Formal Specification and Feature Interaction Detection in the Intelligent Network

Jalel KAMOUN

Thesis Submitted to the School of Graduate Studies and Research
in Partial Fulfillment of the Requirements for the Degree of
Master of Computer Science
under the Auspices of the Ottawa-Carleton Institute of Computer Science

Department of Computer Science,
University of Ottawa,
Ottawa, Ontario, Canada
January 1996

Acknowledgments

I would like to thank all the people who assisted me in completing this work.

First, I would like to express my deepest appreciation to my supervisor, Prof. Luigi Logrippo for his support, encouragement, and fruitful discussions throughout my graduate studies. His accuracy in reviewing the drafts of the thesis have greatly improved the contents and its presentation. The many discussions we have had since I started research work towards this thesis have been of great influence.

I am also much indebted to Mohamed Faci for initiating the idea of this thesis and for the inspiring and fruitful discussions we have had together.

I also owe special thanks to Bernard Stepien for his useful comments about some parts of the thesis and for his careful reading and suggestions on an early versions of some parts of the thesis, which improved the presentation of its final version.

I would like to thank Jacques Sincennes for his technical support and for his help with the use of the LOTOS toolkit and the Goal Oriented Execution tool.

Finally, I would like to express my gratitude for the financial support of the Tunisian Government under the scholarship program sponsored by CIDA, and of Motorola-ARRC.

Abstract

Over the past few years, the subject of *Intelligent Network* (IN) has captured the interest of the telecommunications community. The objective of IN is to allow the introduction of new capabilities in the telecommunications network and to facilitate and accelerate in a cost-effective manner, service implementation and provisioning, in a multivendor environment. However, this objective confronts a major obstacle known as the *feature interaction* problem. The feature interaction problem occurs when a feature is prevented from performing its functionalities in the presence of other features.

In the first part of the thesis, we describe a LOTOS model for structuring the Functional Entities (FEs) that are defined in the Distributed Functional Plane (DFP) of the CS1 IN Conceptual Model (INCM), and that are involved in the establishment of a call/connection and invocation and processing of services. The specification of IN services is achieved using Service Independent building Blocks (SIBs). It is designed in a way that independent specification and rapid introduction of services is provided.

In the second part of the thesis, a method for detecting feature interactions between services is developed. The method is limited to the detection of interactions caused by violation of features properties. It is based on formalization of feature's properties, derivation of goals satisfying the negation of these properties and use of Goal Oriented Execution to detect traces satisfying these goals. A trace satisfying a goal shows that an interaction exists between the specified features by describing a scenario violating one of the properties of the introduced features.

It is concluded that LOTOS is useful as a Formal Description Technique (FDT) in the Service Creation Environment (SCE). The developed specification can be used for adding specifications of new services, and for detecting interactions caused by violation of properties, if there are any.

List of Figures

CHAPTER 1 Introduction: Motivation and Background

1.1 Background and Motivation

The telecommunication industry today is increasingly focusing on growing demand for new services (e.g. Universal Personal Telecommunication UPT). However, with the infrastructure provided by the Plain Old Telephone System (POTS), the task of introducing a new service is tedious and very costly. To overcome the limitations of POTS, Intelligent Networks (IN) are introduced in order to facilitate the creation and provision of telecommunication services.

One of the aims of IN is independent service implementation, which means that every service provider will be able to define its own services independently and deploy them in the network. Each service provider uses its Service Creation Environment (SCE) to define and develop services and then deploy them in the network.

However, the rapid development of services is hindered by the feature interaction problem. Such problem occurs when a feature is prevented from performing its functionalities in the presence of other features. This problem can arise at any stage of the service lifecycle. Interaction can occur at the specification stage as well as at design or implementation stages.

Interactions should be detected as early as possible otherwise they will propagate through the next stages of the service lifecycle. Therefore, the process of interaction detection should start at the specification stage.

Formal Description Techniques (FDTs) such as LOTOS [ISO8807] and SDL [CCITT87] have proven useful in detecting feature interactions at the specification level [Zave93]. In fact, a formal description of the system behavior with the introduced services provides an unambiguous and precise view of the system that can support formal analysis and validation methods. Currently, the most common language for specifying telephony systems is SDL. However, CCITT has been considering the eventual appropriateness of LOTOS as a more advanced description language.

These facts motivated this thesis, which consists in defining in LOTOS a model for structuring IN components with the purpose of providing a suitable model for formalizing and validating IN services and service features, and in developing a methodology for detecting the feature interaction problem between services.

In contrast to earlier work on specification and validation of telephony systems in LOTOS, which concentrated mainly on showing the suitability of LOTOS for specifying such systems and the different specification styles that can be used, the present study attempts to deal with the concepts and objectives of IN which consist mainly in allowing rapid introduction and deployment of services in an implementation independent way. If rapid introduction and independent implementation are desirable, the formal specification and validation process should also be able to be performed rapidly.

1.2 Related Work

1.2.1 Formal Specifications of Telephone Systems in LOTOS

Several specifications and formalizations of telephony systems and services using LOTOS have been presented. They were developed in the context of POTS as well as IN. A formal specification of telephone systems, using the constraint-oriented style was described in [FaLS91], a design methodology for the description in LOTOS of telephone systems with the inclusion of additional services was published in [BoLo93]. The work presented in [SL93] describes a new approach for specifying telephone systems using a mixture of the constraint-oriented style and the state-oriented style also called the status oriented style. Other work which aims at discussing different architectural models for specifying telephony system in LOTOS was presented in [FaLS95]. In the context of IN we mention the work presented in [BZ92] describing the LOTOS formal specification of IN Global Functional Plane, the work of Elie Najm describing a design methodology of IN services in LOTOS [DaNa93] and the formal specification and design of IN services in LOTOS described in [Cheng94].

1.2.2 The Feature Interaction Problem

The feature interaction problem in telecommunication systems has been known for several years [BCDGL88]. Nevertheless, no standards or framework have yet been established to formally define the problem and its terminology. This is due essentially to the huge variety of problems that can be classified under this heading [CoPi94]. Meanwhile, many formulations of the problem and research work in this area have been proposed [BCDGL88], [CaLi91], [Lee92], [SL94], [FaL94].

In this thesis, we use a reformulation of the definition given in [BCDGL88] and which focuses on the manifestation of interactions. It states that there is an interaction between features:

- when a feature inhibits or subverts the expected behavior of another feature considered separately.

- when the joint accurate execution of two features provokes a supplementary phenomenon which cannot occur during the processing of each of the features considered separately.

The feature interaction problem can be addressed according to three approaches: *avoidance*, *detection* and *resolution* [Fits94].

Approaches for *avoidance* are aimed at developing service platforms and service creation environments that lead to service implementation with a minimum probability of interaction. Such approaches can be realized by incorporating Open Distributed Computing Platforms in telecommunication systems to deal with interaction [MiTJ93], [Li94], or by defining additional guidelines that deal with the interaction problem for service creation and service management [Fits94].

Detection of interactions is to analyse a set of independently specified features and determine whether or not there are any possibilities of conflicts between their joint behaviors [CaLi91], [Lee92], [BoLo93], [DaNa93], [SL94], [Faci95]. Detection can be applied through the whole life-cycle of a feature, since the cause of interaction can be related to any phase of the feature life-cycle.

Detection of interaction during the service creation process is known as *off-line* detection, while detection at run-time is known as *on-line* detection.

Once an interaction is detected, one must think of how it can be resolved. Then, similar to detection, *Resolution* can be done both *off-line* or *on-line*, depending on the phase of the feature life-cycle where the interaction was detected [Cain92], [GrVe92], [Chen94].

1.2.3 Detecting Feature Interaction at specification level using FDTs

Since we are dealing with the feature interaction problem at specification level using LOTOS as FDT, we give a brief overview of some approaches for detecting feature interactions at the specification level using FDTs. It should be noted that *feature interaction* is a research area of some importance, and that a number of papers are published every year on the subject. Three International Workshops have been held so far [FWFI92], [Fits94] and [Fits95]. In what follows, we limit ourselves into reviewing work closely related to ours in point form:

- Bouma and Zuidweg [BZ92] modeled IN services as defined in the Global Functional Plane of the IN Conceptual Model in LOTOS and used model checking to validate properties of services when they are integrated together. Interaction is detected when a property of a service is not verified. This approach allows for a complete analysis of the specification. Its limitation is that it requires to produce a full Kripke System for the specification. In our experiences, we found that such systems are too large for the specification presented in this thesis, leading easily to the *state explosion* problem. Thus this approach cannot be used for our case studies.
- Boumezbeur and Logrippo [BoLo93] proposed a LOTOS specification of the a sample telephone system and applied the *step-by-step* execution to detect feature interactions. At each step of the step-by-step execution, the user chooses the next action to be taken among all possible actions that are offered at that point. This is useful for checking the conformance of a system defined informally to its formal description in LOTOS. In practice, this can be done by checking if test sequences that should be allowed according to the informal definition are also accepted by the formal specification; or checking if the test sequences obtained by executing the specification are included in the formal definition of the system; or by checking if test sequences that are not specified informally are not accepted by the formal specification.
- A similar approach to [BZ92] based on model checking was adopted by Combes and Pikin in [CoPi94]. They developed an abstract model, representing the user external view, of the network

and the introduced features using SDL as a formal language. Then, they expressed feature requirements and properties in a temporal logic language and applied the model checker tool to validate the features properties. We believe that the limitations of this method are the same as those mentioned above, i.e. state explosion.

• Stepien and Logrippo [SL94] developed a method to detect feature interaction using backward reasoning, which involves specification of features in LOTOS. Interactions to be detected are caused by ambiguity of actions. An observable action in a LOTOS specification is ambiguous if in the behavior tree of the specification there is a branching point where the action is the first observable one in at least two branches. Ambiguity represents non deterministic behavior of the system being specified, and is a symptom of feature interaction. To prove that an action is ambiguous, backward reasoning for LOTOS is applied. It consists of a combination of backward and forward execution. Forward execution of the specification is applied to reach the action, then, using the resulting behavior expression, backward execution is performed to find a different trace leading to the action. A tool to help carry out backward execution is presented.

• Faci and Logrippo [FaL94], [Faci95] developed a methodology for detecting feature interactions. They proposed a formalization of the notion of *feature interaction.* The formalization is based on the idea that an interaction exists between *n* features if one of the features cannot exhibit its behavior when integrated into POTS in combination with other features. This idea was the basis for defining the notions of *composition* and *integration* of features. *Composition* expresses the synchronization of features on their common actions with POTS and their interleaving on their independent actions. *Integration* expresses the extension of POTS with the *n* features, such that each feature is able to execute all of its actions which are allowed in the context of POTS, when the other features are disabled. Then, they reason about interactions in terms of the *conformance* relation studied in testing theory, in the following way: an interaction exists between *n* features if their *integration* does not *conform* to their *composition*.

• In [SL95], a method for representing and verifying intentions in telephony features using abstract data types is presented. Feature intentions describe the intended behavior of telephony features. The first step of the method is to specify a feature's intentions using abstract data types. Intentions of a feature are described independently of other features without consideration of potential interactions at this stage. They are described for every operation that exists in the system regardless of which feature is actually used, and are implemented as Abstract Data Types operations which specify the intention's violation. The specification language considered is LOTOS. The second step consists in executing a formal specification of the system with features. The abstract data types descriptions of feature intentions are included in the specification, and a monitor for verifying intentions of features described as LOTOS processes is introduced to verify the intentions as described in the abstract data types every time an action of the specification is executed.

1.3 Contribution of the thesis

The major contributions of this thesis consist in developing a model for specifying IN services as described in the CS1 Distributed Functional Plane of the Intelligent Network Conceptual Model and an approach for detecting the problem of feature interaction at the specification level.

1.3.1 Contribution 1: A model for specifying IN services in LOTOS

In Chapter 4, we present a model for specifying IN services as defined in the CS1 Distributed Functional Plane (DFP). In this model, the parts of the DFP involved in call/connection establishment and invocation of services are specified. These parts consist of four Functional Entities which are the Call Control Function, the Service Switching Function, the Service Control Function and the Service Data Function. IN services are specified using the concept of Service Independent building Blocks (SIBs). The model provides independent specification and rapid introduction of IN services in the sense that each IN service can be specified independently using SIBs and added to the global specification without any major modifications. This is one of the main objective of the IN. Concrete examples of IN service specifications are given in Chapter 5.

1.3.2 Contribution 2: Detecting interactions between IN services

In Chapter 5, a method for detecting feature interactions between IN services is developed. This method is limited to interactions occurring at the abstract specification level and resulting in violation of feature properties. It is composed of three steps. The first one consists in specifying the properties and requirements of the introduced features in a formal property language. The temporal logic CTL was chosen for this purpose. In the second step, goals (expressed by an action or a sequence of actions from the specification) which satisfy the negation of the feature properties are derived. Then, the third step consists in applying Goal Oriented Execution to look for traces in the specification satisfying the derived goals. A selected trace shows a scenario which violates one of the feature properties. An application of the method to three pair-wise features was presented.

1.4 Organization of the thesis

The five remaining chapters will cover the following issues:

Chapter 2: *Intelligent Network Concepts*

We review the IN concepts and architectural model by describing the different planes of the Intelligent Network Conceptual Model, with an emphasis on the parts that are treated in this thesis. A mapping between the different planes is also described.

Chapter 3: *Using LOTOS for specifying Intelligent Networks*

We give an overview of the LOTOS specification language by describing its main operators and by giving examples in the context of telephony and Intelligent Network.

Chapter 4: *Specifying IN Call Model and Services in LOTOS*

We present the LOTOS formal specification of the IN call model and services as defined in the Distributed Functional Plane of the Intelligent Network Conceptual Model. We describe the specification architecture and the main processes of which it is composed.

Chapter 5: *Detecting Feature Interaction between IN Services*

We describe the approach adopted for detecting interactions between services and we illustrate it by three pair-wise feature interaction examples. For each feature we give an informal description followed by its formal specification and we show how it can be added to the global specification. Then we show how the interaction is detected by applying the methodology.

Chapter 6: *Conclusion and future work*

Conclusions and future work are presented in this chapter.

CHAPTER 2 Intelligent Network Concepts

In this chapter, we give an overview of Intelligent Network (IN) concepts and architectural model by describing its different planes and the relation between them.

2.1 Introduction

Telephony systems have evolved in several phases. First, telephones were based on central offices where exchanges were manned manually by operators. Later on, automatic switches were introduced. They were operated electromechanically by using electrical relays. The development of transistors permitted the development of electronic switches which made possible the storage of software programs and data within switches.

This has resulted in a transition from a basic telephone system which had provided only the basic functionality of making phone calls, to a more sophisticated system in which new services have been introduced, in order to assist, help and provide more control to the network subscribers in establishing calls.

Before 1980s, services were switch-based, which means that all the data and logic processing required by the services were located within the local node (see figure 1).

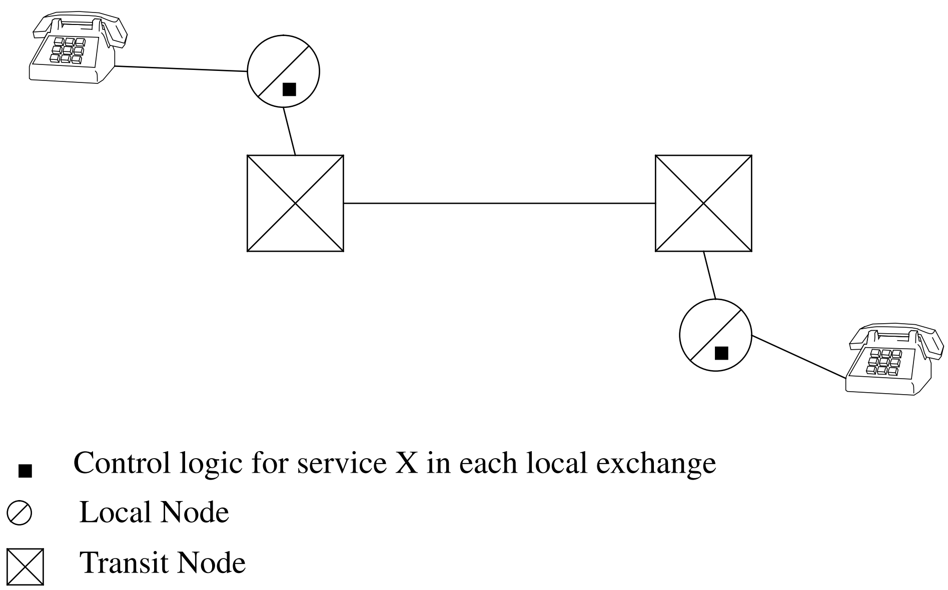


Figure 1: Example of service setup in pre-IN networks

This technique has two major drawbacks. The first one comes from the fact that software related to services must be located in all the local exchange nodes (local switches) to which telephone sets are directly connected. As a result, any software modification should be done in all those local nodes. The second problem is associated with the fact that different types of switches provided by different telecommunication companies could be deployed, therefore the introduction of a new service requires the adaptation of the related software to every type of switch in the network. With these complexities and the effort required, a new service typically requires three or four years to be deployed into the network [Lee92], [Vi95].

2.2 The Essential Elements of Intelligent Networks

By 1980s, in order to allow the introduction of new capabilities in the telecommunications network and to facilitate and accelerate in a cost-effective manner service implementation and provisioning, the concept of Intelligent Network was introduced. It is based on two essential elements which are the Common Channel Signaling and Non-switching nodes [Vi95].

2.2.1 Common Channel Signaling

The Common Channel Signaling is a signaling system where all signaling is performed over transmission paths completely separated from the voice path [Keiser95]. Such a system enables the exchange of different signals, such as supervisory signals and address signals, by transmitting messages between the different nodes over a network of signaling links, instead of using the voice transmission paths. CCITT has defined two common channel signaling systems. These are: CCITT CCSS No 6 using analog voice-band transmission, and CCSS No 7 which evolved from the former, and uses the standard 64 kb/s digital transmission link [Th94].

2.2.2 Non-switching Node

The CCSS No 7 common channel signaling has enabled the introduction of non-switching nodes where service logic and data could be stored. This means that the service control is centralized in some specific nodes. Those nodes are known as Service Control Points (SCPs) and Service Data Points (SDPs) and are defined in section 2.3.4. They are accessible to the switch via protocols using CCSS No 7. Figure 2 describes the two essential elements of IN.

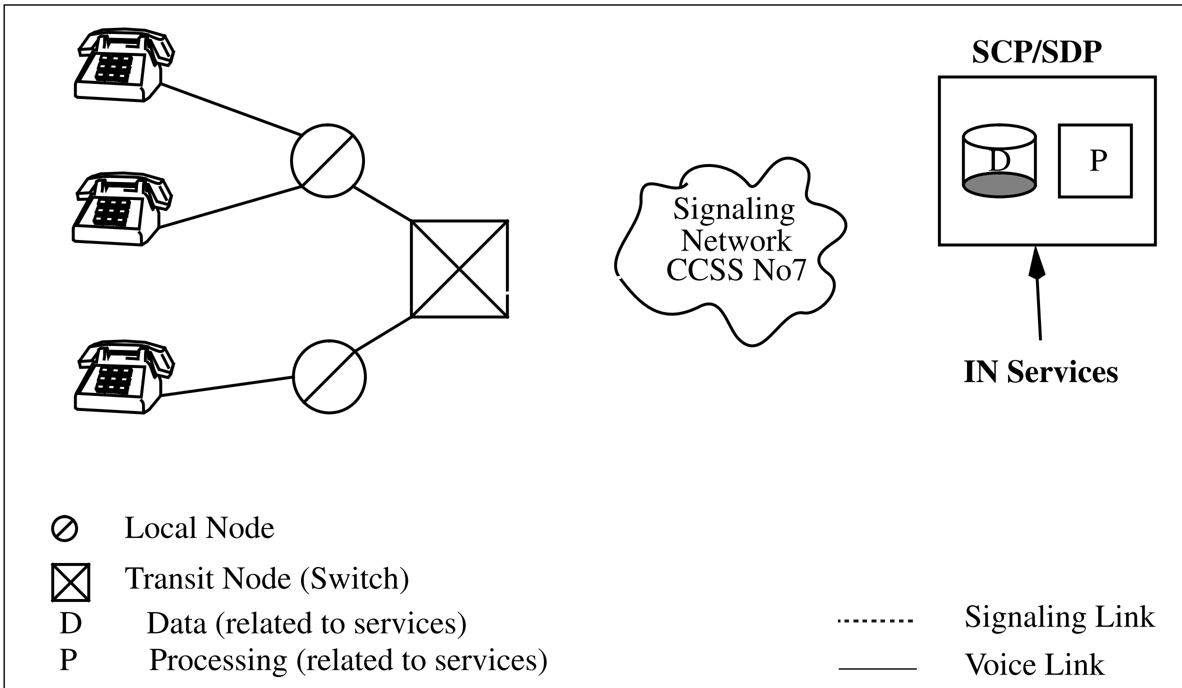


Figure 2: The elementary Intelligent Network

This has resulted in the introduction of the first IN services which are the 800 service (also known as freephone numbers) and Automatic Calling Card Service [Vi95].

When a service is to be invoked, a message indicating a request to process the service is sent from the switch to the SCP via the signaling network, then the service is processed within the SCP. At the end, a message is sent from the SCP back to the switch to continue the processing of the call. During the processing of the service, all the data and information that should be exchanged between the switch and the SCP are transmitted over the signaling network.

However, although services could be located in nodes other than switches, they were dependent on specific trigger events. In fact, each service has its triggering mechanism defined within the switch and a service is launched at the SCP via this triggering mechanism. As a result, any introduction of a new service involves the definition of the process which triggers this service within the switch.

In addition, each service has its specific protocol to communicate with the switch and its specific logic and data. This implies that if a new service is to be created, a new protocol should be defined.

To handle this problem, a new approach was taken. It consists in introducing a number of well-defined service-independent trigger checkpoints within the switch and defining a service-independent interface between the switch and the SCP. As a result, the deployment of a new service does not need the definition of a new protocol for the communication between the SCP and the switch, nor does it need a modification of the switch to introduce the associated triggering mechanism. A simple information to the switch that a new service has been deployed and should be triggered under certain criteria is sufficient.

Figure 3 shows the evolution of the telephone system over the years:

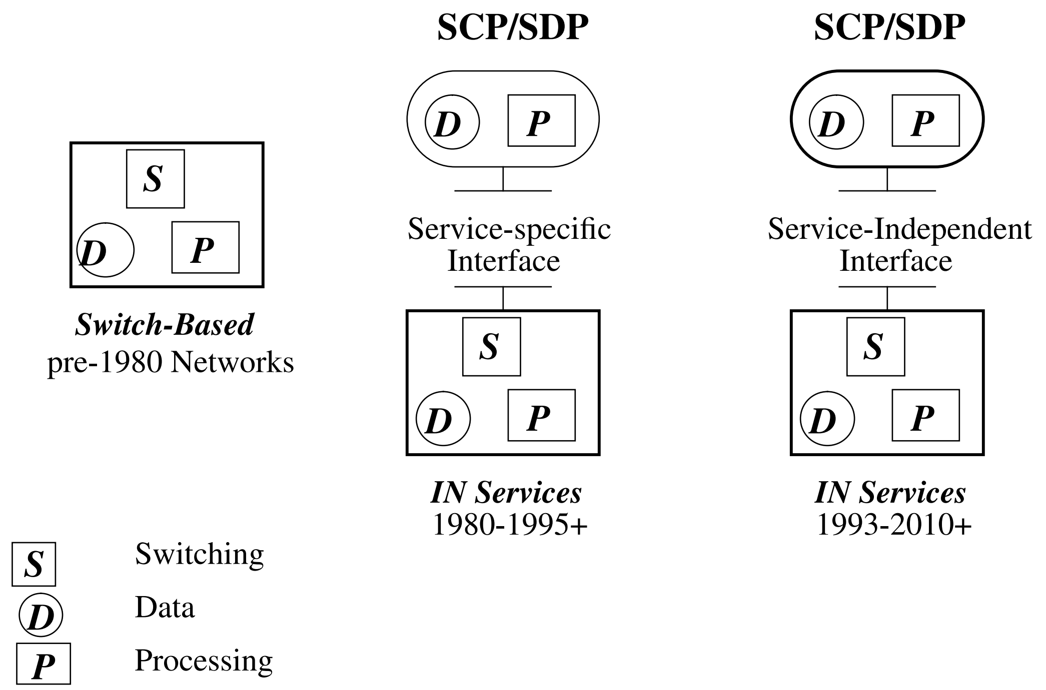


Figure 3: Telephone System Evolution [Vi95]

The next section defines and discusses the first standardized model for IN provided by the **I**nternational **T**elecommunication **U**nion (ITU-T) and the **E**uropean **T**elecommunication **S**tandard **I**nstitute (ETSI).

2.3 Intelligent Network Conceptual Model

In order to encourage better understanding of IN and to enable multi-operating company services, multi-vendor networks and multi-market products, ITU-T has begun to develop recommendations for IN. According to ITU-T objectives, the IN should be applicable to all telecommunication networks, should evolve from existing networks, should enable service providers to define their own services, and should enable network operators to allocate functionality and resources within their network [Keiser95]. The series of recommendations proposed by ITU-T/ ETSI are divided into a number of concepts [Th94], the first ones being Capability Set 1 (CS 1), Capability Set 2 (CS 2), Capability Set 3 (CS3) and so on. The term "Capability Set" refers to the set of services and service features that can be constructed using reusable standard network functions. A number of recommendations, known as Q12xx series, has been proposed for each Capability Set. The Intelligent Network Conceptual Model (INCM) and services are defined in these recommendations. In this thesis, we are interested only in IN CS1 which deals with a subset of services known as *Type A* services and which are *single-ended* which means that they apply to one and only one party in a call, and which have a *single point of control* i.e. the same aspects of a call are influenced by only one SCP at a time.

As mentioned in [Th94], it is important to distinguish between IN architecture and INCM. In fact, the INCM is a framework for the design and description of the IN architecture. It represents an integrated formal framework within which models and concepts used in the standardization of IN are identified, characterized and related. Thus, it is intended to remain consistent.

The INCM proposed by ITU-T/ETSI in [Q1201] consists of four planes. The Service Plane (SP), the Global Functional Plane (GFP), the Distributed Functional Plane (DFP) and the Physical Plane (PP). Each plane represents a different abstract view of the capabilities provided by an IN-structured network (see figure 4). Mappings between adjacent planes establish the correspondence between concepts of the two planes.

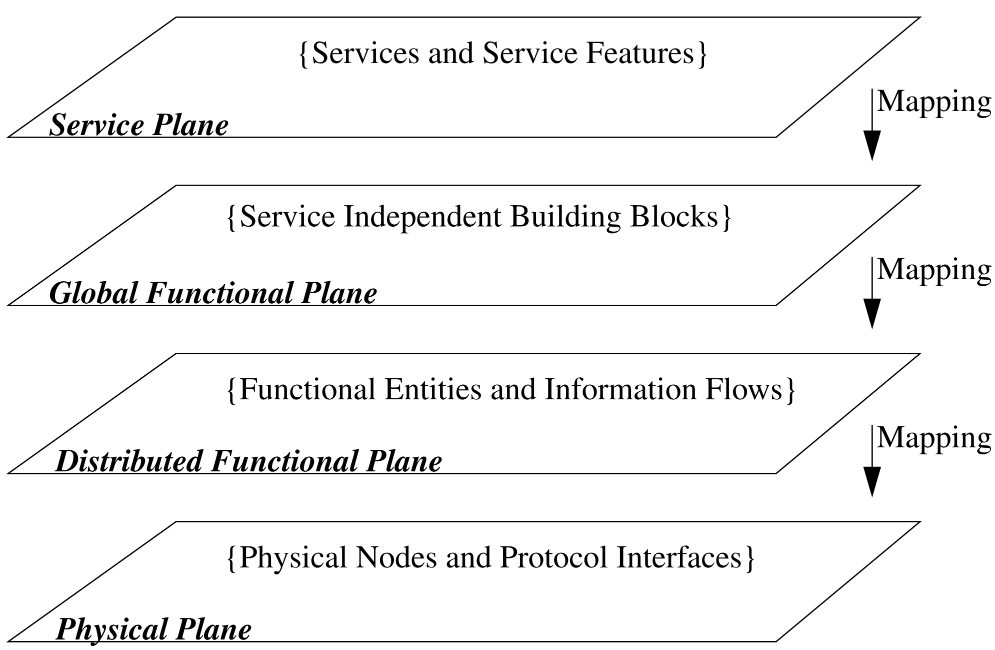


Figure 4: Intelligent Network Conceptual Model

2.3.1 Service Plane (SP)

This plane gives the most abstract view of an IN. It is described in [Q1202] and it provides an exclusively service-oriented view which is implementation-independent, i.e. it does not contain any information about how services are implemented. Only the service behavior is observable.

The only components of this plane are services and service features.

In [Q1211], a service is defined as a stand-alone commercial offering provided to network users. It is composed of service features which are specific aspects of services that can also be used as part of other services.

2.3.2 Global Functional Plane (GFP)

This plane gives a more detailed abstract view of services and service features described in the SP. It is described in [Q1203] and it defines services and service features using Service Independent Building Blocks (SIBs), the Global Service Logic (GSL), The Basic Call Process SIB (BCP) and Points Of Initiation (POI) and Points Of Return (POR) between the BCP and a chain of SIBs, which are described below.

Different elements of the GFP

* <u>SIBs:</u>

As defined in [Q1203], SIBs are standard, reusable network-wide capabilities used to create services and service features. They are used as concepts for service creation, where the service is defined as a combination of SIBs. They are the basic service independent components of services and thus suitable to be reused for different services. They are defined independently of any physical architecture.

A service is viewed in this plane as a combination of SIBs which are chained together in order to provide the functionality of the desired service.

For CS1, ITU-T has defined 13 SIBs in addition to the BCP. These are required to describe the proposed set of services and service features. As example of SIB defined in CS1, we can cite the User Interaction SIB and Translate SIB. The Translate SIB determines an output information from some input information [Vi92]. For example, a dialled number may be translated to a destination directory number. The User Interaction SIB allows the exchange of information between a call party (caller or calling party) and the network.

* <u>BCP:</u>

The BCP is a specialized SIB which identifies at a high level of abstraction all the activities necessary to establish a normal call between parties in the network. It interacts with services (represented through the use of chains of SIBs), at two points which, from the service's point of view, are called:

- Point Of Initiation (POI): It is a point in the BCP at which the service represented by the chain of SIBs is launched.

- Point Of Return (POR): It is the point in the BCP at which call processing should continue after executing the service. Different PORs could exist for one service depending on the logic required to support the service.

The different POIs and PORs which were defined in [Q1203] are described in table 1:

TABLE 1. ***The different POIs and PORs***

POI/POR	**Meaning**
Call originated	At this POI, a user has made a service request without yet specifying a destination address (e.g. off-hook but before dialling).
Address Collected	This POI identifies that the destination address has been received from the user.
Address Analysed	This POI identifies that the destination address has been analysed to determine its characteristics (e.g. Freephone number, long distance number).
Prepared to Complete Call	This POI identifies that the network is ready to attempt completion of the call to the terminating party.
Busy	This POI identifies that the call is designated to a user who is currently busy.
No Answer	This POI identifies that the call has been offered to a user who has not answered.
Call Acceptance	This POI identifies that the call is active but the connection between the calling and called parties is not established (e.g. called party off-hook but no switch-through).
Active State	This POI identifies that the call is active and the connection between the calling and called parties is established.
End of Call	This POI identifies that a call party has disconnected.

POI/POR	Meaning
Initiate Call	This POR identifies that the call should be initiated. This may be independent of an existing call, or may be in the context of an existing call.
Continue with existing data	This POR identifies that the BCP should continue call processing with no modification.
Proceed with new data	This POR identifies that the BCP should proceed call processing with the new call parameters.
Handle as transit	This POR identifies that the BCP should treat the call as if it had just arrived.
Clear Call	This POR identifies that the BCP should clear the call.
Enable call party handling	This POR identifies that the BCP should perform functions to enable call control for individual call parties.

* GSL:

SIBs are service-independent components. A SIB in the chain which represents a service does not have any knowledge of the next SIB in that chain. The only element in the GFP which is service dependent is the Global Service Logic (GSL). It describes how SIBs are chained together and how the chain of SIBs interacts with the BCP to define a service. Then, each service in the SP is defined by a GSL in the GFP.

In this architecture, we no longer need to develop all the components of a service, but we only need to reuse SIBs and define the way they should be combined and chained.Therefore, a drastic amount of time could be gained. It is expected that only few months will be necessary to develop and deploy a new service [Vi95].

When an IN service is to be invoked, its GSL is launched at the POI by a triggering mechanism from the BCP. At the end of the chain, the GSL indicates the corresponding POR to the BCP which depends on the result of processing the service (see figure 5).

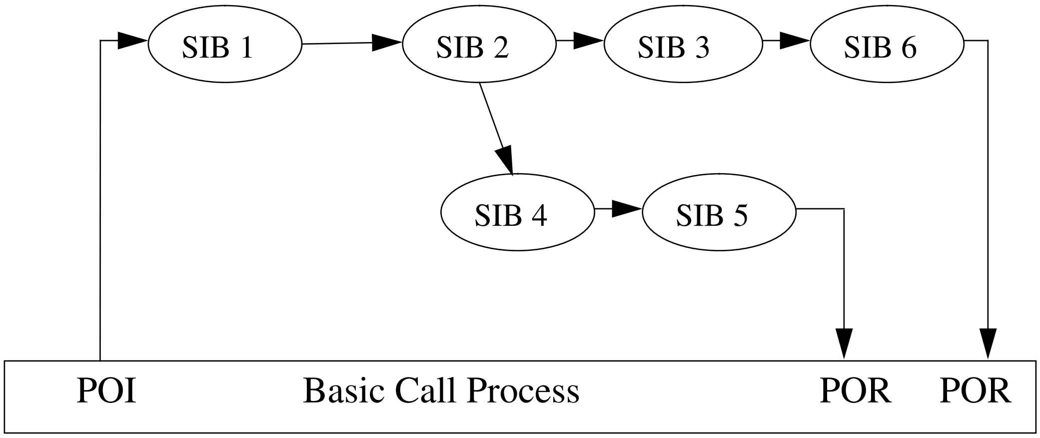


Figure 5: Modelling of the Global Functional Plane

To create a new service, a service designer defines its GSL using the SIBs library. However, some elements which are service dependent are needed to define a service. They can be described using data parameters which enable a SIB to perform the desired functionality. Because the GSL is the only service-dependent element in the GFP, those data parameters are made available to the SIBs through the GSL.

In [Q1203], two types of data parameters that are required for each SIB in order to perform its functionality are defined:

- The Call Instance Data (CID), which describe the dynamic parameters whose values change with each call, such as the dialled number entered by a subscriber or the translated number generated by a SIB.

- The Service Support Data (SSD), which describe the static parameters that are specific to the service and do not change with different calls, such as the File Indicator which identifies the file that should be used by the Translate SIB in order to generate a translated number.

Figure 6 shows a graphical representation of a SIB.

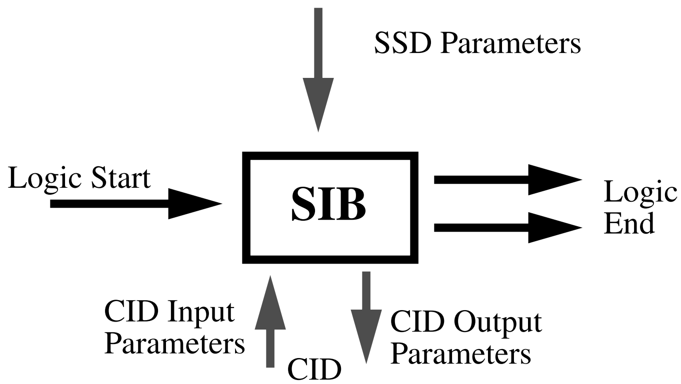


Figure 6: Graphical representation of SIB [Vi95]

2.3.3 Distributed Functional Plane (DFP)

The Distributed Functional Plane (DFP), described in [Q1204], models a distributed view of an IN-structured network. It contains a set of Functional Entities (FEs) which are a unique group of special functions and a part of the total set of functions required to provide a service.

Each FE may perform a variety of Functional Entity Actions (FEAs) and the cooperation between them is achieved by Information Flows (IFs).

Functional Entities

FEs defined in this plane (see figure 7) can be divided into three types of functions:

1- <u>Call Control Related Functions</u>. They include:

- *Call Control Function (CCF)*: It is the function that handles all normal calls by providing the processing and the control of call/connection between network subscribers. It also provides the access to IN and non-IN services for the user.

- *Call Control Agent Function (CCAF)*: It is the interface which provides user access to the CCF in order to establish a call/connection.

- *Service Switching Function (SSF)*: It provides the set of functions required for communication between the CCF and the SCF by managing signaling between them.

- *Specialized Resource Function (SRF)*: It provides the specialized resources required for the execution of IN services such as protocol conversion and speech recognition system.

2 - Service Control Related Functions. They include:

- *Service Control Function (SCF)*: It handles the processing of IN services. It contains all the logic and data required to provide a service. It may interact with other functional entities to access additional logic or data.

- *Service Data Function (SDF)*: It can be defined as a data base which contains all the subscribers information and network data in order to provide them to the SCF when it processes IN-services.

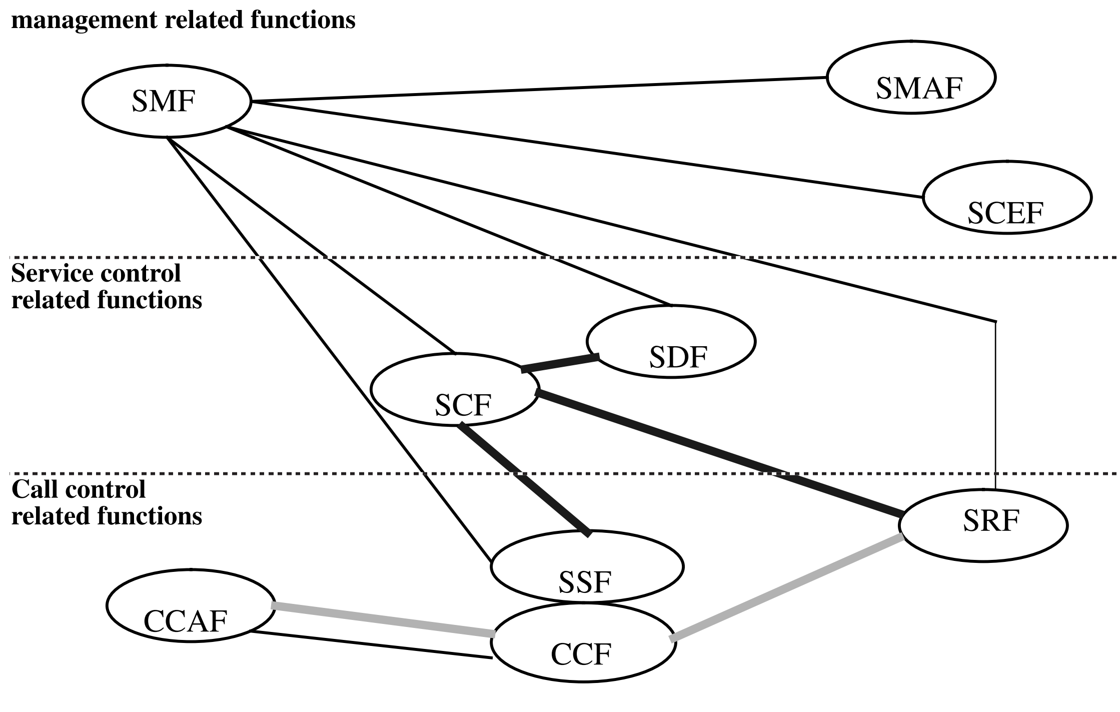
3 - Management Related Functions. They include:

- *Service Management Function (SMF)*: It allows the control of the deployment and provision of IN services provided by the SCEF. It manages, updates and administers service related information in the SRF, SSF and CCF.

- *Service Management Access Function (SMAF)*: It allows service managers to manage services by providing an interface between them and the SMF. As an example, billing and statistic information is received from the SCFs and made available to authorized service managers through the SMAF.

- *Service Creation Environment Function (SCEF)*: This function handles all the operations required for the definition, development and the testing of IN services.

Figure 7 describes the relation between FEs.



SMF: Service Management Function
SMAF: Service Management Access Function
SCEF: Service Creation Environment Function
SDF: Service Data Function
SCF : Service Control Function

SRF: Specialized Resource Function
SSF: Service Switching Function
CCF: Call Control Function
CCAF: Call Control Access Function

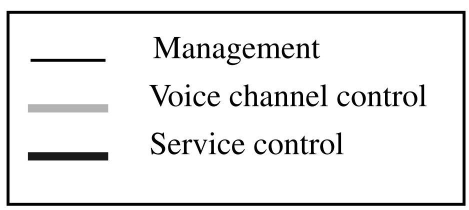


Figure 7: ITU-T/Bellcore IN Functional Architecture

Since there is extensive mutual interaction between the SSF and the CCF, due to the fact that every invocation of the SCF by the CCF in order to execute a service must be done through the SSF, the SSF and the CCF could be treated together as one functional entity which provides the processing and control of a call and which interacts with the SCF when an IN service is invoked and processed.

To establish a call, the user accesses the SSF/CCF via the CCAF. The CCAF receives from the user an indication to setup a call. It passes it to the SSF/CCF for processing. While processing those requests, the SSF/CCF may detect an event (e.g service feature activator) that can lead to the invocation of a service. The service could be an IN or a non-IN service. In the case where an IN service should be involved, the SSF/CCF reports the event to the SCF which invokes the appropriate service, and interacts with the SSF/CCF to provide the IN service to the user. To do so, the SCF can request the SSF/CCF to perform certain call and connection processing functions, and can request the SSF/CCF to make use of resources in the SRF. In addition, it can also request the SDF to perform related service data processing functions.

The Basic Call State Model (BCSM)

It is a high level finite state machine description of the CCF activities required to establish and maintain communication paths for users [Q1204]. It illustrates the different states that a call can go through, from origination (e.g. a user picks up the phone) to termination (e.g. a user hangs up the phone).

It consists of two separate sets of call processing logic. An Originating BCSM (O-BCSM) which models the processing logic supported by the calling party and a Terminating one (T-BCSM) which models the processing logic supported by the called party. Both sides should be active within a given node (see figure 8).

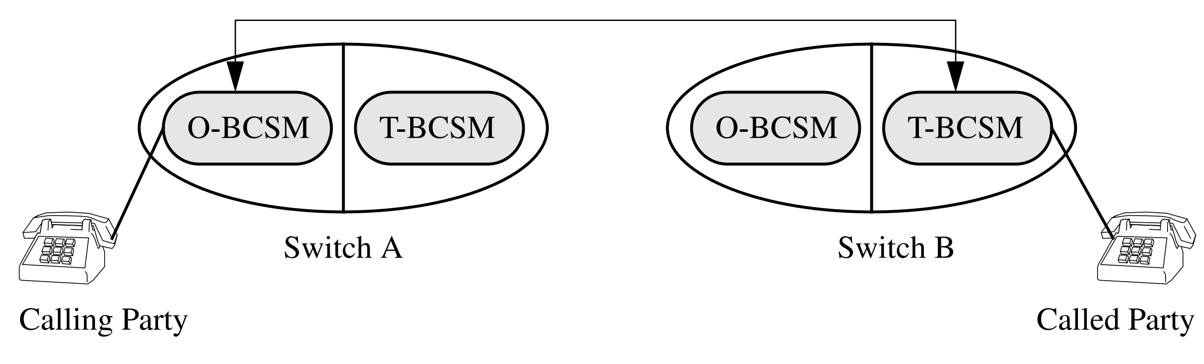


Figure 8: The IN Call Model [Vi95]

The BCSM is composed of a set of Points In Call (PICs), Detection Points (DPs) called also Trigger Check Points (TCPs), transitions and events.

PICs identify CCF activities required to complete one or more basic call transition. DPs indicate points in basic call at which transfer of control from the SSF/CCF to the SCF can occur in order to process an IN service. Transitions indicate the normal flow in basic call from one PIC to another. Events cause transitions into and out of PICs. Figure 9 describes the components of the BCSM.

The BCSM identifies only the states and events that need to be visible to IN service logic. These aspects can be seen by the service logic as the information it receives from the call model. Those are the only aspects that are subject to standardization. Figure 10 and figure 11 describe the O-BCSM and the T-BCSM presented in [Q1204].

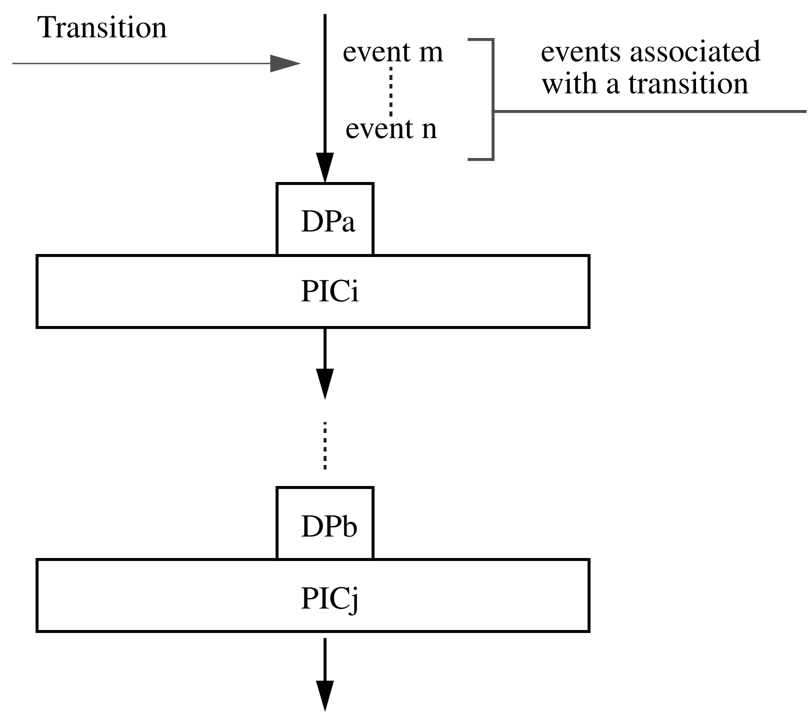


Figure 9: BCSM Components [Q1204]

With each PIC in the BCSM is associated a DP. DPs are points at which specific events are detected and made visible to IN service logic. When a special event is encountered at a DP, call processing may be suspended while waiting for instructions from IN service logic. A call is suspended at a given DP if this DP is armed and a set of trigger conditions are met. A DP can be armed in order to notify that an IN service should be invoked in the SCP at this point of call processing. If a DP is not armed, the SSF/CCF continues call processing without SCF involvement.

DPs are characterized by the following four points:

- *Arming mechanism*: It is the mechanism by which DPs are armed. A DP could be armed statically or dynamically. A DP is statically armed through SMF when a service feature is to be deployed in the network and should be involved at this specific point. It remains armed until explicitly disarmed by the SMF. A DP could be dynamically armed by the SCF within the context of a call associated IN service control relationship (an IN service control relationship is a relationship between the SSF/CCF and the SCF for the purpose of processing a service). It remains armed until detected or until the end of the relationship between the SCF and the SSF/CCF.

- *Criteria*: They define the set of conditions that must be met, in addition to the condition that a DP be armed, in order to notify the SCF that a service must be invoked.

- *Relationship*: When an armed DP is encountered and criteria are met, the SSF may provide an information flow via a relationship. This relationship is considered to be a control relationship if the SCF is capable of influencing call processing via this relationship and it is considered to be a monitor relationship if the SCF is not able to influence the call processing.

- *Call processing suspension*: Given that an armed DP was encountered and DP criteria are met for an IN service control relationship, the SSF/CCF may or may not suspend call processing to allow the SCF to influence subsequent calls. It may provide only notification of an event to the SCF.

If the arming criteria are met at a DP, an information flow is sent from the SSF/CCF to the SCF. The call is suspended and a control relationship is established between the SSF/CCF and the SCF, enabling the SCF to return instructions to the SSF/CCF while executing an IN service.

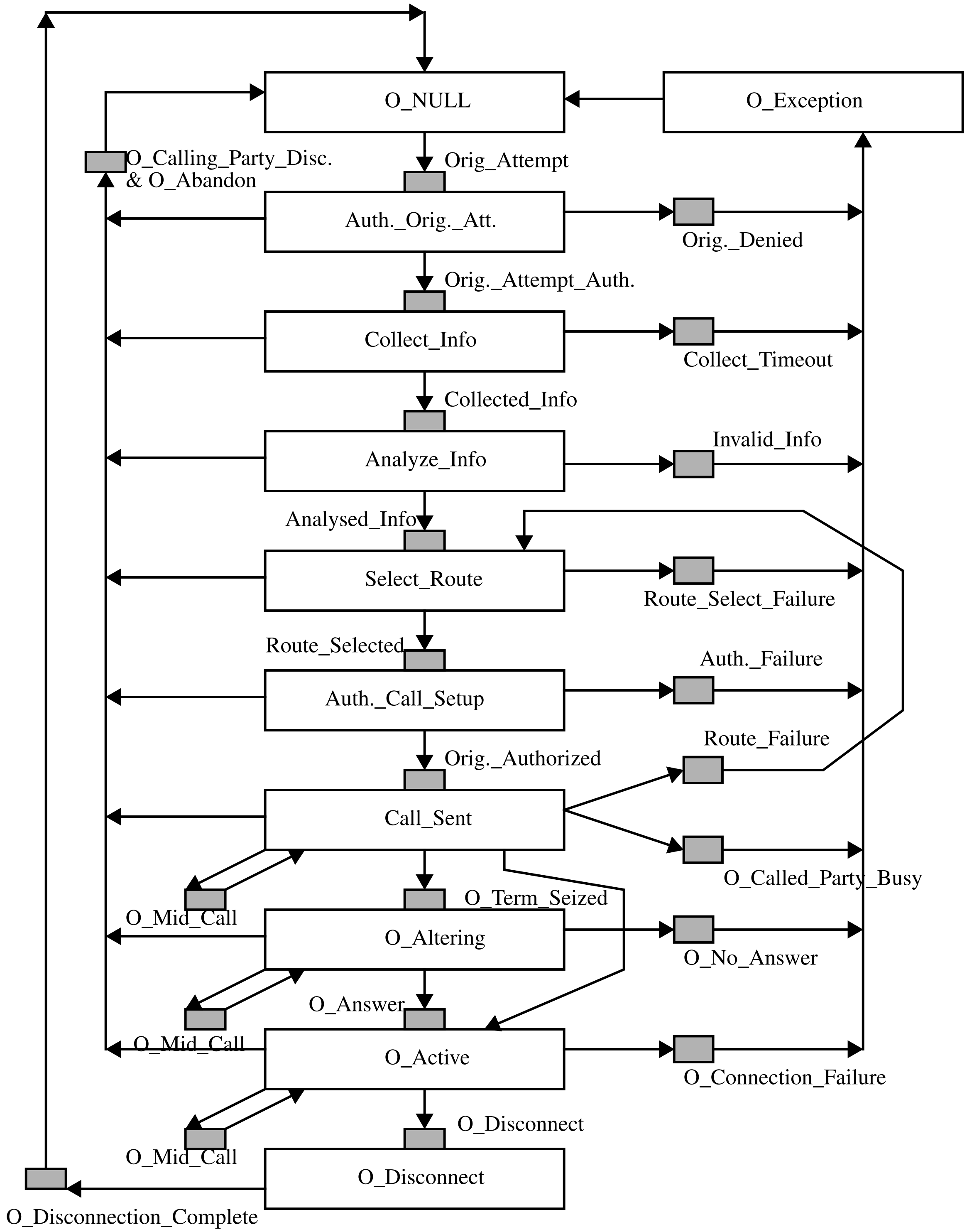


Figure 10: Example of Originating BCSM [Q1204]

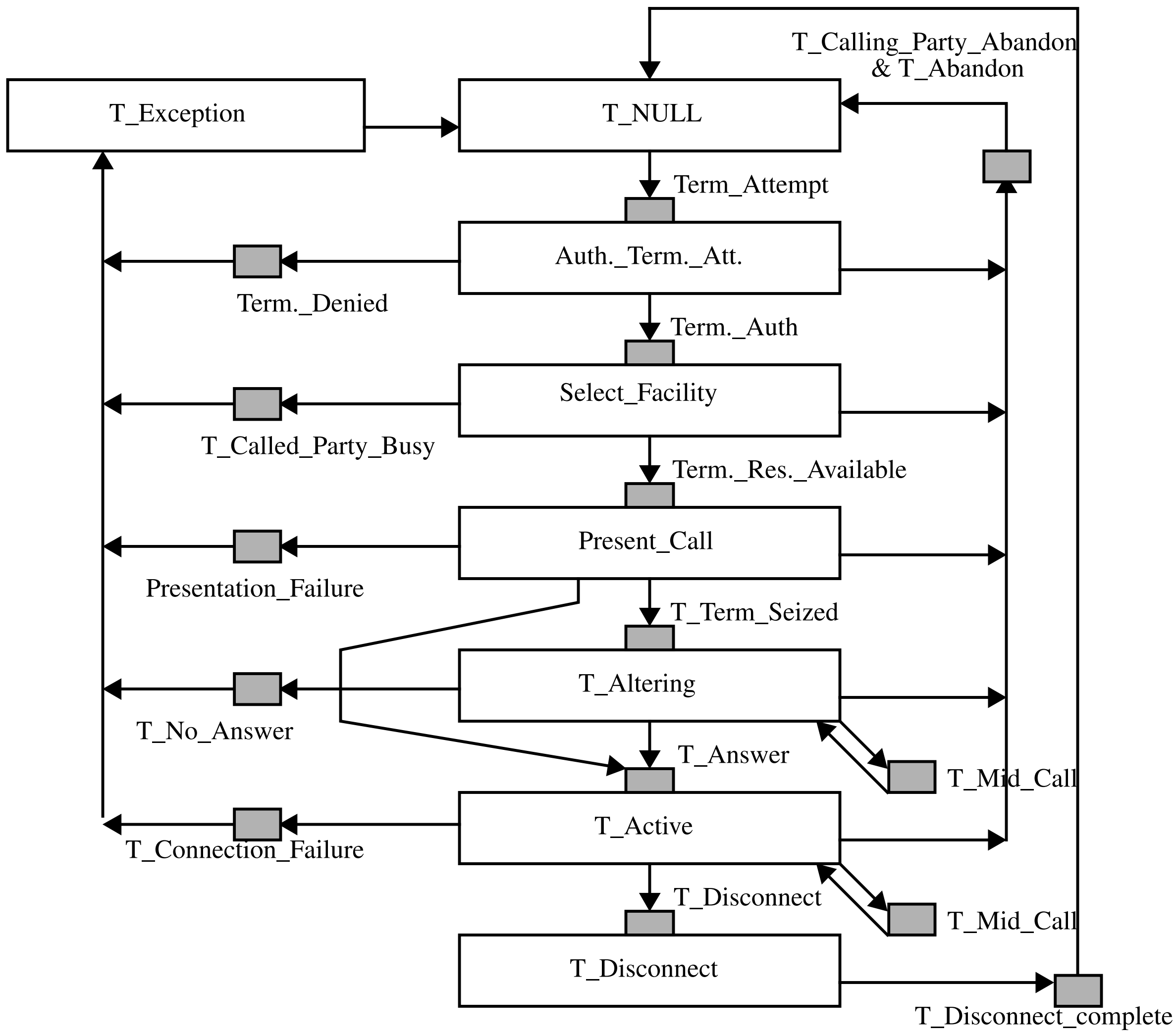


Figure 11: Example of Terminating BCSM [Q1204]

During the processing of the call, trigger condition criteria and arming conditions are checked at each DP. If they are verified, the call is suspended and a request to execute a service is sent from the SSF/CCF to the SCF. The SCF processes the requested service. At the end, it sends a message to the SSF/CCF to indicate at what point the call should continue (Point of Return) with updated call parameters. The return point can be a DP or a PIC. During the processing of the service, the SCF may exchange information with the SSF/CCF, the SDF or the SRF.

Figure 12 describes the scenario of processing an IN service.

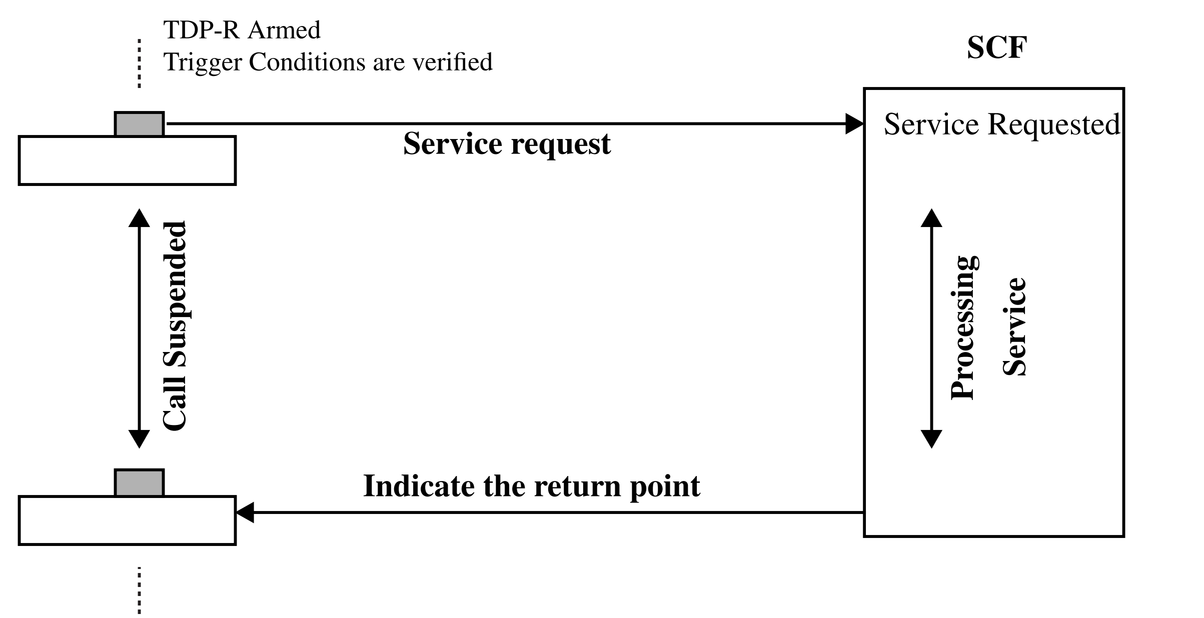


Figure 12: IN service processing scenario

2.3.4 Physical Plane (PP)

The Physical Plane (PP) models the physical aspects of IN-structured networks. It identifies different physical entities (PEs) that may exist and describes the protocols that should be used for communication between those PEs. It also describes the mapping from the DFP to the PP by indicating which functional entities are implemented in which physical entities [Q1205].

Physical Entities

The following PEs are defined in [Q1205], and they describe a selection of PEs which could support the general concepts of IN. They include:

1 - Service Switching Point (SSP):

This node contains the SSF and the CCF. In addition to the switching and call control functionalities it performs, this node makes available to subscribers the set of IN capabilities. It recognizes IN-based services, and communicates with the PE containing the SCF (usually the SCP). If the SSP is a local exchange to which users are connected, it should perform also the functionality of the CCAF (which defines the interface between the user and the SSF/CCF). It also may optionally contain the SRF or the SDF.

2 - Service Control Point (SCP):

This node contains the SCF. All the Service Logic Programs (SLPs) that are used to provide IN services, and optionally customer data are stored in this PE.

3 - Service Data Point (SDP):

The SDP contains data used by SLPs to provide individualized services. Functionally, it contains the SDF. It can be accessed directly by the SCP when some data are needed to provide a service, or by the PE containing the SMF when some data need to be updated within the network.

4 - Service Management Point (SMP):

This node contains the SMF. It is used to perform all the control operations required to deploy, manage and maintain an IN service. Such operations could be data base administration, network data collection, network traffic management and so on. It may optionally contain the SCEF or the SMAF.

5 - Service Management Access Point (SMAP):

This node contains the SMAF and provides access to the SMP to some selected users such as service managers or specially selected customers.

6 - Service Creation Environment Point (SCEP):

The SCEP contains the SCEF and it is used to define, develop and test IN services before their deployment within the network.

7 - Intelligent Peripheral (IP):

This node contains the SRF. It provides special resources for customization of services. It can be used for voice recognition, text to speech synthesis, protocol conversion, customized messages and so on. It could be connected to one or more SCPs or to the signalling network. It might also be integrated directly in the SSP.

8 - Adjunct (AD):

The AD is a PE similar to the SSP (i.e. it contains the same FEs) but it is directly connected to an SSP using a high-speed interface.

9 - Service Node (SN):

This node is similar to an SCP and an AD with the only difference that it has a direct point-to-point connection with the SSP for speech and signalling.

2.4 Mapping between different planes

2.4.1 Mapping from the SP to the GFP

The basic elements of the SP are services and service features. They are realized in the GFP using SIBs which are chained together and interact with the BCP via POIs and PORs. This has been described in section 2.3.2.

2.4.2 Mapping from the GFP to the DFP

The mapping from the GFP to the DFP consists essentially in the mapping of SIBs to FEs and the BCP to the BCSM.

Mapping of SIBs to FEs:

The basic components of the GFP are SIBs. Each SIB is realized in the DFP by a sequence of specific Functional Entity Actions (FEAs) performed by the FEs. Some of these FEAs result in Information Flows (IF) between FEs.

Consider for instance the Screen SIB defined in CS1, which performs a comparison of an identifier against a list to determine whether the identifier has been found in the list [Q1213]. This SIB can be represented in the DFP as a client/server relationship between the SCF and the SDF in which the SCF sends a request to the SDF in order to verify whether the identifier exists in the list, and the SDF sends back a response after consulting the appropriate list (see figure 13).

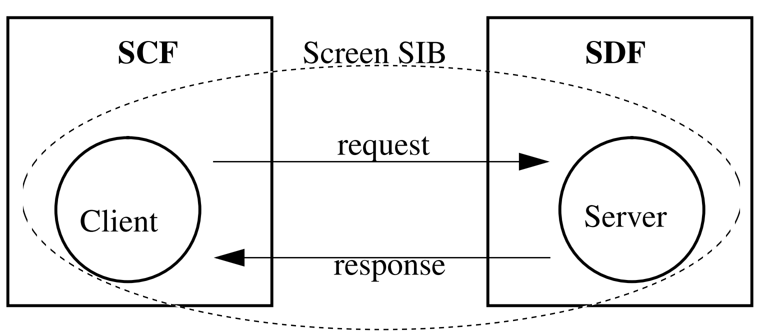


Figure 13: Screen SIB representation in the DFP

The GSL which defines IN services at the GFP using SIBs is represented in the DFP by the Global Service Logic Program (GSLP) which manipulates SIBs via their standardized interfaces.

Mapping of the BCP to the BCSM:

The different POIs and PORs defined in the BCP (table 1) are mapped onto DPs and PICs in the DFP. Table 2 gives insight into the mapping, but a precise mapping may only be determined by the IN services. In fact, some POIs and PORs could have different mappings depending on the IN service.

TABLE 2. Mapping of POIs and PORs to DPs and PICs [Q1214]

POI/ POR	DP	Meaning of the DP/PIC
POI: Call Originated	Orig_Attempt_Authorized	An attempt to make a call is authorized (e.g. off-hook followed by tone which indicates the ability to make a call)
POI: Address Collected	Collected_Info	Initial information is available (e.g. service code, dialled address, prefixes...)
POI: Address Analysed	Analysed_Info	Information has been analysed according to the dialing plan to determine routing address and call type.
POI: Prepared to complete call	Term_Attempt_Authorized	Terminating half has authorized the attempt for making the call
POI: Busy	O_Called_Party_Busy	Indication from the terminating BCSM half that the terminating party is busy
	T_Called_Party_Busy	Terminating party is busy
	Route_Select_Failure	The call cannot be completed because the network is busy
POI: No Answer	O_No_Answer	Indication from the terminating BCSM half that the terminating party did not answer within specified time period
	T_No_Answer	Terminating party does not answer
POI: End of Call	O_Abandon	Originating party abandon call
	T_Abandon	Terminating party abandon call
	O_Disconnect	Disconnect indication is received from Originating party
	T_Disconnect	Disconnect indication is received from Terminating party
POI: Active State	O_Answer	An indication is sent to the O-BCSM that the terminating party has answered the call
	T_Answer	Terminating party has answered the call
	O_Mid_Call	A service request is received from the originating party
	T_Mid_Call	A service request is received from the terminating party

TABLE 2. Mapping of POIs and PORs to DPs and PICs [Q1214]

POI/ POR	DP	Meaning of the DP/PIC
POR: Continue with existing data	Several PICs are possible	Return to the PIC associated to the same DP
POR: Proceed with new data	Several PICs are possible	Return to the PIC associated to the same DP. (Only data have been modified)
POR: Handle as transit	Analyse_Info	Return to the PIC Analyse_Info: analyse and translate information according to the dialing plan.
	O_Altering	Return to the PIC O_Altering: continue the processing of call setup (e.g ringing)
POR: Clear call	O_Null	Return to the PIC O_Null: The calling party is idled
	T_Null	Return to the PIC T_Null: The called party is idled
POR: Enable call party handling	Several	Return to the same DP/PIC
POR: Initiate BCP	Analyze_Info	Return to the PIC Analyse_Info
	O_Altering	Return to the PIC O_Altering

2.4.3 Mapping from the DFP to the PP

The FEs described in the DFP could be mapped onto the PEs in the PP. One or more FEs may be mapped onto the same PE, however, one FE cannot be split between two PEs which means that an FE is mapped entirely within a single PE [Q1205].

Different mappings from functional entities to physical entities are possible. Figure 14 shows one possible mapping from DFP to PP.

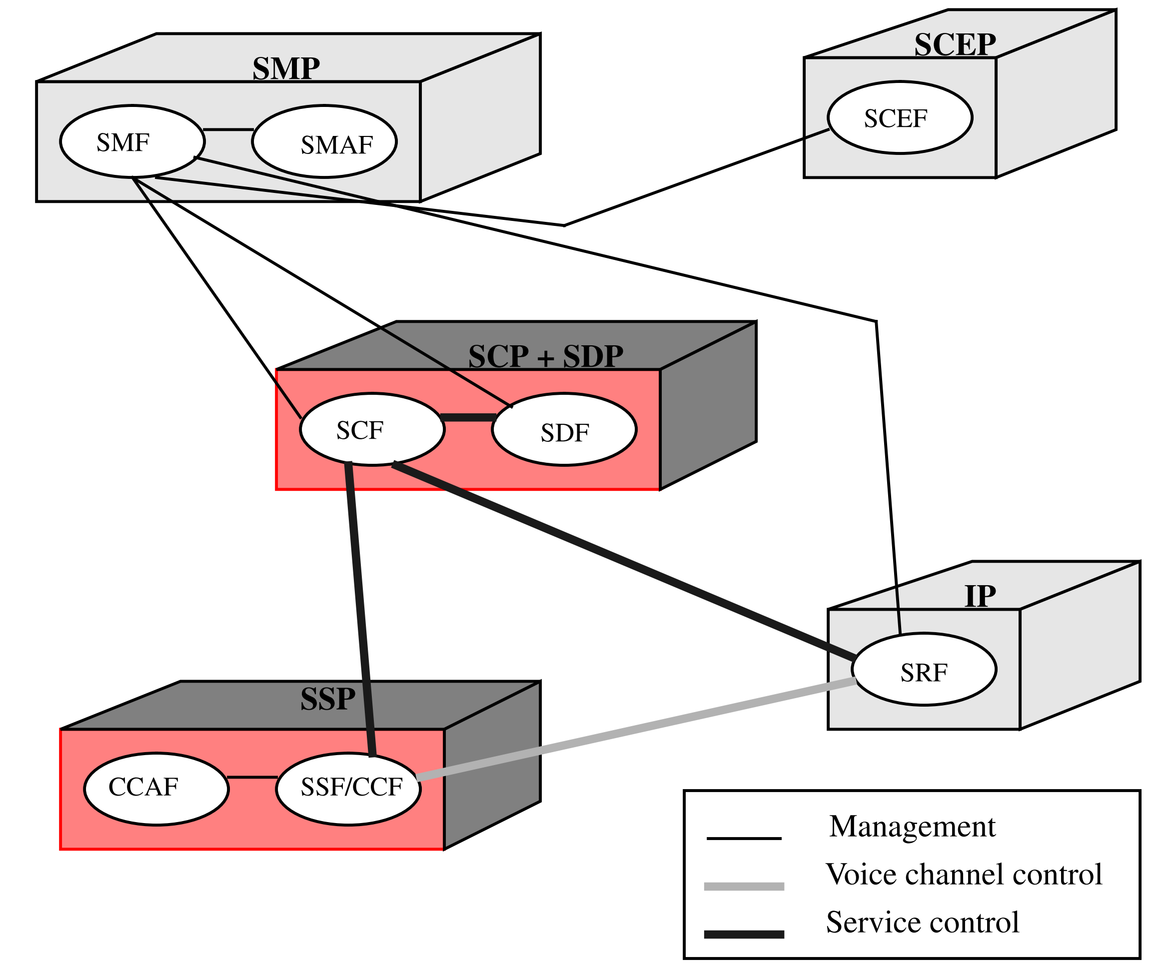


Figure 14:   An example of physical architecture [Vi95]

In this figure, the SCF which handles the processing of services and the SDF, which provides the data required by the IN services, are mapped onto the same physical entity. In fact, most of the IN services need data to be processed and the availability of the data in the same physical entity make easier the access to it. However, if the data are stored into another physical entity, a communication protocol must be defined between the SCP and the SDP.

In addition, the SMAF and the SMF are mapped onto the same physical entity, and the same thing is for the CCAF and the SSF/CCF. In fact both the SMAF and the CCAF are functional entities which provide respectively an interface for the user to access to the SMF and the SSF/CCF.

Figure 15 summarizes the different planes and functions of the INCM. The arrows outline the mappings, although many of these cannot be made precise in such a general figure.

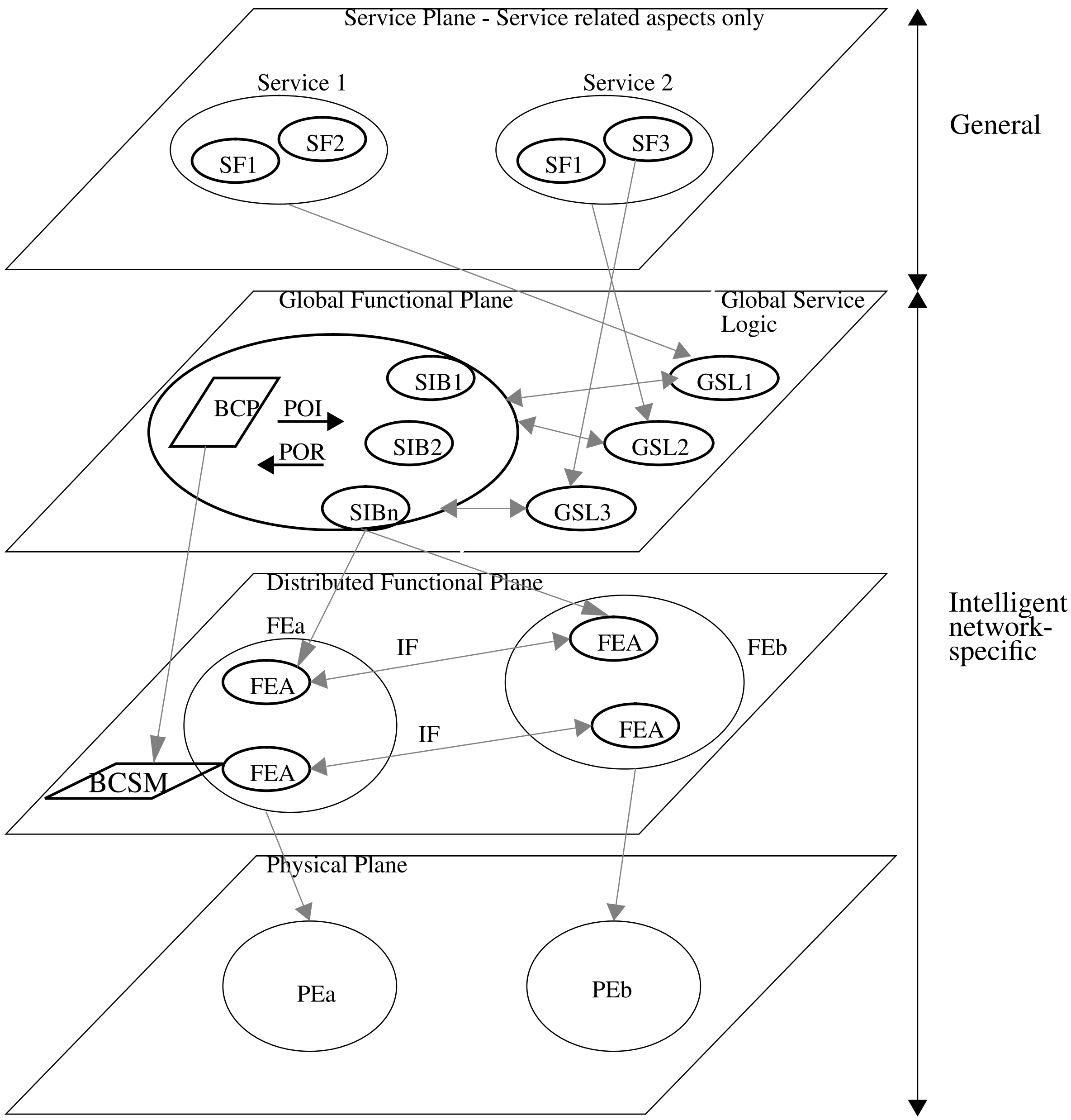


Figure 15: The INCM. [Q1201, Figure 3.1/a]

2.5 Chapter Summary

In this chapter, we have presented an overview of Intelligent Network (IN) concepts and architectural model by describing the essential elements of IN and the different planes of the Intelligent Network Conceptual Model (INCM) defined in ITU-T/ETSI Capability Set 1 (CS1). These different planes are the Service Plane (SP), the Global Functioanl Plane (GFP), the Distributed Functional Plane (DFP) and the Physical Plane (PP). Each plane gives an abstract view of the capabilities provided by an IN-structured network and mappings between adjacent planes establish the correspondence between concepts of the two planes.

CHAPTER 3 Using LOTOS for specifying Intelligent Networks

In this chapter, we give an overview of the LOTOS specification language and its main operators by describing some examples in the context of telephony and Intelligent Networks.

3.1 Introduction

LOTOS (Language Of Temporal Ordering Specification) is a Formal Description Technique (FDT) developed within ISO (International Organization for Standardization) as a formal specification language for the purpose of describing and specifying the different elements of OSI (Open System Interconnection) architecture such as services and protocols. It has been an ISO standard (8807) since 1989 [ISO 8807]. Nowaday, LOTOS applications have been extended to cover some other domains such as hardware [FaL93] and telephony [FaLS91], [SL93].

A LOTOS specification consists of two components. The first one is the *control* component in which the external observable behavior of the system is described. It is based on Milner's Calculus of Communicating Systems (CCS) [Mil80] and Hoare's (CSP) [Hoar85]. The second one is the *data* component, which defines all the data types and value expressions needed to specify the behavior of a system. It is based on the formal theory of algebraic abstract data types ACT-ONE [EM85].

A number of excellent LOTOS tutorials exist in the literature [LoFH92] [To89], therefore, we limit ourselves to a very brief overview of the language and of its use in the context of our research.

All the LOTOS reserved words used in this thesis are written in **bold.**

3.2 LOTOS Data Types

One of the reasons why LOTOS has adopted the algebraic abstract data type language ACT-ONE for defining data types was to abstract the definition of data from the implementation details. The properties and operations of data are defined without any indication about how those data are represented and manipulated in memory.

A data type definition in LOTOS consists of a definition of a *signature* and possibly of a list of *equation*s. A signature of a type is a definition of its *sorts* and *operations*. It includes the domains and ranges of the operations. Equations provide a means to define the semantics of operations.

Consider the following type definition of the network addresses:

```
type Network_Address is Boolean, NaturalNumber
sorts network_address
opns null, adr1, adr2, adr3, adr4     : -> network_address
        _eq_                         : network_address, network_address -> Bool
        _ne_                         : network_address, network_address -> Bool
        to_nat                       : network_address -> Nat
eqns    forall ad1, ad2 : network_address
        ofsort Nat
                to_nat(null) = 0 ;
                to_nat(adr1) = Succ(to_nat(null));
                to_nat(adr2) = Succ(to_nat(adr1));
                to_nat(adr3) = Succ(to_nat(adr2));
                to_nat(adr4) = Succ(to_nat(adr3));
          ofsort Bool
                ad1 eq ad2 = to_nat(ad1) eq to_nat(ad2);
                ad1 ne ad2 = to_nat(ad1) ne to_nat(ad2);
endtype
```

The signature of the type *Network_Address* includes the sort *network_address* and the operations *null, adr1, adr2, adr3, adr4*, *eq*, *ne* and *to_nat*. The operations *null, adr1, adr2, adr3* and *adr4* result in five elements of sort *network_address*, *eq* and *ne* define respectively the equality and inequality between variables of this type, and *to_nat* is the operation which maps a variable into a natural number (the type defining natural numbers is NaturalNumber and is already defined in the abstract data type library). This mapping is used to define the semantics of *eq* and *ne* operations, as described below.

The first five equations define the images of the defined variables, *null, adr1, adr2, adr3* and *adr4* by the operation *to_nat*, then, to each variable of *sort network_address* corresponds a value of *sort Nat* (defined in the type NaturalNumber). The equality and non equality between two variables of *sort network_address* are defined by the last two equations. Two variables of *sort network_address* are equal, respectively not equal, if their images by the operation *to_nat* are equal, respectively not equal. In fact, the operations *eq* and *ne* are already defined in the type NaturalNumber.

3.3 The Control Component

The control component is the part of the specification which deals with the description of the system behavior. In this part, systems are described by means of processes defined in a top down hierarchy.

A process is viewed as a black box interacting with other processes or with the system environment via synchronization on its observable gates. It is basically defined by a set of observable gates, on which synchronizations occur, and by a behavior expression. A behavior expression is built by combining LOTOS actions by means of operators and possibly instantiations of other processes.

The syntax of a process definition is of the form:

process *process_name [gate_list] (formal_parameter_list): functionality*

<behavior expression>

endproc

In addition to the set of observable gates and the behavior expression, a process can also have a set of parameters, denoted in the definition above by *data_parameter_list*. This set represents the set of parameters needed for expressing the behavior of the process. The parameterization of a process enables its reusability.

An action is the basic element of the behavior expression. It consists of a gate name, a list (possibly empty) of events, and possibly a predicate which defines the conditions that should hold for the event to be offered. An event can either *offer* (**!**) or *accept* (**?**) a value. Predicates establish a condition on the values that can be accepted or offered.

An example of an action is:

gate	*event1*	*event2*	*Optional Predicate*
g	**?**Get: Type	**!**Put	**[**Get <> 0**]**

As another example, consider the following action:

N **!**OffHookToCall **?**caller: network_address [caller NotIn BusyList]

where *OffHookToCall* is a parameter of sort *Signal* indicating the type of action performed by a user, *BusyList* is of sort *List* defining a list of busy users and *NotIn* is an operation which specifies the fact that an element is not in a list of users. This action occurs at gate *N* and expects from the environment a value for *caller* of sort *network_address* restricted not to be in the *BusyList*, while at the same time offering the value *OffHookToCall* which specifies the fact that the user having address *caller* picked up the phone in order to make a call.

Actions are considered to be atomic in the sense that they occur instantaneously, without consuming time.Two types of actions exist in LOTOS. There are internal actions, that a process can execute independently, are unobservable to the environment and are represented by the internal action ***i***; and there are actions that need to synchronize with the environment in order to be executed. The environment of a process consists of other processes, or some external world that can be a human observer. If some processes synchronize at an action, they participate in its execution at the same time.

3.4 The main LOTOS Constructors

3.4.1 Basic Behavior Expressions

Inaction: **stop**

It represents a deadlock, i.e. nothing is offered to the environment.

Successful Termination: **exit**

Indicates that a process has successfully performed all its actions.

The keyword **exit** can also be used in the process definition to express the process functionality (denoted in the syntax given above by *functionality*). In fact, a process has functionality **exit** if it can terminate successfully, i.e. it is able to perform an **exit** at the end. If the process cannot perform an **exit**, the functionality is **noexit**.

Process Instantiation: Process_Name [g_1, ...,g_n](actual_parameter_list)

The instantiation of a LOTOS process is equivalent to the invocation of a procedure in a programming language (such as Pascal).

It occurs in the behavior expression of other processes or in the behavior expression of the process itself, thus allowing recursion.

3.4.2 Basic Operators

Action prefix: **;**

The *action prefix* operator, written as a semi-colon (**;**), expresses sequential composition of an action with a behavior expression.

For example, when a user picks up the phone to make a call, she/he will get a tone. This can be expressed by a behavior expression composed of two actions synchronizing at the gate *N* and offering two values of type *Signals* which are *OffHookToCall* and *GetTone,* as:

N **!** *caller* **!** *OffHookToCall* **;** *N* **!** *caller* **!** *GetTone***;** *...*

<u>Choice</u>: B_1 **[]** B_2

The *choice* operator **[]** denotes the choice between two or more alternative behaviors.

For example, when a user picks up the phone to make a call and gets a tone he/she can either dial a number and continue the processing of a call or hang up. This can be expressed by the behavior expression:

N **!** *caller* **!** *OffHookToCall* **;** *N* **!** *caller* **!** *GetTone* **;**

(

N **!** *caller* **!** *Dials* **?** *num: dialled_number* **;** *...*

[]

N **!** *caller* **!** *HangsUp* **; stop**

)

3.4.3 Enabling and Disabling Operators

<u>Enabling</u>: B_1 **>>** B_2

The *enable* operator **>>** has a similar function as the *action prefix* operator but is used to express sequential composition of two behavior expressions. B_1 has to terminate successfully (**exit**) in order for B_2 to be executed.

<u>Disabling</u>: B_1 **[>** B_2

The *disable* operator **[>** is used to express situations where B_1 can be interrupted by B_2 during normal functioning. For example, a normal processing of a call could be interrupted at any point if the caller hangs up. This could be expressed by the behavior expression:

(N **!** *Caller* **!** *GetTone* **;**

N **!** *Caller* **!** *Dial* **?** *num: dialled_number* **;** *N* **!** *Caller* **!** *GetConnection* **;**

N **!** *Caller* **!** *Talk* **;** *N* **!** *Caller* **!** *HangsUp* **; stop***)* **[>** *N* **!** *caller* **!** *HangsUp* **; stop**

<u>Accept</u>: *B1* **>>** ***accept*** X_1: S_1, ...,X_n: S_n ***in*** B_2

The **accept** operator is used with the *enable* operator (**>>**) in order to express sequential composition of behavior expressions with value passing.

A process which successfully terminates can associate with the action **exit** it performs a set of parameters. This is denoted by **exit**(S_1, ...,S_n) where S_1,..., S_n define the sorts of the parameters. These parameters are passed to the enabled behavior if the process enables another behavior expression.

Let us take as example the freephone service. This service allows a reverse charging, the subscriber accepting to receive calls at his expenses and being charged for the whole cost of the call. Then, when a user dials a number in the form a freephone number, the dialled number will be translated to another number representing the network address of the subscriber (the called party). This service uses the *Translate* SIB defined in CS1 in order to translate the dialled number. The *Translate* SIB takes as input parameter the dialled number and gives as output the translated number. This output will be passed to the process defining the freephone service using the *enable* and **accept** operators. This can be expressed in LOTOS as follow:

```
process Translate_SIB (dn: dialled_number) :exit (dialled_number) :=
   (
      exit(get_translated_number(dn))
   )
endproc
```

where *dialled_number* is a sort defining the numbers dialled by users when making calls, and *get_translated_number* is an operation defined in the abstract data type and which gives the translated number for every freephone number. The process *freephone_service,* which uses the *Translate* SIB, is defined as:

```
process freephone_service (dn: dialled_number) :exit (dialled_number) :=
    Translate_SIB (dn) >> accept new_number: dialled_number in exit(new_number)
endproc
```

3.4.4 Composition Operators

<u>*Interleaving*</u>: B_1 **|||** B_2

The fact that B_1 and B_2 are in *interleave* means that they can perform their actions independently of each other. It expresses the concept of parallelism between behaviors where no synchronization is required. For example, users in the network behave independently of each other. Each user is represented by a process *USER* having as parameter its network address. Then, if we define *n* network addresses in the specification, the behavior expression specifying the *n* users in the network is:

USER [list_of_gates] (network_address_1)

|||

...

|||

USER [list_of_gates] (network_address_n)

<u>*Parallel Composition*</u>: B_1 **|**[g_1, ..., g_n]**|** B_2

The *parallel composition* of B_1 and B_2 on the gate list g_1, ..., g_n expresses the fact that B_1 and B_2 behave independently, with the exception that they must synchronize on the gates g_1, ..., g_n, which means that processes B1 and B2 must participate in the execution of every action defined with a gate name g_i, $i \in \{1,..,n\}$. Then interleaving can be defined as parallel composition on an empty gate list.

Synchronization of processes on a gate g_i, $i \in \{1,..,n\}$ occurs, if each process provides an action with a gate name g_i, the list of events offered with the actions match, and the predicates (if any) are satisfied. To better understand this, let us suppose that an action with a gate name g_1 is defined in process B1 as: $\boldsymbol{g_1\ E_1 \ldots E_n}$ where E_i can be an offer or an acceptance of a value, and that an action with the same gate name g_1 is defined in process B2 as: $\boldsymbol{g_1\ E'_1 \ldots E'_m}$,then B1 and B2 synchronize on g1 if and only if:

- $n = m$,

- the parameter value offered/accepted within the event E_i is of the same sort as the parameter value offered/accepted within the event E'_i, and

- if events E_i and E'_i consist of offers (**!**) of two values, respectively x_i and x'_i of the same sort (i.e. $E_i = \mathbf{!}\ x_i$ and $E'_i = \mathbf{!}\ x'_i$), then $x_i = x'_i$.

If events E_i and E'_i consist respectively of an offer (**!**) of a value x_i and an accept (**?**) of a value x'_i, (i.e. $E_i = \mathbf{!}\ x_i$ and $E'_i = \mathbf{?}\ x'_i$*: sort*$_i$), then the parameter x'_i will get the value of x_i after synchronization, which defines a way of exchanging values between processes.

For example, consider two processes *USERS* and *NETWORK* which synchronize on a gate *N* and are described as:

```
Process USERS [N](user: network_address): noexit :=
(
  ...
  N !user !OffHookToCall;
  ...
) endproc

Process NETWORK[N]: noexit :=
(
  ...
  N ?caller: network_address !OffHookToCall;
  ...
) endproc
```

At synchronization on gate *N*, the variable *caller* defined in the process *NETWORK* gets the value of *user* defined in process *USERS.*

Full Synchronization: $B_1 \| B_2$

The full synchronization of B_1 and B_2 is a parallel composition in which B_1 and B_2 must synchronize on all their gates.

3.4.5 Hiding Operator

Hiding: **hide** g_1, ..., g_n **in** B

Used to hide actions synchronizing on gates (g_1, ..., g_n), which become internal (i.e. they become **i**) for the environment. Thus, these actions cannot synchronize with the environment.

3.4.6 Guarded Behavior

Guarded Behavior: **[**P**]** **->** B

The behavior expression B can be executed if and only if the formula P is true, it is equal to **stop** otherwise. For example, a telephone can ring at a called side only if the called is not busy. This could be expressed by the behavior expression:

```
[called NotIn BusyLisy] ->
        N !called !RingsFrom !caller;
```

Where *RingsFrom is* a constant of sort *Signals*.

CHAPTER 4 Specifying Intelligent Network Call Model and Services in LOTOS

In this chapter, we describe the LOTOS formal specification of the IN call model and services as defined in the Distributed Functional Plane (DFP) of the Intelligent Network Conceptual Model (INCM).

4.1 Introduction

Our main objective in specifying the IN call model and services in LOTOS is to provide a specification that can be used as a testbed for specifying, validating and detecting feature interaction between services. Only the external behavior of the system describing call/connection establishment and service activation is of interest. Therefore, the formal specification is limited to the parts of DFP that are involved when a call is being processed and IN services are activated.

The management related functions are not specified since they are not involved in the service processing stage of the service lifecycle. In addition, functions that are related to implementation and deployment of services are not taken into account. Such functions are the SRF which provides specialized resources such as protocol conversion, and the CCAF which defines the interface that provides to network users access to the CCF. The only functions that are relevant to our specification are SCF, CCF, SSF and SDF. Figure 16 describes the part of the DFP subject to formal specification.

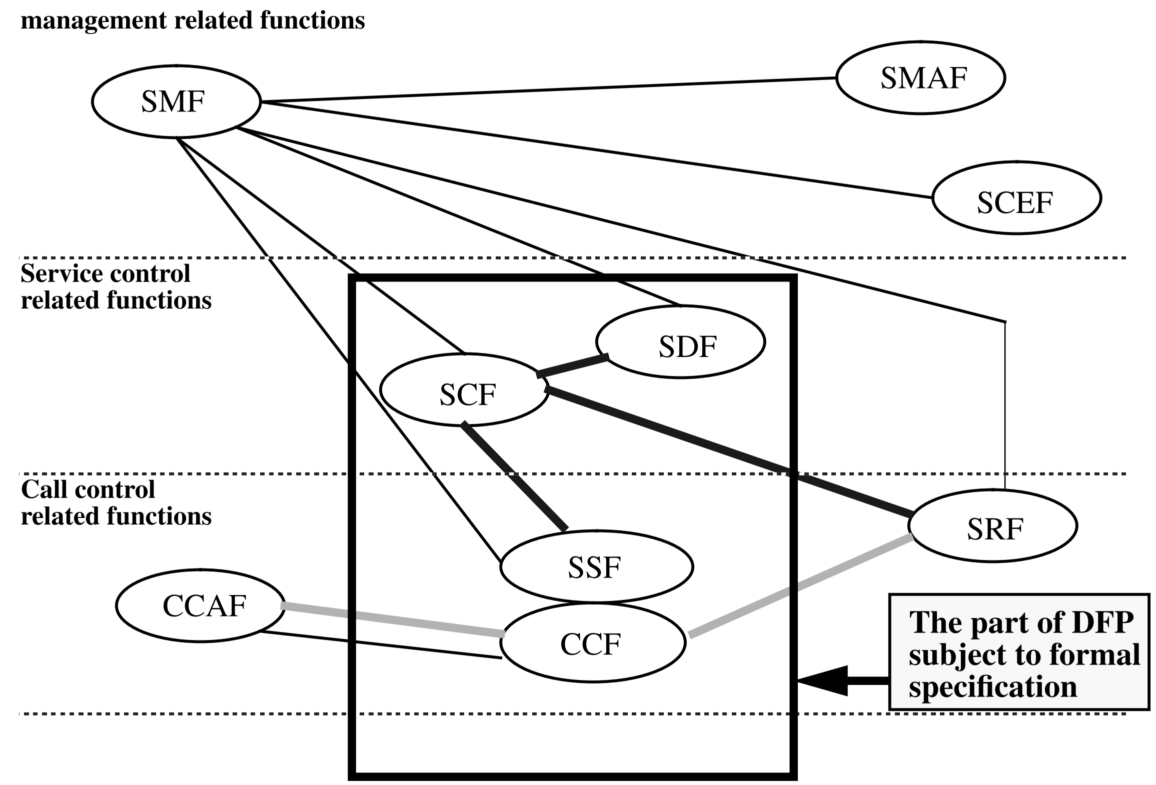


Figure 16: Part of the DFP

As described in Chapter 2, the main objective of IN is to allow rapid introduction and addition of new services from design to deployment, and to provide a service-independent implementation in the sense that every service provider will be able to define its own services independently and deploy them in the network. The formal specification should be designed with these tasks in mind. The specification should enable incremental specification and rapid implementation of services, i.e. each new service must be able to be specified independently and then added easily to the global specification without major modifications.

In addition, the specification should be flexible in the sense that it could be easily extended for call models and services defined in CS2, CS3...

In order to provide such characteristics, we used in our specification, a mixture of the *resource oriented style* and *state oriented style* described in [ViSV88]. With the *resource oriented style* the observable behavior of a system is described as a composition of separate resources whose functionalities are well-defined, and these resources may be specified using any style. This style has been used to preserve the architectural model of the system at the specification level and which is composed of different resources collaborating together such as CCF, SSF, SCF and SDF. In fact, the IN architectural model is designed to provide rapid addition and introduction of services and by preserving this model at the specification stage, we can extend its characteristics at this stage and then provide incremental specification of services in an independent way.

With the *state oriented style*, the system is seen as a single resource whose internal state space is explicitly defined. This style is very appropriate for the specification of the BCSM, which is a finite state machine.

In addition, in order to achieve an open, extensible and modular specification, some general design principles for open distributed systems have been respected. These are mainly generality and open endedness. Generality means that generic and parameterized definitions should be preferred over collections of special-purpose definitions. This can be achieved in LOTOS by using parameterized process definitions. Open endedness supports flexibility of design to ease the modification of the system functionality.

4.2 The LOTOS specification

In this section, we are going to describe the LOTOS specification of the IN call model and services by describing the main abstract datatypes, the structure of the specification and the different processes of which it is composed.

4.2.1 Datatype definitions

Most of the data required to specify our system are defined in IN CS1 GFP and DFP. They have been specified at a high level of abstraction since we are only interested in the observable behavior of the system and not in the implementation details. However, they could always be refined when the implementation details become relevant. For this reason, some data defined in [Q.1213] have been omitted, such as the call reference and the calling line category, since they are not needed for specifying the behavior of our system.

Some of the specified data were not defined in IN CS1 but are still required to complete the specification. Such are the users busy list and the telephone signals.

Table 3 describes the main datatypes that have been specified:

TABLE 3. The main Abstract Datatypes

Datatype	Meaning
Network_address	This type defines points in the network between which connections can be established.
Dialled_Number	This type defines the set of digits dialled by the caller.
Trigger_Detection_Point	This type defines the DPs defined in CS1.
Service_Name	Each IN service is referenced by a variable of this type. This type makes it possible to pass service names as parameters between processes.
BusyList	This type is used to define the set of busy users, i.e. users who are already involved in call/connections.
Signals	This type is used to define the signals exchanged between the switch and the telephone set such as OffHookToCall, GetTone, HangsUp...

The arming of Trigger Detection Points is modeled by the function *trigger_armed* defined in the *Trigger_Detection_Point* type as:

trigger_armed: trigger_detection_point, network_address, service_name -> *Bool*

In fact, for a specific user, subscribing to an IN service, a *Trigger Detection Point* should be armed in order to launch the service at this specific point of the call. The subscriber could be either a caller or a called party.

Some other datatypes which are specific to IN services were also specified such as the *ScreenList* for the Originating Call Screening and Terminating Call Screening services, or the *translated numbers* generated by the Abbreviated Dialling Service. They will be discussed in detail in the next chapter.

4.2.2 Architecture of the specification

In order to achieve a clear and readable specification, it is required to put it together in a step-wise fashion. First, the system is described by the highest level processes that represent the highest abstract view of the different objects composing it, then each resulting process is decomposed into sub-processes. The process of system refinement is repeated until we end up with simpler descriptions where no further decomposition is possible.

At the highest level of abstraction, we can view the IN system as a means to establish connections between network subscribers in order to communicate. These connections are constrained by the fact that at any time, a number is in use at most once and if a network subscriber is in a busy state, it cannot make or receive calls (in normal call processing). This constraint is handled in our specification by using a list of busy subscribers which is updated each time a subscriber becomes busy or turns to idle.

As a result, the top level of the behavior part of our specification consists of two processes, *Connections* and *Update_Busy_List* composed in parallel and synchronizing through gates *N* and *DBRequest*. The two processes synchronize each time an update or a consultation of the list of busy users is needed.

The process *Connections* is composed of two processes: *Network_Subscribers* and *IN_Network*. The process *Network_Subscribers* defines the subscribers of the network. A network subscriber is specified generically by the process *Subscriber* with a parameter representing its network address. An unlimited number of subscribers can thus be produced. The subscribers behave independently and each one can initiate a call at any time. Therefore, the process *Network_Subscribers* consists of a number of interleaved subscribers.

The process *IN_Network* defines the activities required to handle call/connections between network subscribers. These are mainly the functionalities of the Call Control Function/Service Switching Function (CCF/SSF), the Service Control Function (SCF) and the Service Data Function (SDF). Processes *Network_Subscribers* and *IN_Network* communicate through synchronization on gates *N* and *G*. Actions which use *N* as gate name require a three way synchronization between processes *Network_Subscribers, IN_Network* and *Update_Busy_List* in order to be performed. These are the actions that involve an update of the user busy list. However, actions which use gate name *G* require only a two way synchronization between process *Network_Subscribers* and process *IN_Network* since they don't require any modification of the busy list. Synchronization between processes *Connections* and *Update_Busy_List*, on gate *DBRequest*, is done when an update of the busy users list is needed.

Figure 17 describes the top level of the specification:

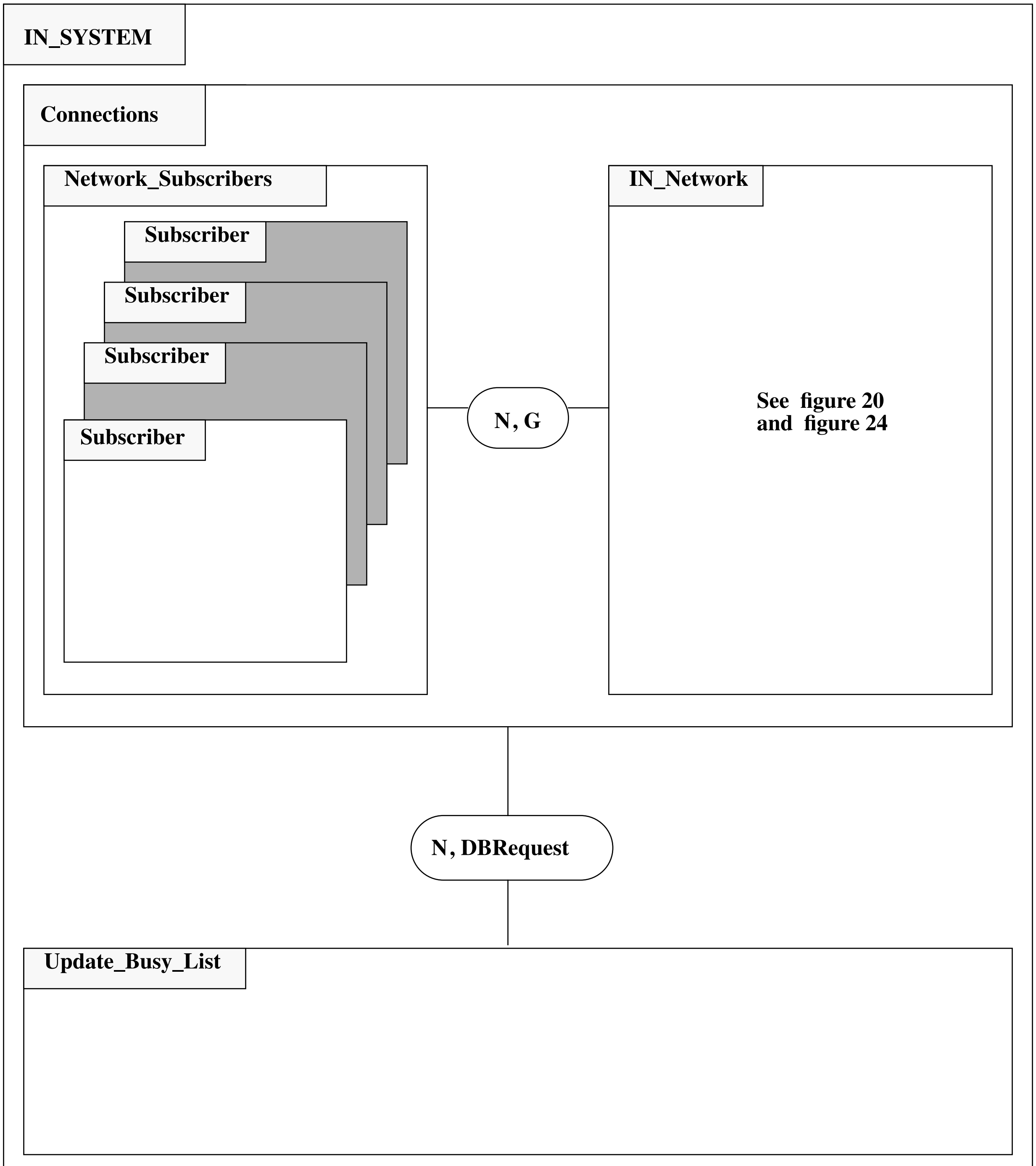


Figure 17: Graphical representation of the top levels of the specification

The specification below represents figure 17 as a LOTOS behavior expression.

```
specification IN_System [S , G, N, D, Detection_P oint, DBRequest] : noexit:=
(*... Abstract datatype Definitions ... *)
 behavior
(
    Connections [S , G, N, D, Detection_P oint, DBRequest]
      |[ N, DBRequest ]|
    Update_Busy_List [N , DBRequest](empty)
)
where

process Connections [S , G, N, D, Detection_P oint, DBRequest] : noexit:=
(
    Network_Subscribers [G , N, DBRequest]
     |[ N, G ]|
    IN_Network [S , G, N, D, Detection_P oint, DBRequest]
)
endproc (* end of process Connections *)

process Network_Subscribers [G , N, DBRequest] : noexit:=
(
   Subscriber [G , N, DBRequest](adr1)
   |||
     ...
   |||
   Subscriber [G , N, DBRequest](adrn)
)
endproc (* end of process Network_Subscriber *)

(*... Process Update_Busy_List Definition ...*)
(*... Process Subscriber Definition ...*)
(*... Process IN_Network Definition ...*)

endspec (* IN_system *)
```

4.2.3 Process Subscriber

Note that although for simplicity only three subscribers were defined, an unbounded number could be allowed by using recursive interleave.

A Subscriber has two roles. It can either initiate or respond to a call, and only one role can be active at any given time. When a subscriber attempts to initiate a call, it cannot receive any call and when its telephone rings, it cannot initiate any call. Therefore, we represent the process *Subscriber* as a choice between two processes: *Caller_Side* and *Called_Side*. When the subscriber is idle, which means that it can initiate or receive calls, either process is ready to synchronize. If one of the processes synchronizes with its environment, the other process dies and cannot synchronize, since only one scenario is permitted. When the subscriber returns to idle, an instantiation of process *Subscriber* is created and becomes ready to synchronize.

The process *Subscriber* is defined as follows:

```
process   Subscriber [G  , N, DBRequest ](adr: network_address)      : noexit:=
(
  Caller_Side[G  , N, DBRequest ](adr)
  []
  Called_Side[G  , N, DBRequest ](adr)
 )
endproc   (* end of process Subscriber *)
```

Process *Caller_Side* specifies the behavior of a subscriber that initiates a call. It describes the actions that are seen and performed by the call initiator. On the other hand, process *Called_Side* describes the actions performed and seen by the call responder.

To specify the caller and the called sides of a subscriber, we have referred to the CS1 Basic Call State Model (BCSM) described in chapter 2. In fact, the Points In Call (PICs) described in the BCSM identify the CCF activities required to complete one or more basic call/connection states of interest to the IN services defined in CS1 [Q1214]. The activity described by a PIC can represent either an internal activity of the switch which is not seen by the subscriber (caller or called side) or can be manifested by an event or a signal seen at the caller or the called side. For example, the two PICs *Analyse_Info* and *Select_Route* define respectively the activities of analysing and translating information according to the dialing plan to determine routing address and call type, and interpreting routing address and call type, and therefore, they not are seen by the subscriber. However, the PIC *Collect_Info* defines the activity of collecting dialing digits (e.g., service code, prefixes, dialed address digits) and is manifested by an event at the caller side which is the dialing of a number by the call initiator. Similarly, the PIC *T_Altering* defines the fact that a processing of call setup is taking place and is manifested by an audible ring indication at the called side. As a result, only the actions defined by the PICs that are seen by the call initiator or the call responder are specified by processes *Caller_Side* and *Called_Side*.

4.2.3.1 Process Caller_Side

The process *Caller_Side* is defined as follows:

```
process Caller_Side [G , N, DBRequest ] (caller: network_address) : noexit:=

 N !caller !OffHookToCall ;
 (
    Caller_GetsTone [G , N, DBRequest ](caller)
   []
    G !caller !NoTone ; User_HangsUp [G , N, DBRequest ](caller)
   []
    User_HangsUp [G , N, DBRequest ](caller)
 )
endproc (* end of process Caller_Side *)
```

The caller side is activated when a subscriber takes the phone off hook in order to make a call. This is described by the action *N* **!***caller* **!***OffHookToCall*. If this action is executed, the subscriber becomes busy and three scenarios are possible. The caller can get a tone and continue the processing of the call, this is described by the process *Caller_GetsTone*; or the origination of making calls is denied and the caller does not get any tone (*G* **!***caller* **!***NoTone*), then it can only hang up the phone; or it can hang up the phone directly after an off hook, and this is described in the third choice by instantiating the process *User_HangsUp* which is described as follows:

```
process User_HangsUp [G , N, DBRequest ] (user: network_address) : noexit:=
(
G !user !HangsUp ;
N !user !Go_Idle ;
Subscriber [G , N, DBRequest ](user)
)
endproc (* end of process User_HangsUp *)
```

The process *User_HangsUp* specifies the fact that a subscriber has hung up the phone and then moved to the idle state where it can again initiate and receive calls. The subscriber can be a call initiator or a call responder. Therefore, we used the parameter *user: network_address* in the process definition to allow the instantiation of this process by the caller and the caller side. This satisfies the principle of generality of processes followed in our specification, which allows their reusability.

The process *Caller_GetsTone* is described as follows:

```
process Caller_GetsT one [G , N, DBRequest ] (caller: network_address) : noexit:=

G !caller !GetT one ;
(
   Caller_Dials [G , N, DBRequest ](caller)
  []
   User_HangsUp [G , N, DBRequest ](caller)
)
endproc (* end of process Caller_GetsT one *)
```

If the caller gets a tone (action *G* **!***caller* **!***GetTone*), it can hang up or dial a number and continue processing the call. The latter case is specified by instantiating the process *Caller_Dials* described as follows:

```
process Caller_Dials [G , N, DBRequest ] (caller: network_address) :noexit:=

G !caller !dials ?dn: dialled_number ;
(
   G !caller !TimeOut ; User_HangsUp[G , N, DBRequest ](caller)
```

```
   []
   G !caller  !GetBusySignal   ; User_HangsUp [G    , N, DBRequest ](caller)
   []
   G !caller  !InvalidInformationIndication      ; User_HangsUp [G    , N, DBRequest](caller)
   []
   G !caller  !IndicationT   erminationDenied    ; User_HangsUp [G    , N, DBRequest](caller)
   []
   Caller_GetsRingT    one [G , N, DBRequest ](caller)
   []
   User_HangsUp [G    , N, DBRequest ](caller)
  )
  endproc    (* end of process Caller_Dials *)
```

When a caller dials a number (action *G* **!***caller* **!***Dials* **?***dn: dialled_number*), six choices can be seen by the caller side (with respect to the CS1 BCSM). The first choice (action *G* **!***caller* **!***TimeOut*) describes the fact that an error has occurred when the user dialled a number. This can be due to an invalid string format or a digit collection time-out. The second choice (action *G* **!***caller* **!***GetBusySignal*) is performed when the called side is busy. The third choice (action *G* **!***caller* **!***InvalidInformationIndication*) can be performed when a user dials an invalid number. The fourth one (action *G* **!***caller* **!***IndicationTerminationDenied*) describes the fact that the attempt of making a call has been denied by the terminating party. The caller can only hang up the phone after one of these four choices. If normal call processing has occurred, the caller gets an audible ring tone indicating that the call setup is taking place. This is described in the fifth choice by instantiating the process *Caller_GetsRingTone*. The sixth choice describes the fact that a caller can hang up the phone right away after dialling a number and before getting any indication from the network.

The process *Caller_GetsRingTone* is defined as follow:

```
process  Caller_GetsRingTone [G , N, DBRequest ] (caller : network_address)   : noexit:=

G !caller !GetRingTone ; (1)
 (
    G !caller !StopRingTone ; (2)
      (
        G !caller !EstablishConnection ; (4)
          (
            Talking_session [G , N, DBRequest ](caller)   (7)
          [>
              (
               G !caller !DisconnectionIndication ; (8)
               User_HangsUp [G , N, DBRequest ](caller)
             []
               User_HangsUp [G , N, DBRequest ](caller)   (9)
              )
          )
      []
        G !caller !GetNoAnswerIndication ; User_HangsUp [G , N, DBRequest ](caller)   (5)
        []
        User_HangsUp [G , N, DBRequest ](caller)   (6)
  )
 []
 User_HangsUp [G , N, DBRequest ](caller)   (3)
 )
endproc  (* end of process Caller_GetsRingTone *)
```

If a caller gets a ring tone (1), two choices are possible. The caller can hang up the phone at any time it wants and this is described by the instantiation of process *User_HangsUp* (3), or the ring tone stops (action *G* **!***caller* **!***StopRingTone*) (2). The second scenario takes place if a ring time-out occurs and no answer has been detected from the terminating side (action *G* **!***caller* **!***GetNoAnswerIndication*) (5), or if the called side answers the phone. The user can also hang up the phone right after the ring tone stops (6).

If the called side answers the phone, the caller establish connection (4) and a talking session starts (7). This is represented by instantiating the process *Talking_session*. This process can be disabled at any time if the caller hangs up (9) or if the connection fails (action *G* **!***caller* **!***DisconnectionFailureIndication*) (8).

The process *Talking_session* which is instantiated when the called side answers the phone is described as follows:

```
process   Talking_session[G   , N, DBRequest ] (user: network_address)        : noexit:=
 (
    G !user   !Talks ;
    Talking_session[G   , N, DBRequest ](user)
 )
endproc   (* end of process T   alking_session *)
```

This process specifies the fact that a user can talk infinitely by instantiating itself recursively after performing the action *G* **!***user* **!***Talks*. The definition of this process satisfies the principle of generality. In fact this process is defined using a parameter *user: network_address* in order to allow its reusability. Figure 18 gives a graphical representation of the process caller Labelled Transition System (LTS).

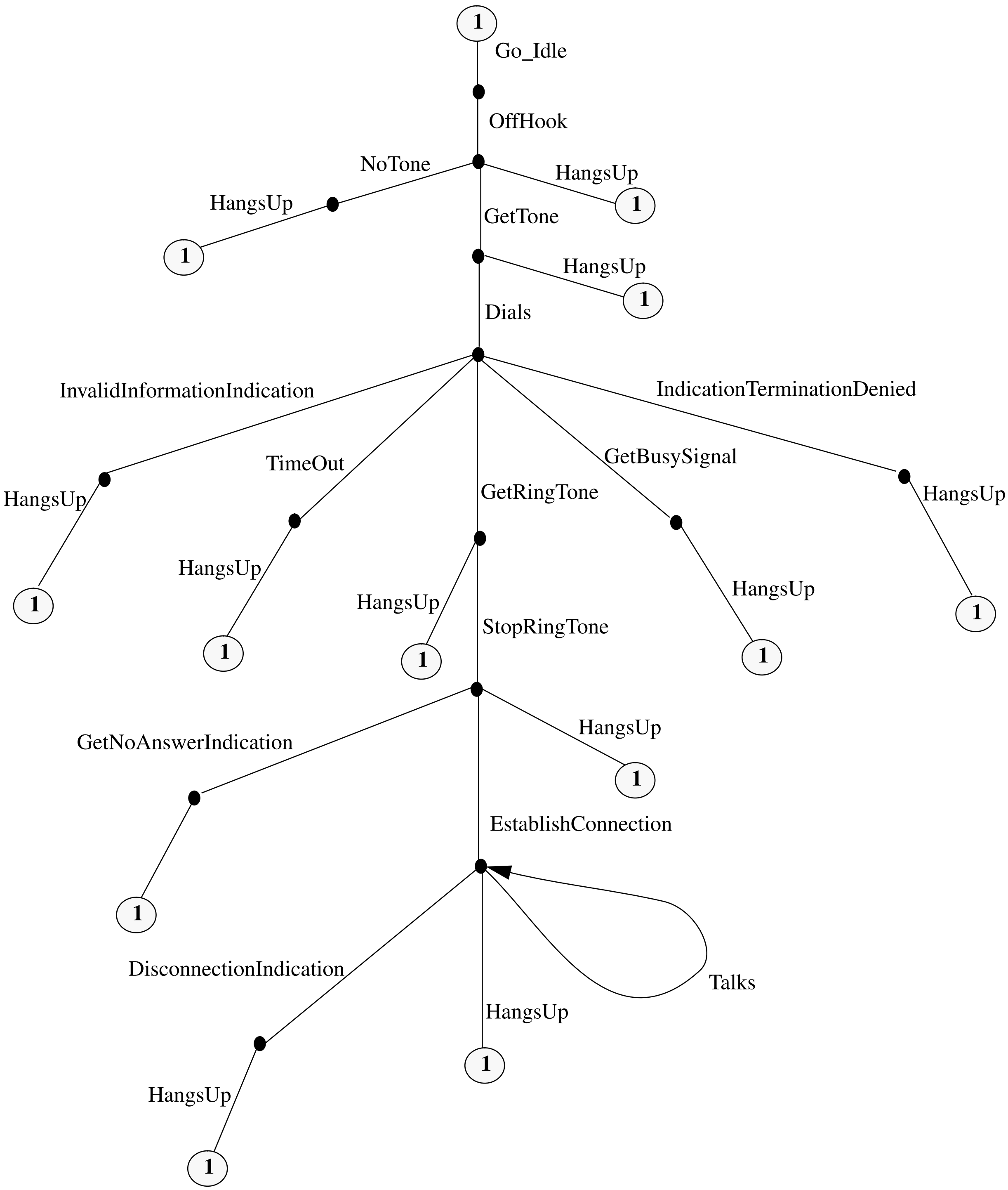


Figure 18: LTS of process Caller

4.2.3.2 Process Called_Side

The responder role of a subscriber is defined by the process *Called_Side* defined as follows:

```
process Called_Side [G , N, DBRequest ] (called: network_address)   : noexit:=

 N !called !RingsF rom ?caller: network_address  ; (1)
 (
   G !called !OffHookT oAnswer ; (2)
    (
      G !called !EstablishConnection  ; (4)
      (
        Talking_session [G , N, DBRequest ](called)  (5)
       [>
         (
          G !called !DisconnectionIndication  ; (7)
          User_HangsUp [G , N, DBRequest ](called)
        []
          User_HangsUp [G , N, DBRequest ](called)  (8)
         )
      )
      []
      User_HangsUp [G , N, DBRequest ](called)  (6)
    )
  []
  G !called !RingTimeOut ; (3)
  N !called !Go_Idle ;
  Subscriber [G , N, DBRequest](called)
)
endproc  (* end of process Called_Side *)
```

The called side of a subscriber becomes active when the phone rings. This is described by the action *N* **!***called* **!***RingsFrom* **?***caller: network_address* (1). The subscriber becomes busy and must be inserted in the busy list. For this reason, gate *N* is used.

When the phone rings, two scenarios are possible. The phone can ring until a time-out occurs and then the called moves to the idle state, this is described by the sequential composition of actions *G* **!***called* **!***RingTimeOut***;** *N* **!***called* **!***Go_Idle***;** *Subscriber [G, N, DBRequest](called)* (3); or the subscriber picks up the phone to answer the call and the ring tone stops. This is described by the action *G* **!***called* **!***OffHookToAnswer;* *G* **!***called* **!***StopRingTone* (2). In this case, the called establishes a connection (4) and either it starts a talking session (6) or hangs up the phone (5).

As mentioned before, a talking session ends when the called hangs up the phone (8) or gets a disconnection indication which happens if the caller side disconnects or if a failure in the system occurs (7). This is represented by applying the disable operator to the process *Talking_session* followed by the choice between the action *G* **!***called* **!***DisconnectionFailureIndication* and the instantiation of the process *User_HangsUp*. Figure 19 describes the LTS of process *Called*.

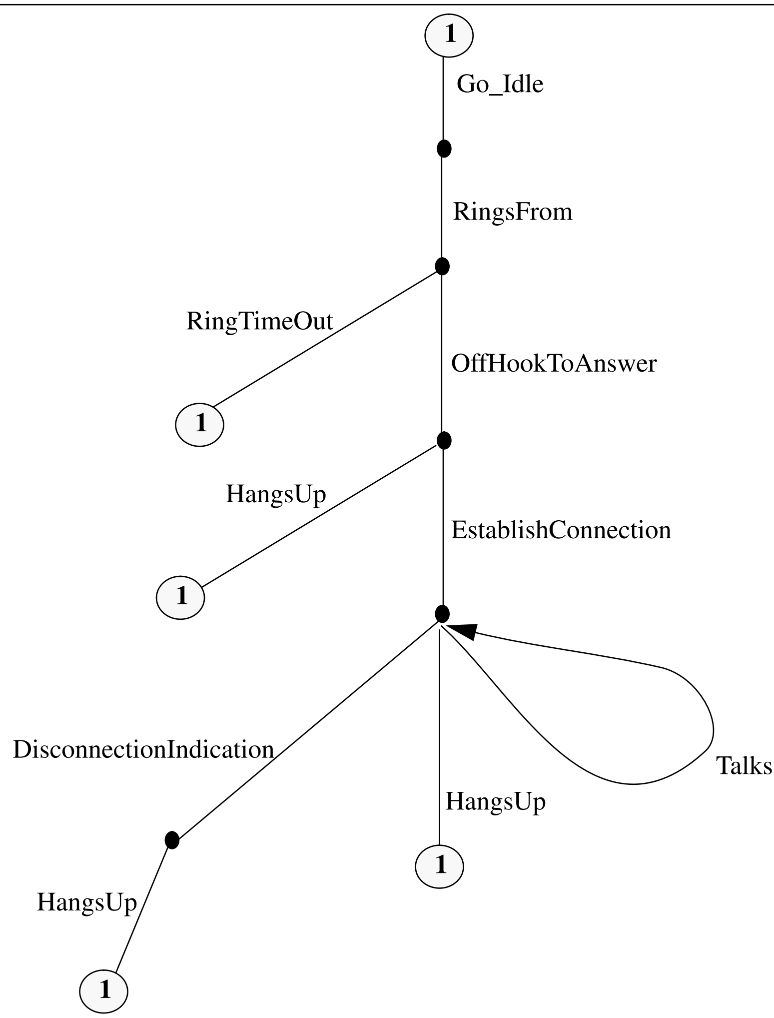


Figure 19: LTS of process Called

4.2.4 Process Update_Data_Base

We have seen that the process *Subscriber* defined as a choice between processes *Caller_Side* and *Called_Side* synchronizes on gates *N* and *G* with the process *IN_Network*, and the resulting behavior synchronizes with process *Update_Busy_List* on gates *N* and *DBRequest*.

The process *Update_Busy_List* is defined as follows:

```
process Update_Busy_List [N , DBRequest] (BusyList: List) :noexit:=
 (
    DBRequest !Consult !BusyList ;
    Update_Busy_List [N , DBRequest] (BusyList)
   []
    N ?caller: network_address !OffHookT oCall [caller NotIn BusyList] ;
    Update_Busy_List [N , DBRequest] (InsertL(caller ,BusyList))
   []
    N ?called: network_address !RingsF rom ?caller: network_address ;
    Update_Busy_List [N , DBRequest] (InsertL(called,BusyList))
   []
    N ?user: network_address !Go_Idle ;
    Update_Busy_List [N , DBRequest] (RemoveL(user , BusyList))
)
endproc (* end of process Update_Busy_List *)
```

This process has as parameter *BusyList* of type List. The parameter is initialized with the value *empty* when it is instantiated at the top level of the specification, representing an empty list of busy users. In fact, at the beginning, we suppose that all the network subscribers defined in our specification are not *busy* and can be involved in any call/connection at any time. A subscriber becomes *busy* when it initiates or responds a call, and must be added to the *BusyList*. This is done by the synchronization which occurs on gate *N* when the two actions *N* **?***caller: network_address* **!** *OffHookToCall [caller NotIn BusyList]* and *N* **?***called: network_address* **!***RingsFrom* **?***caller: network_address* are executed, and by reinstantiating *Update_Busy_List* with the parameter *InsertL(subscriber, BusyList).*

If a subscriber becomes *idle*, it must be removed from the *BusyList*. The synchronization occurs on gate *N* by executing the action *N* **?***user: network_address* **!***Go_Idle,* and the value of the user network address is removed from the *BusyList* by instantiating the process *Update_Busy_List* with the parameter *RemoveL(user, BusyList)*. Operations *InsertL* and *RemoveL* are defined in the abstract data type part and they have respectively the role of inserting and removing an element from a list.

Synchronization on gate *DBRequest* occurs only between the process *IN_Network* and *Update_Busy_List* when a consultation of the *BusyList* is needed.

4.2.5 Process IN_Network

The process *IN_Network* which specifies the DFP entities involved in the establishment of call/connection is defined as follows:

```
process IN_Network [S , G, N, D, Detection_Point, DBRequest] : noexit :=

  CCF_SSF [S , G, N, Detection_Point, DBRequest]
   |[ S]|
  SCF [S , G, N, D, DBRequest]
   |[ D]|
  SDF [D , DBRequest]

endproc (* end of process IN_Network *)
```

This process is defined by a composition of three processes, process *CCF_SSF* which specifies the CCF/SSF functionalities, process *SCF* which specifies the SCF activities and process *SDF* which specifies the SDF. Processes *CCF_SSF* and *SCF* communicate via synchronization on gate *S*, and processes *SCF* and *SDF* communicate via synchronization on gate *D*. Figure 20 gives the top level description of process *IN_Network*.

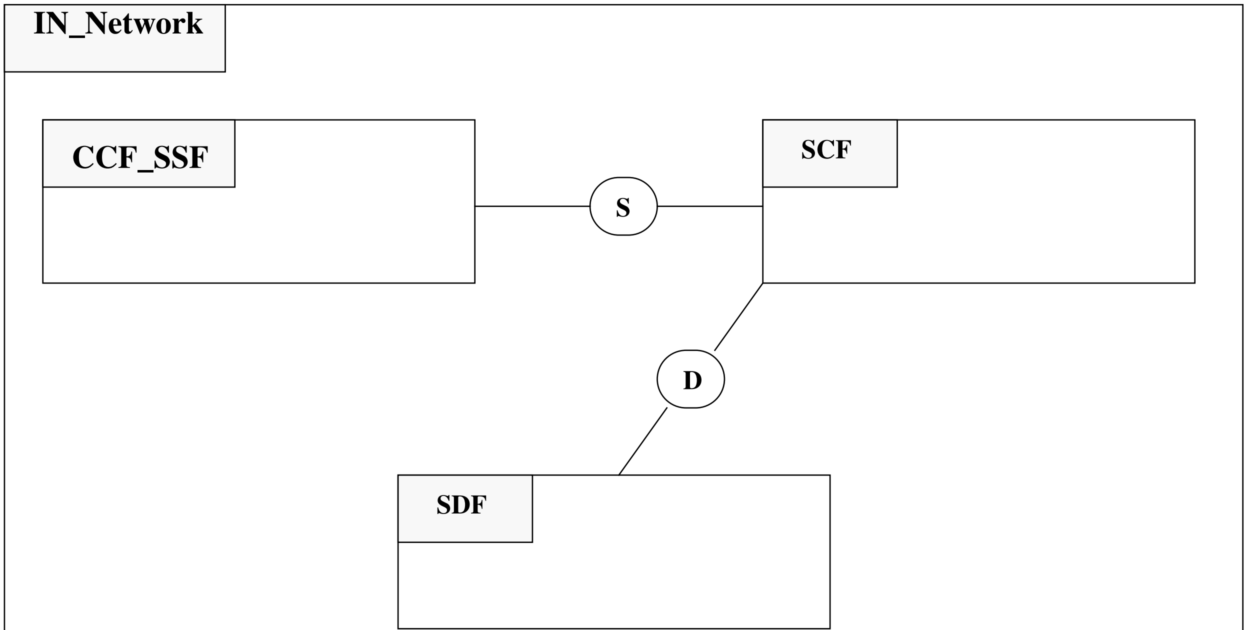


Figure 20: Top level representation of process IN_Network

4.2.5.1 Process CCF_SSF

Process *CCF_SSF* is defined as follows:

```
process CCF_SSF [S, G, N, Detection_Point, DBRequest] : noexit:=

N ?caller: network_address !OffHookToCall ;
(
  (
   Basic_Call_State_Model [S, G, N,Detection_Point, DBRequest](caller, nil, null)
    |[ Detection_Point ]|
   Check_Trigger [Detection_Point]
  )
 |||
  CCF_SSF [S, G, N, Detection_Point, DBRequest]
)
endproc (* end of process CCF_SSF*)
```

where process *Basic_Call_State_Model* specifies the CS1 BCSM, and process *Check_Trigger* checks for the arming criteria at each Detection Point (DP) in the BCSM in order to decide whether an IN service should be launched. The recursive instantiation of process *CCF_SSF* after an interleave operator (|||) induces the fact that an unlimited number of instantiations are ready to synchronize. When a call is initiated by a subscriber and synchronization on gate *N* occurs (action *N* **?***caller: network_address* **!***OffHookToCall* is executed), an instantiation of the process *Basic_Call_State_Model* composed in parallel with the process *Check_Trigger* is created.

The CS1 BCSM defines different states in the call processing, and each state is represented by either a Point in Call (PIC) or a Detection Point (DP). At the DP, arming criteria are checked in order to decide if a service should be invoked or not. If not, the functions defined by the PIC should be processed and the system moves to one of the next possible DPs depending on the results of processing the PIC. If a service is launched, it determines the next state (DP or PIC) at which the processing of the call should continue.

For example, when the trigger criteria are checked at the DP *Origination_Attempt* and no service is invoked, the functions defined by the PIC *Authorize_Origination_Attempt* are processed, and the system moves to one of the next possible states defined by the DPs *O_Calling_Party_Disconnect* & *O_Abandon*, *Origination_Attempt_Authorized* and *Origination_Denied* (see figure 21). If a service is invoked, the return state can be any one of the states defined in the BCSM.

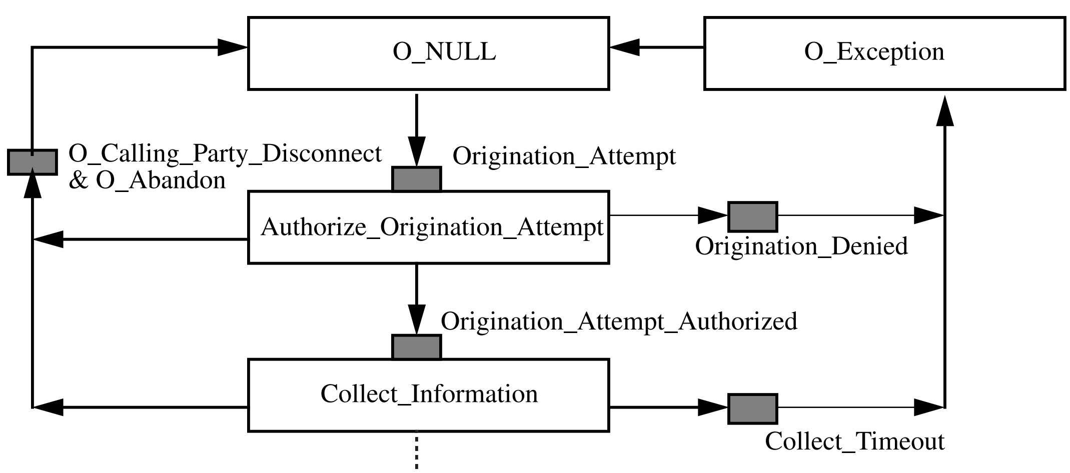


Figure 21: Part of the O-BCSM

The BCSM is composed of two parts, O-BCSM and T-BCSM which model respectively the processing logic supported by the calling and the called parties. These two parts communicate by exchanging intra-switch signals if the two call parties are connected to the same switch, or by exchanging inter-switch signals if each party is connected to a different switch. Since we are interested only in the external behavior of the system, we limited ourselves, in the specification of the BCSM, to the observable actions that can be seen at the caller and the called sides. The intra-switch communication and the signals exchanged between the O-BCSM and the T-BCSM were not specified. We consider O-BCSM and T-BCSM as one entity and we specify the different states and the possible transitions between them.

In order to preserve the structure of the BCSM by specifying explicitly its internal states, the *state oriented style* has been adopted. Normally, a specification in this style takes the form of a process with parameters representing states, usually called state variables. Its behavior expression consists of a series of choices guarded by tests associated with the actual state (the parameters of the process). The process is eventually recursively invoked with new values for the parameters representing the state. However, this form has some drawbacks. In fact, a new type defining the state variables must be defined in the datatypes, and when instantiating the process describing all the states with the value of the state variable representing the state to which we want to move, a series of guards should be evaluated and tested in order to determine the required state. This can slow down the validation tools we are using. To avoid this, a different form of the *state oriented style* has been used. Each state is represented in the process *Basic_Call_State_Model* by a process having the name of the state, and the move to a next state is done by instantiating the process defining this state.

A state can define either a DP or a PIC. A process defining a PIC has the following structure:

```
process PIC_Name [S, G, N, Detection_Point, DBRequest]
(caller: network_address, dn: dialled_number, called: network_address) : noexit:=
(
                    (* ... perform actions ... *)
  Next_DP_1[S, G, N, Detection_Point, DBRequest](caller, dn, called)
  []
  ...
```

```
  []
                    (* ... perform actions ...*)
   Next_DP_n [S  , G, N, Detection_P  oint, DBRequest](caller   , dn, called)
)
endproc   (* end of process PIC_Name *)
```

In a process specifying a PIC, only the possible observable actions that are seen at that PIC are specified, and processes describing the possible DPs to which the system can move are instantiated. For example, the PIC *Authorize_Origination_Attempt* defines the functionality of verifying the ability of placing a call by the originating party. If the ability to originate calls is verified, the caller gets a tone and the system moves to the state defined by the DP *Orig_Attempt_Auth*. If the ability to place outgoing calls is denied, the caller does not get any tone and the transition is to DP *Orig_Denied*. In addition, the caller can hang up the phone at any time, and then the system moves to the state defined by the DP *O_Abandon*. The PIC *Authorize_Origination_Attempt* is described by the following process:

```
process   PIC_Auth_Orig_Attempt[S    , G, N, Detection_P  oint, DBRequest ]
(caller: network_address    , dn: dialled_number   , called: network_address)    : noexit :=
 (
    G !caller  !GetT one ;
    DP_Orig_Attempt_Auth[S    , G, N, Detection_P  oint, DBRequest ](caller    , dn,called)
    []
    G !caller  !NoT one ;
    DP_Orig_Denied[S   , G, N,Detection_P  oint, DBRequest ](caller    , dn, called)
    []
    G !caller  !HangsUp  ;
    DP_O_Abandon[S   , G, N, Detection_P  oint, DBRequest ](caller    , dn, called )
 )
endproc   (* end of process PIC_Auth_Orig_Attempt *)
```

A process defining a DP has the following structure:

```
process DP_Name [S, G, N, Detection_Point, DBRequest]
(caller: network_address, dn: dialled_number, called: network_address) : noexit:=
(
Detection_Point !Name_Of_The_Detection_Point !caller !called ;
Detection_Point !Name_Of_The_Detection_Point ?isarmed: bool ? Ser_Name: service_name ;
  (
   [isarmed eq false ] ->
        Next_State[S, G, N,Detection_Point, DBRequest](caller, dn, called)
  []
  [isarmed eq true ] ->
        SSF [S, G, N, Detection_Point, DBRequest](Ser_Name, caller, dn, called)
  )
)
endproc
```

At a DP, arming criteria are checked. This is done by synchronization between process *Check_Trigger* and the process defining the DP on gate *Detection_Point*.

Process *Check_Trigger* has the following structure:

```
process Check_Trigger [Detection_Point] : noexit:=

Detection_Point ?TDP: trigger_detection_point ?caller: network_address ?called: network_address ;
 (
    [trigger_armed(TDP, caller, Service_Name_1)] ->
    Detection_Point !TDP !true !Service_Name_1 ;
    Check_Trigger [Detection_Point]
   []
    ...
   []
    [trigger_armed(TDP, caller, Service_Name_n )] ->
    Detection_Point !TDP !true !Service_Name_n ;
    Check_Trigger [Detection_Point]
```

```
    []
     [(not(trigger_armed(TDP     , called, Service_Name_1)))
     and...and (not(trigger_armed(TDP       , caller , Service_Name_n)))]    ->
     Detection_P  oint  !TDP  !false  !NO_SERVICE  ;
     Chec k_Trigger [Detection_P     oint]
  )
  endproc    (* Process Chec   k_Trigger *)
```

The synchronization is done by passing parameters to process *Check_Trigger*, parameters that are needed for checking the arming criteria. This is done by executing the action *Detection_Point* **!***Name_Of_The_Detection_Point* **!***caller* **!***called.* These parameters are the name of the DP and the network addresses of the caller and the called. Then, process *Check_Trigger* checks for the arming criteria defined as equations in the datatype part and communicates the result to process *Basic_Call_State_Model* by synchronizing on the gate *Detection_Point* and executing the action *Detection_Point* **!***Name_Of_The_Detection_Point* **?***isarmed: bool* **?***Ser_Name: service_name.* When a service must be invoked, the variable *isarmed* takes the value *true* and the variable *Ser_Name* takes the value of the parameter referencing the service. If no service must be invoked, *isarmed* gets the value *false* and *Ser_Name* gets the value *NO_SERVICE.* The arming of a DP is done each time a new service is specified, by adding arming criteria described as equations in the datatypes (they are explained in details in the next chapter).

Figure 22 describes the communication between *Basic_Call_State_Model* and *Check_Trigger* processes.

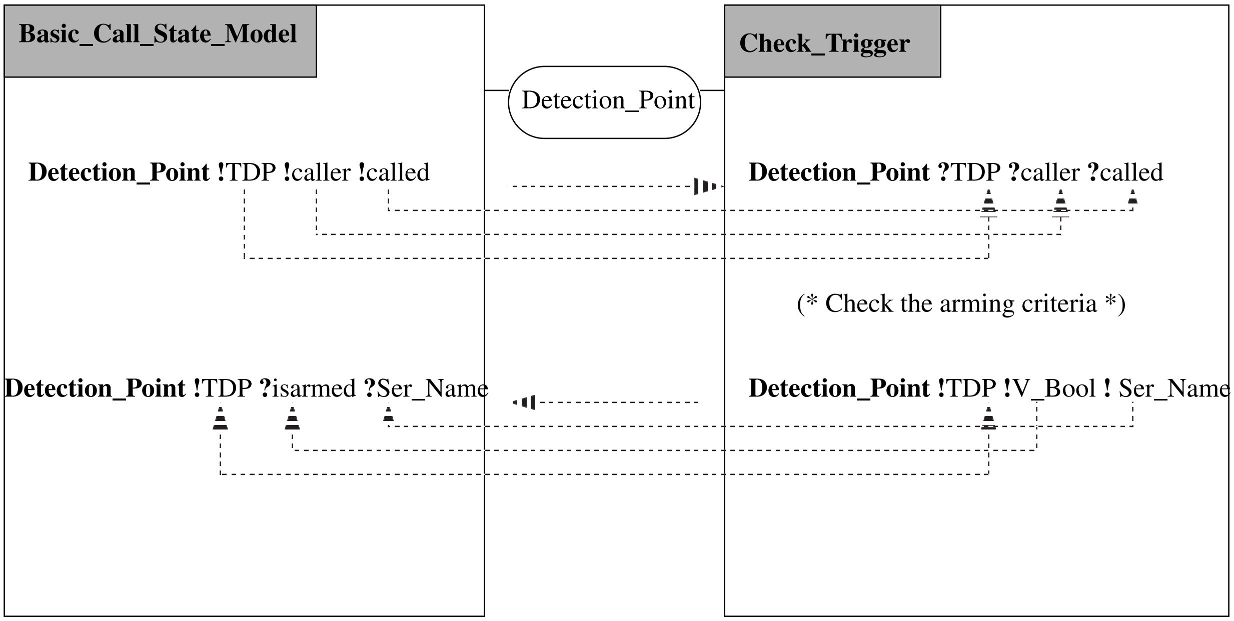


Figure 22: Communication between Basic_Call_State_Model and Check_Trigger processes

The process *Basic_Call_State_Model* is instantiated when a call is initiated by a subscriber. It is specified as follows:

```
process Basic_Call_State_Model [S, G, N, Detection_Point, DBRequest ]
(caller: network_address, dn: dialled_number, called: network_address) : noexit:=

    DP_Orig_Attempt [S, G, N, Detection_Point, DBRequest ](caller, dn, called)

endproc (* end of process Basic_Call_State_Model *)
```

where process *DP_Orig_Attempt* is specified as follows:

```
process DP_Orig_Attempt [S, G, N, Detection_Point, DBRequest ]
(caller: network_address, dn: dialled_number, called: network_address) : noexit:=

  Detection_Point !Orig_Attempt !caller !called ; (* action shown in Figure 22 left box *)
```

```
    Detection_Point !Orig_Attempt ?isarmed: bool ?ser: service_name ; (* action shown in Figure 22
                                                                        right box *)
     (
       [isarmed eq false] ->
       PIC_Auth_Orig_Attempt [S, G, N,Detection_Point, DBRequest ](caller, dn, called)
      []
       [isarmed eq true] ->
       SSF[S, G, N, Detection_Point, DBRequest](Ser_Name, caller, dn, called)
     )
  endproc (* end of process DP_Orig_Attempt *)
```

The process *Basic_Call_State_Model* describes the first DP defined in the BCSM and instantiates the process specifying the associated PIC if a service is not launched. Then each process describing a next possible state instantiates the processes defining the next possible states from that state, an so on.

If the arming criteria are met, an instantiation of the process *SSF* is created. This process specifies the functionalities of SSF which acts as an interface between the CCF (described by the BCSM) and the SCF. This process has the following structure, and is detailed in the next section:

```
  process SSF [S, G, N, Detection_Point, DBRequest]
  (ser: service_name, caller: network_address, dn: dialled_number, called: network_address)
  : noexit:=

     S !ser !caller !dn !called ;
     (
        S !state_1 ?caller: network_address ?dn: dialled_number ?called: network_address ;
        process_state_1 [S, G, N, Detection_Point, DBRequest](caller, dn, called)
        []
         ...
        []
         S !state_n ?caller: network_address ?dn: dialled_number ?called: network_address ;
         process_state_n [S, G, N, Detection_Point, DBRequest](caller, dn, called)
     )
  endproc (* end of process SSF *)
```

4.2.5.2 Process SCF

Process *SSF* synchronizes with process *SCF*, which specifies the SCF functionalities, on gate *S*. When a service must be invoked, both processes synchronize by executing action *S* **!***ser* **!***caller* **!***dn* **!***called*, and the call parameters needed to process the service are passed to the process *SCF*. This process has the following structure:

```
process SCF[S , G, N, D, DBRequest] : noexit:=
(
    S !service_1 ?caller: network_address ?dn: dialled_number ?called: network_address ;
    process_service_1 [S , G, N, D, DBRequest](caller , dn, called)
    []
     ...
    []
    S !service_n ?caller: network_address ?dn: dialled_number ?called: network_address ;
    process_service_n [S , G, N, D, DBRequest](caller , dn, called)
)
endproc (* end of process SCF *)
```

Each choice in process *SCF* is represented by an action with gate name *S* composed sequentially with an instantiation of a process specifying a service. The action with gate *S* is specified with an offer event of a variable of sort *service_name* (*service_1,.., service_n*) indicating the service process to be instantiated. At the synchronization, the action offering the variable of sort *service_name* which is equal to the variable *ser* (of sort *service_name*) specified in the action *S* **!***ser* **!***caller* **!***dn* **!***called* (in process *SSF*) is executed. Then, an instantiation of the process specifying the required service is created with the call parameters that are passed at the synchronization. We shall see that this process will in turn instantiates the necessary SIBs.

When a service is executed, it must indicate the return state at which the call must continue (DP or PIC) and the process describing this state must be instantiated. This is done by specifying a new type named *Process_Name* of sort *process_name* which associates to each process defining a state a constant (described in the process *SSF* by the parameters *state_1*, ..., *state_n*). At the end of processing a service, a synchronization on gate *S* occurs between process *SSF* and the process defining the executed service. This synchronization is achieved by executing the action on gate *S* with an offer of the parameter of sort *process_name* indicating the state process to be instantiated (which corresponds to the return state in the BCSM), and an accept of the new call parameters (if they have been modified). Figure 23 shows the synchronization between processes *SSF* and *SCF.*

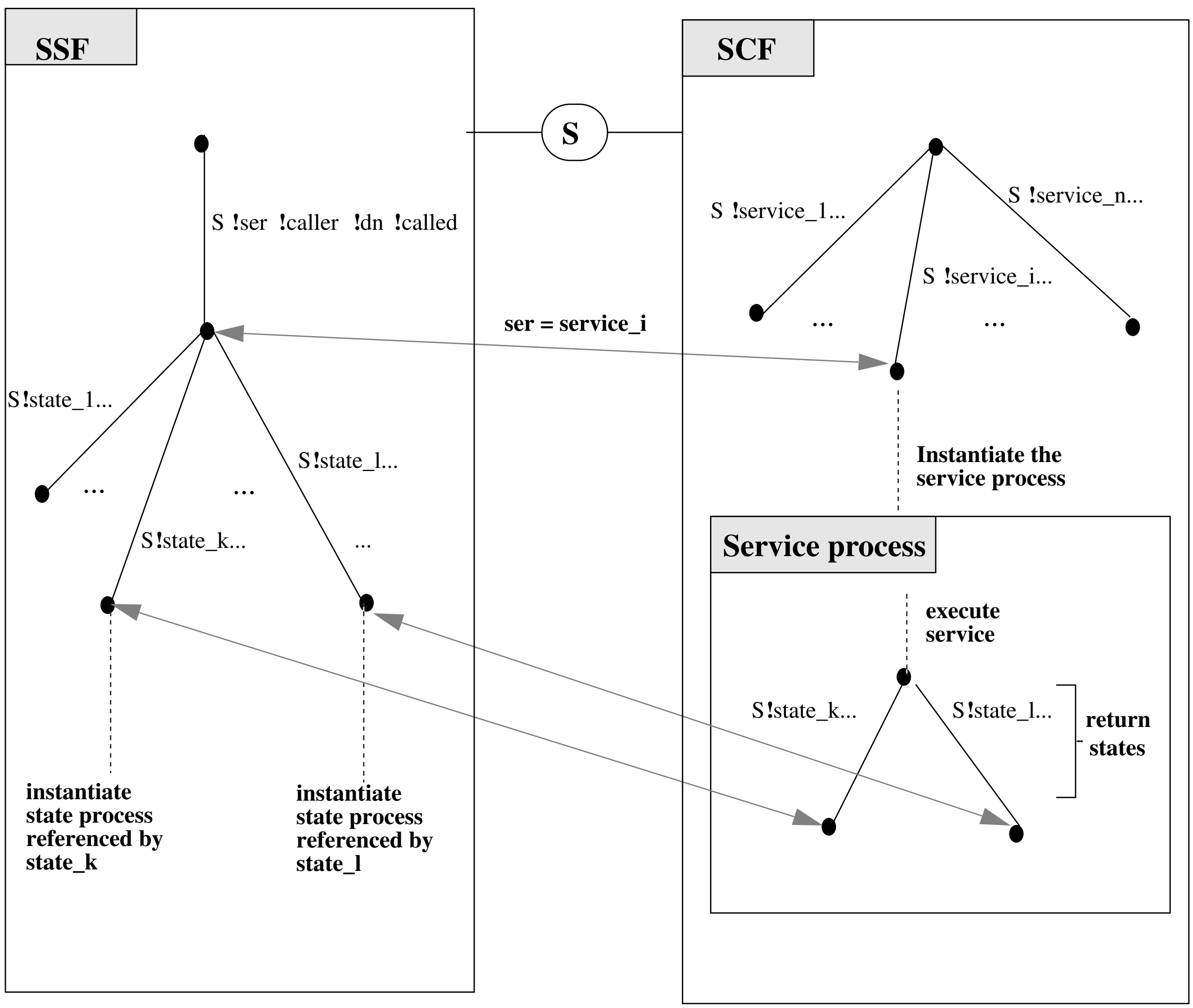


Figure 23: Synchronization between processes SSF and SCF

4.2.5.3 Process SDF

All the data required by IN services are handled by the SDF. This functional entity (FE) can be considered as a data base which the SCF accesses when an IN service needs a consultation or an update of its related data. This FE is described by the process *SDF* which communicates with process *SCF* via synchronization on gate *D*.

In order to perform its functionality, a process defining a service (a service process) needs to communicate with the SDF where the service related data are handled. The service process must send a message to the SDF indicating the type of operation required (consultation, modification or addition of data) and the data involved in this operation. The communication is specified by synchronization on gate *D* between actions defined in the service process (instantiated by process *SCF*) and process *SDF*. Operation type and data are specified as parameters to be exchanged or matched at the synchronization. A new type named *Operation_Name* of sort *operation_name* is specified. It defines the different operations that can be performed by the SDF. After synchronization, process SDF execute the necessary actions needed to perform the operation (consultation, modification or addition of data) required by the service process, and send back the result to the service process.

Process *SDF* is defined as a choice between different operations needed by the different service processes. It has the following structure:

```
process   SDF [D , DBRequest]   : noexit:=

  ...
 []
    D !Service_Name   !Operation_Name    ?parameter_1 ...    ?parameter_n   ; (1)

           (* ... execute actions to perform the operation ... *)

    D !Service_Name   !Operation_Name    !result_1 ...   !result_m  ; (2)
    SDF[D , DBRequest]   (3)
[]
  ...
endproc   (* end of process SDF *)
```

where *parameter_1,..., parameter_n* defined in (1) are the data parameters needed to perform the operation. They are provided by the service process (referenced by the parameter *Service_Name* in (1)) at the synchronization. After performing the necessary actions, the results are sent back to the service process which are represented in (2) by the parameters *result_1, ..., result_m*. Then, process SDF instantiates itself (3) in order to be ready for further synchronizations. A detailed description of process *SDF* is described in the next chapter.

4.2.5.4 Specification of a new service

When a new service is to be specified, a new variable of type *service_name* must be defined and associated to the service, the arming conditions must be defined in the datatypes, checking for the criteria must be added to process *Check_Trigger* and then the service is specified using SIBs. Therefore, a specification of the new service can be done independently and added to the specification of the system without any modifications of structure. This satisfies the principle of open endedness we have mentioned above.

As described in Chapter 2, IN services are defined as a combination of SIBs. Therefore, to specify a new service, we need to specify the SIBs of which it is composed if they are not yet specified.

In order to provide for the reusability of SIBs, their specification should be generic and parametrized so that they can be used by different services.

As an example, let us take the two services *Originating Call Screening* and *Security Screening*. The first service allows a subscriber to authorize outgoing calls, through the use of a screening list. The second one enables a subscriber to protect his line by a user defined key. The caller is asked to dial a PIN code which allows to verify the caller identity before giving access to the subscriber's line. Both services use the *screen* SIB which (as described in Chapter 2) performs a comparison of an identifier against a list to determine whether the identifier is in the list. The first service needs to verify if the called belongs to the originating screening list and the second service needs to check if the PIN code dialled by the caller corresponds to the PIN code defined by the subscriber. This is done by verifying the entered PIN code against the list of subscriber's PIN codes. In order to allow the reusability of this SIB by both services, the list against which the comparison is performed should be defined as a parameter, and this parameter is given by the service. Details are shown in next Chapter.

Each SIB requires two types of data parameters in order to perform its functionality. These are the CID (Call Instance Data) and the SSD (Service Support Data). These data are defined as parameters in the specification of a SIB.

The CID is a record that defines the dynamic parameters, whose values change with each call. The three elements of the CID that are relevant to the specified services and are used as parameters in the SIBs specification are the network addresses of the caller and the called parties, and the number dialled by the caller.

The SSD are the static parameters needed by a SIB and which are specific to each service. They do not change in different calls.

The output data of a SIB are the input data whose values can change during the execution of a service, and some other data depending on the functionality of the SIB.

Thus, the behavior of a process representing a SIB is of the following form:

```
process SIB_Name [G , N, S, DBRequest](caller: network_address , dn: dialled_number ,
called: network_address , SSD_parameters) : exit (network_address , dialled_number ,
 network_address , specific_data) :=
(
    (* perform SIB operations corresponding to the value of SSD_parameters *)
    exit (caller , dn, called, specific_data)
)
endproc (* end of process SIB_Name *)
```

Once the SIBs composing a service are specified, the task of specifying a service becomes easy. In fact, we only need to specify the way these SIBs are combined and this can be done using LOTOS operators. Examples of service specifications using SIBs are described in the next chapter.

Figure 24 provides a schematic description of process *IN_Network.*

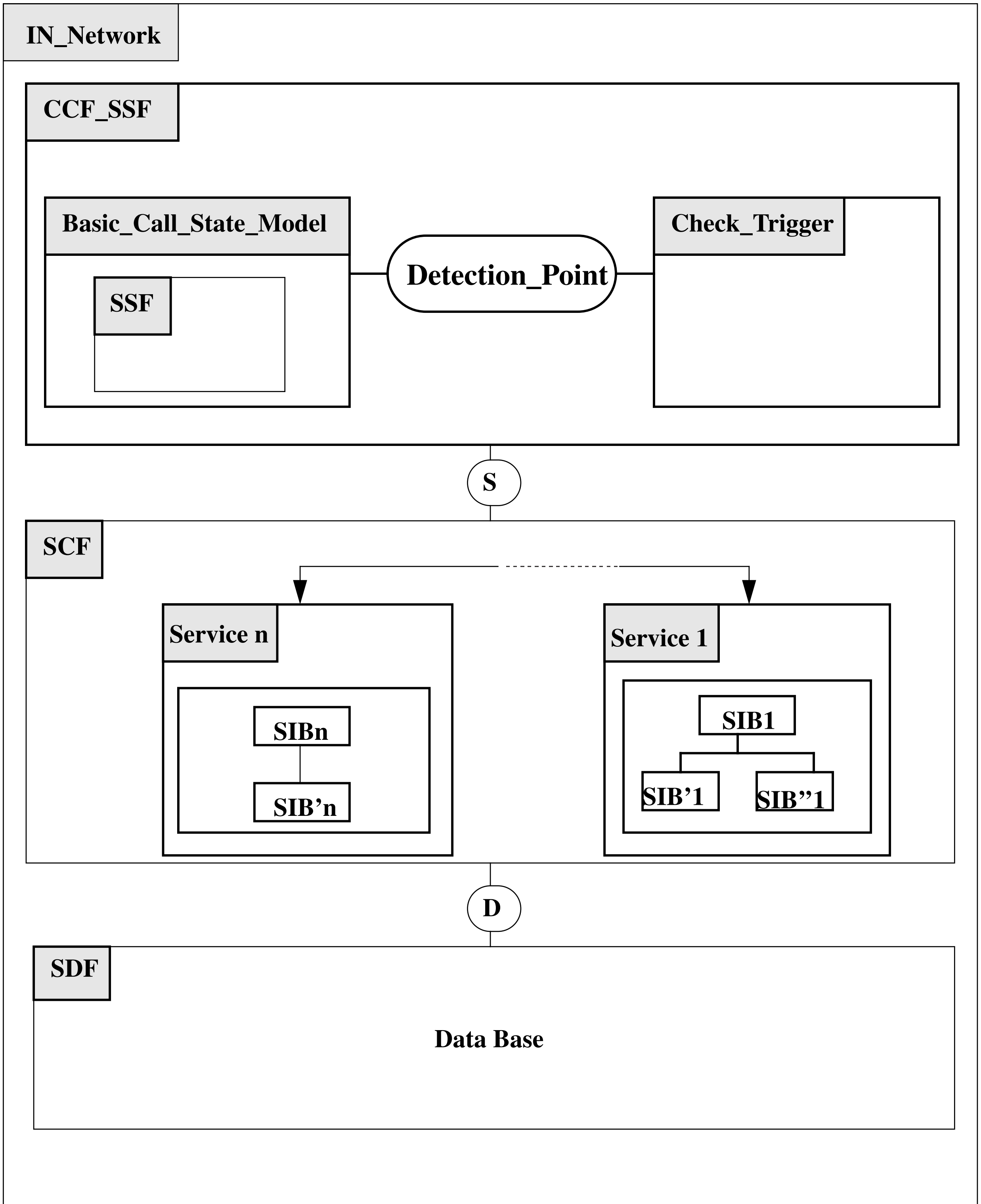


Figure 24: Detailed representation of process IN_Network

4.3 Chapter Summary

In this chapter we have described a LOTOS specification of IN call model and services as defined in the Distributed Functional Plane (DFP). The Functional Entities (FEs) that are involved in the establishment of a call/connection and the control of IN services such as the Call Control Function (CCF), the Service Switching Function (SSF), the Service Control Function (SCF) and the Service Data Function (SDF), as well as the communication between them are specified. IN services are specified using the concept of Service Independent building Blocks (SIBs) described in the INCM. The SIBs used to specify services are defined in IN CS1 [Q1213]. The specification is designed in such a way that independent specification and rapid introduction of services is provided, given that these are two of the main objectives of IN.

CHAPTER 5 Detecting Feature Interaction between IN services

In this chapter, we describe the approach adopted for detecting interactions between IN services and we illustrate it by three pair-wise feature interaction examples. For each example we give an informal description of the two services, followed by their formal specification and we show how we can detect interactions by applying the methodology.

In this section, the word *feature* is used denote both service and service feature.

5.1 Overview of the Detection Method

During the service creation process, a feature is described at several different levels of abstraction, from a high level view to implementation code. According to [BDCG89], interactions occurring at the level of abstract specification are called *logical interactions*, those occurring when the feature specification is mapped onto a network architecture are called *network interactions* and those occurring when the feature software is mapped onto an execution environment are called *implementation interactions*.

Since we are dealing with formal service specification, and we are abstracting from design and implementation details, we are addressing the detection of logical interactions between features. Clearly, this must be done as early as possible, otherwise it will propagate through all the service creation activities.

5.1.1 General Principle

Logical interaction between two or more features occurs when one or some of the requirements or assumptions, that must be satisfied when a feature is introduced in the network, is violated. Therefore, we develop a method based on expressing the feature requirements as properties in a property language and we say that interactions occur when these properties are not satisfied by the system description which is its formal specification [BZ92], [CoPi94].

More precisely, let S be a user-view specification of the basic IN system (without adding features), described in a formal specification language (LOTOS in our case), and let $F_1, F_2, ..., F_n$ be user-view specifications of n features.

We denote by $S \oplus F_1 \oplus F_2 \oplus ... \oplus F_n$, a formal specification of a system obtained by adding features F_i, $1 \leq i \leq n$, to the IN system, denoted by S.

Let $P_1, P_2, ..., P_n$ be n formulae expressing respectively the feature requirements of $F_1, F_2, ..., F_n$, in a suitable property language, and let $N \models P$ denote that a system specification N satisfies formula P, i.e. N is a model of P.

We say that there is interaction between features $F_1, F_2, ..., F_n$ if:

$$S \oplus F_i \models P_i,\ 1 \leq i \leq n$$

but

$$\neg\,(S \oplus F_1 \oplus F_2 \oplus ... \oplus F_n \models P_1 \wedge P_2 \wedge ... \wedge P_n) \quad \textbf{(1)}$$

This method depends heavily on careful and complete formalization of feature requirements. We will usually consider the case where $n=2$ since most interactions reveal themselves in contexts where two features only are active.

It should be noted here that these definitions do not attempt to characterize the feature interaction problem in its most general meaning. We have said before that this is quite difficult and perhaps impossible. Therefore, our definition is consciously a limitative one. In a two features context, we say that there is feature interaction where a second feature modifies the effects of an existing one, although this could be a desired result.

5.1.2 Properties of Features

In order to express in a clear and unambiguous manner the feature requirements and the general properties of the basic system, we need a formal property language. For this purpose we chose the branching time temporal logic CTL [BG93] which is well adapted for concurrent systems since it permits expression of precedence relations between events. A temporal logic language is defined over infinite sequences of states, representing execution states of the specification. Note that the semantics of CTL formulae are defined with respect to Kripke Structures (KS), however a LOTOS specification is seen as a Labelled Transition System (LTS). This does not present any problem since any LTS can be transformed into a KS [BG93]. In this transformation, every transition from a state S_1 to a state S_2 is transformed into a state in a KS.

Definition of CTL

Let *Prop* denote a set of atomic propositions. In the LOTOS context, atomic propositions are LOTOS actions. Intuitively, atomic proposition g is true if and only if we are in a state of the KS where action g is being offered. The set of CTL formulae is defined recursively as follows:

- Every atomic proposition g element of *Prop* is a CTL formula.

- if f_1 and f_2 are CTL formulae, then so are $(\neg f_1)$, $(f_1 \wedge f_2)$, $AX(f_1)$, $EX(f_1)$, $A[f_1 \vee f_2]$, $E[f_1 \vee f_2]$ where

$AX(f_1)$ means that f_1 holds in every immediate successor of the current state.

$EX(f_1)$ means that f_1 holds in some immediate successors of the current state.

$A[f_1 \vee f_2]$ means that for every computation path, starting at the current state, there exists a sequence of transitions satisfying f_2 at last, and f_1 for all the other transitions.

$E[f_1 \vee f_2]$ means that for some computation paths, starting at the current state, there exists a sequence of transitions satisfying f_2 at last, and f_1 for all the other transitions.

The following abbreviations are also used in writing CTL formulae:

$AG(f)$ means that f holds at every state for every future path; that is f holds *globally.*

$EF(f)$ means that f holds at every state for some future path.

$AF(f)$ means that f holds in the future along every path from the initial state; in other words f is *inevitable*.

$EF(f)$ means that f holds along some future paths from the initial state; that is f *potentially* holds.

Figure 25 describes the different CTL operators.

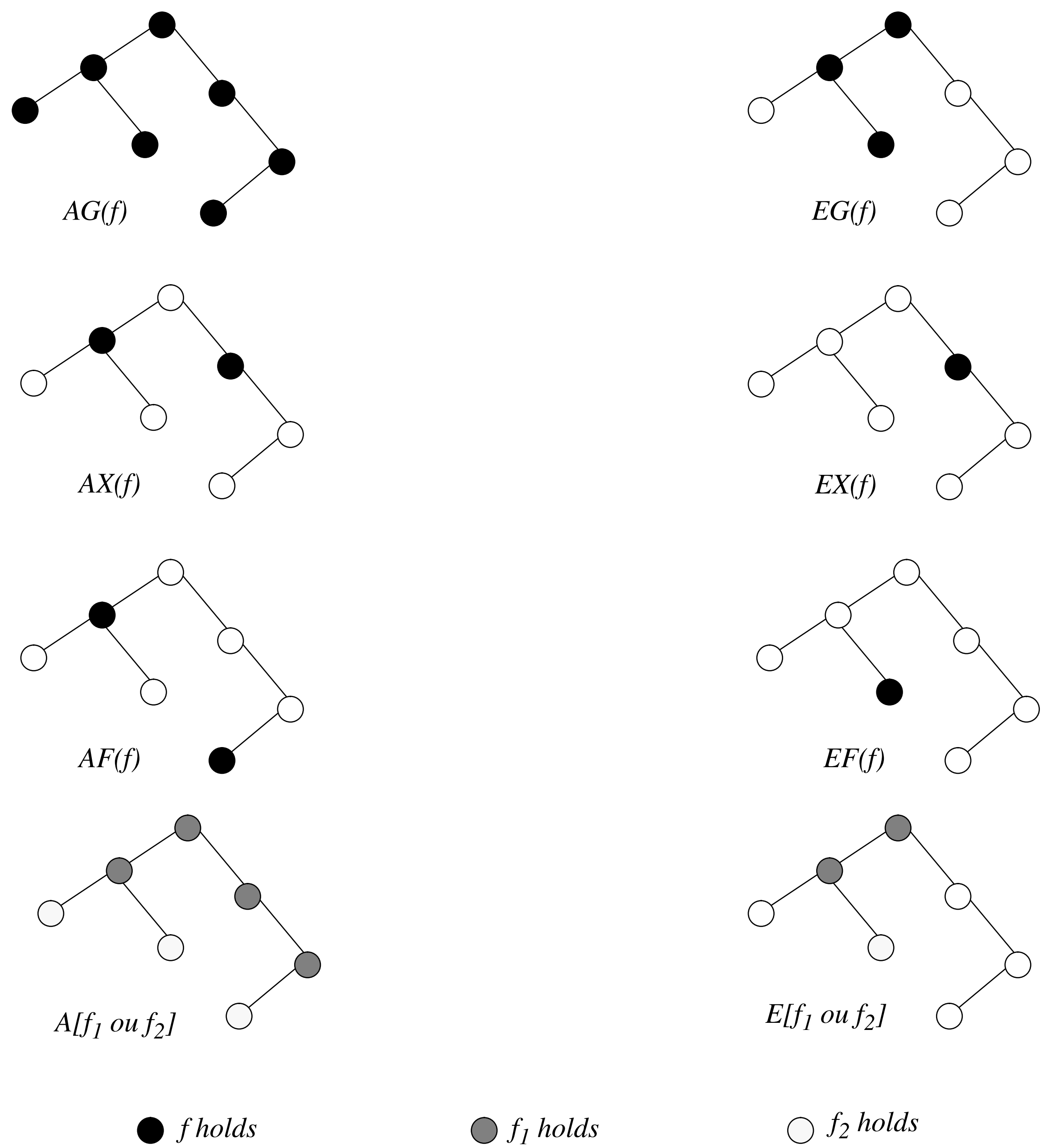


Figure 25: CTL Operators

As example of a CTL formula, if a user takes the phone off hook, it cannot perform another off hook until it hangs up. This property can be described by the following CTL formula:

$$AG\ (\ N\ \mathbf{?}adr\text{: } network\_address\ \mathbf{!}OffHookToCall\ \text{->}$$
$$E\ ((\neg\ (G\ \mathbf{!}adr\ \mathbf{!}OffHookToCall)) \vee (G\ \mathbf{!}adr\ \mathbf{!}HangUp))\ )$$

5.1.3 Verification Tool: Goal Oriented Execution

In the following, we call *trace* a sequence of observable actions that a LOTOS process can offer to the environment.

The presented method is based on the *goal oriented execution* tool developed within the LOTOS group of the University of Ottawa [HLS93], [Ha95]. The *goal oriented execution* allows one to look for execution traces according to several properties. In this type of execution, the user specifies an action to be reached, usually an action that is not immediately derivable. The system then proceeds in a sort of selective eager execution, being able to select traces likely to reach the action. These traces can be found with the help of a static analysis of the behavior expression. For example, if the behavior expression is: (a **;** b **; stop |||** b **;** c **; stop**) **[]** c **;** d **;** f **; stop** and the user wants to be given an (or all) execution trace(s) reaching *f*, then the *goal oriented execution* algorithm is able to see that the left-hand side of the behavior expression does not need to be expanded at all, because it does not contain action *f*. A considerable saving in computing time and space is obvious from the example.

Goal oriented execution allows also to define, instead of one action to be reached, a sequence of actions. The system proceeds to select traces that contain this sequence starting by the first action in the sequence. For example, if the behavior expression is: (a **;** c **; stop []** a **;** e **;** c **; stop []** a **; stop)** and if the sequence of actions to be reached is: [a, c], then the possible traces that can be selected by the system are *a* **;** *c* and *a* **;** *e* **;** *c*.

Events can be associated with actions in the sequence defining the goal to be reached. For example, if the sequence contains an action with gate name *a* and an offer value (**!**) x_1, the selected trace must contain the action with gate *a* and with the offer of value x_1. If the event associated with the action is an accept of a parameter (**?**), the system will instantiate all the possible values of that parameter.

An example of a goal to be reached is the following:

$$[a\ \mathbf{!}x_1\ \mathbf{?}x_2,\ b\sim,\ c]\ \backslash\backslash\ [e,\ f].$$

This goal is satisfied by all traces of actions starting by an action with gate name a, with an offer of the value x_1 and an acceptance of the value x_2, leading to the action represented by gate c (without any event), and having as intermediate action an action with gate name b with an arbitrary event (~). Traces must not include actions with gates e and f.

In addition, the tool has many characteristics that can speed up the search. In fact, the search can be guided by the user by setting limits for the number of instantiations of processes and by avoiding some branches of the corresponding LTS.

If some processes are instantiated recursively, leading to infinite LTS, GOE cannot guarantee the absence of a trace corresponding to the specified goal, because the goal limits the number of instantiations of processes.

Note that, after finding a trace, the tool will ask the user whether another one is desired.

To accelerate the search we can always guide it by adding some intermediate actions, that we know must exist in a trace satisfying the specified goal. For example, if we are looking for a trace leading to an action specifying a connection establishment between two network users, we can add in the goal an action where a user dials a number, since it is evident that before an establishment of a connection, a user must dial a number. If we want to exclude the search from some branches of the behavior tree, we can add some specific gates and exclude them from the search. Then, the search process will not go in those branches where these specific gates are inserted.

Following the principle described in section 5.2.1, an interaction occurs if one of the properties P_i is not verified with respect to the resulting system by adding the n features f_i to S. **(1)** in section 5.2.1 can be expressed by:

$$\exists\, P_i,\ 1 \leq i \leq n \text{ such that: } \neg\, (S \oplus F_1 \oplus F_2 \oplus \ldots \oplus F_n \models P_i)$$

A property P_i is not satisfied whenever there is a trace t_i in the specification describing a scenario which does not satisfy P_i. This means that t_i satisfies the property ($\neg\, P_i$).

Therefore, for each P_i, we construct a goal g_i which satisfies the property ($\neg\, P_i$) and we apply *Goal Oriented Execution* in order to see if a trace t_i satisfying g_i can be found. If t_i exists, then there is interaction.

Figure 26 describes the method.

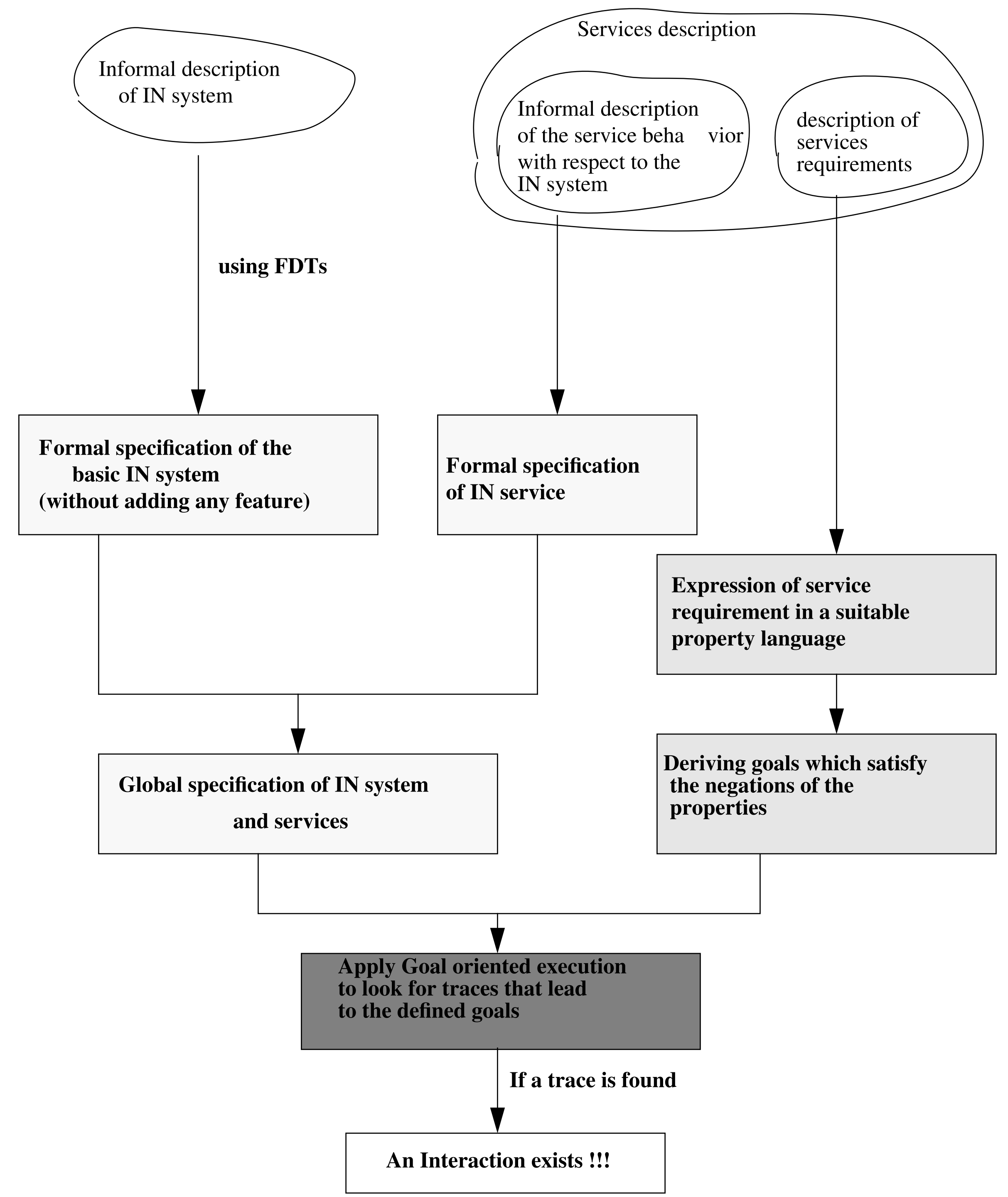


Figure 26: Methodology for detecting Feature Interaction

5.2 Application of the Method

In this section, we apply the method to three pair-wise combinations of features. For each feature, we give its informal description, followed by its formal specification, and then we show how the interaction is detected.

Interaction between features depends on the way these features are assigned to subscribers. For example, an interaction may exist between feature f_1 and feature f_2 if f_1 and f_2 are assigned to the same subscriber. On the other hand, it is possible that there is interaction between *f*2 and f_3 only if they are assigned to different subscribers. Therefore, to detect interaction, we should look at all possible assignments of features to subscribers. In the case of two features, we should look at the system when both features are assigned to the same subscriber, and when the two features are assigned to different subscribers. In our specification, the assignment of features to subscribers is done statically, i.e. a feature is specified independently and assigned to a subscriber by defining the arming conditions of the feature. As described in Chapter 4 section 4.2.1, the arming conditions are specified in the abstract data type by the operation: *trigger_armed: trigger_detection_point, network_address, service_name* -> *Bool* defined in the type *Trigger_Detection_Point*. In this operation, the parameters of sort *service_name* and *trigger_detection_point* represent respectively the name of the feature and the DP at which it must be invoked, and the parameter of sort *network_address* represents the network address of the feature subscriber. To assign a feature to a subscriber, we simply assign the value of its network address to the parameter of sort *network_address* defined in the operation above. Therefore, we can easily change the subscriber of the feature and assign it to another subscriber, by just changing the value of the subscriber's network address in the equations.

Some additional parameters must be associated to a feature in order to allow it to perform its functionalities. For example Call Forwarding Always is a feature that allows a subscriber to forward all his incoming calls to another line, then the network address to which calls are forwarded must be specified among the parameters. Screening list features such as Originating and Terminating Call Screening are defined with associated lists against which an identifier is screened, and depending on what elements are in the screening lists, an interaction may or may not occur. Therefore, to detect interactions we should also look at all the possible combinations of the feature's associated parameters. This would be very complex and even impossible in some cases where the number of associated data is quite large. However, by following the principle of equivalence partitioning known in software testing theory [Myers79] we can always reduce this complexity. With this principle, the different possible parameters manipulated within a system are partitioned into a finite number of equivalence classes such that one can reasonably assume that using an arbitrary value of a class is equivalent to a scenario using another value of that class. That is, if one scenario in an equivalence class detects an interaction, all other scenarios in the same equivalence class would be expected to detect that interaction. The identification of such equivalence classes is beyond the scope of this thesis. Therefore, in this chapter, we limit ourselves to scenarios which describe cases where interactions occurs and we only show how these interactions are detected by following our method.

At the validation stage of the specification, we are mainly interested in the verification of the system properties, and only the part of the system that involves these properties is relevant. For example, in the specification of the IN system, the checking for the arming criteria is done at each DP in order to check if a service must be invoked. This involves the instantiation of process *Check_Trigger* at each DP and an evaluation of guards and equations defining these criteria. However, if we are interested in the validation of the specified services only, we don't need to check for the arming criteria at every DP and we can limit checking to the DPs where these services are invoked. This can accelerate the validation process. As a result, some simplifications of the basic specification were carried out. They consist in modifying the specification in a way that the checking for the arming criteria is done only at the DPs where the specified services are invoked. further, checking is done only for these services.

5.2.1 Originating Call Screening and Call Forward Always

In this example we show how the interaction between the Originating Call Screening and Call Forward Always features is detected.

5.2.1.1 Informal Description of Originating Call Screening (OCS)

OCS is a feature that allows a subscriber to prevent outgoing calls to be made, according to a screening list for the creation and modification of which the subscriber is responsible. However, it can still be reached from subscribers whose telephone numbers are included in the list.

5.2.1.2 Formal Specification of OCS

The OCS feature is composed of the *Screen* SIB which performs a comparison of an identifier against a list to determine whether the identifier is in the list. As input data, this SIB needs the call parameters and a screen list indicator which identifies the list to be used for screening. Call parameters include network addresses of calling and called parties and the number dialled by the caller. Some other input data are defined in [Q1213] but they are not relevant to the formal specification since they are related to implementation details.

As described in Chapter 4 section 4.2.5.3, in order to perform its functionality, the *Screen* SIB (instantiated by a service process) needs to communicate with the SDF where the service related data are handled. The *Screen* SIB must send a message to the SDF indicating the type of operation required (consultation, modification or addition of data) and the data involved in this operation. The communication is specified by synchronization on gate *D* between process *Screen_SIB* which defines the *Screen* SIB and process *SDF*. Operation type and data are specified as parameters to be exchanged or matched by synchronization. A new type named *Operation_Name* of sort *operation_name* is specified. It defines the different operations that can be performed by the SDF.

Process *Screen_SIB* is defined as follows:

```
process Screen_SIB [S , G, N, D, DBRequest]
(ser: service_name , caller: network_address , dn: dialled_number , called: network_address) :
exit (bool) :=

 [ser eq OCS] ->
  (
    D !OCS !IsInScreenList !caller !called ;
    D !OCS !IsInScreenList ?result: bool ;
    exit (result)
  )
endproc (* end of process Screen_SIB *)
```

The *Screen* SIB can be used by different services that need to screen different lists and for each calling service, certain actions must be executed. For this reason, the variable *ser* of sort *service_name* referencing the invoked service is passed as parameter to the *Screen_SIB* process to indicate what actions must be executed. The value of the parameter of sort *service_name* referencing the Originating Call Screening feature is *OCS*.

The output of process *Screen_SIB* is a boolean variable which is set to *true* if the identifier is in the screening list and to *false* if not.

Processes *Screen_SIB* and *SDF* synchronize on gate *D* by executing the action specified in the former process as *D* **!***OCS* **!***IsInScreenList* **!***caller* **!***called*. This action offers four parameters: *OCS* of sort *service_name* indicating the service being processed, *IsInScreenList* of sort *operation_name* indicating the type of operation to be performed by SDF and *caller* and *called* which represent the call parameters needed to perform the operation. When process *SDF* perform the required operation, it sends back the result to process *Screen_SIB*. This is done by synchronizing on gate *D* and executing the action defined in process *Screen_SIB* as *D* **!***OCS* **!***IsInScreenList* **?***result: bool*. Process *SDF* is defined as a choice between different operations needed by the different SIBs. It is described as follows:

```
process  SDF [D , DBRequest]  : noexit:=

 D !OCS  !IsInScreenList    ?caller: network_address     ?called: network_address    ;
      (
        [called NotIn ScreenList(caller) ]       ->
          (
            D !OCS  !IsInScreenList    !false ;
            SDF[S , G, N, D, DBRequest]
          )
        []
        [called IsIn ScreenList(caller)]       ->
          (
            D !OCS  !IsInScreenList    !true ;
            SDF[S , G, N, D, DBRequest ]
          )
      )
[]
  (* ... definition of other operations ...*)

endproc   (* end of process SDF *)
```

Process SDF reinstantiates itself to allow other synchronizations with other service processes. This process can always be modified when new operations needed by other SIBs must be defined. The modification consists in adding a choice with the new operation actions.

The list to be screened is created and updated by the feature subscriber. It is defined in the type *List* by the operation: *OCS_ScreenList: network_address -> list*, which defines for each feature subscriber the OCS list.

The process defining the OCS feature is defined as follows:

```
process Originating_Call_Screening [S , G, N, D, DBRequest]
(caller: network_address , dn: dialled_number , called: network_address) : noexit :=

  Screen_SIB[S , G, N, D, DBRequest](OCS , caller , dn, called) >> accept result: bool in
  (
     [result eq true] ->
        (
         S !PIC_Analyse_Info !caller !dn !called ;
         SCF[S , G, N, D, DBRequest]
        )
    []
     [result eq false] ->
        (
         S !PIC_O_Exception !caller !dn !called ;
         SCF[S , G, N, D, DBRequest]
        )
  )
endproc (* end of process Originating_Call_Screening *)
```

Process *Originating_Call_Screening* instantiates process *Screen_SIB* then, depending on the output of the latter process, it indicates the return point at which the call must continue. This is done, as explained in Chapter 4, by synchronizing with process *SSF* on gate *S*. If the called was found in the screening list, the caller must abandon the call and hang up, this is done by synchronization with process *SSF* on action *S* **!***PIC_O_Exception* **!***caller* **!***dn* **!***called*. If the called does not appear in the screening list, the call must continue and move to the next point defined by the PIC *Analyse_Info*. This is done by performing action *S* **!***PIC_Analyse_Info* **!***caller* **!***dn* **!***called*.

The verification of the authority to originate a call is checked by the OCS feature when a subscriber finishes dialling a number. Thus, the service must be invoked at the DP *Collected_Info* after the dialling string is collected. Figure 27 describes the invocation of the OCS feature.

In our specification we suppose that the user represented by the network address *adr1* has subscribed to the OCS feature and has the user represented by the network address *adr2* in his screening list.

The arming condition is defined by the operation *trigger_armed* of sort *bool* defined in the type *Trigger_Detection_Point* as:

trigger_armed(t, adr, ser_name) =

((t eq Collected_Info) and (adr eq adr1) and (serv_ind eq OCS)) where *t* is of sort *trigger_detection_point*, *adr* is of sort *network_address* and *ser* is of sort *service_name*.

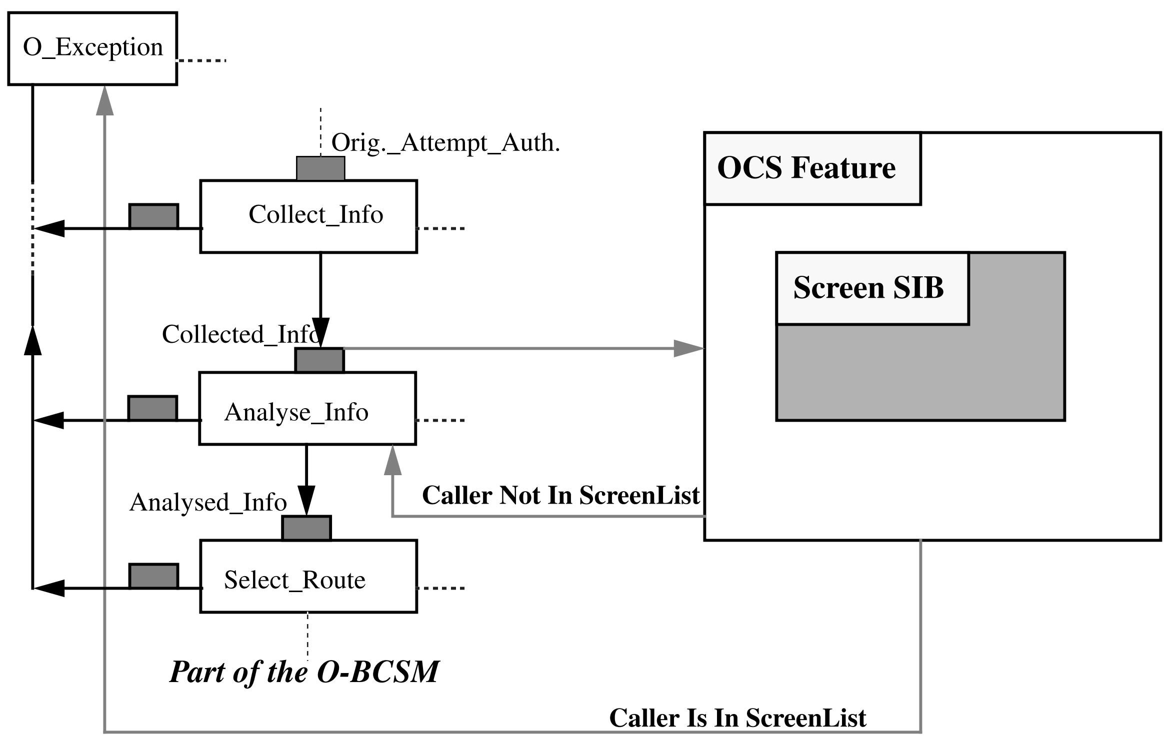


Figure 27: The invocation of the OCS feature

5.2.1.3 Informal Description of Call Forward Always (CFA)

CFA is a feature that allows a subscriber to forward all incoming calls to another telephone number. With this service, all calls destined to the subscriber's number are redirected to the new phone number, no matter what the called party line status is.

5.2.1.4 Formal Specification of CFA

The CFA service is composed of the *Translate* SIB which, as defined in [Q1213], translates input information and provides output information, based on various other input parameters. These parameters include the file indicator which indicates what file contains the translation data, and the call parameters (described by the caller and the called network addresses and the number dialled by the caller). The *Translate* SIB is specified by the process *Translate_SIB*. Its inputs are the service name and the call parameters. The outputs are the new values of the call parameters after performing the translation. The translation can affect one of the call parameters depending on what feature is being processed.

The process *Translate_SIB* is defined as follows:

```
process Translate_SIB [S, G, N, D, DBRequest]
(ser: service_name, caller: network_address, dn: dialled_number, called: network_address)
:exit (network_address, dialled_number, network_address) :=

  [ser eq CFA] ->
  (
     D !CFA !GetForwardedAddress !called ;
     D !CFA !GetForwardedAddress ?forwarded_called_address: network_address ;
    exit (caller, dn, forwarded_called_address)
  )

endproc (* end of process Translate_SIB *)
```

When the CFA feature is invoked, processes *Translate_SIB* and *SDF* synchronize on gate *D* by performing action *D* **!***CFA* **!***GetForwardedAddress* **!***called. CFA* is a parameter of sort *service_name* referencing CFA feature. *GetForwrdedAddress* is a parameter of sort *operation_name* indicating the type of operation to be performed at the SDF and which consists in determining the forwarded network address of the CFA subscriber. The parameter called of sort *network_address* indicates the network address of the subscriber. This new operation must be added to process *SDF* as follows:

```
process SDF [D , DBRequest] : noexit:=

  ...
 []
   D !CFA !GetForwardedAddress ?called: network_address ;
   D !CFA !GetForwardedAddress !get_CFAddress(called) ;
   SDF[D , DBRequest]
[]
  ...
endproc (* end of process SDF *)
```

The operation *get_CFAddress: network_address -> network_address* is defined in the datatypes to specify the network address to which calls are forwarded.

The CFA feature is specified by the process *Call_Forward_Always* defined as follows:

```
process Call_Forward_Always [S , G, N, D, DBRequest]
(caller: network_address , dn: dialled_number , called: network_address) : noexit:=
(
  Translate_SIB[S , G, N, D,DBRequest] (CFA, caller , dn, called) >> accept
  caller: network_address , dn: dialled_number , new_called_address: network_address in
   (
    S !DP_Term_Attempt !caller !dn !new_called_address ;
    SCF [S , G, N, D, DBRequest]
   )
 )
endproc (* end of process Call_Forward_Always *)
```

Process *Call_Forward_Always* instantiates process *Translate_SIB* in order to determine the new network address to which the call must be forwarded. Then, the call must continue at DP *Term_Attempt* since a new attempt to call a new number is taking place. This is described in the process definition by action *S* **!***DP_Term_Attempt* **!***caller* **!***dn* **!***new_called_address* on which process *SSF* and process *Call_Forward_Always* synchronize.

The CFA feature is invoked when an indication of incoming call is received by the called line. Thus, the DP *Term_Attempt* must be armed in order to launch this service (see figure 28). In our specification, we suppose that the subscriber defined by the network address *adr3* subscribes to the CFA service and its incoming calls are forwarded to the subscriber *adr2*. The arming condition is defined by the following equation:

trigger_armed(t, adr, ser_name) =

((t eq Term_Attempt) and (adr eq adr3) and (ser_name eq CFA));

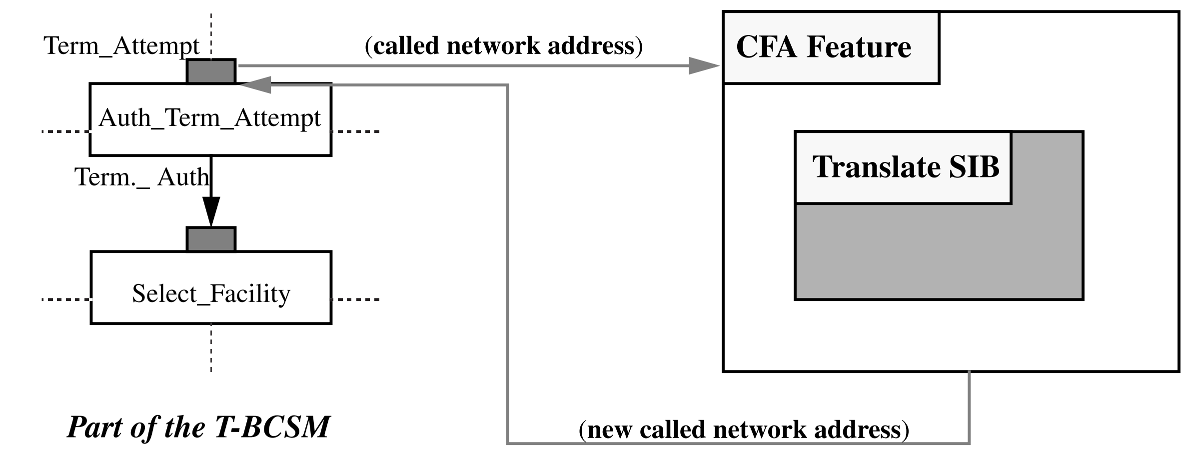


Figure 28: The invocation of the CFA feature

5.2.1.5 Detection of interaction between OCS and CFA

<u>Expressing the requirement of OCS in CTL</u>

In this example, we suppose that *adr1* subscribes to the OCS feature. Then, it cannot be connected to any user in the screening list. This means that whenever *adr1* picks up the phone in order to make a call, it cannot reach *adr2* (since *adr2* is the only user in the screening list). This property is described as follows:

***P1**: AG((N !adr1 !OffHookToCall) -> ¬(EG(N !adr1 !RingsFrom !adr2)))*

<u>Expressing the requirement of CFA in CTL</u>

In our example, *adr3* subscribes to CFA feature, and forwards all the calls to *adr2*. Then, if any user tries to call *adr3*, the call will be forwarded to *adr2* and a connection between *adr2* and the calling party takes place. This property is described in CTL as:

***P2**: AG((Detection_Point !Term_Attempt ?caller !adr3) ->*
AF(!Detection_Point !O_Term_Seized !caller !adr2)).

<u>*Deriving Goals that satisfy the negation of the OCS property*</u>

(¬ *P1*) is satisfied if there exists a trace that starts with action *N* !*adr1* !*OffHookToCall* and leads to the action *N* !*adr1* !*RingsFrom* !*adr2*. This can be expressed by the goal:

G1: [*N* !*adr1* !*OffHookToCall*, *N* !*adr1* !*RingsFrom* !*adr2*].

After applying *Goal Oriented Execution*, a trace satisfying goal ***G1*** has been found (trace 1).

1-N !adr1: network_address !OffHookToCall: Signal [true] **line(s): [1198,681,547]**

2-Detection_Point !Orig_Attempt: trigger_detection_point !adr1: network_address !null: network_address **line(s): [703]**

3-G !adr1: network_address !GetTone: Signal **line(s): [560,711]**

4-Detection_Point !Orig_Attempt_Auth: trigger_detection_point !adr1: network_address !null: network_address **line(s): [767**]

5-G !adr1: network_address !Dials: Signal ?dn,dn=num3: dialled_number **line(s): [571,775]**

6-Detection_Point !Collected_Info: trigger_detection_point !adr1: network_address !adr3: network_address **line(s): [804]**

7-S !OCS: service_indicator !adr1: network_address !num3: dialled_number !adr3: network_address **line(s): [1389,1215]**

8-D !OCS:service_indicator !IsInScreenList:operation_name !adr1:network_address !adr3:network_address **line(s):[1463,1451]**

9-D !OCS: service_indicator !IsInScreenList: operation_name !false: Bool **line(s): [1467,1452]**

10-**i/exit** (false:Bool) **line(s): [1453]**

11-S !PIC_Analyse_Info:state_name !adr1:network_address !num3:dialled_number !adr3:network_address **line(s): [1432,1219]**

12-Detection_Point !Analysed_Info: trigger_detection_point !adr1: network_address !adr3: network_address **line(s): [846]**

13-Detection_Point !Route_Selected: trigger_detection_point !adr1: network_address !adr2: network_address **line(s): [895]**

14-Detection_Point !Orig_Auth: trigger_detection_point !adr1: network_address !adr3: network_address **line(s): [865]**

15-Detection_Point !Term_Attempt: trigger_detection_point !adr1: network_address !adr3: network_address **line(s): [885]**

16-S !CFA: service_indicator !adr1: network_address !num3: dialled_number !adr3: network_address **line(s): [1394,1215]**

17-D !CFA: service_indicator !GetForwrdedAddress: operation_name !adr3: network_address **line(s): [1479,1418]**

18-D !CFA: service_indicator !GetForwrdedAddress: operation_name !adr2: network_address **line(s): [1480,1419]**

19-**i/exit** (adr1: network_address, num3: dialled_number, adr2: network_address) **line(s): [1420]**

20-S !DP_Term_Attempt:state_name !adr1:network_address !num3:dialled_number !adr2:network_address **line(s): [1405,1223]**

21-Detection_Point !Term_Attempt: trigger_detection_point !adr1: network_address !adr2: network_address **line(s): [885]**

22-Detection_Point !Term_Auth: trigger_detection_point !adr1: network_address !adr2: network_address **line(s): [927]**

23-DBRequest !Consult: dboperations !ADD(adr1, empty): List **line(s): [1194,935]**

24-Detection_Point !Term_Res_Avail: trigger_detection_point !adr1: network_address !adr2: network_address **line(s): [966]**

25-N !adr2: network_address !RingsFrom: Signal !adr1: network_address **line(s): [1206,974,648]**

Trace1: Trace showing interaction between OCS and CFA

This trace shows a scenario which violates the property of the OCS feature. Therefore, an interaction between OCS and CFA features exists. In this scenario, *adr1* dials phone number *num3* corresponding to *adr3* (line 5), then the OCS feature is invoked to check whether the called number is in the screening list (line 7). *Adr3* was not found in the list, and the processing of the call continue (line 11 to line 14). Then, the CFA feature is invoked (line 15) and the call is forwarded to *adr2* (line 19). A connection is then established between *adr1* and *adr2* (line 25). Figure 29 describes how the property of the OCS feature is violated by the introduction of the CFA features.

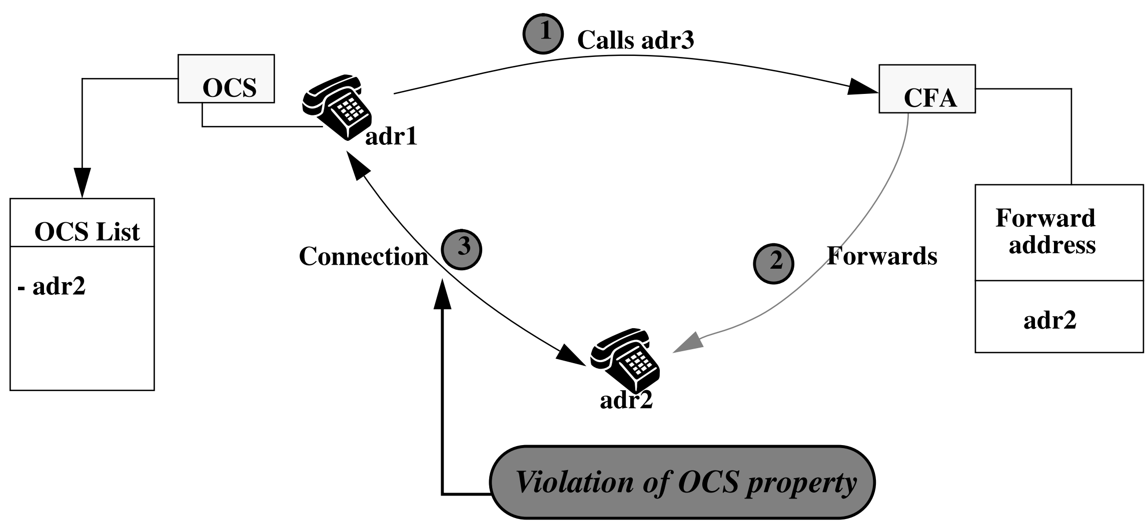


Figure 29: Interaction between OCS and CFA

5.2.2 Originating Call Screening and Abbreviated Dialling

In this example we show how the interaction between Originating Call Screening and Abbreviated Dialling features is detected. The OCS service was described in section 5.3.2, then only the specification of ABD feature is described in this section.

5.2.2.1 Informal Description of Abbreviated Dialling (ABD)

According to [Q1213], ABD is an originating line feature that allows the definition of an abbreviated dialling digit sequence to represent an actual dialling digit sequence. For example, business subscribers can dial others in their company using, e.g. only four digits even if the calling user’s line and the called user’s line are served by different switches. This extends switch based intercom calling beyond the switch boundary.

5.2.2.2 Formal Specification of ABD

The ABD feature uses the *Translate* SIB to translate the abbreviated digit sequence dialled by a calling party into a real phone number. Then, a message must be sent to the SDF in order to ask for the number corresponding to the dialled string. Similarly to the operation defined for the CFA feature, a new operation *GetTranslatedNumber* of sort *operation_name* is defined, and actions to be performed are added to the *Translate_SIB* process as follows:

```
process Translate_SIB [S , G, N, D, DBRequest]
(ser: service_name , caller: network_address , dn: dialled_number , called: network_address)
:exit (network_address , dialled_number , network_address) :=

  [ser eq CF A] ->
        (* ... see previous definition of the process *)
  []
  [ser eq ABD] ->
     (
        D !ABD !GetTranslatedNumber !dn ;
        D !ABD !GetTranslatedNumber ?TranslatedNumber: dialled_number ;
        exit (caller , dn, get_address(TranslatedNumber))
      )
endproc (* end of process Translate_SIB *)
```

Process *SDF* is modified by adding the following actions:

```
process SDF [D , DBRequest] : noexit:=

  ...
 []
  D !CFA !GetTranslatedNumber ?dn: dialled_number ;
  D !CFA !GetTranslatedNumber !get_translated_number(dn) ;
  SDF[D , DBRequest]
[]
  ...

endproc (* end of process SDF *)
```

Translated numbers are defined in the datatypes by the operation:

get_translated_number: *dialled_number -> dialled_number* which defines for each abbreviated number the corresponding phone number.

The process *Abbreviated_Dialling* specifying the ABD feature is defined as follows:

```
process Abbreviated_Dialling[S , G, N, D, DBRequest]
(caller: network_address , dn: dialled_number , called: network_address) : noexit:=
(
    Translate_SIB[S , G, N, D, DBRequest](ABD , caller , dn, called) >> accept
    caller: network_address , new_dn: dialled_number , called: network_address in
       (
        S !PIC_Select_Route !caller !dn !new_called_address ;
        SCF [S , G, N, D, DBRequest]
       )
)
endproc (* end of process Abbreviated_Dialling *)
```

After analysing the collected information (dialled string), the ABD is launched in order to translate the abbreviated number, thus, the DP *Analysed_Info* must be armed. In addition, we suppose in our specification that the network subscriber defined by the network address *adr1* subscribes to ABD feature. Then, the arming conditions are defined as follows:

trigger_armed(t, adr, ser_name) =

((t eq Analysed_Info) and (adr eq adr1) and (ser_name eq ABD));

Process *Abbreviated_Dialling* instantiates process *Translates_SIB* which gives as output the new phone number to which the call must be routed. Then, the processing of the call continues at the PIC *Select_Route*.

Figure 30 describes the invocation of ABD feature.

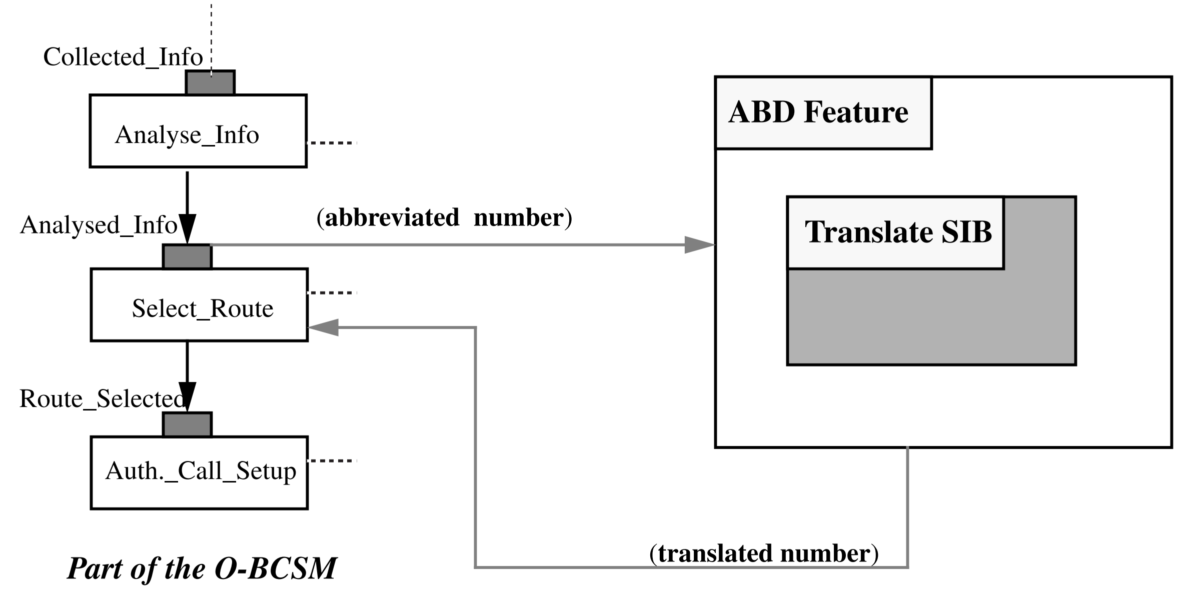


Figure 30: The invocation of ABD feature

5.2.2.3 Detection of interaction between OCS and ABD

Expressing the requirement of OCS in CTL

In this example, we suppose that *adr1* subscribes to the OCS feature, and has *adr2* in his screening list. Then, similarly to the example of section 5.3.1, the property of the feature is as follows:

***P1**: AG((N !adr1 !OffHookToCall) -> ¬(EG(N !adr1 !RingsFrom !adr2)))*

Expressing the requirement of ABD in CTL

In this example, we suppose that *adr1* also subscribes to the ABD feature, and uses two abbreviated numbers: *num4* corresponding number *num2*, and *num5* corresponding to number *num3*. No network addresses are associated to these abbreviated numbers since they don't represent real phone numbers and the basic network does not know how to transform them into network addresses. Only the ABD feature can translate them into real phone numbers using a predefined list of abbreviated numbers.

This feature induces the fact that if a subscriber dials an abbreviated number (*num4 or num5*), he will be connected to the corresponding line defined by the corresponding phone number. This property is described in CTL as:

P3*: AG(*

((N !adr1 !Dials !num4) ->

AF(N !adr1 !RingsFrom !get_address(get_translated_number(num4)))

or

((N !adr1 !Dials !num5) ->

AF(N !adr1 !RingsFrom !get_address(get_translated_number(num5)))

)

<u>Deriving Goals that satisfy the negation of the property of OCS</u>

As described in section 5.3.1, (¬ *P1*) is satisfied if there exists a trace that starts with the action *N* !*adr1* !*OffHookToCall* and leads to the action *N* !*adr1* !*RingsFrom* !*adr2*, which can be expressed by the goal:

G1: [*N* !*adr1* !*OffHookToCall*, *N* !*adr1* !*RingsFrom* !*adr2*].

After applying *Goal Oriented Execution*, a trace satisfying ***G1*** was found (trace 2). Then, an interaction between the OCS and ABD features exist.

1-N **!**adr1: network_address **!**OffHookToCall: Signal [true] **line(s): [1238,699,565]**

2-Detection_Point **!**Orig_Attempt: trigger_detection_point **!**adr1: network_address **!**null: network_address **line(s): [721]**

3-G **!**adr1: network_address **!**GetTone: Signal **line(s): [578,729]**

4-Detection_Point **!**Orig_Attempt_Auth: trigger_detection_point **!**adr1: network_address **!**null: network_address **line(s): [785]**

5-G **!**adr1: network_address **!**Dials: Signal **?**dn,dn=num4: dialled_number **line(s): [589,793]**

6-Detection_Point **!**Collected_Info: trigger_detection_point **!**adr1:network_address **!**null: network_address **line(s): [822]**

7-S **!**OCS:service_indicator **!**adr1: network_address **!**num4: dialled_number **!**null: network_address **line(s): [1432,1255]**

8-D **!**OCS:service_indicator **!**IsInScreenList:operation_name **!**adr1:network_address **!**null:network_address **line(s): [1506,1494]**

9-D **!**OCS: service_indicator **!**IsInScreenList: operation_name **!**false: Bool **line(s): [1510,1495]**

10-**i/exit** (false:Bool) **line(s): [1496]**

11-S **!**PIC_Analyse_Info:state_name **!**adr1:network_address **!**num4:dialled_number **!**null:network_address **line(s): [1475,1258]**

12-Detection_Point **!**Analysed_Info: trigger_detection_point **!**adr1: network_address **!**null: network_address **line(s): [864]**

13-S **!**ABD: service_indicator **!**adr1: network_address **!**num4: dialled_number **!**null: network_address **line(s): [1437,1255]**

14-D **!**ABD: service_indicator **!**GetTranslatedNumber: operation_name **!**num4: dialled_number **line(s): [1522,1461]**

15-D **!**ABD: service_indicator **!**GetTranslatedNumber: operation_name **!**num2: dialled_number **line(s): [1523,1462]**

16-**i/exit** (adr1: network_address, num4: dialled_number, adr2: network_address) **line(s): [1463]**

17-S **!**PIC_Select_Route: state_name **!**adr1:network_address **!**num4:dialled_number **!**adr2:network_address **line(s): [1448,1262]**

18-Detection_Point **!**Route_Selected: trigger_detection_point **!**adr1: network_address **!**adr2: network_address **line(s): [895]**

19-Detection_Point **!**Orig_Auth: trigger_detection_point **!**adr1: network_address **!**adr2: network_address **line(s): [915]**

20-Detection_Point **!**Term_Attempt: trigger_detection_point **!**adr1: network_address **!**adr2: network_address **line(s): [935]**

21-Detection_Point **!**Term_Auth: trigger_detection_point **!**adr1: network_address **!**adr2: network_address **line(s): [967]**

22-DBRequest **!**Consult: dboperations **!**ADD(adr1, empty): List **line(s): [1234,975]**

23-Detection_Point **!**Term_Res_Avail: trigger_detection_point **!**adr1: network_address **!**adr2: network_address **line(s): [1006]**

24-N **!**adr2: network_address **!**RingsFrom: Signal **!**adr1: network_address **line(s): [1246,1014,666]**

Trace 2: Trace showing the interaction between OCS and ABD

This trace shows a scenario which violates the property of the OCS feature. In this scenario *adr1* dials the abbreviated number *num4* corresponding to *adr2* (line 5), then the OCS feature is invoked to check whether the called number is in the screening list (line 7). The abbreviated number does not correspond to any network address (in the specification, it is represented by null), thus the processing of the call continues (line 11). Then, the ABD feature is invoked (line 13) and the abbreviated number is translated into its associated phone number which is *num2*. The processing of the call continues with the new call parameters and connection is then established between *adr1* and *adr2* (line 24).

Figure 31 describes how the property of OCS is violated by the introduction of the ABD.

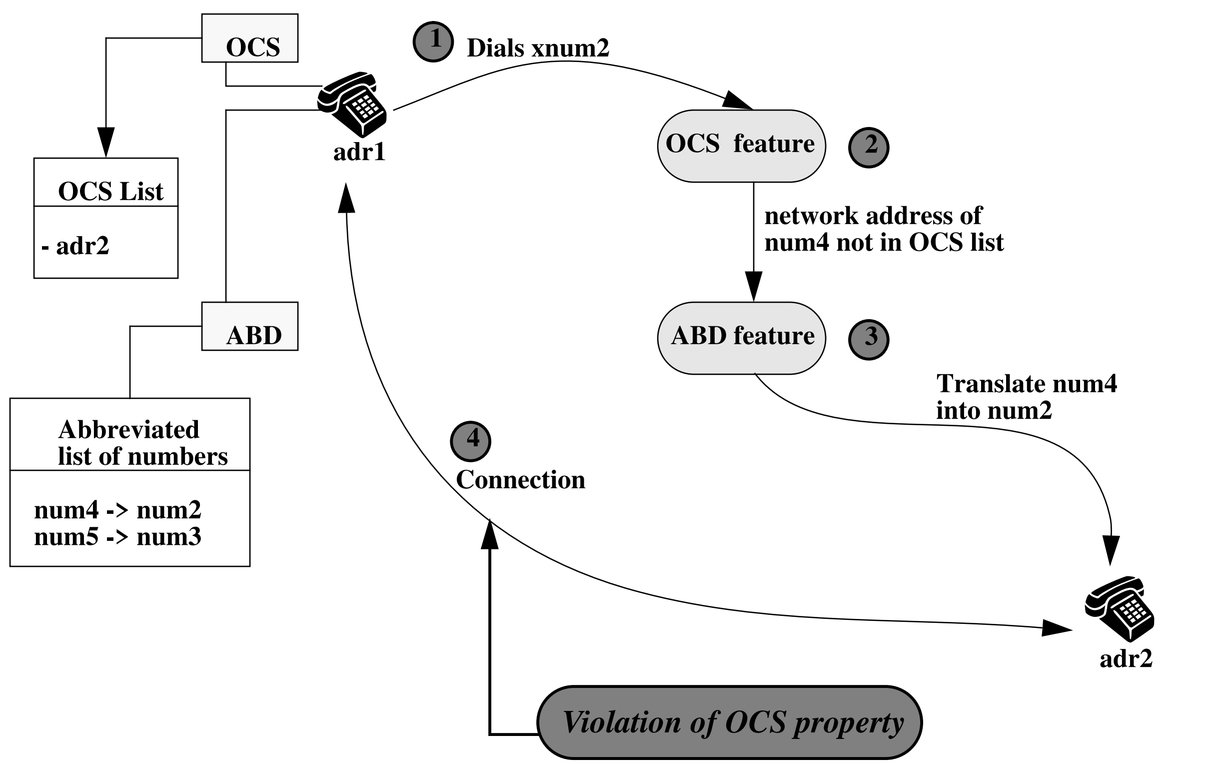


Figure 31: Interaction between OCS and ABD

It should be noticed that if the order of invocation of OCS and ABD features is reversed, this case of interaction will not occur.

5.2.3 Security Screening and Call Forward Always

In this example we show how the interaction between Security Screening (SS) and Call Forward Always (CFA) features is detected. The CFA feature was described in section 5.3.2, then only the specification of CFA feature is described in this section.

5.2.3.1 Informal Description of Security Screening (SS)

This feature enables a subscriber to protect his line by a user defined key. The feature asks the calling party to dial a PIN code, which allows the verification of the caller identity before giving the access permission to the subscriber's line. In other words, a subscriber can be called successfully only by subscribers who know the code.

5.2.3.2 Formal Specification of Security Screening (SS)

SS feature is composed of three SIBs. The *User Interaction* SIB, the *Verify* SIB and the *Screen* SIB. *User Interaction* SIB allows information to be exchanged between the network and a call party, where a call party can have either a calling or a called role. It provides a call party with information such as announcements (e.g. customized or generic audio message, network progression tones), and/or collects information from a call party such as audio messages or sequences of digits representing PIN codes or string text. The input parameters are of two types: the announcement parameters and the collect information parameters. The former type of parameters specifies the announcement identifier (which announcement is to be sent), the repetition option (if the announcement is to be repeated), the repetition interval and the duration. The latter type of parameters stands for the control values for user entered information such as the minimum and maximum number of characters and the inter-character waiting time.

However, all these parameters are related to implementation details, and for the specification, we need only to describe the fact that an interaction has occurred between a call party and the network. Such interaction can be specified by a synchronization on a gate *G* between the process *User_Interaction* defining this SIB and the process describing the call party involved in the interaction (*caller* or *called* processes).

When the SS feature is being processed, the caller party is asked to enter a PIN code in order to be connected to the called party. This event is specified by the action *G* **!***caller* **!***EnterPinCode* **?***entered_code: code* on which process *User_Interaction* and process *Caller* synchronize, and where *code* is the sort name of the type *Code* defined to describe the different codes and digit strings that must be entered by a call party after an announcement from the network.

If the caller enters a wrong PIN code, a message must be announced by the network to notify the caller that the entered code is not valid. This is specified by the action *G* **!***caller* **!***InvalidCodeAnnouncement*. The process *User_Interaction* is then defined as follows:

```
process User_Interaction [S, G, N, D, DBRequest]
(ser: service_name, caller: network_address, dn: dialled_number, called: network_address)
:exit (code) :=
(
 [ser eq SS] ->
  (
  G !caller !EnterPinCode ?entered_code: code ;
  exit (entered_code)
  []
  G !caller !InvalidCodeAnnouncement ;
  exit (no_code) (*no_code is a value of sort code indicating that an invalid code has been entered *)
  )
)
endproc (* end of process User_Interaction *)
```

The two actions specified above must be added to process *Caller*. The caller is asked to enter the PIN code after dialing a number of a subscriber to the SS feature. Then, in addition to the next possible actions a calling party can execute (already defined in the process *Caller*) when it dials a number, it can get an announcement to enter a code. This is specified by instantiating the process *Caller_GetsEnterCodeAnnouncement* described below in the process *Caller_Dials* (defined in Chapter 4) as a choice after executing the action *G* **!***caller* **!***dials* **?***dn: dialled_number* (see figure 32).

```
process Caller_GetsEnterCodeAnnouncement [G, N, DBRequest]
(caller: network_address) : noexit:=

G !caller !EnterVpnCode ?entered_code: code ;
 (
   Caller_GetsRingTone [G , N](caller)
   []
   G !caller !InvalidCodeAnnouncement ; User_HangsUp [G, N](caller)
   []
   User_HangsUp [G, N](caller)
 )
endproc (* end of process Caller_GetsEnterCodeAnnouncement *)
```

The information collected from a call party after an announcement from the network is of type *Code*. It can represent PIN codes as well as some other types of codes. Each sub-type of the type *Code* is defined by an operation which indicates all its different values. For example, the operation which specifies the PIN codes is defined in the definition of the type code as: *IS_PIN: code -> bool*. This operation takes a value of sort code and indicates whether it is a PIN code by defining an image of sort bool. If the image has a value *true*, then the value is a PIN code. If not, the value does not represent a PIN code.

The *Verify* SIB provides the confirmation that received information is syntactically consistent with the expected form of such information. This SIB normally follows the *User Interaction* SIB when information has been collected from a call party. The input required by this SIB includes the format of the collected information which depends on the processed feature.

The *Verify* SIB is specified by the process *Verify_SIB* defined as follows:

```
process Verify_SIB [S , G, N, D, DBRequest]
(ser: service_name , entered_code: code , caller: network_address , dn: dialled_number ,
called: network_address) : exit (bool) :=

 [ser eq SS] ->
    (
      [IS_PIN(entered_code) eq true]    -> exit (true)
     []
      [IS_PIN(entered_code) eq false]   -> exit (false)
    )

endproc (* end of process V erify_SIB *)
```

After verifying whether the syntax of the collected information correspond to the syntax of a PIN code, the validity of the code must also be verified. In fact, the feature must check if the PIN code entered by a user corresponds to the PIN code defined by the feature subscriber. This is done by the *Screen* SIB which verifies whether an identifier belongs to a given list or not.

One can think why the verification of the PIN code validity is not done directly after collecting the information from the caller party, i.e. without verification of the syntax. This can be explained by the fact that the process of verification provided by the *Screen* SIB needs a consultation of a data base containing the PIN code list of all the subscribers, which can be very costly in time if the data base is a remote one. If the syntax of the entered code is wrong, the data base consultation is not needed.

Since the PIN code list is maintained by the SDF, the verification of the entered PIN code is done by process *SDF*. Processes *Screen_SIB* must communicate the parameters to be verified by process *SDF*, which sends back the verification result after performing the appropriate operation. Then a new operation name *VerifyPinValidity* of sort *operation_name* is defined. When both processes synchronize on gate *D* by offering this operation parameter, process *SDF* checks for the validity and sends back the result to process *Screen_SIB*. The entered code is passed as parameter to process *Screen_SIB* since it is provided by process *User_Interaction*.

Process *Screen_SIB* is modified by adding the appropriate actions that must be performed when the SS feature is activated. It becomes as follows:

```
process Screen_SIB [S , G, N, D, DBRequest]
(ser: service_name , entered_code: code , caller: network_address , dn: dialled_number ,
 called: network_address) : exit (bool):=
 ...
[]
[ser eq SS] ->
   ( D !SS !VerifyPinValidity !called !entered_code ;
     D !SS !VerifyPinValidity ?validity: bool ;
     exit (validity))
[]
...
endproc (* end of process Screen_SIB *)
```

The subscriber's PIN codes are defined by the operation *PIN_List: network_address -> code* specified in the definition of type *code*, operation which associates for each SS subscriber the PIN code that must be entered by the caller. In addition, the code entered by the caller is added to the parameter list of *Screen_SIB* process. Process SDF is modified by adding the necessary operation to perform PIN code verification by adding the following actions:

```
process  SDF [D , DBRequest]  : noexit:=

  ...
[]
  D !SS !VerifyPinV alidity ?called: network_address   ?entered_code: code ;
       (
        [PIN_List (called) eq entered_code ]    ->
           (
            D !SS !VerifyPinV alidity !true ;
            SDF[S , G, N, D, DBRequest ]
          )
       []
       [not(PIN_List(called) eq entered_code)]    ->
           (
            D !SS !VerifyPinV alidity !false ;
            SDF[S , G, N, D, DBRequest ]
           )
      )
[]
  ...
endproc  (* end of process SDF *  )
```

The SS feature is then specified by composing the three processes *User_Interaction_SIB, Verify_SIB* and *Screen_SIB*. The process *Service_Screening* which defines the SS feature is defined as follows:

```
process Security_Screening[S , G, N, D, DBRequest]
(caller: network_address , dn: dialled_number , called: network_address) : noexit:=

User_Interaction [S , G, N, D, DBRequest](SS , caller , dn, called) >> accept entered_code: code in
(
 Verify_SIB [S , G, N,D, DBRequest](SS , entered_code , caller , dn, called) >> accept
   PIN_syntax: bool in
    (
    [PIN_syntax eq true] ->
        (
          Screen_SIB[S , G, N, D, DBRequest](SS , entered_code , caller , dn, called) >> accept
          result: bool in
            (
              [result eq true] ->
                        (S !PIC_Select_F acility !caller !dn !called ;
                         SCF[S , G, N, DBRequest])
              []
              [result eq false] -> User_Interaction [S , G, N, D, DBRequest](SS , caller , dn, called)
               >> accept no_code: code in
                        (S !PIC_O_Exception !caller !dn !called ;
                          SCF[S , G, N, DBRequest])
              )
        )
   []
   [PIN_syntax eq false] -> User_Interaction [S , G, N, D, DBRequest](SS , caller , dn, called)
   >> accept no_code: code in
                        (S !PIC_O_Exception !caller !dn !called ;
                         SCF[S , G, N, DBRequest])
   )
) endproc (* end of process Security Screening *)
```

Figure 32 describes the synchronization between processes *Caller* and *Security_Screening* which, in fact, is realized by the synchronization on the actions specified in processes *Caller* and *User_Interaction_SIB*. We can see that process *User_Interaction* is instantiated three times (right hand side of figure 32), to synchronize with successive stages of process *Caller*.

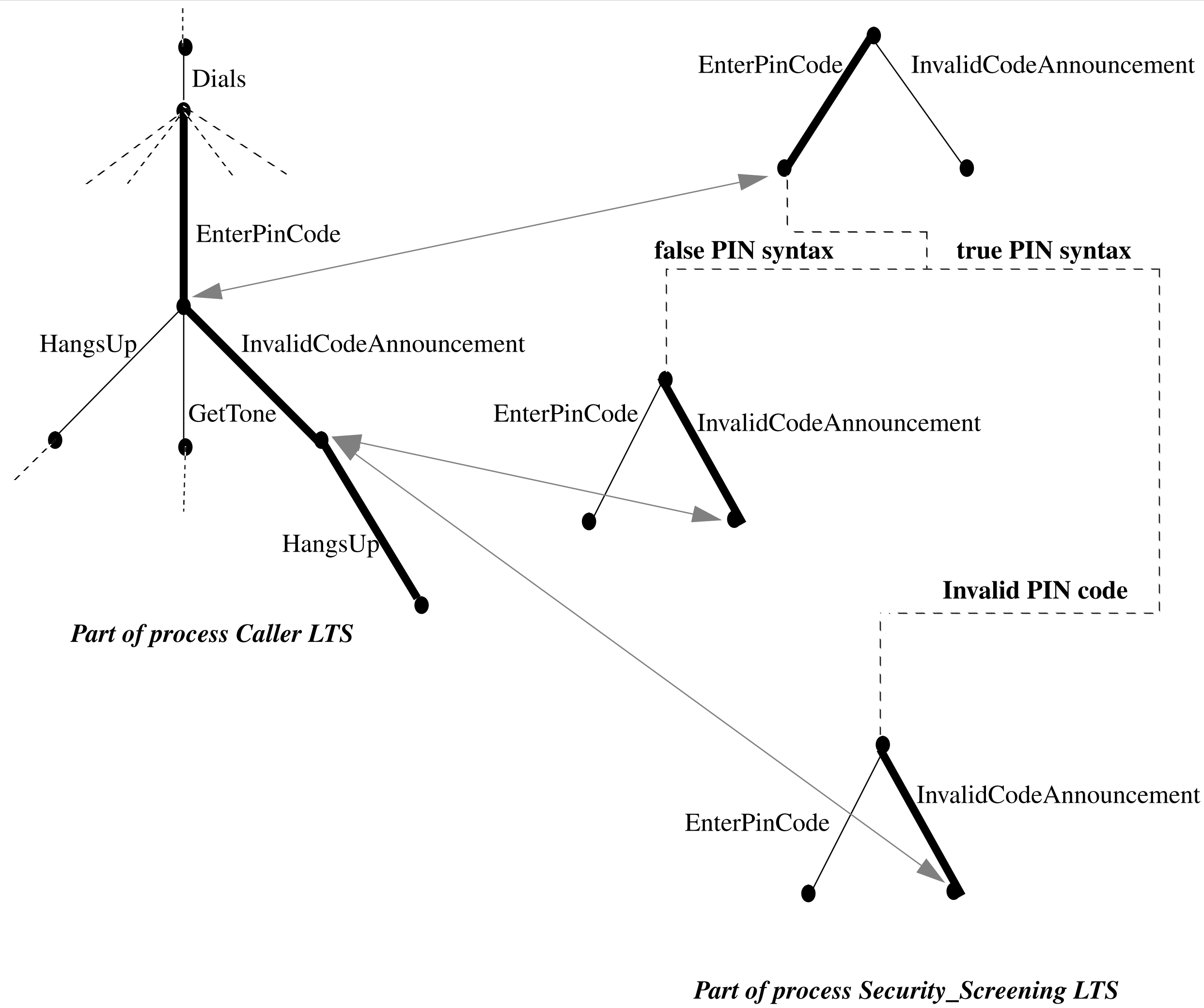


Figure 32: Synchronization between processes Caller and Security_Screening (page 97)

The SS feature is invoked when an indication of incoming call is received by the called line. Thus, the DP *Term_Attempt* must be armed in order to invoke this feature. In our specification, we suppose that the subscriber defined by the network address *adr3* is subscribed to the SS. The arming condition is then:

trigger_armed(t, adr, ser_name) =

((t eq Term_Attempt) and (adr eq adr3) and (ser_name eq SS));

Figure 33 describes the invocation of the SS feature. The control flow starts at the T-BCSM, then if a valid code indication is obtained from the feature, the call continues at the DP *Term_Attempt*. Otherwise, an invalid code indication is given to the caller, who will be forced to hang up (PIC *O_Exception*).

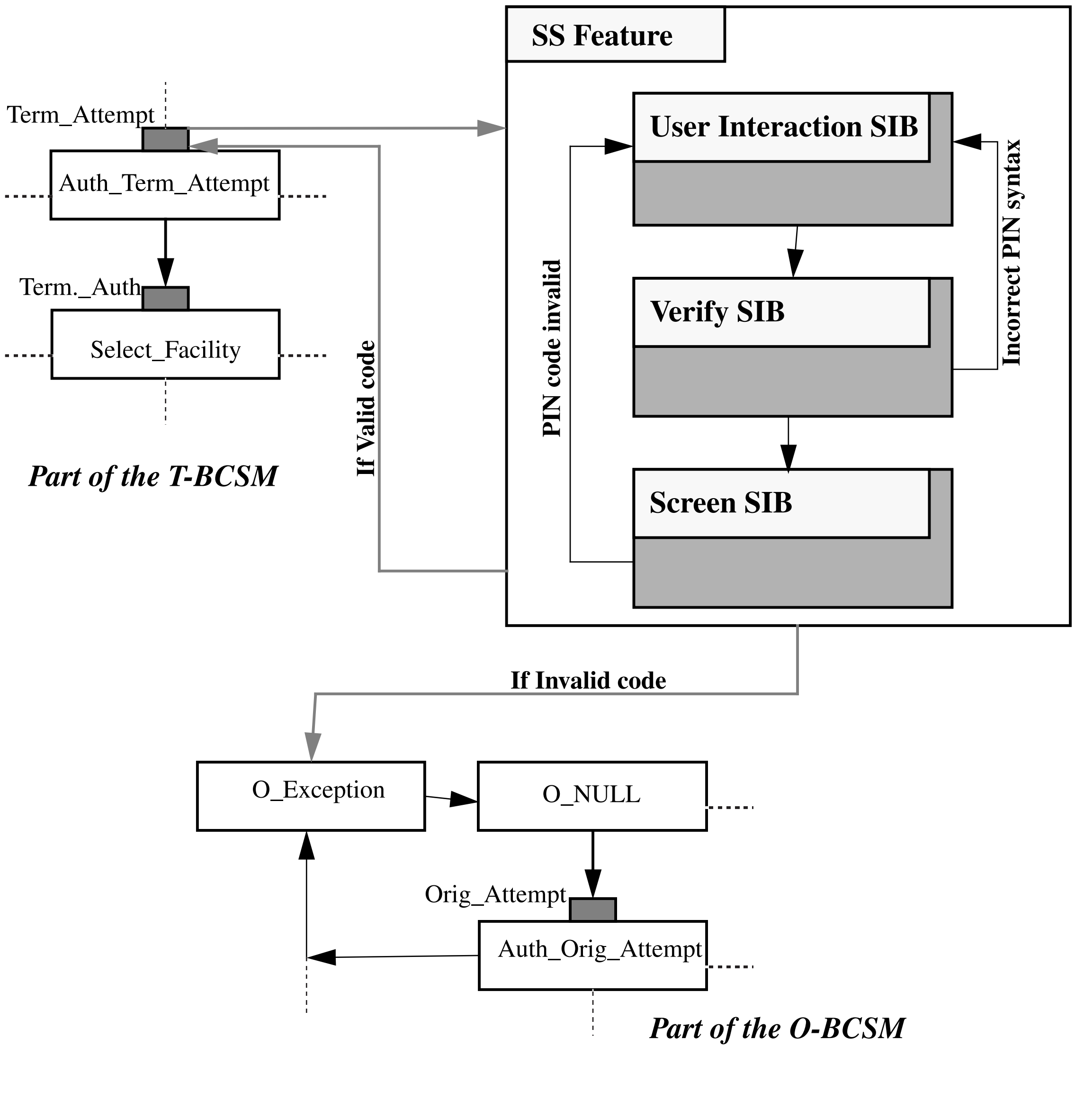


Figure 33: The invocation of SS feature

5.2.3.3 Detection of interaction between SS and CFA

The intention of a subscriber to the SS feature is to protect his line by a PIN code. Only network users who know the code can establish a connection with the subscriber. It could debated what is meant to be achieved with the SS feature. We will suppose here that a user who has protected his line with this feature would like to be reached only by network users he wants i.e. users who know the PIN code.

Expressing the requirement of SS in CTL

In this example, we suppose that *adr2* subscribes to the SS feature, and has as PIN code, *code2*. Then, if a caller attempts to call *adr2*, he must enter the PIN code *code2* of *adr2*.

The requirement of this feature can be described by the fact that any user who attempts to call an SS subscriber i.e. dials a phone number of an SS subscriber, must enter the PIN code before establishing any connection. This requirement can be specified by the following CTL formula:

***P3**: AG(*
(N ?adr: network_address !Dials !num2) ->
AF (¬(Detection_Point !O_Term_Seized !adr !) ∨ (G !adr !EnterPinCode ?PIN: code))*
)

P3 means that if a calling party (specified by *adr* of *sort network_address*) dials phone number *num2* (corresponding to *adr2*), then in every future path, the calling party cannot establish a connection *(Detection_Point !O_Term_Seized !adr !*)* without being asked to enter a PIN code *(G !adr !EnterPinCode ?PIN: code)* because otherwise, the protection is violated. Evidently, the code must be correct, however we will not consider this requirement here.

Expressing the requirement of CFA in CTL

In this example, *adr2* subscribes to the CFA feature and forwards all his calls to *adr*3. Then, the requirements of this feature are specified by the following CTL formula:

***P4**: AG((Detection_Point !Term_Attempt ?caller !adr2) ->*
AF(!Detection_Point !O_Term_Seized !caller !adr3)).

<u>*Deriving Goals satisfying the negation of the properties*</u>

The property *P3* is violated when a user attempts to call an SS feature subscriber and to establish a connection without entering the corresponding valid PIN code. Then, if a trace describing a scenario which violates the property *P3* can be found in the specification, an interaction exist between SS and CFA features. Such a scenario can be described by a trace in which a user picks up the phone and dials a phone number of an SS subscriber, and establishes a connection without entering the PIN code.

With the *Goal Oriented Execution* tool, we can define goals by excluding some gates. The expected trace will not contain the excluded gates. However, the tool does not enable the exclusion of specified actions i.e. actions synchronizing on a gate and offering or accepting specific parameters. Then, to exclude some specific actions from the search, we add to the specification a specific gate just before these actions and we exclude that gate from the search. In our example, we added a specific gate *X* before the action *G* **!***caller* **!***EnterPinCode* **?***PIN: code* which specifies the fact that a user must enter a PIN code, and we exclude the gate *X* from the search. A goal satisfying (*not P3*) is described as follows:

G3: *[G* **?***adr* **!***OffHookToCall, G* **!***adr* **!***Dials* **!***num2,*
Detection_Point **!***O_Term_Seized* **!***adr* **?***called]* \\ *[X].*

After applying *Goal Oriented Execution*, a trace satisfying *G3* was found (trace 3). Then, *P3* is not satisfied and an interaction exist between CFA and SS features.

1-N **!**adr1: network_address **!**OffHookToCall: Signal [true] **line(s): [1206,685,551]**

2-Detection_Point **!**Orig_Attempt: trigger_detection_point **!**adr1: network_address **!**null: network_address **line(s): [707]**

3-G **!**adr1: network_address **!**GetTone: Signal **line(s): [715,564]**

4-Detection_Point **!**Orig_Attempt_Auth: trigger_detection_point **!**adr1: network_address **!**null: network_address **line(s): [771]**

5-G **!**adr1: network_address **!**Dials: Signal **!**num2: dialled_number **line(s): [779,575]**

6-N **!**adr3: network_address **!**OffHookToCall: Signal [true] **line(s): [1206,685,551]**

7-Detection_Point **!**Orig_Attempt: trigger_detection_point **!**adr3: network_address **!**null: network_address **line(s): [707]**

8-G **!**adr3: network_address **!**GetTone: Signal **line(s): [715,564]**

9-Detection_Point **!**Orig_Attempt_Auth: trigger_detection_point **!**adr3: network_address **!**null: network_address **line(s): [771]**

10-G **!**adr3: network_address **!**Dials: Signal **?**dn,dn=num2: dialled_number **line(s): [779,575]**

11-G **!**adr3: network_address **!**TimeOut: Signal **line(s): [783,577]**

12-Detection_Point **!**Collect_Timeout: trigger_detection_point **!**adr3: network_address **!**adr2: network_address **line(s): [799]**

13-G **!**adr3: network_address **!**HangsUp: Signal **line(s): [737,642]**

14-N !adr3: network_address **!**Go_Idle: Signal **line(s): [1210,744,643]**

15-Detection_Point **!**Collected_Info: trigger_detection_point **!**adr1: network_address **!**adr2: network_address **line(s): [808]**

16-Detection_Point **!**Analysed_Info: trigger_detection_point **!**adr1: network_address **!**adr2: network_address **line(s): [840]**

17-Detection_Point **!**Orig_Auth: trigger_detection_point **!**adr1: network_address **!**adr2: network_address **line(s): [859]**

18-Detection_Point **!**Term_Attempt: trigger_detection_point **!**adr1: network_address **!**adr2: network_address **line(s): [879]**

19-S **!**CFA: service_indicator **!**adr1: network_address **!**num2: dialled_number **!**adr2: network_address **line(s): [1399,1223]**

20-D **!**CFA: service_indicator **!**GetForwrdedAddress: operation_name **!**adr2: network_address **line(s): [1532,1423]**

21-D **!**CFA: service_indicator **!**GetForwrdedAddress: operation_name **!**adr3: network_address **line(s): [1533,1424]**

22-**i/exit** (adr1: network_address, num2: dialled_number, adr3: network_address) **line(s): [1425]**

23-S **!**DP_Term_Attempt:state_name **!**adr1:network_address **!**num2:dialled_number **!**adr3:network_address **line(s): [1410,1226]**

24-Detection_Point **!**Term_Attempt: trigger_detection_point **!**adr1: network_address **!**adr3: network_address **line(s): [879]**

25-Detection_Point **!**Term_Auth: trigger_detection_point **!**adr1: network_address **!**adr3: network_address **line(s): [937]**

26-DBRequest **!**Consult: dboperations **!**ADD(adr1, empty): List **line(s): [1202,945]**

27-Detection_Point **!**Term_Res_Avail: trigger_detection_point **!**adr1: network_address **!**adr3: network_address **line(s): [974]**

28-N **!**adr3: network_address **!**RingsFrom: Signal **!**adr1: network_address **line(s): [1214,982,652]**

29-Detection_Point **!**T_Term_Seized: trigger_detection_point **!**adr1: network_address **!**adr3:network_address **line(s): [993]**

30-Detection_Point **!**O_Term_Seized: trigger_detection_point **!**adr1: network_address **!**adr3: network_address **line(s): [1001]**

Trace3: Trace showing the interaction between SS and CFA

The interaction occurs since the CFA feature was invoked before the SS feature. In fact, when the call is forwarded to *adr3* (line 21), the SS feature is no longer active since a new call attempt to *adr3* takes place and *adr3* does not subscribe to the SS feature. The calling party is then connected without being asked to enter the PIN code although it dialled a phone number of a SS subscriber (*num2*). Figure 34 describes interaction between CFA and SS features.

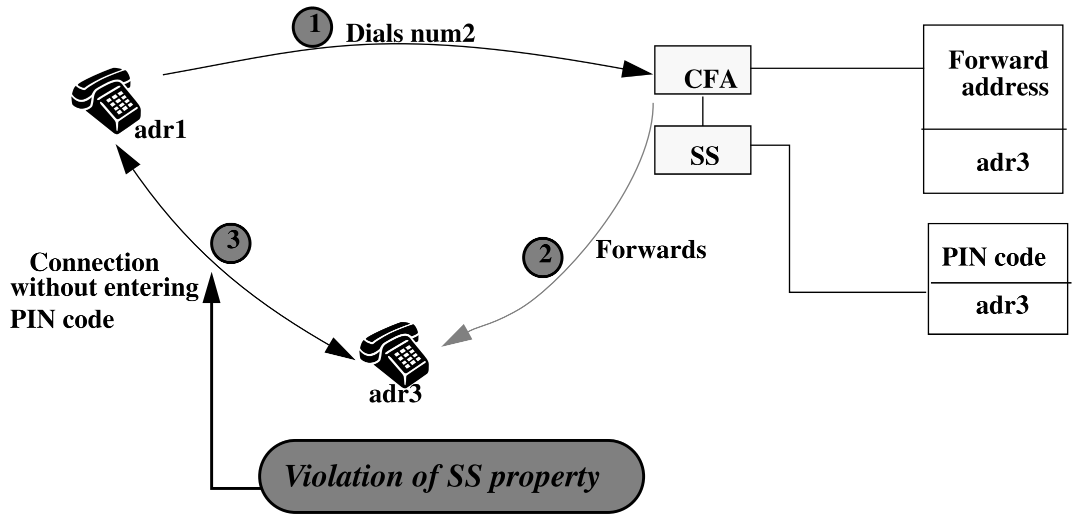


Figure 34: Interaction between SS and CFA

5.3 Discussion

The method we have presented has some advantages compared to the state based approaches which use the full LTS of the specification to verify the desired properties. This is because state based approaches are confronted with the state explosion problem which occurs when the number of states to generate is so big that it cannot fit in memory. Our method is based on Goal Oriented Execution which does not generate the whole LTS but uses the inference rules to look for traces verifying certain properties. These traces can be found with the help of a static analysis of the behavior expression which allows to exclude subtrees of the LTS in which the desired actions cannot occur. The search can be accelerated by adding some intermediate actions in the goal which must exist in the trace or by excluding some actions or some branches from the search which cannot exist in the trace.

Goal Oriented Execution does not eliminate completely the state explosion problem because the search can be conducted deeply and the expected trace could be too long to fit in memory. In addition, since the tool limits the number of instantiations of processes in the case of recursion, the fact that no trace is selected does not mean that no trace exists. In other words, this fact does not guarantee the absence of interactions.

Goal Oriented Execution is able to provide a number of traces for a given goal.

5.4 Chapter Summary

In this chapter, we have presented the method adopted to detect logical interactions between features. This method is based on formalization of feature's properties and derivation of goals satisfying the negation of these properties. Goal Oriented Execution is used to detect traces satisfying these goals. A trace satisfying a goal shows that an interaction exists between the specified features. In fact, it describes a scenario violating one of the properties of the features.

Three cases studies are presented. They are limited to interactions caused by pairs of features since most of the interactions involve two features only. For each feature, its informal specification followed by its formal specification using SIBs is given, its properties using the temporal logic CTL are described, goals satisfying the negation of the properties are derived, and Goal Oriented Execution is applied to look for traces satisfying the derived goals.

Note that, although all our examples have considered interactions between two features only, in principle the method can be applied to the case of more than two features.

CHAPTER 6 Conclusion and Future Directions

The main objective of Intelligent Networks (IN) is to provide independent development and rapid introduction of new services. However, this objective is confronted by the feature interaction problem [BDCG89]. This thesis describes a design method, based on a formal approach, for specifying IN call model and services using the concepts defined in IN Capability Sets 1 (CS1), and a method for detecting feature interaction at specification level based on *Goal Oriented Execution*.

6.1 Summary

This thesis consists of six chapters. The motivation for our work is given in Chapter 1.

Chapter 2 presents an overview of the IN concepts and architectures and describes the IN Conceptual Model (INCM). The INCM consists of four planes, the Service Plane (SS), the Global Functional Plane (GFP), the Distributed Functional Plane (DFP) and the Physical Plane (PP). The mapping between these four planes is also described.

Chapter 3 gives an overview of the LOTOS specification language by describing its main operators and some examples in the context of telephony and IN.

Chapter 4 shows the use of LOTOS as a Formal Description Technique (FDT) in the Service Creation Environment (SCE). It describes the LOTOS specification of IN call model and services as defined in the DFP. The Functional Entities (FEs) that are involved in the establishment of a call/ connection and the control of IN services such as the Call Control Function (CCF), the Service

Switching Function (SSF), the Service Control Function (SCF) and the Service Data Function (SDF), as well as the communication between them are specified. IN services are specified using the concept of Service Independent building Blocks (SIBs) described in the INCM. The SIBs used to specify services are defined in IN CS1 [Q1213]. The specification is designed in a way that independent specification and rapid introduction of services is provided, given that these are two of the main objectives of IN.

In Chapter 5, an approach to detect feature interaction between IN services is given. This approach is limited to the detection of logical interactions which occur when one or some of the requirements or assumptions, that must be satisfied when a feature is introduced separately in the network, is violated. This method is based on formalization of feature's properties, which is followed by derivation of goals satisfying the negation of these properties. Goal Oriented Execution is used to detect traces satisfying these goals. A trace satisfying a goal shows that an interaction exists between the specified features. In fact, it describes a scenario violating one of the properties of the features.

Three cases studies are presented. They are limited to interactions caused by pairs of features since most of the interactions involve two features only. For each feature, its informal specification followed by its formal specification using SIBs is given, its properties using the temporal logic CTL are described, goals satisfying the negation of the properties are derived, and Goal Oriented Execution is applied to look for traces satisfying the derived goals. This method depends heavily on the formalizations of properties and derivation of goals.

The method presented to detect interaction between features does not give a general solution to the feature interaction problem but it gives a partial solution limited to the detection of logical interactions which must be performed as early as possible, otherwise they will propagate through the other stages of the feature lifecycle.

This work shows the usefulness of LOTOS and formal methods in the SCE. In fact, the SCE defines all the functions which handle all the operations required for the definition, development and testing of IN services. Therefore, the developed specification described in Chapter 4 can be used as a testbed for adding specifications of new services, testing them and detecting logical interactions between them, if there are any, using the approach described in Chapter 5.

6.2 Research Directions

6.2.1 Extension to IN CS2

The specification of IN call model and services is done with respect to IN Capability Set 1 (CS1) which deals only with *type A* services which, as described in Chapter 2 section 2.3, are single-ended and have a single point of control. Some new concepts are introduced in IN CS2 in order to include *Type B* services which allow multiple subscribers i.e. services that are visible to multiple parties. These new concepts consist mainly in the introduction of high level SIBs and service processes to enhance the modelling of *type B* services [Hu95]. Extending our model to fit IN CS2 concepts can be an interesting continuation of our work.

6.2.2 Extension to Switch based features

Switch-based features are features that are deployed in the switch (SSP) such as Call Waiting and Three Way Calling. This type of features was out of the scope of this thesis but it would be useful to investigate it since it can be in conflict with IN features.

6.2.3 Extension to other types of interaction

As mentioned above, the approach adopted for detecting feature interactions between IN services concerns only logical interactions. Many other types of interactions that can be detected at the specification level are worth investigating. Such interactions include the ones caused by non-determinism of actions i.e. more than one action is possible at a certain point [BoLo93], [SL94].

Bibliography

[BDCG89] T.F. Bowen, F.S. Dworak, C.H. Chow, N. Griffeth, G.E. Herman, and Y-J. Lin. The Feature Interaction Problem in Telecommunications Systems. 7th International Conference on Software Engineering for Telecommunication Switching Systems, July 1989, 59-62.

[BG93] B, Ghribi. A Model Checker for LOTOS. Master Thesis, Department of Computer Science, University of Ottawa, 1993.

[BoLo93] R. Boumezbeur, L. Logrippo. Specifying Telephone Systems in LOTOS. IEEE Communications Magazine, Aug. 1993, 38-45.

[BZ92] W. Bouma and H. Zuidweg. Formal Analysis of Feature Interactions by Model Checking. PTT research, the Netherlands, December 1992.

[Cain92] M. Cain. Managing Run-Time Interactions Between Call-Processing Features. IEEE Communications Magazine, February 1992, 44-50.

[CaLi91] E. J. Cameron and Y.J. Lin. A Real-Time Transition Model for Analyzing Behavioral Compatibility of Telecommunications Services. In Proceedings of the ACM SIGSOFT 1991 Conference on Software for Critical Systems, December 1991, New Orleans, Louisiana, 101-111.

[CCITT87] Recommendation Z.100. Specification and Description Language SDL. CCITT SG X, contribution com X-R 15-E, 1987.

[Cheng94] K.E. Cheng. Towards a Formal Model for Incremental Service Specification and Interaction Management Support. Second International Workshop on Feature Interactions in Telecommunications Software Systems. eds. L. G. Bouma and H. Velthuijsen, IOS Press 1994, pages 152 - 166.

[CoPi94] P.Combe and S. Pickin. Formalization of a User View of Network and Services for Feature Interaction Detection. Second International Workshop on Feature Interactions in Telecommunications Software Systems, eds. L. G. Bouma and H. Velthuijsen, IOS Press 1994, pages 120 - 135.

[DaNa93] O. Dahl and E. Najm. Specification and Detection of IN Service Interference Using LOTOS. Proceedings Forte ‘93, eds. R. L. Tenney, P. D. Amer, M. U. Uyar, Boston 1993, 53-71.

[EM85] B. Ehrig, B. Mahr. Fundamentals of Algebraic Specifications. Springer-Verlag, 1985.

[Faci95] M. Faci. Detecting Feature Interaction in Telecommunications Systems Designs. Phd Thesis, Department of Computer Science, University of Ottawa, November 1995.

[FaL93] M. Faci and L. Logrippo. Specifying Hardware in LOTOS. Proceedings of the IFIP WG 10.2. 11th International Conference on Computer Hardware Description Languages and their Applications, Ottawa, Canada, April 1993, 305-312.

[FaL94] M. Faci and L. Logrippo. Specifying Features and Analysing their Interactions in a LOTOS Environment. Second International Workshop on Feature Interactions in Telecommunications Software Systems, eds. L. G. Bouma and H. Velthuijsen, IOS Press 1994, 136-151.

[FaLS91] M. Faci, L. Logrippo and B. Stepien. Formal Specifications of Telephone Systems in LOTOS: The Constraint-Oriented Style Approach. Computer Networks and ISDN Systems, 21, North Holland, 1991, 52-67.

[FaLS95] M. Faci, L. Logrippo and B. Stepien. Structural Models for Telephone Specifications. To appear in Computer Networks & ISDN Systems.

[Fits94] Second International Workshop on Feature Interactions in Telecommunications Software Systems. eds. L. G. Bouma and H. Velthuijsen, IOS Press 1994.

[Fits95] Third International Workshop on Feature Interactions in Telecommunications Software Systems. eds. K. E. Cheng and T. Ohta, ISO Press 1995.

[FWFI92] The First International Workshop on Feature Interactions in Telecommunications Software Systems. Florida, 1992.

[GrVe92] N. D. Griffeth and H. Velthuijsen. Conflict Resolution in Telecommunications Systems: A Formalism for Proposal Generation, 1992.

[Ha95] M. Haj-Hussein. Goal Oriented Execution for LOTOS. Phd Thesis, Department of Computer Science, University of Ottawa, 1995.

[HLS93] M. Haj-Hussein, L. Logrippo and J, Sincennes. Goal Oriented Execution of LOTOS Specifications. in M. Diaz and R. Groz (Eds) Formal Description Techniques, V. North Holland, 1993, 311 - 327.

[Hoar85] C. A. R. Hoare. Communicating Sequential Processes. Prentice-Hall, 1985.

[Hu95] Y. Hu. IN CS2 Enhancements to the global functional plane. IEEE Intelligent Network Workshop, Ottawa, Canada, May 1995.

[ISO8807] ISO, IS 8807. Informal Processing Systems - Open Systems Interconnection - LOTOS: A formal Description Technique based on the Temporal Ordering of Observational behavior, May 1989.

[Keiser95] K. Bernhard. Digital telephony and network integration. Van Nostrand Reinhold, 1995. 2nd ed.

[Lee92] A. Lee. Formal Specification and Analysis of Intelligent Network Services and their Interaction. Ph. D. Thesis, Dept. of Computer Science, University of Queensland, 1992.

[Li94] R. Linden. Using an Architecture to Help Beat Feature Interaction. Second International Workshop on Feature Interactions in Telecommunications Software Systems, eds. L. G. Bouma and H. Velthuijsen, IOS Press 1994, pages 24 - 35.

[LoFH92] L. Logrippo, M. Faci and M. Haj-Hussein. An Introduction to LOTOS: Learning by Examples. Computer Networks & ISDN Systems, Vol. 23, No. 5, 1992, 325-342.

[Mil80] R. Milner. A Calculus of Communicating Systems. Lecture Notes in Computer Science, (Springer-Verlag, 1980) No. 92.

[MiTJ93] J. Mierop, S. Tax, R. Janmaat. Service Interaction in an Object Oriented Environment. IEEE Communications Magazine, Aug. 1993.

[Myers79] C. Myers. The art of Software Testing. John Wiley & Sons, 1979.

[Q1201] ITU-T/ETSI Recommendation Q1201, 1993.

[Q1202] ITU-T/ETSI Recommendation Q1202, 1993.

[Q1203] ITU-T/ETSI Recommendation Q1203, 1993.

[Q1204] ITU-T/ETSI Recommendation Q1204, 1993.

[Q1205] ITU-T/ETSI Recommendation Q1205, 1993.

[Q1211] ITU-T/ETSI Recommendation Q1211, 1993.

[Q1213] ITU-T/ETSI Recommendation Q1213, 1993.

[Q1214] ITU-T/ETSI Recommendation Q1214, 1993.

[SL93] B. Stepien and L. Logrippo. Status-Oriented Telephone Service Specification. In: T.Rus and C.Rattray (eds.) Theories and Experiences for Real-Time System Development. AMAST Series in computing, Vol.2, World Scientific, 1994, pages 265-286.

[SL94] B. Stepien and L. Logrippo. Feature Interaction Detection using Backward Reasoning with LOTOS. In: S. Vuong (ed.) Protocol Specification, Testing and Verification, XIV Proc. of the 14th International Symposium on Protocol Specification, Testing and Verification, organized by IFIP WG 6.1, Vancouver, 1995, pages 71-86.

[SL95] B. Stepien and L. Logrippo. Representing and Verifying Intentions in Telephony Features using Absract Data Types. Third International Workshop on Feature Interactions in Telecommunications Software Systems, eds. K. E. Cheng and T. Ohta, IOS Press 1995, pages 141 - 155.

[Th94] J. Thorner. Intelligent Networks. Artech House, 1994.

[To89] A.J. Tocher. LOTOS and the Formal Specification of Communication Standards: An example. Formal Methods: Theory and Practice, Chapter 2, edited by P.N. Scharbach, BP Research, 1989.

[Vi92] J. Visser. International Standards for Intelligent Networks. IEEE Communications Magazine, February 1992.

[Vi95] J. Visser. Tutorial on Intelligent Network Basics. IEEE IN'95 Workshop, Ottawa, May 1995.

[ViSV88] C. A. Vissers, G. Scollo, and M. van Sinderen. Architecture and Specification Style in Formal Descriptions of Distributed Systems. In: Aggarwal, S., and Sabnani, K., (eds.)

Protocol Specification, Testing and Verification, VIII, North-Holland, 1988, 189-204.

[Zave93] P. Zave. Feature Interactions and Formal Specifications in Telecommunications. IEEE Computer, Aug. 1993, 20-30.

List of Acronyms

S

T