Paper appeared in: K. Kimbler and L.G. Bouma (Eds.). Feature Interactions in Telecommunications and Software Systems V. IOS Press, 1998, 172-186.

Goal-Oriented Feature Interaction Detection in the Intelligent Network Model

Jalel KAMOUN and Luigi LOGRIPPO
University of Ottawa
Telecommunications Software Engineering Research Group
School of Information Technology and Engineering
Ottawa, Ontario, Canada K1N 6N5
luigi@site.uottawa.ca

Abstract. In the first part of the paper, a LOTOS model for specifying the Intelligent Network call model and services belonging to the Distributed Functional Plane is described. The functional entities involved in the establishment of call connection and invocation of services are formally specified. In the second part of the paper, an approach to detect feature interactions between IN services is presented. Interactions caused by violation of features properties are detected. The approach is based on stating feature properties, on deriving goals satisfying the negation of these properties, and on use of Goal Oriented Execution to detect traces satisfying these goals. Such traces, if found, show that interactions exists between the specified features by showing that a scenario violating one of the properties of the features can be found. An example showing the detection of interaction between Originating Call Screening and Call Forward Always is given.

1. Introduction and Motivation

With the infrastructure provided by the Plain Old Telephone System (POTS), the task of introducing a new service was tedious and very costly. To overcome the limitations of POTS, Intelligent Networks (IN) were 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 within its Service Creation Environment (SCE) and deploy them in the network.

However, the rapid development of services is hindered by the feature interaction problem [1]. Formal Description Techniques (FDTs) have proven useful in detecting feature interactions at the specification level (see numerous papers in [5][2][7]). A formal description of the system behavior with the features provides an unambiguous and precise view of the system that can support formal analysis and validation methods.

In this paper, we describe an architectural model for structuring the specification of IN components in the formal description technique LOTOS [11], as well as a LOTOS-based method for detecting feature interactions.

Our view of the Intelligent Network is contained in its Global Functional Plane and Distributed Functional Plane (DFP). In order to write a specification that constitutes a functioning model of some of the essential elements of the IN, we were obliged to combine features of the two planes. Only the characteristics of Capability Set 1 (C1), the ‘basic’ one, were considered [12] [13] [14].

2. Specification of IN Call Model and Services in LOTOS

Our main objective in specifying the IN call model and services in LOTOS is to provide an executable specification, or model, that can be used as a test bed for specifying, validating and detecting feature interactions. The model 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.

Only the external behavior of the system describing call/connection establishment and service activation is of interest. Fig. 1 shows the elements of the DFP of CS1 that are involved when a call is being processed and the features are activated, as well as the elements that were formalized.

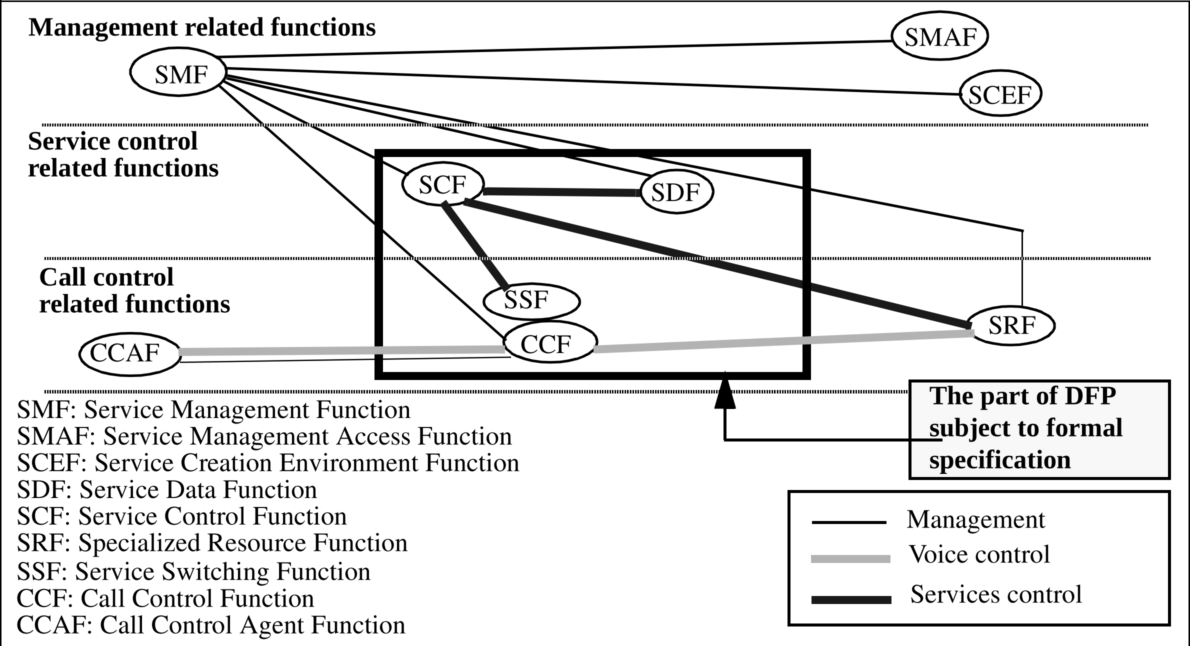
Management related functions
SMF
SMAF
SCEF
Service control related functions
SCF
SDF
Call control related functions
SSF
CCF
CCAF
SRF
The part of DFP subject to formal specification
SMF: Service Management Function
SMAF: Service Management Access Function
SCEF: Service Creation Environment Function
SDF: Service Data Function
SCF: Service Control Function
SRF: Specialized Resource Function
SSF: Service Switching Function
CCF: Call Control Function
CCAF: Call Control Agent Function
Management
Voice control
Services control

Figure 1: Functional Entities of the DFP

These are the SCF, CCF, SSF and SDF. 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. Among these are the SRF which provides specialized resources such as protocol conversion and speech recognition, and the CCAF which defines the interface that provides to network users access to the CCF.

Principles for the specification of telephony systems in LOTOS were discussed in [8]. In this work, we decided to specify IN in a way that reflects closely the architecture described in the standard documents. This required further adaptation of the resource-oriented model presented in that paper.

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. For simplicity, in our model we assume that directory number, line, and network subscriber are the same thing. Therefore, connections are constrained by the fact that at any time, a directory number or line 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 (Fig.2).

The process *Connections* is composed of two processes: *Network_Subscribers* and *IN_Network*. 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 CCF/SSF, the SCF and the 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.

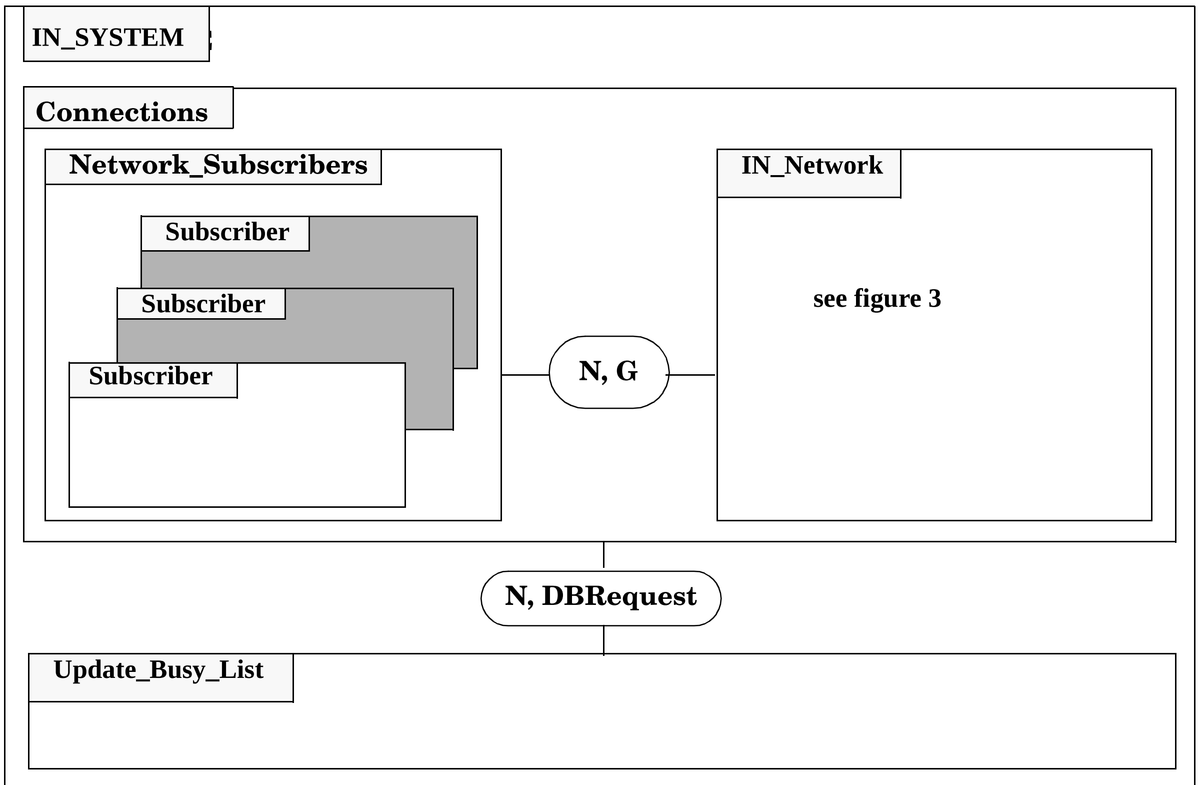

Figure 2: Graphical representation of the top levels of the specification

2.1 Process Subscriber

A Subscriber can have one of 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), where 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 [15]. 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 PIC *Analyse_Info* defines the activities of analyzing and translating information according to the dialing plan to determine routing address and call type, therefore, it is not 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. 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*.

2.2 Process IN_Network

Process *IN_Network* which specifies the DFP entities involved in the establishment of call/ connection is defined as follows (Fig. 3):

```
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 *)
```

where process *CCF_SSF* specifies the CCF/SSF functionalities, process *SCF* specifies the SCF activities and process *SDF* 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*. Process *CCF_SSF* is defined as follows:

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

N ?caller: network_address !OffHookToCall; (1)
(
  (
   Basic_Call_State_Model [S, G, N,Detection_Point, DBRequest](caller, nil, null)
```

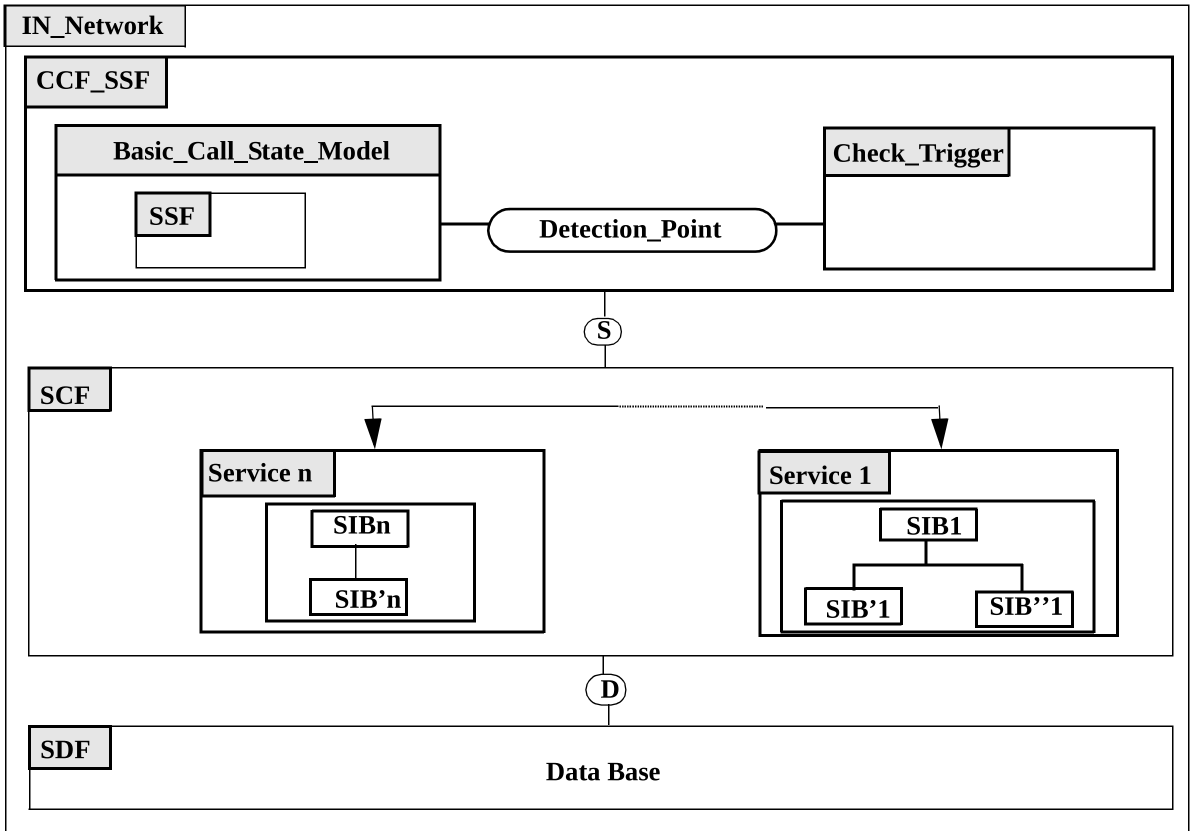

Figure 3: Detailed representation of process IN_Network

```
    |[ 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 **(1)** in the specification of process *CCF_SSF* 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 PIC or a 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.

2.3 Process SCF

Process *SSF* synchronizes with process *SCF*, which specifies the SCF functionalities, on gate *S*. 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 *)
```

When a service must be invoked, the two 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*.

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 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)

2.4 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 access to its data. This FE is described by the process *SDF* which communicates with process *SCF* via synchronization on gate *D*. It has the following structure:

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

  ...
 []
    D !Service_Name !Operation_Name ?parameter_1 ... ?parameter_n;

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

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

In order to perform its functionality, a service process must send a message to the SDF indicating the type of operation required 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. Process *SDF* executes the actions needed, and sends back the results to the service process. Hence process *SDF* is defined as a choice between different operations needed by the different service processes.

2.5 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 data types, 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.

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.

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). 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 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.

3. Detecting Feature Interaction between IN services

In our method, feature requirements are expressed as properties in a property language and interaction is said to occur when these properties are not satisfied by the formal specification [3] [6].

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

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

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

3.1 Properties of Features

To be able to verify the correctness of the behavior of the resulting system (after adding features to it), we have to express formally the feature requirements and the general properties of the basic system. Thus we need a formal property language. For this purpose we chose the branching time temporal logic CTL [4] 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 is defined with respect to Kripke Structures, however a LOTOS specification is seen as a Labelled Transition System (LTS). This does not present a problem since any LTS can be transformed into a Kripke Structure. In this transformation, every transition from a state *S1* to a state *S2* is transformed into a state in a Kripke Structure. It is possible also to use other temporal logics more appropriate for LTSs but this was left for further research.

3.2 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 method we present is based on the *Goal Oriented Execution* tool developed within the LOTOS group of the University of Ottawa [9] [10]. Goal oriented execution allows one to look for execution traces according to several properties. In the simplest instance of this execution method, 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 uses the fact 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 (possibly non-contiguous) 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 **;** f ; c **; stop []** a **; stop)** and the goal is: [a, c], then the following traces satisfy the goal and will be found: *a* **;** *c* and *a* **;** *e* **;** *f ; c*. Events can be associated with actions in the expression defining the goal to be reached. For example, if the sequence contains an action $a!x_1$, the selected trace must contain an action with gate *a* and with the offer of value x_1. If the event associated with the action is the acceptance of a parameter (**?**), the system will accept any possible value of that parameter. An example of a goal to be satisfied is the following:

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

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

The tool has many characteristics that can speed up the search. If the search is slow or unsuccessful we can 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 known that before the 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 exclude some gates from the search. The search process will not search further those branches. It is also possible to set limits on the number of instantiations of processes, thereby limiting the depth of the search.

Note that goal-oriented execution is a state exploration method that is suitable for infinite-state systems, since there is no requirement that the global state space of the system be computed. This is a main difference between the techniques used in [17] and ours. While the execution times reported by this author are in the order of dozens of hours, our execution times are in the order of dozens of minutes.

Following the principle described in section 3.2, 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 3.2 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 verified 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.

4. Case Study

In this example we show how the interaction between *Originating Call Screening* (OCS) and *Call Forward Always* (CFA) features is detected.

4.1 Specification 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, the subscriber can still be reached from subscribers whose telephone numbers are included in the list.

OCS 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 [14] but they are not relevant to the formal specification since they are related to implementation details.

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 !IsInScreenList !caller !called; (1)
    D !IsInScreenList ?result: bool; (2)
    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 action (1) specified in the former process. This action offers three parameters: *IsInScreenList* of sort *operation_name* indicating the type of operation to be performed by the SDF and *caller* and *called* which represent the call parameters needed to perform the operation. When process *SDF* performs the required operation, it sends back the result to process *Screen_SIB*. This is done by synchronizing on gate *D* and executing action (2)(*D* **!***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  !IsInScreenList ?caller: network_address ?called: network_address;
      (
        [called NotIn ScreenList(caller) ] ->
           (
             D !IsInScreenList !false;
             SDF[S, G, N, D, DBRequest]
           )
        []
        [called IsIn ScreenList(caller)] ->
           (
                D  !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 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.

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 her screening list. The arming condition is defined by an operation *trigger_armed* of sort *bool* defined in the type *Trigger_Detection_Point*.

4.2 Specification 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.

The CFA service is composed of the *Translate* SIB which, as defined in [14], 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.

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  !GetForwardedAddress !called; (1)
     D  !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 (1). *CFA* is a parameter of sort *service_name* referencing CFA feature. *GetForwrdedAddress* is a parameter of sort *operation_name* which indicates 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  !GetForwardedAddress ?called: network_address;
   D  !GetForwardedAddress !get_CFAddress(called);
   SDF[D, DBRequest]
[]
  ...
endproc (* end of process SDF *)
```

The operation *get_CFAddress: network_address* -> *network_address* is defined in the data types 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 (note that this is a deliberate simplification in our model).

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. 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*.

4.3 Detection of interaction between OCS and CFA

Expressing the requirement of OCS in CTL

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 in CTL as follows:

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

Expressing the requirement of CFA in CTL

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)).*

Deriving Goals that satisfy the negation of the OCS property

(¬ *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*** was found (Trace 1), consisting of 25 actions. This trace shows a scenario which violates the property of the OCS feature, proving that 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 continues (line 11 to line 14). Then, the CFA feature is invoked (line 15) and the call is forwarded to *adr2* (line 19). A *ring* is then performed between *adr1* and *adr2* (line 25). Figure 4 describes how the property of the OCS feature is violated by the introduction of the CFA features.

5. Conclusion

An architecture for formally specifying the IN Call Model and services as defined in the DFP of IN CS1 was presented. 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 the second part of the paper, an approach to detect feature interaction between IN services was presented. This approach is applicable 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

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

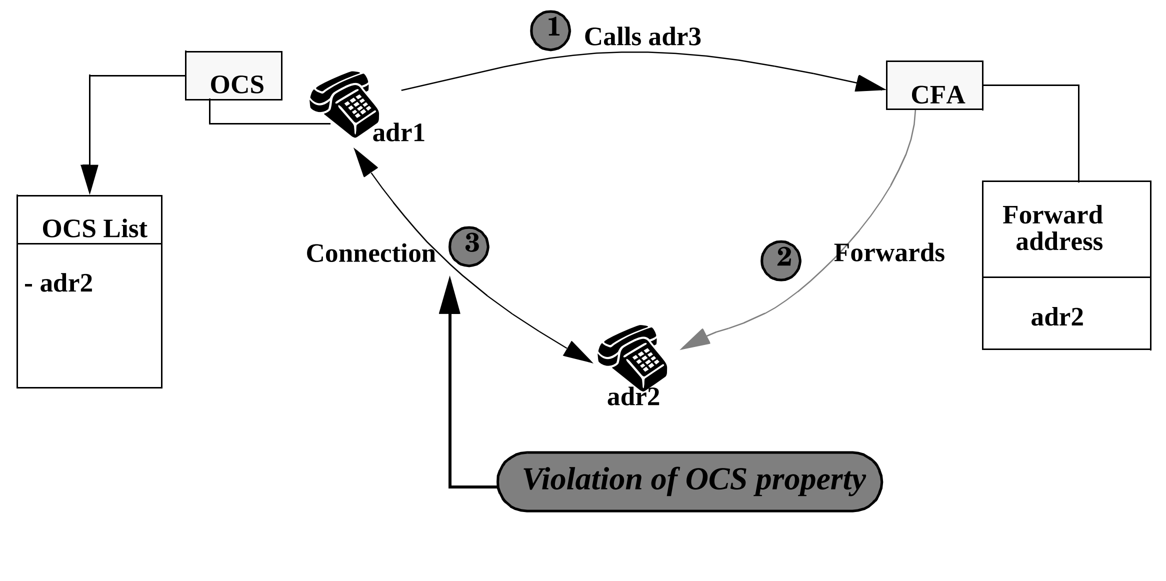

Figure 4: Interaction between OCS and CFA

formalization of feature 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. An example of detecting interaction between Call Forward Always and Originating Call Screening was presented. Two other cases of interaction detection following this approach are presented in [16]. They concern the interactions between OCS and Abbreviated Dialling, and between Security Screening and CFA.Execution times for this method compare very favorably with those reported for a related LOTOS-based method.

Acknowledgments

We would like to thank Motorola-ARRC and NSERC for their financial support and all the members of the University of Ottawa LOTOS group for the fruitful discussions we have had. We are particularly indebted to Bernard Stepien for his useful comments about some parts of the specification and to Jacques Sincennes for his technical support in using the Goal Oriented Execution tool.

References

[1] E.J. Cameron, N. Griffeth, Y. Lin, M.E. Nilson, W.K. Schnure, H. Velthuijsen. A feature interaction benchmark for IN and beyond. IEEE Communications 31 (1993), 64-69. Reprinted in extended form in: L.G. Bouma and H. Velthuijsen (eds.). Second International Workshop on Feature Interactions in Telecommunications Software Systems, IOS Press, 1994, 1-23.

[2] W. Bouma and H. Velthuijsen (Eds.) Feature Interactions in Telecommunications Systems. IOS Press, 1994.

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

[4] E.M. Clarke, E.A. Emerson and A.P. Sistla. Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications, ACM TOPLAS, 8(2):244-263, April 1986.

[5] K.E. Cheng and T Ohta (Eds.) Feature Interaction in Telecommunications III. IOS Press, 1995.

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

[7] P. Dini, R. Boutaba, and L.Logrippo. Feature Interactions in Telecommunications Networks IV. IOS Press, 1997.

[8] M. Faci, L. Logrippo, and B. Stepien. Structural Models for Specifying Telephone Systems. Computer Networks and ISDN Systems 29, 501-528.

[9] M. Haj-Hussein. Goal Oriented Execution for LOTOS, PhD Thesis in Computer science, University of Ottawa, Canada, 1995.(http://LOTOS.csi.UOttawa.ca/ftp/pub/Lotos/Theses/)

[10] 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.

[11] ISO, IS 8807. Information Processing Systems - Open Systems Interconnection - LOTOS: A formal Description Technique Based on the Temporal Ordering of Observational Behavior, May 1989 (E. Brinksma, editor).

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

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

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

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

[16] J. Kamoun. Formal Specification and Feature Interaction Detection in Intelligent Networks, Master thesis in Computer Science, University of Ottawa, Canada, 1996.(http://LOTOS.csi.UOttawa.ca/ftp/pub/Lotos/Theses/)

[17] M. Thomas. Modeling and Analysing User Views of Telecommunications Services. In: Feature Interactions in Telecommunications and Distributed Systems IV. P. Dini, R. Boutaba, L. Logrippo (eds.) IOS Press, 1997. 168-182.