entity
X ! map_open_req(...) X ! map_perform_handover_req(...) X ! map_delimiter_req(...) y ! tc_invoke_req(...) y ! tc_begin_req(...) .... y ! tc_invoke_req(...) X ! map_delimiter_req(...) y ! tc_begin_req(...)

MAP/E protocol entity + TCAP service-provider
X ! map_open_req(...) X ! map_perform_handover_req(...) X ! map_delimiter_req(...) y ! tc_begin_ind(...) y ! tc_invoke_ind(...)

Both simulation trees show that SP *map_perform_handover_req* can only be performed after the performance of SP *map_open_req*. In a test case this could be seen as the preample.

In case of the left simulation tree the derivation of a test case is rather straight forward. The resulting test case is given below. However in the other case the SP *tc_invoke_req* can not be observed

directly, because an underlying service-provider is used. Therefore the tester must look at the indication SP related to *tc_invoke_req* (i.e. *tc_invoke_ind*). The simulation tree on the right shows that a *tc_invoke_ind* SP is performed not sooner then after the performance of SPs *map_delimiter_req* and *tc_begin_ind*. Below, both test cases are given.

Single MAP/E protocol entity	
AA ! map_open_req(...) AA ! map_perform_handover_req(...) a ? tc_invoke_req(...) Otherwise Otherwise	Pass Fail Fail

MAP/E protocol entity + TCAP service-provider	
AA ! map_open_req(...) AA ! map_perform_handover_req(...) AA ! map_delimiter_req(...) b ? tc_begin_ind(...) b ? tc_invoke_ind(...) Otherwise Otherwise Otherwise	Pass Fail Fail Fail

□

At last it must be said that the outstanding purpose of the current LOTOS specification is the provision of a basis for the development of more detailed and/or new MAP/x protocol specifications. Besides, the current specification is structured in a way that it is easy to extend and/or enhance. Accordingly, time could be saved which could be used to direct the attention to the derivation of (more) useful test cases.

Conclusion

The protocol and service descriptions presented in part II are already sufficient to specify a MAP/E protocol in LOTOS which is able to handle one dialogue in order to support the performance of a handover procedure. It can also be used to derive descriptions and LOTOS specifications of other MAP/x protocols. The TOs presented in this part are very useful to show the observable behaviour of these protocols. The specification of the MAP/E protocol presented in part III shows that such descriptions are easily captured in LOTOS. By holding strongly to the resource-oriented and constraint-oriented specification styles a framework is obtained which could be used to specify other MAP/x protocols. Besides it enables easy extension and enhancement of the current LOTOS specification of the MAP/E protocol.

With the given descriptions and LOTOS specifications, test cases for conformance testing MAP/E protocol implementations can already be derived. The specification of the underlying service offers the possibility to consider different test architectures.

The way MAP/x protocols handle multiple dialogues should be examined. This also applies for TCAP, especially for TCAP SPs. It will lead to a more complete description and LOTOS specification of the MAP/x protocols, and in particular of the MAP/E protocol. This will be necessary to obtain the possibility to test all requirements.

Literature

[BB] E. Brinksma, T. Bolognesi. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN systems 14 (1987)*, pages 25-59.

[CCITT] CCITT Recommendations Q.771 to Q.775.

[ETSI] ETSI. GSM 09.02, European Digital Cellular Telecommunication System (phase 2); Mobile Application Part Specification.

[JS] John Scourias. Overview of the Global System for Mobile Communications. WWW,http://ccnga.uwaterloo.ca/~jscouria/.

[MP] M. Mouly, M.-B. Pautet. An Introduction to GSM. Presented by Nortel Matra, published by authors, 1993.

[NC] N. Carrere. DSM Specification in LOTOS and Test Cases Generation. Technical report, 1995.

[OTS] University of Twente. Design of Telematic Systems, 1993 (lecture notes).

[RWO] S.M. Redl, M.K. Weber, M.W. Oliphant, An Introduction to GSM. Artech House Publishers, 1995.

Appendix I: Overview of specific MAP services

The following tables give an overview of the MAP service-providers, the services they provide, and the service elements of which these services are built from. The title of each table contains the service-provider name, and which pair of NSS-entities make use of this service-provider. Every other row is divided in three columns. The first column contains the name of service element, the second column gives the section in [ETSI] for more details, and the third column gives the name of the service to which the service element belongs.

MAP/A: MSC <--> BSC		
map_forward_check_ss_indication	6.10.2	Fault Recovery service

MAP/B: MSC <--> VLR		
map_update_location_area	6.1.1	Location Registration service
map_detach_imsi	6.1.5	
map_page	6.2.1	Paging Management service
map_search_for_ms	6.2.2	
map_process_access_request	6.3.1	Access Management service
map_allocate_handover_number	6.4.6	Handover service
map_send_handover_report	6.4.7	
map_authenticate	6.5.1	Authentication Management service
map_set_ciphering_mode	6.6.1	Security Management service
map_check_imei	6.7.1	IMEI Management service
map_obtain_imei	6.7.2	
map_provide_imsi	6.9.1	Identity Management service
map_forward_new_imsi	6.9.2	
map_forward_check_ss_indication	6.10.2	Fault Recovery service
map_trace_subscriber_activity	7.1.3	Subscriber Tracing service
map_send_info_for_incoming_call	8.1	Call Handling service
map_send_info_for_outgoing_call	8.2	
map_complete_call	8.3	
map_process_call_waiting	8.4	
map_register_ss	9.1	Supplementary Services Related service
map_erase_ss	9.2	
map_activate_ss	9.3	
map_deactivate_ss	9.4	
map_interrogate_ss	9.5	
map_invoke_ss	9.6	
map_register_password	9.7	
map_get_password	9.8	

map_process_ss_request	9.9	
map_ss_get_digits	9.10	
map_ss_get_string	9.11	
map_unstructured_notify_ss	9.12	
map_ready_for_sm	10.4	Short Message Service Management service
map_send_info_for_mt_sms	10.7	
map_send_info_for_mo_sms	10.8	

MAP/C: GMSC <--> HLR		
map_send_routing_information	8.3	Call Handling service
map_send_routing_info_for_sm	10.1	Short Message Service Management service
map_report_sm_delivery_status	10.3	
map_alert_service_centre	10.5	
map_inform_service_centre	10.6	

MAP/D: HLR <--> VLR		
map_update_location	6.1.2	Location Registration service
map_cancel_location	6.1.3	
map_send_authentication	6.5.2	Authentication Management service
map_insert_subscriber_data	6.8.1	Subscriber Management service
map_delete_subscriber_data	6.8.2	
map_reset	6.10.1	Fault Recovery service
map_forward_check_ss_indication	6.10.2	
map_restore_data	6.10.3	
map_activate_trace_mode	7.1.1	Subscriber Tracing service
map_deactivate_trace_mode	7.1.2	
map_provider_roaming_number	8.4	Call Handling service
map_register_ss	9.1	Supplementary Services Related service
map_erase_ss	9.2	
map_activate_ss	9.3	
map_deactivate_ss	9.4	
map_interrogate_ss	9.5	
map_register_password	9.7	
map_get_password	9.8	
map_process_ss_requets	9.9	
map_ss_get_digits	9.10	
map_ss_get_string	9.11	
map_unstructured_notify_ss	9.12	
map_ready_for_sm	10.4	Short Message Service Management service

MAP/E: MSC-A <--> MSC-B		
map_perform_handover	6.4.1	Handover service
map_send_end_signal	6.4.3	
map_process_access_signalling	6.4.3	
map_forward_access_signalling	6.4.4	
map_perform_subsequent_handover	6.4.5	
map_trace_subscriber_activity	7.1.3	Subscriber Tracing service
map_note_internal_handover	7.2.1	Other Operation & Maintenance service
map_forward_short_message	10.2	Short Message Service Management service

MAP/F: MSC <--> EIR		
map_check_imei	6.7.1	IMEI Management service

MAP/G: VLR-A <--> VLR-B		
map_send_identification	6.1.4	Location Management service

Appendix II: LOTOS spec of the MAP/E service-provider (ADT part)

```
(*** mape_serv.final.lot ***)

(*** This is the LOTOS specification of the MAP/E service-provider. ***)
(*** It consists of two MAP/E protocol entities and an underlying   ***)
(*** service (TCAP).                                                ***)

specification MapService[AA, BB] : noexit

library
      Boolean, FBoolean, NaturalNumber, Element
endlib

(*----------- Parameter Types. Can also be found in Nancys part. --------------------*)

type EnrichedElement is Element
     formalopns _lt_    : Element, Element -> FBool
     formaleqns forall e1, e2: Element
         ofsort FBool
          not(e1 ne e2) => e1 lt e2 = not(true);
          e1 lt e2      => e2 lt e1 = not(true);
endtype (* EnrichedElement *)

type Queue is (* SetElement *) EnrichedElement
     sorts Queue
     opns  empty    :                -> Queue
           add      : Element, Queue -> Queue
           dda      : Element, Queue -> Queue
           concat   : Queue, Queue   -> Queue
           head     : Queue          -> Element
           tail     : Queue          -> Queue
           last     : Queue          -> Element
           init     : Queue          -> Queue
           IsEmpty  : Queue          -> FBool
           NotEmpty : Queue          -> FBool
     eqns  forall x,y,z: Element, q, q1, q2: Queue
     ofsort Element
           last(add(x, empty))     = x;
           last(add(x, add(y, q))) =last(add(y, q));
     ofsort Element
           head(add(x, empty))     = x;
           head(add(x, add(y, q))) = x;
     ofsort Queue
           dda(y, empty)           = add(y, empty);
           dda(y, add(x, empty))   = add(x, add(y, empty));
           dda(y, add(x, q))       = add(x, dda(y, q));
     ofsort Queue
           tail(empty)             = empty;
           tail(add(x, empty))     = empty;
           tail(add(x, add(y, q))) = add(y, q);
     ofsort Queue
           init(empty)             = empty;
           init(add(x, empty))     = empty;
           init(add(x, add(y, q))) = add(x, init(add(y, q)));
     ofsort Queue
           concat(empty, empty)     = empty;
           concat(empty, q)         = q;
           concat(q, empty)         = q;  (* not necesary *)
           concat(add(y, empty), q) = add(y, q);
           concat(add(y, q1), q2)   = add(y, concat(q1, q2));
     ofsort FBool
           IsEmpty(empty)     = true;
           IsEmpty(add(x, q)) = not(true);
           NotEmpty(q)        = not(IsEmpty(q));
endtype

type BSS_APDU is NaturalNumber, Boolean
      sorts BSS_APDU
      opns bp_1             :                     -> BSS_APDU
           _eq_, _ne_, _lt_ : BSS_APDU, BSS_APDU -> Bool
           NatNum           : BSS_APDU           -> Nat
     eqns forall x, y : BSS_APDU
     ofsort Nat
           NatNum(bp_1) = 0;
     ofsort Bool
            x eq y = NatNum(x) eq NatNum(y);
            x ne y = NatNum(x) ne NatNum(y);
            x lt y = NatNum(x) lt NatNum(y);
endtype

type OtherParas is Boolean, NaturalNumber
```

```
      sorts OtherParas
      opns op_1             :                          -> OtherParas
           _eq_, _ne_, _lt_ : OtherParas, OtherParas -> Bool
           NatNum           : OtherParas              -> Nat
     eqns forall x, y : OtherParas
     ofsort Nat
           NatNum(op_1) = 0;
     ofsort Bool
            x eq y = NatNum(x) eq NatNum(y);
            x ne y = NatNum(x) ne NatNum(y);
            x lt y = NatNum(x) lt NatNum(y);
endtype

(*------*)

type Argument is NaturalNumber, Boolean
      sorts Argument
      opns arg_1            :                      -> Argument
           _eq_, _ne_, _lt_ : Argument, Argument -> Bool
           NatNum           : Argument           -> Nat
     eqns forall x, y : Argument
     ofsort Nat
           NatNum(arg_1) = 0;
     ofsort Bool
            x eq y = NatNum(x) eq NatNum(y);
            x ne y = NatNum(x) ne NatNum(y);
            x lt y = NatNum(x) lt NatNum(y);
endtype

type UserInfo is NaturalNumber, Boolean
      sorts UserInfo
      opns ui_1             :                      -> UserInfo
           _eq_, _ne_, _lt_ : UserInfo, UserInfo -> Bool
           NatNum           : UserInfo           -> Nat
     eqns forall x, y : UserInfo
     ofsort Nat
           NatNum(ui_1) = 0;
     ofsort Bool
            x eq y = NatNum(x) eq NatNum(y);
            x ne y = NatNum(x) ne NatNum(y);
            x lt y = NatNum(x) lt NatNum(y);
endtype

(*-------*)

type AC is Boolean, NaturalNumber
     sorts AC
     opns
            Access_Location_Updating,
            Location_Updating,
            Location_Cancellation,
            Roaming_Number_Inquiry,
            Incoming_Call_Handling,
            Location_Information_Retrieval,
            Outgoing_Call_Setup,
            Access_Functionnal_SS_Handling,
            Access_SS_Invocation,
            Access_Unstructured_SS_Handling,
            Location_Register_Restart,
             Handover_Control,
            Handover_Number_Allocation,
            Information_Retrieval,
            Inter_VLR_Information_Retrieval,
            Stand_Alone_Subscriber_Data_Management,
            Tracing,
            Network_Functionnal_SS_Handling,
            Network_Unstructured_SS_Handling        :        ->AC
            V1                                      : AC     -> Bool
            NatNum                                  : AC     -> Nat
           _eq_, _ne_, _lt_                         : AC, AC -> Bool
     eqns forall x, y : AC
     ofsort Bool
            V1(Access_Location_Updating)         = true;
            V1(Location_Updating)                = true;
            V1(Location_Cancellation)            = true;
            V1(Roaming_Number_Inquiry)           = true;
            V1(Incoming_Call_Handling)           = true;
            V1(Location_Information_Retrieval)   = true;
            V1(Outgoing_Call_Setup)              = true;
            V1(Access_Functionnal_SS_Handling)   = true;
            V1(Access_SS_Invocation)             = false;
            V1(Access_Unstructured_SS_Handling)  = false;
            V1(Location_Register_Restart)        = true;
            V1(Handover_Control)                 = true;
            V1(Handover_Number_Allocation)       = true;
```

```
          V1(Information_Retrieval)            = true;
          V1(Inter_VLR_Information_Retrieval)  = true;
          V1(Stand_Alone_Subscriber_Data_Management) = true;
          V1(Tracing)                          = true;
          V1(Network_Functionnal_SS_Handling)  = false;
          V1(Network_Unstructured_SS_Handling) = false;
    ofsort Bool
          x eq y = NatNum(x) eq NatNum(y);
          x ne y = NatNum(x) ne NatNum(y);
          x lt y = NatNum(x) lt NatNum(y);
    ofsort Nat
       NatNum(Access_Location_Updating)        = 0;
       NatNum(Location_Updating)               = Succ(NatNum(Access_Location_Updating));
       NatNum(Location_Cancellation)           = Succ(NatNum(Location_Updating));
       NatNum(Roaming_Number_Inquiry)          = Succ(NatNum(Location_Cancellation));
       NatNum(Incoming_Call_Handling)          = Succ(NatNum(Roaming_Number_Inquiry));
       NatNum(Location_Information_Retrieval)  = Succ(NatNum(Incoming_Call_Handling));
       NatNum(Outgoing_Call_Setup)             = Succ(NatNum(Location_Information_Retrieval));
       NatNum(Access_Functionnal_SS_Handling)  = Succ(NatNum(Outgoing_Call_Setup));
       NatNum(Access_SS_Invocation)            = Succ(NatNum(Access_Functionnal_SS_Handling));
       NatNum(Access_Unstructured_SS_Handling) = Succ(NatNum(Access_SS_Invocation));
       NatNum(Location_Register_Restart)       = Succ(NatNum(Access_Unstructured_SS_Handling));
       NatNum(Handover_Control)                = Succ(NatNum(Location_Register_Restart));
       NatNum(Handover_Number_Allocation)      = Succ(NatNum(Handover_Control));
       NatNum(Information_Retrieval)           = Succ(NatNum(Handover_Number_Allocation));
       NatNum(Inter_VLR_Information_Retrieval) = Succ(NatNum(Information_Retrieval));
  NatNum(Stand_Alone_Subscriber_Data_Management) = Succ(NatNum(Inter_VLR_Information_Retrieval));
       NatNum(Tracing)                  = Succ(NatNum(Stand_Alone_Subscriber_Data_Management));
       NatNum(Network_Functionnal_SS_Handling)  = Succ(NatNum(Tracing));
       NatNum(Network_Unstructured_SS_Handling) = Succ(NatNum(Network_Functionnal_SS_Handling));
endtype

type OpCode is NaturalNumber, Boolean    (* OpCode <-> InvokeId *)
       sorts OpCode
       opns
       UpdateLocationArea,         UpdateLocation,            CancelLocation,
       SendIdentification,         DetachIMSI,                AttachIMSI,
       Page,                       SearchForMS,               ProcessAccessRequest,
        PerformHandover,            SendEndSignal,            ProcessAccessSignalling,
       ForwardAccessSignalling,    PerformSubsequentHandover, AllocateHandoverNumber,
       SendHandoverReport,         Authenticate,              SendAuthentificationInfo,
       SetCipheringMode,           SendParameters,            InsertSubscriberData,
       DeleteSubscriberData,       ProvideIMSI,               ForwardNewTMSI,
       Reset,                      ForwardCheckSS_Indication, RestoreData,
       ActivateTraceMode,          DeactivateTraceMode,       TraceSubscriberActivity,
       NoteInternalHandover,       SendIMSI,                  SendInfoForIncomingCall,
       ConnectToFollowingAddress,  ProcessCallWaiting,        SendInfoForOutgoingCall,
       SendRoutingInfo,            ProvideRoamingNumber,      CompleteCall,
       RegisterSS,                 EraseSS,                   ActivateSS,
       DeactivateSS,               InterrogateSS,             InvokeSS,
       ProcessUnstructuredSS_Data, ProcessSS_Request,         SS_GetDigits,
       SS_GetString,               UnstructuredNotifySS,      RegisterPassword,
       GetPassword,                BeginSubscriberActivity    :      -> OpCode

       NatNum           : OpCode          -> Nat
       _eq_, _ne_, _lt_ : OpCode, OpCode  -> Bool

    eqns forall oc1, oc2:OpCode
    ofsort Nat
       NatNum( UpdateLocationArea )         = 0;
       NatNum( UpdateLocation )             = Succ(NatNum(UpdateLocationArea));
       NatNum( CancelLocation )             = Succ(NatNum(UpdateLocation));
       NatNum( SendIdentification )         = Succ(NatNum(CancelLocation));
       NatNum( DetachIMSI )                 = Succ(NatNum(SendIdentification));
       NatNum( AttachIMSI )                 = Succ(NatNum(DetachIMSI));
       NatNum( Page )                       = Succ(NatNum(AttachIMSI));
       NatNum( SearchForMS )                = Succ(NatNum(Page));
       NatNum( ProcessAccessRequest )       = Succ(NatNum(SearchForMS));
         NatNum( PerformHandover )            = Succ(NatNum(ProcessAccessRequest));
         NatNum( SendEndSignal )              = Succ(NatNum(PerformHandover));
       NatNum( ProcessAccessSignalling )    = Succ(NatNum(SendEndSignal));
       NatNum( ForwardAccessSignalling )    = Succ(NatNum(ProcessAccessSignalling));
       NatNum( PerformSubsequentHandover )  = Succ(NatNum(ForwardAccessSignalling));
       NatNum( AllocateHandoverNumber )     = Succ(NatNum(PerformSubsequentHandover));
       NatNum( SendHandoverReport )         = Succ(NatNum(AllocateHandoverNumber));
       NatNum( Authenticate )               = Succ(NatNum(SendHandoverReport));
       NatNum( SendAuthentificationInfo )   = Succ(NatNum(Authenticate));
       NatNum( SetCipheringMode )           = Succ(NatNum(SendAuthentificationInfo));
       NatNum( SendParameters )             = Succ(NatNum(SetCipheringMode));
       NatNum( InsertSubscriberData )       = Succ(NatNum(SendParameters));
       NatNum( DeleteSubscriberData )       = Succ(NatNum(InsertSubscriberData));
       NatNum( ProvideIMSI )                = Succ(NatNum(DeleteSubscriberData));
       NatNum( ForwardNewTMSI )             = Succ(NatNum(ProvideIMSI));
       NatNum( Reset )                      = Succ(NatNum(ForwardNewTMSI));
       NatNum( ForwardCheckSS_Indication )  = Succ(NatNum(Reset));
```

```
        NatNum( RestoreData )                = Succ(NatNum(ForwardCheckSS_Indication));
        NatNum( ActivateTraceMode )          = Succ(NatNum(RestoreData));
        NatNum( DeactivateTraceMode )        = Succ(NatNum(ActivateTraceMode));
        NatNum( TraceSubscriberActivity )    = Succ(NatNum(DeactivateTraceMode));
        NatNum( NoteInternalHandover )       = Succ(NatNum(TraceSubscriberActivity));
        NatNum( SendIMSI )                   = Succ(NatNum(NoteInternalHandover));
        NatNum( SendInfoForIncomingCall )    = Succ(NatNum(SendIMSI));
        NatNum( ConnectToFollowingAddress )  = Succ(NatNum(SendInfoForIncomingCall));
        NatNum( ProcessCallWaiting )         = Succ(NatNum(ConnectToFollowingAddress));
        NatNum( SendInfoForOutgoingCall )    = Succ(NatNum(ProcessCallWaiting));
        NatNum( SendRoutingInfo )            = Succ(NatNum(SendInfoForOutgoingCall));
        NatNum( ProvideRoamingNumber )       = Succ(NatNum(SendRoutingInfo));
        NatNum( CompleteCall )               = Succ(NatNum(ProvideRoamingNumber));
        NatNum( RegisterSS )                 = Succ(NatNum(CompleteCall));
        NatNum( EraseSS )                    = Succ(NatNum(RegisterSS));
        NatNum( ActivateSS )                 = Succ(NatNum(EraseSS));
        NatNum( DeactivateSS )               = Succ(NatNum(ActivateSS));
        NatNum( InterrogateSS )              = Succ(NatNum(DeactivateSS));
        NatNum( InvokeSS )                   = Succ(NatNum(InterrogateSS));
        NatNum( ProcessUnstructuredSS_Data ) = Succ(NatNum(InvokeSS));
        NatNum( ProcessSS_Request )          = Succ(NatNum(ProcessUnstructuredSS_Data));
        NatNum( SS_GetDigits )               = Succ(NatNum(ProcessSS_Request));
        NatNum( SS_GetString )               = Succ(NatNum(SS_GetDigits));
        NatNum( UnstructuredNotifySS )       = Succ(NatNum(SS_GetString));
        NatNum( RegisterPassword )           = Succ(NatNum(UnstructuredNotifySS));
        NatNum( GetPassword )                = Succ(NatNum(RegisterPassword));
        NatNum( BeginSubscriberActivity )    = Succ(NatNum(GetPassword));
     ofsort Bool
        oc1 eq oc2 = NatNum(oc1) eq NatNum(oc2);
        oc1 ne oc2 = NatNum(oc1) ne NatNum(oc2);
        oc1 lt oc2 = NatNum(oc1) lt NatNum(oc2);
endtype

(*-------------------------------------------------------------------------------*)

type SpecParameterTypes is OpCode, BSS_APDU, OtherParas, Boolean
endtype

type ComParameterTypes is AC, OtherParas, Boolean
endtype

type TcapParameterTypes is AC, UserInfo, OpCode, Argument, Boolean
endtype

(*------------------------ MAP Common Service Primitives ----------------------------*)

type MapComServPrim is Boolean, NaturalNumber, ComParameterTypes
     sorts MAPCS
     opns
          map_open_req      : AC, OtherParas -> MAPCS
          map_open_ind      : AC, OtherParas -> MAPCS
          map_open_rsp      : AC, OtherParas -> MAPCS
          map_open_cnf      : AC, OtherParas -> MAPCS
          map_close_req     :                -> MAPCS
          map_close_ind     :                -> MAPCS
          map_delimiter_req :                -> MAPCS
          map_delimiter_ind :                -> MAPCS
          (* necessary operations to make SPs usable in queues and sets *)
          _eq_, _ne_, _lt_  : MAPCS, MAPCS   -> Bool
          NatNum            : MAPCS          -> Nat
     eqns forall sp1, sp2:MAPCS
     ofsort Bool
           sp1 eq sp2 = NatNum(sp1) eq NatNum(sp2);
           sp1 ne sp2 = NatNum(sp1) ne NatNum(sp2);
           sp1 lt sp2 = NatNum(sp1) lt NatNum(sp2);
endtype

type MapComServPrimDiscr is MapComServPrim
     opns
          IsMapOpenReq      : MAPCS -> Bool
          IsMapOpenInd      : MAPCS -> Bool
          IsMapOpenRsp      : MAPCS -> Bool
          IsMapOpenCnf      : MAPCS -> Bool
          IsMapCloseReq     : MAPCS -> Bool
          IsMapCloseInd     : MAPCS -> Bool
          IsMapDelimiterReq : MAPCS -> Bool
          IsMapDelimiterInd : MAPCS -> Bool
     eqns forall a:AC, o:OtherParas
          ofsort Bool
          IsMapOpenReq(map_open_req(a, o)) = true;
          IsMapOpenReq(map_open_ind(a, o)) = false;
          IsMapOpenReq(map_open_rsp(a, o)) = false;
          IsMapOpenReq(map_open_cnf(a, o)) = false;
          IsMapOpenReq(map_close_req) = false;
          IsMapOpenReq(map_close_ind) = false;
```

```
          IsMapOpenReq(map_delimiter_req) = false;
          IsMapOpenReq(map_delimiter_ind) = false;

          IsMapOpenInd(map_open_req(a, o)) =  false;
          IsMapOpenInd(map_open_ind(a, o)) = true;
          IsMapOpenInd(map_open_rsp(a, o)) =  false;
          IsMapOpenInd(map_open_cnf(a, o)) =  false;
          IsMapOpenInd(map_close_req) =  false;
          IsMapOpenInd(map_close_ind) = false;
          IsMapOpenInd(map_delimiter_req) = false;
          IsMapOpenInd(map_delimiter_ind) = false;

          IsMapOpenRsp(map_open_req(a, o)) =  false;
          IsMapOpenRsp(map_open_ind(a, o)) = false;
          IsMapOpenRsp(map_open_rsp(a, o)) = true;
          IsMapOpenRsp(map_open_cnf(a, o)) =  false;
          IsMapOpenRsp(map_close_req) =  false;
          IsMapOpenRsp(map_close_ind) =  false;
          IsMapOpenRsp(map_delimiter_req) =  false;
          IsMapOpenRsp(map_delimiter_ind) = false;

          IsMapOpenCnf(map_open_req(a, o)) =  false;
          IsMapOpenCnf(map_open_ind(a, o)) = false;
          IsMapOpenCnf(map_open_rsp(a, o)) =  false;
          IsMapOpenCnf(map_open_cnf(a, o)) = true;
          IsMapOpenCnf(map_close_req) =  false;
          IsMapOpenCnf(map_close_ind) =  false;
          IsMapOpenCnf(map_delimiter_req) =  false;
          IsMapOpenCnf(map_delimiter_ind) = false;

          IsMapCloseReq(map_open_req(a, o)) =  false;
          IsMapCloseReq(map_open_ind(a, o)) = false;
          IsMapCloseReq(map_open_rsp(a, o)) =  false;
          IsMapCloseReq(map_open_cnf(a, o)) =  false;
          IsMapCloseReq(map_close_req) = true;
          IsMapCloseReq(map_close_ind) =  false;
          IsMapCloseReq(map_delimiter_req) = false;
          IsMapCloseReq(map_delimiter_ind) = false;

          IsMapCloseInd(map_open_req(a, o)) =  false;
          IsMapCloseInd(map_open_ind(a, o)) = false;
          IsMapCloseInd(map_open_rsp(a, o)) =  false;
          IsMapCloseInd(map_open_cnf(a, o)) =  false;
          IsMapCloseInd(map_close_req) =  false;
          IsMapCloseInd(map_close_ind) = true;
          IsMapCloseInd(map_delimiter_req) =  false;
          IsMapCloseInd(map_delimiter_ind) = false;

          IsMapDelimiterReq(map_open_req(a, o)) =  false;
          IsMapDelimiterReq(map_open_ind(a, o)) = false;
          IsMapDelimiterReq(map_open_rsp(a, o)) =  false;
          IsMapDelimiterReq(map_open_cnf(a, o)) =  false;
          IsMapDelimiterReq(map_close_req) =  false;
          IsMapDelimiterReq(map_close_ind) =  false;
          IsMapDelimiterReq(map_delimiter_req) =  true;
          IsMapDelimiterReq(map_delimiter_ind) = false;

          IsMapDelimiterInd(map_open_req(a, o)) = false;
          IsMapDelimiterInd(map_open_ind(a, o)) = false;
          IsMapDelimiterInd(map_open_rsp(a, o)) = false;
          IsMapDelimiterInd(map_open_cnf(a, o)) = false;
          IsMapDelimiterInd(map_close_req) = false;
          IsMapDelimiterInd(map_close_ind) = false;
          IsMapDelimiterInd(map_delimiter_req) = false;
          IsMapDelimiterInd(map_delimiter_ind) = true;

endtype

type MapComServPrimPars is MapComServPrimDiscr
     opns
          ACOf       : MAPCS -> AC
          OthParaOf  : MAPCS -> OtherParas
(*        DestAddrOf : MAPCS -> DestAddr  *)
(*        DestRefOf  : MAPCS -> DestRef   *)
(*        OrAddrOf   : MAPCS -> OrAddr    *)
(*        OrRefOf    : MAPCS -> OrRef     *)
(*        SpecInfOf  : MAPCS -> SpecInfo  *)
(*        RespAddrOf : MAPCS -> RespAddr  *)
(*        ResultOf   : MAPCS -> Result    *)
(*        RefuseOf   : MAPCS -> Refuse    *)
(*        RelMethOf : MAPCS -> RelMeth    *)
     eqns forall a:AC, o:OtherParas
     ofsort AC
          ACOf(map_open_req(a, o))       = a; (* mandatory *)
          ACOf(map_open_ind(a, o))       = a; (* mandatory *)
```

```
          ACOf(map_open_rsp(a, o))       = a; (* optional: not used yet *)
          ACOf(map_open_cnf(a, o))       = a; (* optional: not used yet *)
     ofsort OtherParas
          OthParaOf(map_open_req(a, o)) = o;
          OthParaOf(map_open_ind(a, o)) = o;
          OthParaOf(map_open_rsp(a, o)) = o;
          OthParaOf(map_open_cnf(a, o)) = o;

(*      ofsort DestAddr *)
(*           DestAddrOf(map_open_req(a, b, c, d, e, f)) = b; *) (* mandatory *)
(*           DestAddrOf(map_open_ind(a, b, c, d, e, f)) = b; *) (* mandatory *)
(*      ofsort DestRef *)
(*           DestRefOf(map_open_req(a, b, c, d, e, f))  = c; *) (* optional: not used yet *)
(*           DestRefOf(map_open_ind(a, b, c, d, e, f))  = c; *) (* optional: not used yet *)
(*      ofsort OrAddr *)
(*           OrAddrOf(map_open_req(a, b, c, d, e, f))   = d; *) (* optional: not used yet *)
(*           OrAddrOf(map_open_ind(a, b, c, d, e, f))   = d; *) (* optional: not used yet *)
(*      ofsort OrRef *)
(*           OrRefOf(map_open_req(a, b, c, d, e, f))    = e; *) (* optional: not used yet *)
(*           OrRefOf(map_open_ind(a, b, c, d, e, f))    = e; *) (* optional: not used yet *)
(*      ofsort SpecInfo *)
(*           SpecInfOf(map_open_req(a, b, c, d, e, f))  = f; *) (* optional: not used yet *)
(*           SpecInfOf(map_open_ind(a, b, c, d, e, f))  = f; *) (* optional: not used yet *)
(*           SpecInfOf(map_open_rsp(a, f, g, h, r))    = f; *) (* optional: not used yet *)
(*           SpecInfOf(map_open_cnf(a, f, g, h, r))    = f; *) (* optional: not used yet *)
(*           SpecInfOf(map_close_req(j, f))             = f; *) (* optional: not used yet *)
(*           SpecInfOf(map_close_ind(f))                = f; *) (* optional: not used yet *)
(*      ofsort RespAddr *)
(*           RespAddrOf(map_open_rsp(a, f, g, h, r))   = g; *) (* optional: not used yet *)
(*           RespAddrOf(map_open_cnf(a, f, g, h, r))   = g; *) (* optional: not used yet *)
(*      ofsort Result *)
(*           ResultOf(map_open_rsp(a, f, g, h, r))     = h; *) (* mandatory *)
(*           ResultOf(map_open_cnf(a, f, g, h, r))     = h; *) (* mandatory *)
(*      ofsort Refuse *)
(*           RefuseOf(map_open_rsp(a, f, g, h, r))     = r; *) (* optional: not used yet *)
(*           RefuseOf(map_open_cnf(a, f, g, h, r))     = r; *) (* optional: not used yet *)
(*      ofsort RelMeth *)
(*           RelMethOf(map_close_req(j, f))             = j; *) (* mandatory *)
endtype


(*------------------------ MAP Specific Service Primitives --------------------------*)

type MapESpecServPrim is Boolean, NaturalNumber, SpecParameterTypes
     sorts MAPSS
     opns (* handover services *)
          map_perform_handover_req            : OpCode, OtherParas -> MAPSS
          map_perform_handover_ind            : OpCode, OtherParas -> MAPSS
          map_perform_handover_rsp            : OpCode, OtherParas -> MAPSS
          map_perform_handover_cnf            : OpCode, OtherParas -> MAPSS
          map_send_end_signal_req             : OpCode             -> MAPSS
          map_send_end_signal_ind             : OpCode             -> MAPSS
          map_send_end_signal_rsp             : OpCode             -> MAPSS
          map_send_end_signal_cnf             : OpCode, OtherParas -> MAPSS
          map_process_access_signalling_req   : OpCode, BSS_APDU   -> MAPSS
          map_process_access_signalling_ind   : OpCode, BSS_APDU   -> MAPSS
          map_forward_access_signalling_req   : OpCode, BSS_APDU   -> MAPSS
          map_forward_access_signalling_ind   : OpCode, BSS_APDU   -> MAPSS
          map_perform_subsequent_handover_req : OpCode, OtherParas -> MAPSS
          map_perform_subsequent_handover_ind : OpCode, OtherParas -> MAPSS
          map_perform_subsequent_handover_rsp : OpCode, OtherParas -> MAPSS
          map_perform_subsequent_handover_cnf : OpCode, OtherParas -> MAPSS
          (* subscriber tracing services *)
          (* other operation & maintenance services *)
          (* short message service management services *)
           _eq_, _ne_, _lt_  : MAPSS, MAPSS                                  -> Bool
           NatNum            : MAPSS                                         -> Nat
     eqns forall sp1, sp2:MAPSS
     ofsort Bool
            sp1 eq sp2 = NatNum(sp1) eq NatNum(sp2);
            sp1 ne sp2 = NatNum(sp1) ne NatNum(sp2);
            sp1 lt sp2 = NatNum(sp1) lt NatNum(sp2);
endtype

type MapESpecServPrimDiscr is MapESpecServPrim
     opns (* handover services *)
          IsMapPerformHandoverReq     : MAPSS -> Bool
          IsMapPerformHandoverInd     : MAPSS -> Bool
          IsMapPerformHandoverRsp     : MAPSS -> Bool
          IsMapPerformHandoverCnf     : MAPSS -> Bool
          IsMapSendEndSignalReq       : MAPSS -> Bool
          IsMapSendEndSignalInd       : MAPSS -> Bool
          IsMapSendEndSignalRsp       : MAPSS -> Bool
          IsMapSendEndSignalCnf       : MAPSS -> Bool
          IsMapProcessAccessSignReq   : MAPSS -> Bool
```

```
      IsMapProcessAccessSignInd   : MAPSS -> Bool
      IsMapForwardAccessSignReq   : MAPSS -> Bool
      IsMapForwardAccessSignInd   : MAPSS -> Bool
      IsMapPerformSubsequentHOReq : MAPSS -> Bool
      IsMapPerformSubsequentHOInd : MAPSS -> Bool
      IsMapPerformSubsequentHORsp : MAPSS -> Bool
      IsMapPerformSubsequentHOCnf : MAPSS -> Bool
      (* subscriber tracing services *)
      (* other operation & maintenance services *)
      (* short message service management services *)
eqns forall a:OpCode, b:BSS_APDU, c:OtherParas
      ofsort Bool

      IsMapPerformHandoverReq(map_perform_handover_req(a, c))            = true;
      IsMapPerformHandoverReq(map_perform_handover_ind(a, c))            = false;
      IsMapPerformHandoverReq(map_perform_handover_rsp(a, c))            = false;
      IsMapPerformHandoverReq(map_perform_handover_cnf(a, c))            = false;
      IsMapPerformHandoverReq(map_send_end_signal_req(a))                = false;
      IsMapPerformHandoverReq(map_send_end_signal_ind(a))                = false;
      IsMapPerformHandoverReq(map_send_end_signal_rsp(a))                = false;
      IsMapPerformHandoverReq(map_send_end_signal_cnf(a, c))             = false;
      IsMapPerformHandoverReq(map_process_access_signalling_req(a, b))   = false;
      IsMapPerformHandoverReq(map_process_access_signalling_ind(a, b))   = false;
      IsMapPerformHandoverReq(map_forward_access_signalling_req(a, b))   = false;
      IsMapPerformHandoverReq(map_forward_access_signalling_ind(a, b))   = false;
      IsMapPerformHandoverReq(map_perform_subsequent_handover_req(a, c)) = false;
      IsMapPerformHandoverReq(map_perform_subsequent_handover_ind(a, c)) = false;
      IsMapPerformHandoverReq(map_perform_subsequent_handover_rsp(a, c)) = false;
      IsMapPerformHandoverReq(map_perform_subsequent_handover_cnf(a, c)) = false;

      IsMapPerformHandoverInd(map_perform_handover_req(a, c))            = false;
      IsMapPerformHandoverInd(map_perform_handover_ind(a, c))            = true;
      IsMapPerformHandoverInd(map_perform_handover_rsp(a, c))            = false;
      IsMapPerformHandoverInd(map_perform_handover_cnf(a, c))            = false;
      IsMapPerformHandoverInd(map_send_end_signal_req(a))                = false;
      IsMapPerformHandoverInd(map_send_end_signal_ind(a))                = false;
      IsMapPerformHandoverInd(map_send_end_signal_rsp(a))                = false;
      IsMapPerformHandoverInd(map_send_end_signal_cnf(a, c))             = false;
      IsMapPerformHandoverInd(map_process_access_signalling_req(a, b))   = false;
      IsMapPerformHandoverInd(map_process_access_signalling_ind(a, b))   = false;
      IsMapPerformHandoverInd(map_forward_access_signalling_req(a, b))   = false;
      IsMapPerformHandoverInd(map_forward_access_signalling_ind(a, b))   = false;
      IsMapPerformHandoverInd(map_perform_subsequent_handover_req(a, c)) = false;
      IsMapPerformHandoverInd(map_perform_subsequent_handover_ind(a, c)) = false;
      IsMapPerformHandoverInd(map_perform_subsequent_handover_rsp(a, c)) = false;
      IsMapPerformHandoverInd(map_perform_subsequent_handover_cnf(a, c)) = false;

      IsMapPerformHandoverRsp(map_perform_handover_req(a, c))            = false;
      IsMapPerformHandoverRsp(map_perform_handover_ind(a, c))            = false;
      IsMapPerformHandoverRsp(map_perform_handover_rsp(a, c))            = true;
      IsMapPerformHandoverRsp(map_perform_handover_cnf(a, c))            = false;
      IsMapPerformHandoverRsp(map_send_end_signal_req(a))                = false;
      IsMapPerformHandoverRsp(map_send_end_signal_ind(a))                = false;
      IsMapPerformHandoverRsp(map_send_end_signal_rsp(a))                = false;
      IsMapPerformHandoverRsp(map_send_end_signal_cnf(a, c))             = false;
      IsMapPerformHandoverRsp(map_process_access_signalling_req(a, b))   = false;
      IsMapPerformHandoverRsp(map_process_access_signalling_ind(a, b))   = false;
      IsMapPerformHandoverRsp(map_forward_access_signalling_req(a, b))   = false;
      IsMapPerformHandoverRsp(map_forward_access_signalling_ind(a, b))   = false;
      IsMapPerformHandoverRsp(map_perform_subsequent_handover_req(a, c)) = false;
      IsMapPerformHandoverRsp(map_perform_subsequent_handover_ind(a, c)) = false;
      IsMapPerformHandoverRsp(map_perform_subsequent_handover_rsp(a, c)) = false;
      IsMapPerformHandoverRsp(map_perform_subsequent_handover_cnf(a, c)) = false;

      IsMapPerformHandoverCnf(map_perform_handover_req(a, c))            = false;
      IsMapPerformHandoverCnf(map_perform_handover_ind(a, c))            = false;
      IsMapPerformHandoverCnf(map_perform_handover_rsp(a, c))            = false;
      IsMapPerformHandoverCnf(map_perform_handover_cnf(a, c))            = true;
      IsMapPerformHandoverCnf(map_send_end_signal_req(a))                = false;
      IsMapPerformHandoverCnf(map_send_end_signal_ind(a))                = false;
      IsMapPerformHandoverCnf(map_send_end_signal_rsp(a))                = false;
      IsMapPerformHandoverCnf(map_send_end_signal_cnf(a, c))             = false;
      IsMapPerformHandoverCnf(map_process_access_signalling_req(a, b))   = false;
      IsMapPerformHandoverCnf(map_process_access_signalling_ind(a, b))   = false;
      IsMapPerformHandoverCnf(map_forward_access_signalling_req(a, b))   = false;
      IsMapPerformHandoverCnf(map_forward_access_signalling_ind(a, b))   = false;
      IsMapPerformHandoverCnf(map_perform_subsequent_handover_req(a, c)) = false;
      IsMapPerformHandoverCnf(map_perform_subsequent_handover_ind(a, c)) = false;
      IsMapPerformHandoverCnf(map_perform_subsequent_handover_rsp(a, c)) = false;
      IsMapPerformHandoverCnf(map_perform_subsequent_handover_cnf(a, c)) = false;

      IsMapSendEndSignalReq(map_perform_handover_req(a, c))           = false;
      IsMapSendEndSignalReq(map_perform_handover_ind(a, c))           = false;
      IsMapSendEndSignalReq(map_perform_handover_rsp(a, c))           = false;
      IsMapSendEndSignalReq(map_perform_handover_cnf(a, c))           = false;
```

```
IsMapSendEndSignalReq(map_send_end_signal_req(a))                = true;
IsMapSendEndSignalReq(map_send_end_signal_ind(a))                = false;
IsMapSendEndSignalReq(map_send_end_signal_rsp(a))                = false;
IsMapSendEndSignalReq(map_send_end_signal_cnf(a, c))             = false;
IsMapSendEndSignalReq(map_process_access_signalling_req(a, b))   = false;
IsMapSendEndSignalReq(map_process_access_signalling_ind(a, b))   = false;
IsMapSendEndSignalReq(map_forward_access_signalling_req(a, b))   = false;
IsMapSendEndSignalReq(map_forward_access_signalling_ind(a, b))   = false;
IsMapSendEndSignalReq(map_perform_subsequent_handover_req(a, c)) = false;
IsMapSendEndSignalReq(map_perform_subsequent_handover_ind(a, c)) = false;
IsMapSendEndSignalReq(map_perform_subsequent_handover_rsp(a, c)) = false;
IsMapSendEndSignalReq(map_perform_subsequent_handover_cnf(a, c)) = false;

IsMapSendEndSignalInd(map_perform_handover_req(a, c))            = false;
IsMapSendEndSignalInd(map_perform_handover_ind(a, c))            = false;
IsMapSendEndSignalInd(map_perform_handover_rsp(a, c))            = false;
IsMapSendEndSignalInd(map_perform_handover_cnf(a, c))            = false;
IsMapSendEndSignalInd(map_send_end_signal_req(a))                = false;
IsMapSendEndSignalInd(map_send_end_signal_ind(a))                = true;
IsMapSendEndSignalInd(map_send_end_signal_rsp(a))                = false;
IsMapSendEndSignalInd(map_send_end_signal_cnf(a, c))             = false;
IsMapSendEndSignalInd(map_process_access_signalling_req(a, b))   = false;
IsMapSendEndSignalInd(map_process_access_signalling_ind(a, b))   = false;
IsMapSendEndSignalInd(map_forward_access_signalling_req(a, b))   = false;
IsMapSendEndSignalInd(map_forward_access_signalling_ind(a, b))   = false;
IsMapSendEndSignalInd(map_perform_subsequent_handover_req(a, c)) = false;
IsMapSendEndSignalInd(map_perform_subsequent_handover_ind(a, c)) = false;
IsMapSendEndSignalInd(map_perform_subsequent_handover_rsp(a, c)) = false;
IsMapSendEndSignalInd(map_perform_subsequent_handover_cnf(a, c)) = false;

IsMapSendEndSignalRsp(map_perform_handover_req(a, c))            = false;
IsMapSendEndSignalRsp(map_perform_handover_ind(a, c))            = false;
IsMapSendEndSignalRsp(map_perform_handover_rsp(a, c))            = false;
IsMapSendEndSignalRsp(map_perform_handover_cnf(a, c))            = false;
IsMapSendEndSignalRsp(map_send_end_signal_req(a))                = false;
IsMapSendEndSignalRsp(map_send_end_signal_ind(a))                = false;
IsMapSendEndSignalRsp(map_send_end_signal_rsp(a))                = true;
IsMapSendEndSignalRsp(map_send_end_signal_cnf(a, c))             = false;
IsMapSendEndSignalRsp(map_process_access_signalling_req(a, b))   = false;
IsMapSendEndSignalRsp(map_process_access_signalling_ind(a, b))   = false;
IsMapSendEndSignalRsp(map_forward_access_signalling_req(a, b))   = false;
IsMapSendEndSignalRsp(map_forward_access_signalling_ind(a, b))   = false;
IsMapSendEndSignalRsp(map_perform_subsequent_handover_req(a, c)) = false;
IsMapSendEndSignalRsp(map_perform_subsequent_handover_ind(a, c)) = false;
IsMapSendEndSignalRsp(map_perform_subsequent_handover_rsp(a, c)) = false;
IsMapSendEndSignalRsp(map_perform_subsequent_handover_cnf(a, c)) = false;

IsMapSendEndSignalCnf(map_perform_handover_req(a, c))            = false;
IsMapSendEndSignalCnf(map_perform_handover_ind(a, c))            = false;
IsMapSendEndSignalCnf(map_perform_handover_rsp(a, c))            = false;
IsMapSendEndSignalCnf(map_perform_handover_cnf(a, c))            = false;
IsMapSendEndSignalCnf(map_send_end_signal_req(a))                = false;
IsMapSendEndSignalCnf(map_send_end_signal_ind(a))                = false;
IsMapSendEndSignalCnf(map_send_end_signal_rsp(a))                = false;
IsMapSendEndSignalCnf(map_send_end_signal_cnf(a, c))             = true;
IsMapSendEndSignalCnf(map_process_access_signalling_req(a, b))   = false;
IsMapSendEndSignalCnf(map_process_access_signalling_ind(a, b))   = false;
IsMapSendEndSignalCnf(map_forward_access_signalling_req(a, b))   = false;
IsMapSendEndSignalCnf(map_forward_access_signalling_ind(a, b))   = false;
IsMapSendEndSignalCnf(map_perform_subsequent_handover_req(a, c)) = false;
IsMapSendEndSignalCnf(map_perform_subsequent_handover_ind(a, c)) = false;
IsMapSendEndSignalCnf(map_perform_subsequent_handover_rsp(a, c)) = false;
IsMapSendEndSignalCnf(map_perform_subsequent_handover_cnf(a, c)) = false;

IsMapProcessAccessSignReq(map_perform_handover_req(a, c))            = false;
IsMapProcessAccessSignReq(map_perform_handover_ind(a, c))            = false;
IsMapProcessAccessSignReq(map_perform_handover_rsp(a, c))            = false;
IsMapProcessAccessSignReq(map_perform_handover_cnf(a, c))            = false;
IsMapProcessAccessSignReq(map_send_end_signal_req(a))                = false;
IsMapProcessAccessSignReq(map_send_end_signal_ind(a))                = false;
IsMapProcessAccessSignReq(map_send_end_signal_rsp(a))                = false;
IsMapProcessAccessSignReq(map_send_end_signal_cnf(a, c))             = false;
IsMapProcessAccessSignReq(map_process_access_signalling_req(a, b))   = true;
IsMapProcessAccessSignReq(map_process_access_signalling_ind(a, b))   = false;
IsMapProcessAccessSignReq(map_forward_access_signalling_req(a, b))   = false;
IsMapProcessAccessSignReq(map_forward_access_signalling_ind(a, b))   = false;
IsMapProcessAccessSignReq(map_perform_subsequent_handover_req(a, c)) = false;
IsMapProcessAccessSignReq(map_perform_subsequent_handover_ind(a, c)) = false;
IsMapProcessAccessSignReq(map_perform_subsequent_handover_rsp(a, c)) = false;
IsMapProcessAccessSignReq(map_perform_subsequent_handover_cnf(a, c)) = false;

IsMapProcessAccessSignInd(map_perform_handover_req(a, c))            = false;
IsMapProcessAccessSignInd(map_perform_handover_ind(a, c))            = false;
IsMapProcessAccessSignInd(map_perform_handover_rsp(a, c))            = false;
IsMapProcessAccessSignInd(map_perform_handover_cnf(a, c))            = false;
```

```
IsMapProcessAccessSignInd(map_send_end_signal_req(a))                = false;
IsMapProcessAccessSignInd(map_send_end_signal_ind(a))                = false;
IsMapProcessAccessSignInd(map_send_end_signal_rsp(a))                = false;
IsMapProcessAccessSignInd(map_send_end_signal_cnf(a, c))             = false;
IsMapProcessAccessSignInd(map_process_access_signalling_req(a, b))   = false;
IsMapProcessAccessSignInd(map_process_access_signalling_ind(a, b))   = true;
IsMapProcessAccessSignInd(map_forward_access_signalling_req(a, b))   = false;
IsMapProcessAccessSignInd(map_forward_access_signalling_ind(a, b))   = false;
IsMapProcessAccessSignInd(map_perform_subsequent_handover_req(a, c)) = false;
IsMapProcessAccessSignInd(map_perform_subsequent_handover_ind(a, c)) = false;
IsMapProcessAccessSignInd(map_perform_subsequent_handover_rsp(a, c)) = false;
IsMapProcessAccessSignInd(map_perform_subsequent_handover_cnf(a, c)) = false;

IsMapForwardAccessSignReq(map_perform_handover_req(a, c))            = false;
IsMapForwardAccessSignReq(map_perform_handover_ind(a, c))            = false;
IsMapForwardAccessSignReq(map_perform_handover_rsp(a, c))            = false;
IsMapForwardAccessSignReq(map_perform_handover_cnf(a, c))            = false;
IsMapForwardAccessSignReq(map_send_end_signal_req(a))                = false;
IsMapForwardAccessSignReq(map_send_end_signal_ind(a))                = false;
IsMapForwardAccessSignReq(map_send_end_signal_rsp(a))                = false;
IsMapForwardAccessSignReq(map_send_end_signal_cnf(a, c))             = false;
IsMapForwardAccessSignReq(map_process_access_signalling_req(a, b))   = false;
IsMapForwardAccessSignReq(map_process_access_signalling_ind(a, b))   = false;
IsMapForwardAccessSignReq(map_forward_access_signalling_req(a, b))   = true;
IsMapForwardAccessSignReq(map_forward_access_signalling_ind(a, b))   = false;
IsMapForwardAccessSignReq(map_perform_subsequent_handover_req(a, c)) = false;
IsMapForwardAccessSignReq(map_perform_subsequent_handover_ind(a, c)) = false;
IsMapForwardAccessSignReq(map_perform_subsequent_handover_rsp(a, c)) = false;
IsMapForwardAccessSignReq(map_perform_subsequent_handover_cnf(a, c)) = false;

IsMapForwardAccessSignInd(map_perform_handover_req(a, c))            = false;
IsMapForwardAccessSignInd(map_perform_handover_ind(a, c))            = false;
IsMapForwardAccessSignInd(map_perform_handover_rsp(a, c))            = false;
IsMapForwardAccessSignInd(map_perform_handover_cnf(a, c))            = false;
IsMapForwardAccessSignInd(map_send_end_signal_req(a))                = false;
IsMapForwardAccessSignInd(map_send_end_signal_ind(a))                = false;
IsMapForwardAccessSignInd(map_send_end_signal_rsp(a))                = false;
IsMapForwardAccessSignInd(map_send_end_signal_cnf(a, c))             = false;
IsMapForwardAccessSignInd(map_process_access_signalling_req(a, b))   = false;
IsMapForwardAccessSignInd(map_process_access_signalling_ind(a, b))   = false;
IsMapForwardAccessSignInd(map_forward_access_signalling_req(a, b))   = false;
IsMapForwardAccessSignInd(map_forward_access_signalling_ind(a, b))   = true;
IsMapForwardAccessSignInd(map_perform_subsequent_handover_req(a, c)) = false;
IsMapForwardAccessSignInd(map_perform_subsequent_handover_ind(a, c)) = false;
IsMapForwardAccessSignInd(map_perform_subsequent_handover_rsp(a, c)) = false;
IsMapForwardAccessSignInd(map_perform_subsequent_handover_cnf(a, c)) = false;

IsMapPerformSubsequentHOReq(map_perform_handover_req(a, c))            = false;
IsMapPerformSubsequentHOReq(map_perform_handover_ind(a, c))            = false;
IsMapPerformSubsequentHOReq(map_perform_handover_rsp(a, c))            = false;
IsMapPerformSubsequentHOReq(map_perform_handover_cnf(a, c))            = false;
IsMapPerformSubsequentHOReq(map_send_end_signal_req(a))                = false;
IsMapPerformSubsequentHOReq(map_send_end_signal_ind(a))                = false;
IsMapPerformSubsequentHOReq(map_send_end_signal_rsp(a))                = false;
IsMapPerformSubsequentHOReq(map_send_end_signal_cnf(a, c))             = false;
IsMapPerformSubsequentHOReq(map_process_access_signalling_req(a, b))   = false;
IsMapPerformSubsequentHOReq(map_process_access_signalling_ind(a, b))   = false;
IsMapPerformSubsequentHOReq(map_forward_access_signalling_req(a, b))   = false;
IsMapPerformSubsequentHOReq(map_forward_access_signalling_ind(a, b))   = false;
IsMapPerformSubsequentHOReq(map_perform_subsequent_handover_req(a, c)) = true;
IsMapPerformSubsequentHOReq(map_perform_subsequent_handover_ind(a, c)) = false;
IsMapPerformSubsequentHOReq(map_perform_subsequent_handover_rsp(a, c)) = false;
IsMapPerformSubsequentHOReq(map_perform_subsequent_handover_cnf(a, c)) = false;

IsMapPerformSubsequentHOInd(map_perform_handover_req(a, c))            = false;
IsMapPerformSubsequentHOInd(map_perform_handover_ind(a, c))            = false;
IsMapPerformSubsequentHOInd(map_perform_handover_rsp(a, c))            = false;
IsMapPerformSubsequentHOInd(map_perform_handover_cnf(a, c))            = false;
IsMapPerformSubsequentHOInd(map_send_end_signal_req(a))                = false;
IsMapPerformSubsequentHOInd(map_send_end_signal_ind(a))                = false;
IsMapPerformSubsequentHOInd(map_send_end_signal_rsp(a))                = false;
IsMapPerformSubsequentHOInd(map_send_end_signal_cnf(a, c))             = false;
IsMapPerformSubsequentHOInd(map_process_access_signalling_req(a, b))   = false;
IsMapPerformSubsequentHOInd(map_process_access_signalling_ind(a, b))   = false;
IsMapPerformSubsequentHOInd(map_forward_access_signalling_req(a, b))   = false;
IsMapPerformSubsequentHOInd(map_forward_access_signalling_ind(a, b))   = false;
IsMapPerformSubsequentHOInd(map_perform_subsequent_handover_req(a, c)) = false;
IsMapPerformSubsequentHOInd(map_perform_subsequent_handover_ind(a, c)) = true;
IsMapPerformSubsequentHOInd(map_perform_subsequent_handover_rsp(a, c)) = false;
IsMapPerformSubsequentHOInd(map_perform_subsequent_handover_cnf(a, c)) = false;

IsMapPerformSubsequentHORsp(map_perform_handover_req(a, c))            = false;
IsMapPerformSubsequentHORsp(map_perform_handover_ind(a, c))            = false;
IsMapPerformSubsequentHORsp(map_perform_handover_rsp(a, c))            = false;
IsMapPerformSubsequentHORsp(map_perform_handover_cnf(a, c))            = false;
```

```
          IsMapPerformSubsequentHORsp(map_send_end_signal_req(a))                = false;
          IsMapPerformSubsequentHORsp(map_send_end_signal_ind(a))                = false;
          IsMapPerformSubsequentHORsp(map_send_end_signal_rsp(a))                = false;
          IsMapPerformSubsequentHORsp(map_send_end_signal_cnf(a, c))             = false;
          IsMapPerformSubsequentHORsp(map_process_access_signalling_req(a, b))   = false;
          IsMapPerformSubsequentHORsp(map_process_access_signalling_ind(a, b))   = false;
          IsMapPerformSubsequentHORsp(map_forward_access_signalling_req(a, b))   = false;
          IsMapPerformSubsequentHORsp(map_forward_access_signalling_ind(a, b))   = false;
          IsMapPerformSubsequentHORsp(map_perform_subsequent_handover_req(a, c)) = false;
          IsMapPerformSubsequentHORsp(map_perform_subsequent_handover_ind(a, c)) = false;
          IsMapPerformSubsequentHORsp(map_perform_subsequent_handover_rsp(a, c)) = true;
          IsMapPerformSubsequentHORsp(map_perform_subsequent_handover_cnf(a, c)) = false;

          IsMapPerformSubsequentHOCnf(map_perform_handover_req(a, c))            = false;
          IsMapPerformSubsequentHOCnf(map_perform_handover_ind(a, c))            = false;
          IsMapPerformSubsequentHOCnf(map_perform_handover_rsp(a, c))            = false;
          IsMapPerformSubsequentHOCnf(map_perform_handover_cnf(a, c))            = false;
          IsMapPerformSubsequentHOCnf(map_send_end_signal_req(a))                = false;
          IsMapPerformSubsequentHOCnf(map_send_end_signal_ind(a))                = false;
          IsMapPerformSubsequentHOCnf(map_send_end_signal_rsp(a))                = false;
          IsMapPerformSubsequentHOCnf(map_send_end_signal_cnf(a, c))             = false;
          IsMapPerformSubsequentHOCnf(map_process_access_signalling_req(a, b))   = false;
          IsMapPerformSubsequentHOCnf(map_process_access_signalling_ind(a, b))   = false;
          IsMapPerformSubsequentHOCnf(map_forward_access_signalling_req(a, b))   = false;
          IsMapPerformSubsequentHOCnf(map_forward_access_signalling_ind(a, b))   = false;
          IsMapPerformSubsequentHOCnf(map_perform_subsequent_handover_req(a, c)) = false;
          IsMapPerformSubsequentHOCnf(map_perform_subsequent_handover_ind(a, c)) = false;
          IsMapPerformSubsequentHOCnf(map_perform_subsequent_handover_rsp(a, c)) = false;
          IsMapPerformSubsequentHOCnf(map_perform_subsequent_handover_cnf(a, c)) = true;

endtype

type MapESpecServPrimPars is MapESpecServPrimDiscr
     opns
          InvokeIdOf   : MAPSS -> OpCode
          ApduOF       : MAPSS -> BSS_APDU
          OtherParasOf : MAPSS -> OtherParas
     eqns forall a:OpCode, b:BSS_APDU, c:OtherParas
     ofsort OpCode
          InvokeIdOf(map_perform_handover_req(a, c)) = a;
          InvokeIdOf(map_perform_handover_ind(a, c)) = a;
          InvokeIdOf(map_perform_handover_rsp(a, c)) = a;
          InvokeIdOf(map_perform_handover_cnf(a, c)) = a;
          InvokeIdOf(map_send_end_signal_req(a)) = a;
          InvokeIdOf(map_send_end_signal_ind(a)) = a;
          InvokeIdOf(map_send_end_signal_rsp(a)) = a;
          InvokeIdOf(map_send_end_signal_cnf(a, c)) = a;
          InvokeIdOf(map_process_access_signalling_req(a, b)) = a;
          InvokeIdOf(map_process_access_signalling_ind(a, b)) = a;
          InvokeIdOf(map_forward_access_signalling_req(a, b)) = a;
          InvokeIdOf(map_forward_access_signalling_ind(a, b)) = a;
          InvokeIdOf(map_perform_subsequent_handover_req(a, c)) = a;
          InvokeIdOf(map_perform_subsequent_handover_ind(a, c)) = a;
          InvokeIdOf(map_perform_subsequent_handover_rsp(a, c)) = a;
          InvokeIdOf(map_perform_subsequent_handover_cnf(a, c)) = a;
     ofsort BSS_APDU
          ApduOF(map_process_access_signalling_req(a, b)) = b;
          ApduOF(map_process_access_signalling_ind(a, b)) = b;
          ApduOF(map_forward_access_signalling_req(a, b)) = b;
          ApduOF(map_forward_access_signalling_ind(a, b)) = b;
     ofsort OtherParas
          OtherParasOf(map_perform_handover_req(a, c)) = c;
          OtherParasOf(map_perform_handover_ind(a, c)) = c;
          OtherParasOf(map_perform_handover_rsp(a, c)) = c;
          OtherParasOf(map_perform_handover_cnf(a, c)) = c;
          OtherParasOf(map_send_end_signal_cnf(a, c)) = c;
          OtherParasOf(map_perform_subsequent_handover_req(a, c)) = c;
          OtherParasOf(map_perform_subsequent_handover_ind(a, c)) = c;
          OtherParasOf(map_perform_subsequent_handover_rsp(a, c)) = c;
          OtherParasOf(map_perform_subsequent_handover_cnf(a, c)) = c;
endtype

(*------------------------- TCAP Service Primitives ------------------------------*)

type TcapServPrim is Boolean, NaturalNumber, TcapParameterTypes
      sorts TCAP
      opns
            tc_begin_req     : AC, UserInfo       -> TCAP
            tc_begin_ind     : AC, UserInfo       -> TCAP
            tc_invoke_req    : OpCode, Argument   -> TCAP
            tc_invoke_ind    : OpCode, Argument   -> TCAP
            tc_continue_req  : AC, UserInfo       -> TCAP
            tc_continue_ind  : AC, UserInfo       -> TCAP
            tc_end_req       :                    -> TCAP
            tc_end_ind       :                    -> TCAP
```

```
            _eq_, _ne_, _lt_ : TCAP, TCAP         -> Bool
            NatNum           : TCAP               -> Nat
      eqns forall sp1, sp2:TCAP
      ofsort Bool
            sp1 eq sp2 = NatNum(sp1) eq NatNum(sp2);
            sp1 ne sp2 = NatNum(sp1) ne NatNum(sp2);
            sp1 lt sp2 = NatNum(sp1) lt NatNum(sp2);
endtype

type TcapServPrimDiscr is TcapServPrim
      opns
            IsTcBeginReq     : TCAP -> Bool
            IsTcBeginInd     : TCAP -> Bool
            IsTcInvokeReq    : TCAP -> Bool
            IsTcInvokeInd    : TCAP -> Bool
            IsTcContinueReq  : TCAP -> Bool
            IsTcContinueInd  : TCAP -> Bool
            IsTcEndReq       : TCAP -> Bool
            IsTcEndInd       : TCAP -> Bool
      eqns forall a:AC, u:UserInfo, id:OpCode, g:Argument
           ofsort Bool

            IsTcBeginReq(tc_begin_req(a, u))    = true;
            IsTcBeginReq(tc_begin_ind(a, u))    = false;
            IsTcBeginReq(tc_invoke_req(id, g))   = false;
            IsTcBeginReq(tc_invoke_ind(id, g))   = false;
            IsTcBeginReq(tc_continue_req(a, u)) = false;
            IsTcBeginReq(tc_continue_ind(a, u)) = false;
            IsTcBeginReq(tc_end_req) = false;
            IsTcBeginReq(tc_end_ind) = false;

            IsTcBeginInd(tc_begin_req(a, u))    = false;
            IsTcBeginInd(tc_begin_ind(a, u))    = true;
            IsTcBeginInd(tc_invoke_req(id, g))   = false;
            IsTcBeginInd(tc_invoke_ind(id, g))   = false;
            IsTcBeginInd(tc_continue_req(a, u)) = false;
            IsTcBeginInd(tc_continue_ind(a, u)) = false;
            IsTcBeginInd(tc_end_req) = false;
            IsTcBeginInd(tc_end_ind) = false;

            IsTcInvokeReq(tc_begin_req(a, u))    = false;
            IsTcInvokeReq(tc_begin_ind(a, u))    = false;
            IsTcInvokeReq(tc_invoke_req(id, g))   = true;
            IsTcInvokeReq(tc_invoke_ind(id, g))   = false;
            IsTcInvokeReq(tc_continue_req(a, u)) = false;
            IsTcInvokeReq(tc_continue_ind(a, u)) = false;
            IsTcInvokeReq(tc_end_req) = false;
            IsTcInvokeReq(tc_end_ind) = false;

            IsTcInvokeInd(tc_begin_req(a, u))    = false;
            IsTcInvokeInd(tc_begin_ind(a, u))    = false;
            IsTcInvokeInd(tc_invoke_req(id, g))   = false;
            IsTcInvokeInd(tc_invoke_ind(id, g))   = true;
            IsTcInvokeInd(tc_continue_req(a, u)) = false;
            IsTcInvokeInd(tc_continue_ind(a, u)) = false;
            IsTcInvokeInd(tc_end_req) = false;
            IsTcInvokeInd(tc_end_ind) = false;

            IsTcContinueReq(tc_begin_req(a, u))    = false;
            IsTcContinueReq(tc_begin_ind(a, u))    = false;
            IsTcContinueReq(tc_invoke_req(id, g))   = false;
            IsTcContinueReq(tc_invoke_ind(id, g))   = false;
            IsTcContinueReq(tc_continue_req(a, u)) = true;
            IsTcContinueReq(tc_continue_ind(a, u)) = false;
            IsTcContinueReq(tc_end_req) = false;
            IsTcContinueReq(tc_end_ind) = false;

            IsTcContinueInd(tc_begin_req(a, u))    = false;
            IsTcContinueInd(tc_begin_ind(a, u))    = false;
            IsTcContinueInd(tc_invoke_req(id, g))   = false;
            IsTcContinueInd(tc_invoke_ind(id, g))   = false;
            IsTcContinueInd(tc_continue_req(a, u)) = false;
            IsTcContinueInd(tc_continue_ind(a, u)) = true;
            IsTcContinueInd(tc_end_req) = false;
            IsTcContinueInd(tc_end_ind) = false;

            IsTcEndReq(tc_begin_req(a, u))    = false;
            IsTcEndReq(tc_begin_ind(a, u))    = false;
            IsTcEndReq(tc_invoke_req(id, g))   = false;
            IsTcEndReq(tc_invoke_ind(id, g))   = false;
            IsTcEndReq(tc_continue_req(a, u)) = false;
            IsTcEndReq(tc_continue_ind(a, u)) = false;
            IsTcEndReq(tc_end_req)        = true;
            IsTcEndReq(tc_end_ind)        = false;
```

```
          IsTcEndInd(tc_begin_req(a, u))    = false;
          IsTcEndInd(tc_begin_ind(a, u))    = false;
          IsTcEndInd(tc_invoke_req(id, g))   = false;
          IsTcEndInd(tc_invoke_ind(id, g))   = false;
          IsTcEndInd(tc_continue_req(a, u)) = false;
          IsTcEndInd(tc_continue_ind(a, u)) = false;
          IsTcEndInd(tc_end_req)        = false;
          IsTcEndInd(tc_end_ind)        = true;
endtype

type TcapServPrimPars is TcapServPrimDiscr
     opns
          ACOf       : TCAP -> AC
          UserInfoOf : TCAP -> UserInfo
          InvokeIdOf : TCAP -> OpCode
          ArgumentOf : TCAP -> Argument
     eqns forall a:AC, u:UserInfo, id:OpCode, g:Argument
     ofsort AC
          ACOf(tc_begin_req(a, u)) = a;
          ACOf(tc_begin_ind(a, u)) = a;
          ACOf(tc_continue_req(a, u)) = a;
          ACOf(tc_continue_ind(a, u)) = a;
     ofsort UserInfo
          UserInfoOf(tc_begin_req(a, u)) = u;
          UserInfoOf(tc_begin_ind(a, u)) = u;
          UserInfoOf(tc_continue_req(a, u)) = u;
          UserInfoOf(tc_continue_ind(a, u)) = u;
     ofsort OpCode
          InvokeIdOf(tc_invoke_req(id, g)) = id;
          InvokeIdOf(tc_invoke_ind(id, g)) = id;
     ofsort Argument
          ArgumentOf(tc_invoke_req(id, g)) = g;
          ArgumentOf(tc_invoke_ind(id, g)) = g;
endtype

(*------------------------ Specific Queues and Buffers ------------------------*)

type TCAPQ is Queue actualizedby TcapServPrim using
     sortnames  TCAPQ for Queue
                TCAP  for Element
                Bool  for FBool
endtype

(*------------------------------ Constants -----------------------------------*)

type Constraints is
    sorts Constraints
    opns  Calling, Called : -> Constraints
endtype (*  *)

(*------------------- End of Specific Service Primitives ADT part --------------*)
```

Appendix III: LOTOS spec of the MAP/E service-provider (behaviour part)

```
behaviour
   MAP_E_Service[AA, BB]

where
   process MAP_E_Service[AA, BB] : noexit :=
   hide a, b in
      (PE[AA, a] ||| PE[BB, b])
      |[a, b]|
      TCAPServices[a, b]

   where
      process PE[X, y] : noexit :=
         (UpperLSI[X] ||| LowerLSI[y])
         ||
         PFs[X, y]

      where
         process UpperLSI[X] : noexit :=
            HandoverULSI[X]

         where
            process HandoverULSI[X] : noexit :=
               CallingHOULSI[X] ||| CalledHOULSI[X]

            where
               process CallingHOULSI[X] : noexit :=
                  X ? sp:MAPCS ! Calling [IsMapOpenReq(sp) and (ACOf(sp) eq Handover_Control)];
                  X ? sp:MAPSS ! Calling [IsMapPerformHandoverReq(sp)];
                  X ? sp:MAPCS ! Calling [IsMapDelimiterReq(sp)];
                  X ? sp:MAPCS ! Calling [IsMapOpenCnf(sp) and (ACOf(sp) eq Handover_Control)];
                  X ? sp:MAPSS ! Calling [IsMapPerformHandoverCnf(sp)];
                  X ? sp:MAPCS ! Calling [IsMapDelimiterInd(sp)];
                  X ? sp:MAPSS ! Calling [IsMapSendEndSignalInd(sp)];
                  X ? sp:MAPCS ! Calling [IsMapDelimiterInd(sp)];
                  X ? sp:MAPSS ! Calling [IsMapSendEndSignalRsp(sp)];
                  X ? sp:MAPCS ! Calling [IsMapCloseReq(sp)];
                  CallingHOULSI[X]
               endproc (*  CallingULSI *)

               process CalledHOULSI[X] : noexit :=
                  X ? sp:MAPCS ! Called [IsMapOpenInd(sp) and (ACOf(sp) eq Handover_Control)];
                  X ? sp:MAPSS ! Called [IsMapPerformHandoverInd(sp)];
                  X ? sp:MAPCS ! Called [IsMapDelimiterInd(sp)];
                  X ? sp:MAPCS ! Called [IsMapOpenRsp(sp) and (ACOf(sp) eq Handover_Control)];
                  X ? sp:MAPSS ! Called [IsMapPerformHandoverRsp(sp)];
                  X ? sp:MAPCS ! Called [IsMapDelimiterReq(sp)];
                  X ? sp:MAPSS ! Called [IsMapSendEndSignalReq(sp)];
                  X ? sp:MAPCS ! Called [IsMapDelimiterReq(sp)];
                  X ? sp:MAPSS ! Called [IsMapSendEndSignalCnf(sp)];
                  X ? sp:MAPCS ! Called [IsMapCloseInd(sp)];
                  CalledHOULSI[X]
               endproc (* CalledHOULSI *)
            endproc (* HandoverUlSI *)
         endproc (* UpperLSI *)

         process PFs[X, y] : noexit :=
            HandoverPFs[X, y]

         where
            process HandoverPFs[X, y] : noexit :=
               CallingPFs[X, y] ||| CalledPFs[X, y]

            where
               process CallingPFs[X, y] : noexit :=
                  X ? sp1:MAPCS ! Calling [ IsMapOpenReq(sp1) and
                                            (ACOf(sp1) eq Handover_Control)];
                  X ? sp2:MAPSS ! Calling [IsMapPerformHandoverReq(sp2)];
                  y ! tc_invoke_req(InvokeIdOf(sp2), arg_1) ! Calling ; stop
                  |||
                  X ? sp3:MAPCS ! Calling [IsMapDelimiterReq(sp3)];
                  y ! tc_begin_req(Handover_Control, ui_1 ) ! Calling ; stop
                  |||
                  y ? sp1:TCAP ! Calling [ IsTcContinueInd(sp1) and           (* opening phase *)
                                           (ACOf(sp1) eq Handover_Control)];
                  X ! map_open_cnf(ACOf(sp1), op_1) ! Calling;
                  y ? sp1:TCAP ! Calling [ IsTcContinueInd(sp1) and        (* continuing phase *)
                                           (ACOf(sp1) eq Handover_Control)]; stop
                  |||
                  y ? sp2:TCAP ! Calling [ IsTcInvokeInd(sp2) and
                                           (InvokeIdOf(sp2) eq PerformHandover)];
                  X ! map_perform_handover_cnf(InvokeIdOf(sp2 of TCAP), op_1) ! Calling;
```

```
            X ! map_delimiter_ind ! Calling; stop
            |||
            y ? sp1:TCAP ! Calling [ IsTcInvokeInd(sp1) and
                                     (InvokeIdOf(sp1) eq SendEndSignal)];
            X ! map_send_end_signal_ind(InvokeIdOf(sp1)) ! Calling;
            X ! map_delimiter_ind ! Calling; stop
            |||
            X ? sp1:MAPSS ! Calling [IsMapSendEndSignalRsp(sp1)];
            y ! tc_invoke_req(InvokeIdOf(sp1), arg_1) ! Calling; stop
            |||
            X ? sp2:MAPCS ! Calling [IsMapCloseReq(sp2)];
            y ! tc_end_req ! Calling; CallingPFs[X, y]
         endproc (* CallingPFs *)

         process CalledPFs[X, y] : noexit :=
            y ? sp1:TCAP ! Called [IsTcBeginInd(sp1) and (ACOf(sp1) eq Handover_Control)];
            X ! map_open_ind(ACOf(sp1), op_1) ! Called; stop
            |||
            y ? sp2:TCAP ! Called [ IsTcInvokeInd(sp2) and
                                    (InvokeIdOf(sp2) eq PerformHandover)];
            X ! map_perform_handover_ind(InvokeIdOf(sp2), op_1) ! Called;
            X ! map_delimiter_ind ! Called; stop
            |||
            X ? sp1:MAPCS ! Called [IsMapOpenRsp(sp1) and (ACOf(sp1) eq Handover_Control)];
            X ? sp2:MAPSS ! Called [IsMapPerformHandoverRsp(sp2)];
            y ! tc_invoke_req(InvokeIdOf(sp2), arg_1) ! Called; stop
            |||
            X ? sp3:MAPCS ! Called [IsMapDelimiterReq(sp3)];         (* Opening phase *)
            y ! tc_continue_req(Handover_Control, ui_1) ! Called; stop
            |||
            X ? sp3:MAPCS ! Called [IsMapDelimiterReq(sp3)];         (* Continuing phase *)
            y ! tc_continue_req(Handover_Control, ui_1) ! Called; stop
            |||
            X ? sp1:MAPSS ! Called [IsMapSendEndSignalReq(sp1)];
            y ! tc_invoke_req(InvokeIdOf(sp1), arg_1) ! Called; stop
            |||
            y ? sp1:TCAP ! Called [IsTcEndInd(sp1)];
            y ? sp2:TCAP ! Called [ IsTcInvokeInd(sp2) and
                                    (InvokeIdOf(sp2) eq SendEndSignal)];
            X ! map_send_end_signal_cnf(InvokeIdOf(sp2), op_1) ! Called;
            X ! map_close_ind ! Called; CalledPFs[X, y]
         endproc (* CalledPFs *)
      endproc (* PFs *)
   endproc (* HandoverPFs *)

   process LowerLSI[y] : noexit :=
      HandoverLLSI[y]

   where
      process HandoverLLSI[y] : noexit :=
         CallingHOLLSI[y] ||| CalledHOLLSI[y]

      where
         process CalledHOLLSI[y] : noexit :=
            y ? sp:TCAP ! Called [IsTcBeginInd(sp) and (ACOf(sp) eq Handover_Control)];
            y ? sp:TCAP ! Called [ IsTcInvokeInd(sp) and
                                   (InvokeIdOf(sp) eq PerformHandover)];
            y ? sp:TCAP ! Called [ IsTcInvokeReq(sp) and
                                   (InvokeIdOf(sp) eq PerformHandover)];
            y ? sp:TCAP ! Called [IsTcContinueReq(sp) and (ACOf(sp) eq Handover_Control)];
            y ? sp:TCAP ! Called [IsTcInvokeReq(sp) and (InvokeIdOf(sp) eq SendEndSignal)];
            y ? sp:TCAP ! Called [IsTcContinueReq(sp) and (ACOf(sp) eq Handover_Control)];
            y ? sp:TCAP ! Called [IsTcEndInd(sp)];
            y ? sp:TCAP ! Called [IsTcInvokeInd(sp) and (InvokeIdOf(sp) eq SendEndSignal)];
            CalledHOLLSI[y]
         endproc (* CalledHOLLSI *)

         process CallingHOLLSI[y] : noexit :=
            y ? sp:TCAP ! Calling [ IsTcInvokeReq(sp) and
                                    (InvokeIdOf(sp) eq PerformHandover)];
            y ? sp:TCAP ! Calling [IsTcBeginReq(sp) and (ACOf(sp) eq Handover_Control)];
            y ? sp:TCAP ! Calling [IsTcContinueInd(sp) and (ACOf(sp) eq Handover_Control)];
            y ? sp:TCAP ! Calling [ IsTcInvokeInd(sp) and
                                    (InvokeIdOf(sp) eq PerformHandover)];
            y ? sp:TCAP ! Calling [IsTcContinueInd(sp) and (ACOf(sp) eq Handover_Control)];
            y ? sp:TCAP ! Calling [ IsTcInvokeInd(sp) and
                                    (InvokeIdOf(sp) eq SendEndSignal)];
            y ? sp:TCAP ! Calling [ IsTcInvokeReq(sp) and
                                    (InvokeIdOf(sp) eq SendEndSignal)];
            y ? sp:TCAP ! Calling [IsTcEndReq(sp)];
            CallingHOLLSI[y]
         endproc (* CallingHOLLSI *)
      endproc (* HandoverLLSI *)
   endproc (* LowerLSI *)
endproc (* PE *)
```

```
      process TCAPServices[a, b] : noexit :=
         TCAPService[a, b](empty of TCAPQ, empty of TCAPQ)
         |||
         TCAPService[b, a](empty of TCAPQ, empty of TCAPQ)

      where
         process TCAPService[a, b](stack:TCAPQ, qstack:TCAPQ) : noexit :=
               a ? sp:TCAP [IsTcInvokeReq(sp)];
                     TCAPService[a, b](add(sp, stack), qstack)
            [] a ? sp:TCAP [IsTcBeginReq(sp)];
                     TCAPService[a, b](empty of TCAPQ, concat(dda(sp, stack), qstack))
            [] a ? sp:TCAP [IsTcContinueReq(sp)];
                     TCAPService[a, b](empty of TCAPQ, concat(dda(sp, stack), qstack))
            [] a ? sp:TCAP [IsTcEndReq(sp)];
                     TCAPService[a, b](empty of TCAPQ, concat(dda(sp, stack), qstack))
            [] [NotEmpty(qstack)] ->
            (  let sp:TCAP = last(qstack) in
               b ! tc_invoke_ind(InvokeIdOf(sp), ArgumentOf(sp)) [IsTcInvokeReq(sp)];
                     TCAPService[a, b](stack, init(qstack))
            [] b ! tc_begin_ind(ACOf(sp), UserInfoOf(sp)) [IsTcBeginReq(sp)];
                     TCAPService[a, b](stack, init(qstack))
            [] b ! tc_continue_ind(ACOf(sp), UserInfoOf(sp)) [IsTcContinueReq(sp)];
                     TCAPService[a, b](stack, init(qstack))
            [] b ! tc_end_ind [IsTcEndReq(sp)];
                     TCAPService[a, b](stack, init(qstack))
            )
         endproc (* TCAPservice *)
      endproc (* TCAPServices *)
   endproc (* MAP_E_Service *)
endspec (* MapService *)
```