Feature Interaction Detection in a Telephony Network Integrated with Switch-based Features and IN Features

Jennifer Qifang Fu

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 for Computer Science

University of Ottawa
Ottawa, Ontario, Canada
September, 1999

Abstract

Telephony systems have evolved from the Plain Old Telephony System providing only the basic functionality of making phone calls, to sophisticated systems in which many features have been introduced, providing network subscribers more control on the call establishment process. The concept of Intelligent Network was developed to facilitate and accelerate the introduction of new features in a cost-effective manner. However, this objective confronts a major obstacle known as the *feature interaction* problem. A feature interaction occurs when at least one feature is prevented from performing its functionality or when the system functions incorrectly due to the presence of features.

In the first part of the thesis, we present a model for specifying a telephony network integrated with both switch-based features and IN features using a mixture of resource-oriented style and state-oriented style as well as a specially organized Abstract Data Type hierarchy. The model is designed in a way that independent specification and rapid introduction of features is provided.

In the second part of the thesis, we present an improved formal definition of the concept of Feature Interaction and a Feature Interaction Detection System. The system is limited to interactions caused by violation of features or system properties. Feature Interaction between the given features can be detected and presented in the format of Message Sequence Charts via five steps: *Test Scenario Designer, Integrator, Feature Interaction Hunter, Property Checker* and *Message Sequence Charts Translator.*

It is concluded that LOTOS is useful as a Formal Description Technique in specifying the telephony system with features and for detecting feature interactions at the abstract specification level.

Acknowledgement

I would like to thank all the people who helped me in completing this thesis.

First, I would like to thank my supervisor, Prof. Luigi Logrippo, for his encouragement, guidance and financial assistance through my graduate studies.

Second, I would also like to thank the members of University Ottawa LOTOS group for all their advise and assistance. In particular, I would like to thank Jacques Sincennes, Laurent Andriantsiferana and Rossana M. Castro Andrade.

Most of all, I should like to thank my parents, Gui Heng Fu and Su Qin Tian, and my fiancé Tie Xu for their love, support and encouragement.

Table of Contents

List of Figures

List of Tables

List of Charts

Chapter 1. Introduction: Motivation and Background

1.1 Introduction

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

This has resulted in a transition from basic telephony systems providing only the basic functionality of making phone calls, to sophisticated systems in which many features have been introduced, providing the network subscribers more control in the call establishment process. However, with the infrastructure provided by the Plain Old Telephony System (POTS), the task of introducing new features was tedious and very costly. This is because, before 1980s, features were switch-based. All the data and logic processing required by the services were located within the local node. This technique has two major drawbacks. First, since the software related to the new introduced features must be located in all the local exchange nodes (local switches) to which end-users are directly connected, any software modification should be done to all those local nodes. Second, due to the fact that different types of switches provided by different telecommunication companies could be deployed, the introduction of a new feature requires the adaptation of the related software to every type of switch in the network. With these complexities and effort required, a new feature typically requires three or four years to be deployed into the network [Lee92], [Viss95].

To overcome the limitations of POTS, Intelligent Networks (IN) were introduced to facilitate and accelerate in a cost-effective manner service implementation and provisioning. One of the aims of IN is independent service implementation. That is, every service provider will be able to define and develop its own services independently and then deploy them in the network.

IN has two essential elements: Common Channel Signaling and Non-switching nodes [Viss95].

- Common Channel Signaling

  The Common Channel Signaling is a signaling system where all signalings are performed over transmission paths completely separated from the voice path [Bern95]. 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 System: CCSS6 using analog voice-band transmission and CCSS7 that evolved from the former, using the standard 64 kb/s digital transmission link [Thor94].

- Non-switching Node

  The CCSS7 common channel signaling has enabled the introduction of non-switching nodes where feature logic and data could be stored. This means that the service control can be centralized in some specific nodes. Those nodes are known as Service Control Points (SCPs) and Service Data Points (SDP). They are accessible to the switch via protocols using CCSS7.

The first IN services introduced are the 800 services (known as freephone numbers) and Automatic Calling Card Service [Viss95].

When an IN feature is to be invoked, a message indicating a request to process the feature and other related information, i.e. the caller and callee's addresses, the calling time, etc. is sent from the switch to the SCP via the signaling network. Then the feature is processed within the SCP. When the SCP finishes processing, a response of instructions, i.e. rerouting or terminating the call, is sent from the SCP back to the switch. The switch will process the call as instructed, i.e. rerouting or terminating the call.

However, although the CCSS7 and the SCP technology free the features from being located in the switches, the features are dependent on specific activation events and each feature has its own activating mechanism defined within the switch. A new approach was developed to handle this problem by introducing a number of well-defined feature independent activation checkpoints within the switch and defining a feature independent interface between the switch and the SCP. As a result, the deployment of a new feature does not need a modification on the switch for the specific activating mechanism. A simple information to the switch that a new feature has been deployed and should be activated under certain criteria is sufficient.

1.2 Feature Interaction Problem

The introduction of the IN technology eased the difficulty of feature creation, deployment and maintenance. However, with the abundance of new features and their co-existence in the networks, a new problem called *Feature Interaction* (FI) problem was discovered [BDCG89]. A Feature Interaction is understood to be any kind of unexpected interference among multiple features. These interferences may prevent at least one of the features from behaving correctly

[BDCG89]. The FI problem is complex. After several years of exploration [1stFITS92] [2ndFITS94] [3rdFITS95][4thFITS97][5thFITS98], researchers have generally agreed that it is probably not feasible to resolve all possible feature interactions at any single stage of a feature lifecycle or with any single technique [Kell94].

Our work is motivated by the challenges, from a designer's point of view, of detecting FIs in telecommunication network systems.

The feature interaction problem can arise at any stage of the feature development lifecycle. Therefore, the feature interaction problem can be approached from three different angles: detection, avoidance, and resolution [CaVe93] [2ndFITS94]. Furthermore, detection and resolution may be divided into on-line and off-line techniques, as discussed in the introduction of [2ndFITS94]. Off-line methods deal with the problem before deployment. On-line methods deal with it after deployment.

- Detection

  The objective of a detection approach is to analyze a set of independently specified features and determine whether or not there are any interactions between their joint behavior [BoLo93], [Thom97], [FaL397], [NaKK97], [KaLo98]. Detection can be applied through the whole lifecycle of a feature, since the cause of interaction can be related to any phase of the feature lifecycle.

- Avoidance

  An avoidance mechanism for unwanted interaction assumes that the causes of the interaction are known and an architectural or analytical approach is defined to prevent the manifestation of such interactions [MiTJ93]. The avoidance approach is most suitable in the

early phases of specification and design of features. An example of the application using the avoidance approach is the Wireless Intelligent Network (WIN) protocol, where the feature interaction problem is solved by giving pre-defined priorities to different features [Grin97].

- Resolution

The objective of a resolution mechanism is to find appropriate solutions to interactions that manifest themselves at execution time. Several approaches have been proposed in [Chen94][IrEr97] [BAEQ98].

Formal Description Techniques (FDTs) such as LOTOS [ISO8807] and SDL [CCITT87] have proven useful in detecting feature interactions at the specification level [Zave93]. A formal description of the system behavior with the introduced features can provide an unambiguous and precise view of the system and of the new integrated features. The formal analysis and validation methods are also based on this formal description of the system.

The main subject of this thesis is the investigation of techniques based on FDT LOTOS for the *detection* of *unwanted* feature interactions.

1.3 Feature Interaction Contest

On the occasion of the Fifth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'98), an international Feature Interaction (FI) detection contest was held [GTGB98]. The contest offered an opportunity to compare the efficiency and the adaptability of different methods and tools in the detection of feature interactions. Inviting research teams of all schools of thought to compete, under controlled conditions, in accomplishing

a specific and predetermined task, set a basis to permit the assessment of the advantages and capabilities of the various methodologies.

The contest specifications modeled a telephony network as a collection of black boxes, communicating with each other via defined interfaces. They defined POTS and 12 switch-based and IN features as sequences of events taking place on these interfaces. The contestants were required to develop an automated FI detection tool and apply it to detect the FI between these features. The tool was evaluated by 1) its coverage, that is, the number of features actually defined in the tool language, 2) the number of valid feature interactions found [NBGO98].

Six teams joined the contest as follows,

- AT&T Research Labs, in New Jersey, USA
- Institute d'Informatique et de Mathematiques Appliquees de Grenoble, in Grenoble, France
- University of Ottawa
- University of Sherbrooke, in Quebec, Canada
- Uppsala University, in Sweden
- University of Waterloo, in Ontario, Canada.

The winner was the IMAG team from France and the University of Ottawa team was ranked second.

The IMAG team adopted a synchronous approach using Lustre as the specification language and Lustess as the FI detection tool [BORZ98], where the system, the features and the

property checker are modeled as synchronous systems. Synchronous systems have cyclic behaviors: at each tick of a global clock (also called instant of time), all inputs are read and all outputs are emitted. Every reaction to inputs is theoretically instantaneous. The FI detection system using Lustess consists of three components: the system under test, an input data generator and an oracle system. The input and output of the system are both boolean. The input data generator is built by Lustess according to the description of the environment that is interacting with the system and randomly produces test data at each instant of time in response to the system outputs. The oracle system plays the role of property checker, which checks the validity of the system based on the dynamically produced input to and program-reaction output from the system under test and outputs the verdict as Boolean.

The University of Ottawa team adopted two methods, both based on the use of the FDT LOTOS and its tools: one of the methods is presented in this thesis; another method, is based on the concept of Observers [QPLS99].

1.4 Contributions of the Thesis

The contributions of this thesis are in two areas: 1) a model of telephony network integrated with both switch-based features and IN features and 2) a system for detecting feature interactions at the specification level.

1.4.1 Contribution 1: A model for specifying in LOTOS a telephony system integrated with both switch-based features and IN features

In chapter 3 and 4, we present a model for specifying a telephony system integrated with both switch-based features and IN features using a mixture of resource-oriented style and state-

oriented style as well as a specially organized Abstract Data Type hierarchy. The billing database and the user status database are specified using ADT pyramids to reflect their hierarchical architectures. The system framework is specified in a resource-oriented style to preserve its interface integrity. The Basic Call Process and the integrated features are specified using the state-oriented style to enhance reusability and to preserve consistency with their scenario definition. By introducing Feature Activation Process, rapid feature integration is achieved in the sense that any feature, switch-based feature or IN feature, can be added to the global specification without any major modification.

1.4.2 Contribution 2: Adding the system properties into the traditional formal FI definition

In Chapter 5, we expand the traditional formal FI definition presented by P. Combes et. al [CoPi94] and W. Bouma [BoZu92], which addresses the FI problem as a violation of integrated feature properties due to the introduction of new features into the network, by adding system properties into the set of properties to be checked. Such properties include the correctness of billing and the consistency of successive signals given to user.

1.4.3 Contribution 3: A Feature Interaction Detection System

In Chapter 5, a Feature Interaction Detection System is described for detecting feature interactions between switch-based features and IN features. This system is developed upon the improved FI formal definition described in contribution 2. It is limited to interactions occurring at the abstract specification level and resulting in violation of system/feature properties. The FI Detection System consists of five parts: *Scenario Designer* designs specific test scenarios for the pair of features to be considered; *Integrator* integrates the test scenario and a global process monitoring system property violations into the system specification; *FI Hunter* detects those FI

traces violating the system properties, e.g. conflicting signals given to users or incorrect billing actions, and those potential FI traces where both features have been executed and need to be further analyzed; *Property Checker* checks the potential FI traces found by FI hunter with the properties of the activated features and filters out the real FI traces where the features properties do not hold; the final step, *MSC Translator* translates FI traces found by both FI Hunter (FIs violating the system properties) and Property Checker (FIs violating the feature properties) from LOTOS traces into the format of Message Sequence Charts (MSC) and compiles them into a final FI report.

An application of the system on two switch-based (CFBL, TWC) and two IN features (INTL, INFB) is presented in this thesis. However, all pair-wise combinations of twelve features were analyzed for the contest.

1.4.4 Contribution 4: Towards a method for feature interaction detection.

Although we do not claim that a general method for feature interaction detection was developed in this thesis, some contributions towards a method were presented. The main ideas are described in §5.3 §5.4 §5.5 §5.6. We start from the specification of system and feature properties, and then we provide several mechanisms for detecting violations of these properties.

1.5 Organization of the thesis

The four remaining chapters will cover the following issues:

Chapter 2: *Related Work: Formal Methods for Specifying the telephony networks and Detecting FI*

We present a survey of related work on formalisms used to specify telephony systems and on FI detection methods using FDT.

Chapter 3: *System Model Design*

We describe the design to a telephony system model integrated with both switch-based features and IN features and discuss the Basic Call Process, the classification of features, and the concepts of feature integration and activation. Two switch-based and two IN features are used as examples to illustrate the feature integration and the feature activation in the system model.

Chapter 4: *LOTOS Specification of the System Model*

First, we give a brief introduction to the LOTOS specification language by describing its main operators and by giving examples of their use in the context of telephony network specification. Then, we discuss four main styles of writing LOTOS specifications for telecommunication systems. Finally, we present a LOTOS formal specification developed for the telephony system model defined in Chapter 3. This is done using a mixture of resource-oriented style and state-oriented style, as well as a specially organized ADT hierarchy.

Chapter 5: *Feature Interaction Detection System*

We describe an improved formal definition of FI and a Feature Interaction Detection System (FIDS) developed based upon this definition. Two switch-based and two IN features are used as examples to illustrate how FIDS is applied to detect feature interactions between the given features. Finally, an evaluation of FIDS is performed by comparing the FIs detected by FIDS with those in the FI benchmark issued by the contest committee.

Chapter 6: *Conclusions and Future Work*

Conclusions and future work are presented in this chapter.

Chapter 2. Related Work: Formal Methods for Specifying the Telephony Networks and Detecting FIs

Traditional engineering disciplines rely heavily on mathematical models and calculation to make judgments about designs. For example, aeronautical engineers make extensive use of Computational Fluid Dynamics (CFD) to calculate and predict how particular airframe designs will behave in flight. A variety of methods with similar goals are available in computer science and engineering. Quantitative simulation methods are among them; however, they do not relate to the research area of this thesis. We concentrate on methods that have their foundation in logic and formal semantics. Such methods are called "formal methods"[Turn93].

Formal methods can be used to determine the logical properties of systems with respect to their functional behaviors. Very well-known properties in this family are "deadlock" properties. Others all relate to the fact that certain post-condition are satisfied or not. We will see a number of such properties in this thesis.

In this chapter we conduct, with no attempt to be exhaustive, a survey of a number of formal methods and languages that are used for the specification of telephone systems and features, as well as of FI detection methods using Formal Description Techniques (FDT).

2.1 Formal Specification Methodologies for Telephony Systems

2.1.1 Finite State Machines

A Finite State Machine (FSM) is an abstract machine that is used to represent the behavior of a given system in terms of *states* and *transitions*. The most common notation used to represent a

FSM is a directed graph whose nodes are system states and whose arcs are system transitions; the other notation being state transition matrices. The machine can be in only one state at a time. Upon receiving an input, the machine generates an output and may change to a new state. Both the output and the new state are functions of the input and the current state. A state is a mean by which one can describe an aspect of the system's behavior. For example, one may talk about a *Dialing* state, a *Ringing* state, or a *Talking* state while describing the behavior of a telephone system. Telephony applications described using FSM can be found in [KaWa71] [WhCh81].

2.1.2 Petri Nets

Petri nets [Pete 77] [NaKa97] are abstract machines that are used to describe the behavior of systems. They are represented by a directed graph containing two types of elements: places and transitions. Places, which contain tokens, are represented by circles; transitions, which allow tokens to move between places, are represented by lines. Directed graphs connect places to transitions. A transition is said to fire if 1) it is triggered by a clock pulse and 2) all arrows entering the transition originate from places that contain tokens.

The Petri-net based model has been used to describe, among other applications [Ager79], the behavior of telephone switching systems [YoBa79]. Yoeli and Barzalai introduce the concept of extended Petri Nets (EPN) and use it to model the call processing operations in an automatic telephone exchange. In their approach, the telephone system is decomposed into a set of virtual subsystems: a virtual station subsystem (VSS) representing the user's station, a virtual station control (VSC) representing the central exchange, a virtual dial control (VDC) collecting the dialed digits, and a virtual central control (VCC) representing the module which handles the establishment

of a connection between two users. When a user dials a digit, it is transmitted to the VDC through the VSC. Once the caller has reached the callee, the connection is handled by the VCC.

Two common problems with the FSM and Petri-nets are: 1) the limited role they assign to data. Many features rely on data values and data structures for the essential aspects of their functionalities. However, data aspects take a secondary role in these formalisms. 2) the lack of process structure, which is very useful for design. Extended Finite State Machine (EFSM) methods, such as SDL, remedy this situation.

2.1.3 SDL

SDL (Specification and Description Language) is the most widely used FDT in the field of telecommunications [BeHo89]. It has been developed and standardized by CCITT (the International Telegraph and Telephone Consultative Committee) and ITU (International Telecommunication Union). SDL is used to describe both the behavior and structure of systems, from a high description level down to a detailed design level. The behavior of a system is described in terms of a set of processes, which are extended finite state machines. Processes work concurrently and communicate asynchronously with each other by sending and receiving discrete messages called *signals*. Signals are also the means by which SDL processes communicate with the environment. When signals are used to communicate between processes, they always carry the unique identifiers of the sending and receiving processes, along with possible data values. Examples of specifying telephony systems using SDL are presented in [CHCk89] [CoPi94].

A problem with SDL formal language is that it enforces rigid system boundaries in the form of process and blocks. Although these are useful to represent system architecture, they may

increase the diffculties in the early design stage when the system architecture is not quite clear. LOTOS structure, which consists of only processes, is more flexible.

2.1.4 LOTOS

An early study [FaLS90] has shown that LOTOS is well suited for specifying elementary telephone systems, basically the Plain Old Telephone System (POTS). The results of that study further motivated the research on specification styles. A formal specification of telephone systems, using the constraints-oriented style was described in [FaLS91]. The work presented in [StLo93] describes a new approach for specifying telephony systems using a mixture of the constraint-oriented style and the state-oriented style. In [KaLo98], a formal specification of IN network model was developed using the resource-oriented style. The telephony network system model in this thesis is specified using a mixture of resource-oriented style and the state-oriented style. More details of the specification styles can be found in § 5.2 and in [FaLS97]

2.2 FI Detection Methods using FDTs

Feature interaction is a research area of some importance, and a number of papers are published every year on the subject. Five International Workshops have been held so far [1stInt.92] [2ndInt.94] [3rdInt.95][4thInt.97][5thInt.98], where detection approaches from various research areas, e.g. software engineering theory, formal description techniques etc., are presented.

In the following, we limit ourselves to briefly reviewing work closely related to ours, which uses LOTOS as FDT to detect FI at the specification level.

2.2.1 Step-by-Step Execution

Boumezberur and Logrippo [BoLo93] proposed a LOTOS specification of 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 methodology 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 1) test sequences derived from the informal definition are accepted by the formal specification, 2) test sequences obtained by executing the specification are included in the formal definition of the system, 3) test sequences that are not specified in the informal definition are not accepted by the formal specification.

2.2.2 Model Checking

Model checking is a method for formally verifying finite-state concurrent systems. Specifications about the system are expressed as temporal logic formulas, and efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not.

Many FI detection methods have been developed using Model Checking: 1) [BoZu92] modeled IN services as defined in the Global Functional Plan 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. 2) [CoPi94] 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. 3)

Using LOTOS as a formal language, [Thom97] modeled features as user view behavior trees, which are synchronized to form a network of users interacting with a "network manager" to complete the call process. Features' properties are specified using μ-calculus and verified using CAESAR model-checking evaluator.

2.2.3 Backward Reasoning

Stepien and Logrippo [StLo95] 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.

2.2.4 Conformance Theory

In [FaLS97], Faci and Logrippo developed a methodology for detecting feature interactions using conformance theory. First, they defined two 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 that 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*.

2.2.5 Abstract Data Types

In [SteL95], 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.

2.2.6 Goal Oriented Method

In [KaLS98], Kamoun and Logrippo developed a method for detecting feature interactions between IN services using the Goal Oriented method. 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 the feature 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.

Our FI Detection method presented in chapter 5, was first inspired from the idea of the Goal Oriented method. However, considering the burden that a complex goal may cause on the goal-matching tool, we simplified the goal to be just an “error reporting” event and a “call process finish” event. We also let a global monitoring process capture the violation of system properties and perform a static feature property check on the snapshot of the billing data taken right before the “call process finish” event.

Chapter 3. System Model Design

In this chapter, we present a telephony system model integrated with both switch-based features and IN features. It should be noted that the system is modeled after the definition of the FI detection contest rather than after the functionality of a 'real' telephony system.

3.1 General Architecture of the System Model

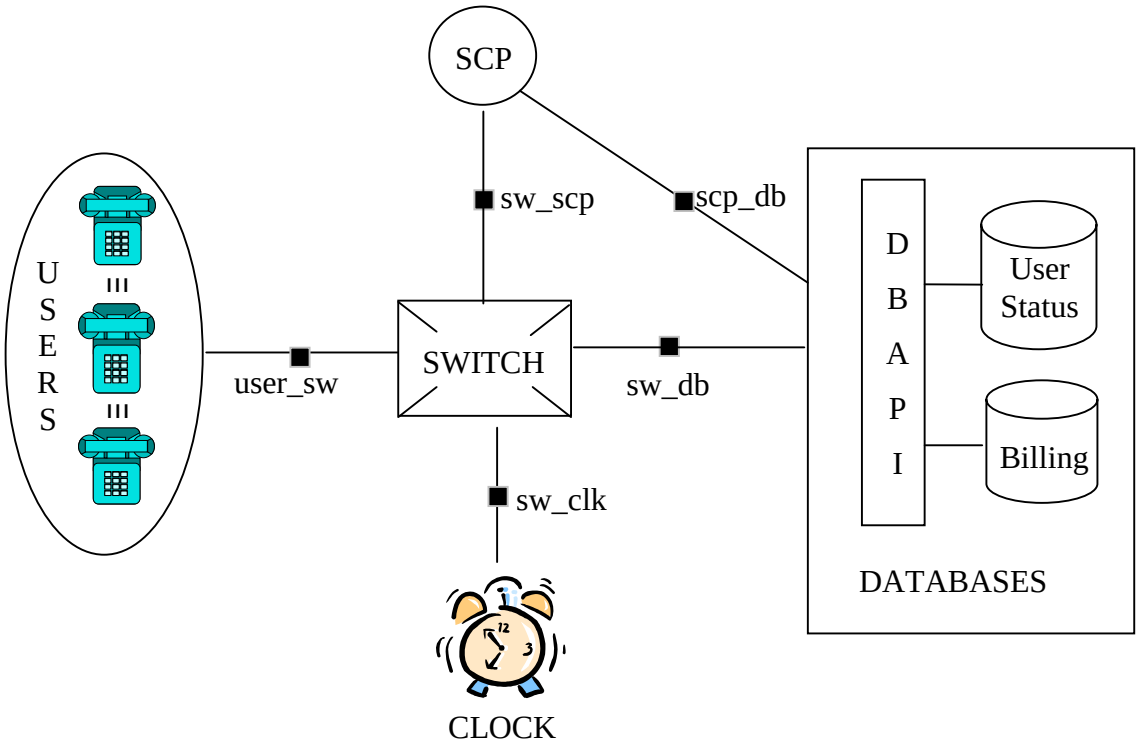


Fig 3.1 Architecture of the System Model

As depicted in Fig 3.1, our telephony network system is modeled as a collection of black boxes communicating with each other via defined interfaces between them. Interfaces are represented by solid black squares and named after the two components involved. For instance, the interface between the switch and the clock is named by "sw_clk".

- Users

Three users *A*, *B* and *C* send corresponding signals to the switch when users perform offhook, onhook, dial actions on them; and respond with ringing, audibleringing (an audible tone to the caller indicating that the destination phone is ringing) etc., when they receive corresponding signals from the switch. Note that in this thesis, for simplicity, users model both the "machines" and the people who operates them.

- Switch

The switch is the main engine of the whole system model. It consists of three parts: Basic Call Process (BCP) providing only the basic functionality of making phone calls, 12 features providing the network subscribers more control in the call establishment process and Feature Activation Process linking the BCP and the integrated features together.

The BCP processes the signals that come from users, consults user status information stored in the user status database, establishes the call connection and logs the billing actions into the billing database while the called party answers the call.

The twelve integrated features are as follows:

- CFBL

Call Forwarding on Busy Line (CFBL) that redirects all calls to the subscribing line to a predetermined number when the line is busy.

- CND

Call Number Delivery (CND) that allows the called telephone to receive a calling party's directory number, and the date and time.

- INFB

IN Freephone Billing (INFB) that allows the subscriber to pay for incoming calls.

- INFR

IN Freephone Routing (INFR) that allows the subscriber to redirect a call to various telephones potentially using the whole or part of the calling number and/or the time of day.

- INTL

IN Teen Line (INTL) that restricts outgoing calls based on the time of a day. This feature can be overridden on a per-call basis by anyone with the proper identity code (PIN).

- TCS

Terminating Call Screening (TCS) that restricts incoming calls by redirecting calls from lines that appear on a screening list to a vague but polite message.

- TWC

Three-way Calling (TWC) that allows the connection of three parties in a single conversation.

- INCF

IN Call Forwarding (INCF) that permits the subscriber to have incoming calls redirected to another number.

- CW

Call Waiting (CW) that allows the subscriber to be notified that another party is trying to reach him/her while the line is busy, and to accept the new call by placing the original call on hold.

- CC

Charge Call (CC) that allows a caller to be automatically charged on a different telephone number than the calling number.

- CELL

Cellular (Cell) that charges cellular subscribers a fixed fee for each minute when a call is in progress

- RC

Return Call (RC) by which the subscriber can set up a call to the last caller by dialing *69.

- SCP

If the user subscribes to Intelligent Network (IN) features, the Service Control Point (SCP) will replace the switch to control the call process when IN features are activated.

- Clock

The clock provides the switch with the global time to log billing actions or make time-dependant decisions. For example, the user subscribing to INTL has to dial a valid PIN to originate a call during the specific INTL time period while no PIN is required outside that time period. The switch will get the current time from the clock and send the information to

the SCP. The SCP compares the current time with the INTL time period stored in the user status record and decides if a PIN validation procedure is necessary or not.

- Databases and DBAPI

Our database system consists of two parts: *Databases* and *DB Application Interface (DBAPI)*. Databases store billing data and subscription data. External applications (e.g. the switch) get access to these information via the DBAPI. DBAPI hides implementation details of the databases from external users (the switch and the SCP).

3.2 Notation Description

The notation used in this thesis to define BCP, FAP and the features is based on Labeled Transition Systems.

Labeled Transition Systems (LTS) are a variation of the Finite State Machine formalism where transitions are labeled with action names [Miln89]. The most common notation used to represent LTS is a directed graph whose nodes are system states and whose arcs are system transitions. The machine can be in only one state at a time. Upon executing the labeled action, the system moves to a new state along that arc.

- *States*

In our model, LTS have three kinds of states: 1) start state which has only “out” arcs, 2) intermediate states which have both “in’ and “out” arcs and 3) end states which have only “in” arcs. For reference, the intermediate states of LTS presented in this thesis are numbered from “1”, the start state and the end state are marked with “S” and “E” respectively, in gray circles.

- *Transitions*

In our model, transitions are labeled by actions of format as follows:

[[Guard] ->] Interface_name, Signal_name, [Parameter1, Parameter2,…]

Guard

Guard is optional and only used for restricting the transition's occurrence: only if the guard is satisfied, could the transition be executed. In our model, guard is usually a logic expression or a check on user status, e.g. guard "Busy B" means "user B is busy".

Interface_name

Interface_name indicates the name of the interface where the signal occurs. We have five interfaces in our system model: *user_sw* (the interface between the users and the switch); *sw_db* (the interface between the switch and DBAPI); *scp_db* (the interface between the SCP and DBAPI); *sw_clk* (the interface between the switch and the clock); *sw_scp* (the interface between the switch and the SCP).

Signal_name

Signal_name identifies the name of the signal. Signal_name is unique and belongs to only one interface. For example, the "offhook" signal can only occur at "user_sw" because only the users can send out the "offhook" signal.

Different features may have different set of signals. We will discuss it along with the individual description of features later.

Parameters

Parameters carry the data information of signals. Signals can have 0 or more parameters depending on the type of signals. For example, user signals (signals on "user_sw" have 1 or 2 parameters where the first one indicates the address to or from which the signal is given, the second parameter carries additional information such as, the identification of the user that causes the signal, e.g. "user_sw Ringing B A" means the ringing signal is sent to B because of A (dials B). We will explain this formula along with signals in *§3.3 Description of Features.*

- *Multiple-action Transitions*

For simplicity, we compress LTS Trees using multiple-action transitions. Multiple-action transitions are transitions labeled by a series of actions. To execute a multiple-action transition is to execute all actions belonging to that transition sequentially.

3.3 BCP

Basic Call Process provides basic telephony functionalities. It identifies at a high level of abstraction all the activities necessary to establish a normal call between parties in the system. As described in Fig 3.2, BCP starts when the caller performs *offhook* and ends when both calling and called parties hang up. Since the switch takes the role of controlling the whole call process, BCP is implemented within the switch. Thus, signals from the user such as: *offhook, onhook, dial,* etc are input signals to BCP and signals going to the user such as: *ringing, audibleringing, announcement, dialtone* are responses from BCP to those input signals. Besides the end-users, BCP also communicates with DBAPI and the clock, inquiring user's status information from the user status database to establish the call or adding new billing records with timestamps into the billing database when the call is connected. Fig 3.2 shows the LTS tree of BCP.

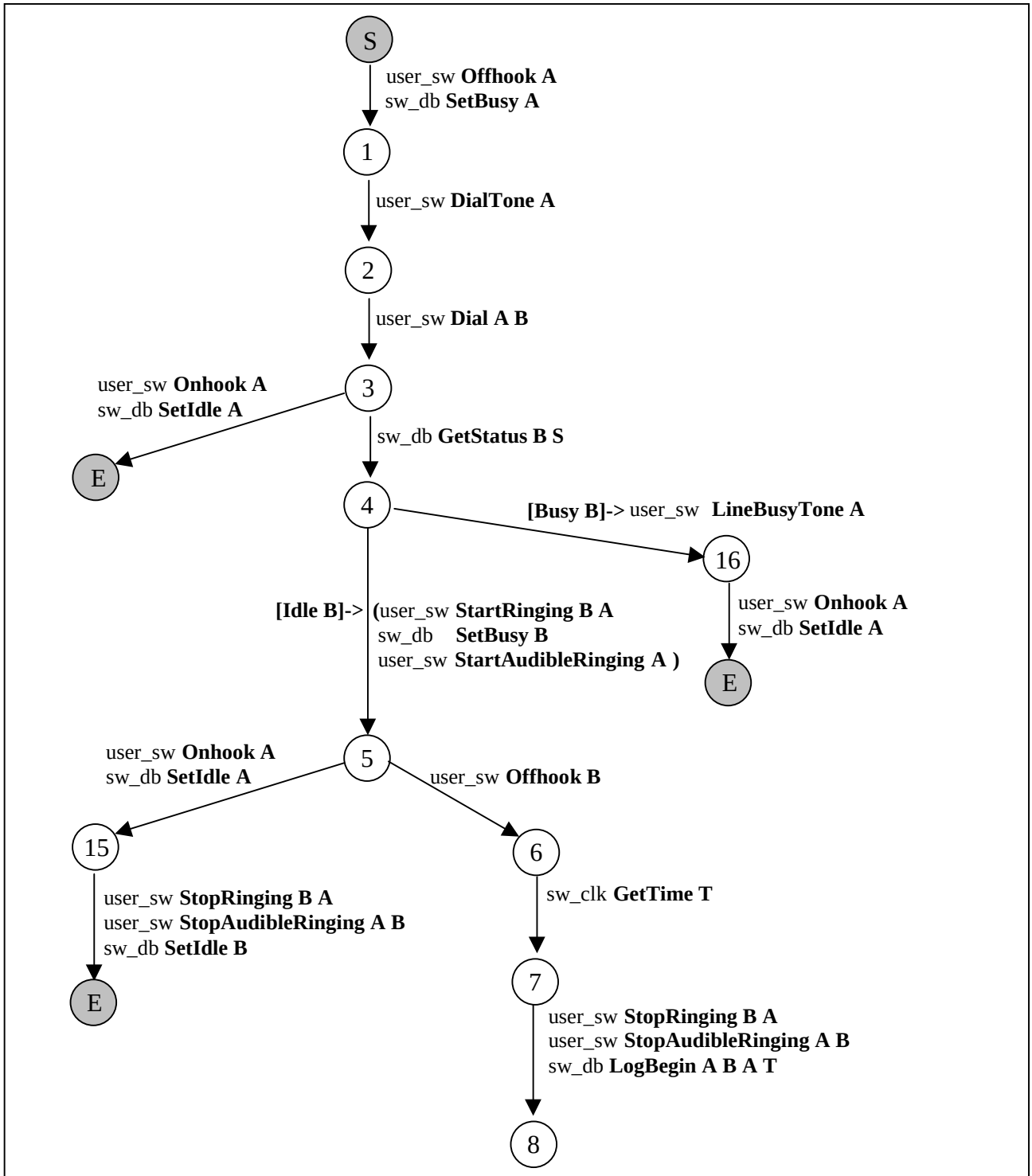


Fig 3.2 The LTS Tree of BCP (To be continued)

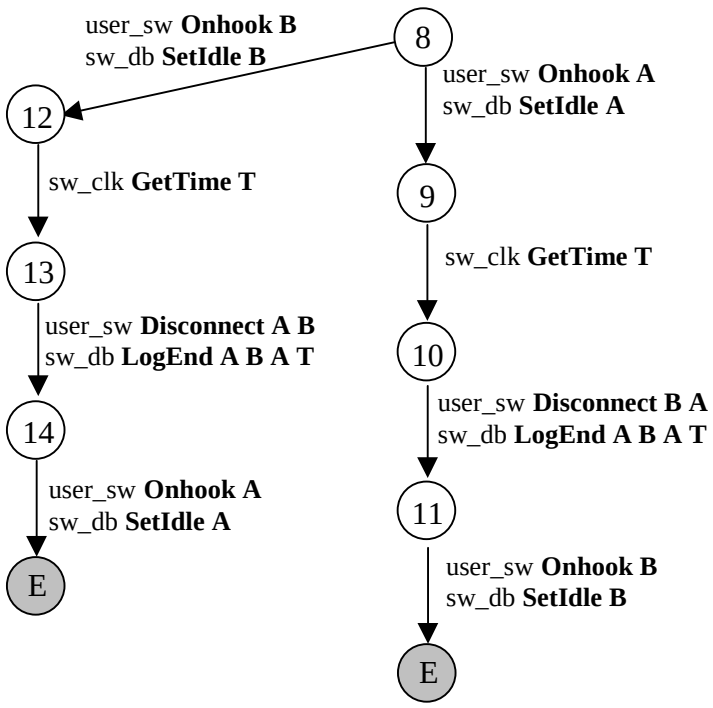


Fig 3.2 The LTS Tree of BCP (Continued)

- Points In Call:

The numbered intermediate states in Fig 3.2 are also called *Points In Call* (PIC), where 1) the Feature Activation Process will be attached to activate corresponding features and (Point of Initialization or POI) 2) the activated features return to BCP upon completion. (Point of Return or POR) (*see §3.4 Feature Activation Process)*

- Interfaces

Three interfaces, *user_sw, sw_clk, sw_db*, are used by BCP.

- Signals

In BCP, 1) signals at "user_sw" are: *Offhook, Onhook, Dial, Dialtone, LineBusyTone, StartRinging, StartAudibleRinging* (when the called party is rung, the caller can also hear a corresponding ringing tone) and *Disconnect*; 2) signals at "sw_db" is *GetStatus* (inquire user's status), *SetBusy* (set user's status to be busy), *SetIdle* (set use's status to be Idle), *LogBegin* (Add a new billing record and log the call beginning time), *LogEnd* (log the call ending time); 3) the only signal at "sw_clk" is *GetTime* (read the time from the clock).

- Parameters

In BCP, 1) signals at "user_sw" have one or two parameters, where the first parameter indicates the user who sends or receives the signal. For example, A is the sender of the signal "*user_sw* Offhook A" and B is the receiver of the signal "*user_sw* StartRinging B A". The second parameter usually describes who causes the received signal. As in the previous example, the second parameter specifies that B is rung by A. However, in "*user_sw* Dial A B", the second parameter indicates the callee's number that is dialed by A. 2) "Get status" , "SetBusy" and "SetIdle" signals at "sw_db" have one parameter indicating whose status is inquired or changed. 3)"LogBegin" and "LogEnd" signals at "sw_db" have four parameters, the first and the second parameters respectively specify the caller and the callee of the call, the third parameter tells who is the payer of the call and the last parameter holds the beginning (or ending) time of the call. (see transition from state 7 to state 8) 4)"GetTime" signal at "sw_clk" has one parameter, storing the current time read from the clock.

- Possible Exits

In BCP, after "Offhook A", a call process has five possible exits: 1) the caller A onhooks after it dials the callee's number; 2) the callee B is busy when A is calling, thus after hearing the linebusytone, A hangs up; 3) A onhooks while B is being rung; 4) the caller A onhooks first when finishes talking with B; 5) the callee B onhooks first when it finishes talking with A. Only in the last two cases, a real connection between A and B is successfully established.

3.4 Feature Activation Process

3.4.1 Feature Integration: POI & POR

In an IN-like model, all telephony features other than BCP are built upon BCP and interact with it at two points which, from the feature's point of view, are called:

i. *Point Of Initialization* (POI) is a PIC in BCP where the feature is activated. All telephony features other than BCP have one and only one corresponding POIs.

ii. *Point Of Return* (POR) is a PIC in BCP where normal call processing should continue after executing the feature. One feature could have 0 (if it never returns to BCP) or more PORs.

Figure 3.3 illustrates the integrating relationship between BCP and feature INTL, CFBL, TWC and CW.

For feature INTL (see *§3.6.1 INTL*), the POI is PIC_ 1. If the caller A subscribes to INTL, the feature is activated right after caller A offhooks. The POR of INTL is PIC_2. If INTL does not

block the call (either not in INTL time period or in INTL time period but the user has a valid PIN for the call origination), a dialtone is given to caller A. Then, INTL finishes and the call process returns to BCP and resumes from PIC_2.

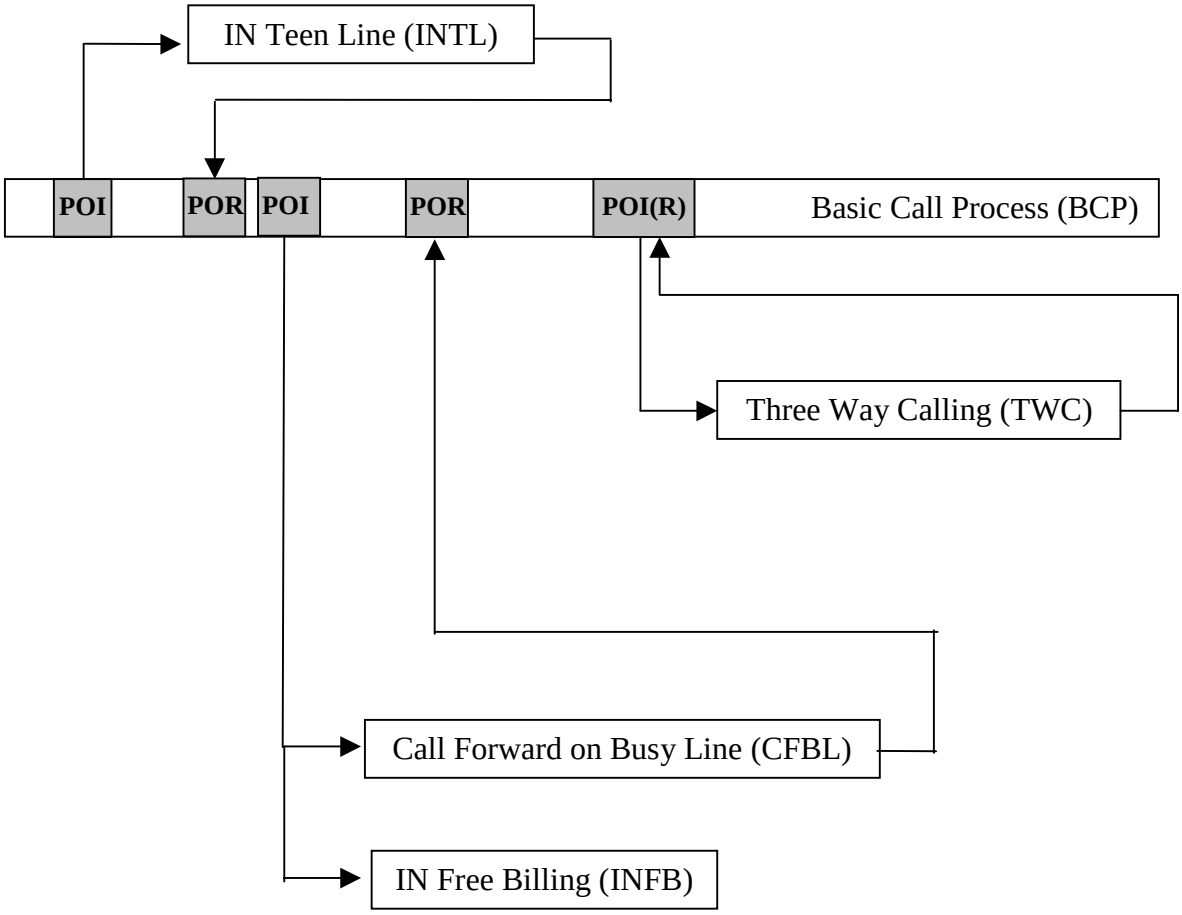


Fig 3.3 The Integrating Relationship between BCP and Other Features

The POI of CFBL is PIC_3. If the callee B subscribes to CFBL, the feature is activated after caller A dials the callee's number B. If B is busy when A calls, CFBL of B forwards the call to C, a predefined forwarded address. The POR of CFBL is PIC_5. If A calls B when B is idle, the call process returns to BCP and continues from PIC_5.

The POI of INFB is PIC_3. If the callee B subscribes to INFB, the feature is activated after the caller A dials the callee's number B. If the call is connected to B, INFB of B charges the call to the callee B. INFB has no POR.

The POI of TWC is PIC_8. If the caller A (or the callee B) has TWC, the feature can be activated after A and B enter the talking state. A (or B) can dial the third party C during the call with B (or A) by performing flashhook and can establish a three-way connection among A B C. When one of A B C onhooks, TWC finishes and returns to PIC_8, which is the two-way connection state.

3.4.2 Feature Activation Process

Feature Activation Process (FAP) is a process that is instantiated at every POI of the integrated features to activate the subscribing features.

Two parameters are passed to FAP from BCP, user address and feature name. FAP will do two things: 1) check if the user subscribes to the feature or not. The subscribing information is stored in TheUser database; 2) If the feature is subscribed, FAD calls the feature process and passes all associated parameters, such as the caller and or callee's name to it. Otherwise, FAP returns to BCP and the call process is resumed.

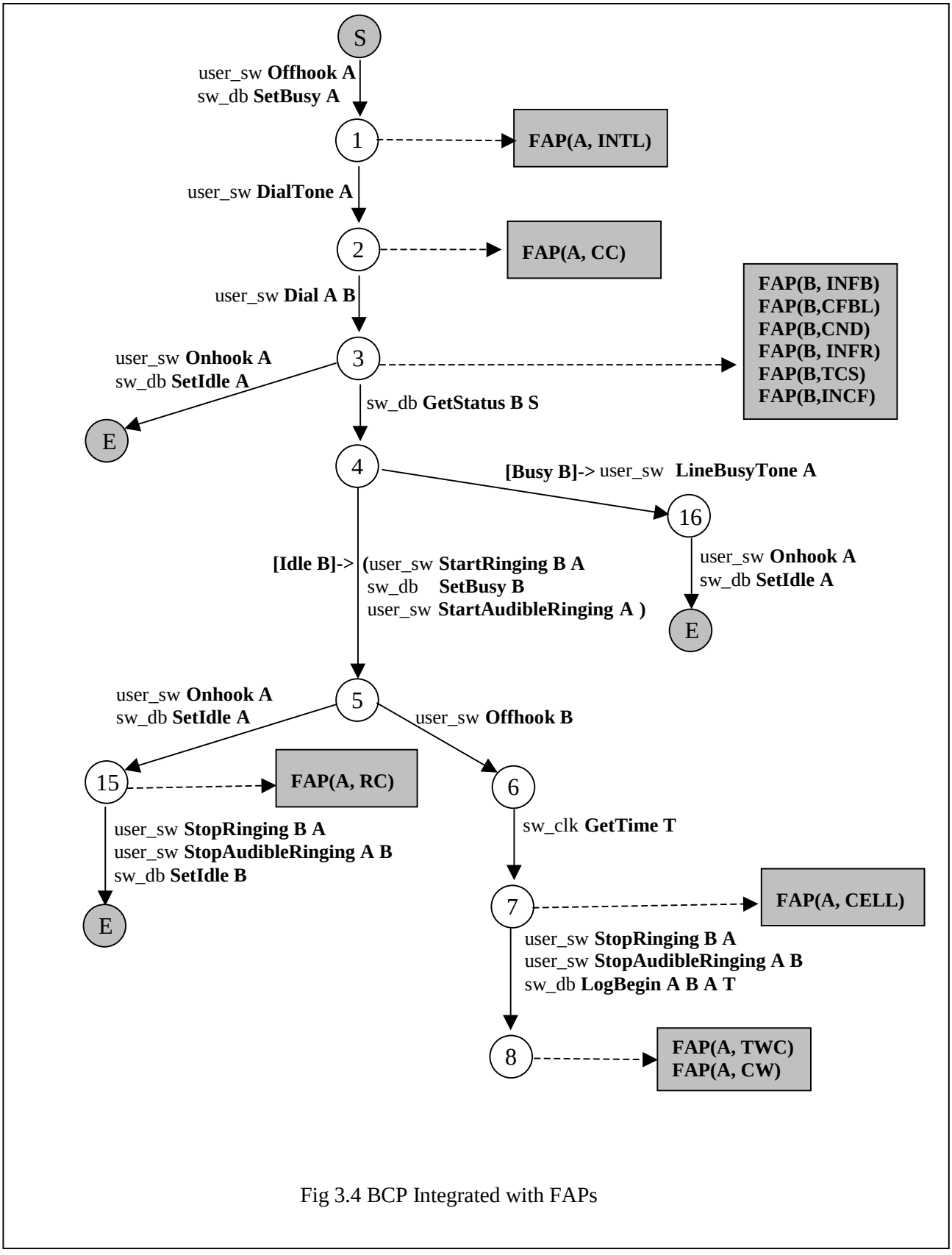


Fig 3.4 BCP Integrated with FAPs

At PICs that are POIs for more than one feature, FAP instances of different features are mutually independent and of the same priority. Fig 3.4 lists FAPs integrated into BCP.

3.5 Feature Classification

Features in telecommunication systems are packages of incrementally added call functions providing advanced call features to subscribers [Bowe89]. These packages are provided to users on a subscribe-and-use basis.

We use a classification of features that is based on the way they are integrated to the system and on the way that they can be activated only once or repeatedly. From the first point of view, we talk about switch-based (Non-IN) and IN feature. From the second point of view, we talk of finite and infinite feature.

3.5.1 Switch-based Features

Features that are implemented within the switch are called switch-based features. This is the traditional way to add new features (before 1980s). In this method, since all data and processing required by the features are located within the local node (the switch), new features must be added to all local switches. Moreover, since different types of switches provided by different vendors are deployed in a telephony network, the introduction of new features requires the adaptation of the related software for every type of switch in the network.

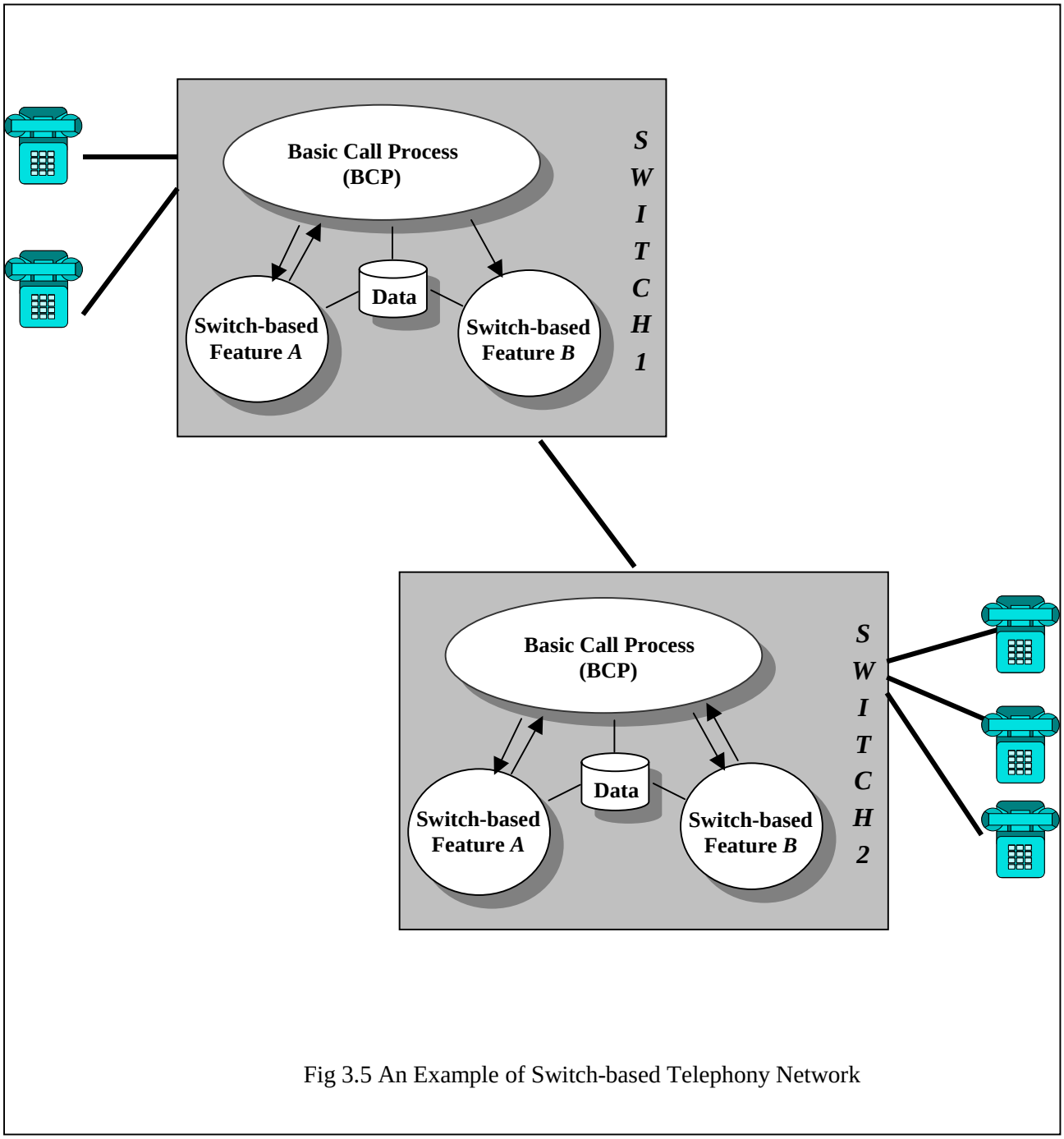


Fig 3.5 An Example of Switch-based Telephony Network

Fig 3.5 depicts an example of switch-based telephony network. Three features, BCP plus two new added features are implemented in two local switches. Each switch has its own local database, which stores data required by BCP and other switch-based features.

In our system model, CFBL, CND, TCS, TWC, CW,CC, CELL and RC are switch_based features.

3.5.2 IN features

As mentioned before, switch-based telephony features and corresponding data must be implemented in every local switch in the network. This method is tedious and it is costly to introduce new features or improve old features. The introduction of Intelligent Network (IN) eased the difficulty of feature creation, deployment and maintenance by facilitating creation and provision of telecommunication services. In IN telephony networks, new features are implemented in Service Control Point (SCP) and corresponding data required by IN features are stored and managed by Service Data Point (SDP). Unlike BCP and other switch-based features that are completely implemented within the local switch, part of the functionality of IN features is carried out by the SCP. During the execution of IN features, the call process control remains in the switch while the feature process control is done by the SCP. The switch provides the SCP with collected information and follows the decision made by the SCP. The interface between the switch and the SCP is service-independent, which means that the communication style between the SCP and the switch remains the same for all IN features.

Fig 3.6 depicts a simple IN telephony network, which consists of two local switches and one SCP/SDP. We can see the advantage of the IN features directly from the picture. Unlike

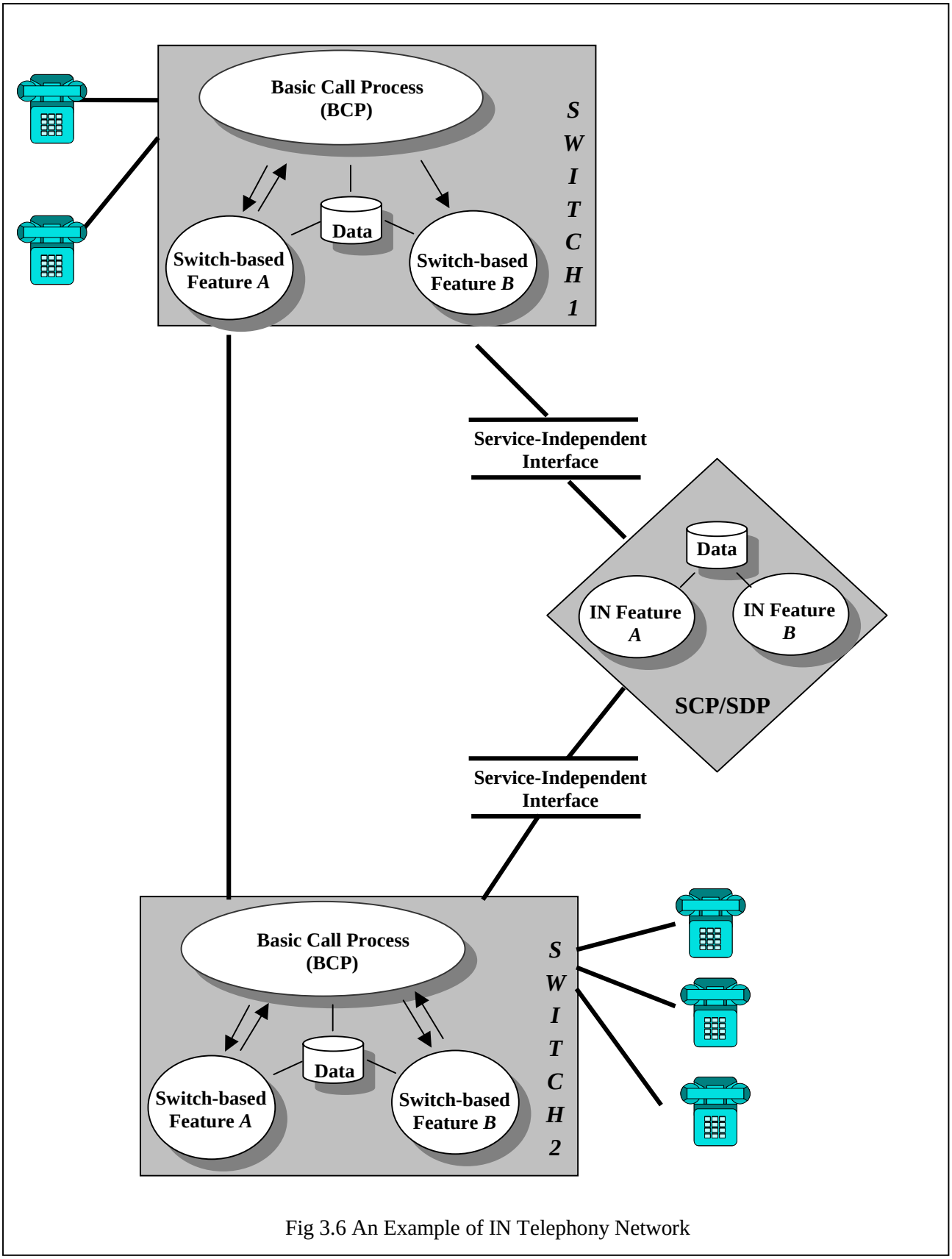


Fig 3.6 An Example of IN Telephony Network

switch-based features, which need to be implemented in both switches in the network, IN features and data are deployed only in the SCP/SDP.

Due to legacy, nowadays telephony systems usually have both switch-based features and IN features.

In our model, IN features are: INTL, INFB, INFR, INCF and CC

3.5.3 Finite Features

Finite Features are those features that can be executed only once during a single call process. The main property of finite features is that PORs of finite features can only occur after the POIs.

In our model, finite features are CFBL, CND, INFB, INFR, INTL, TCS, INCF, CC, CELL and RC.

3.5.4 Infinite Features

Infinite Features are those features that can be executed repeatedly during one call process. The main property of infinite features is that their PORs occur at the same time or earlier than the POI.

In our model, TWC and CW are infinite features for which the POR and the POI are the same PIC (PIC_9).

Fig 3.7 shows the classifications of features.

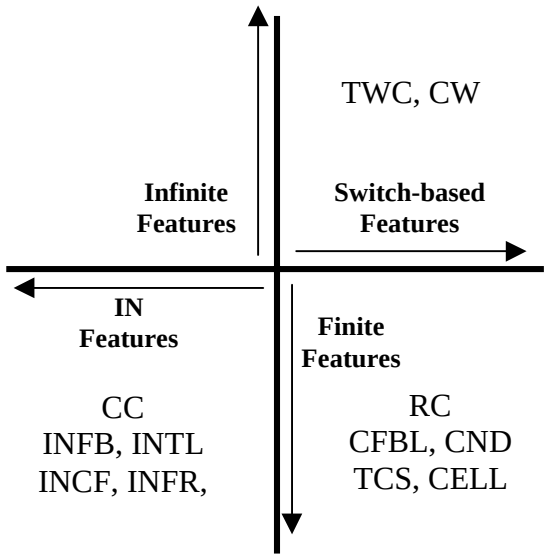


Fig 3.7 Classifications of Features

3.6 Descriptions of Features

Since it would be very long to describe all 12 features we implemented and analyzed, 4 representatives, INTL, INFB, INCF, TWC are selected as examples to show how different kinds of features are designed.

3.6.1 INTL

INTL restricts outgoing calls based on the time of the day, such as hours when homework should be the primary activity. However, the restriction of INTL can be overridden by entering the correct PIN. When the user subscribes to INTL, the following information is required from the user:

1) *TeenTime1 TeenTime2*: a time period from TeenTime1 to TeenTime2 when the outgoing calls are restricted.

2) *TeenPIN:* a PIN used to originate a call during the TeenTime period

When the caller A, who subscribes to INTL, offhooks, INTL is activated by FAP from PIC_1. Fig 3.8 illustrates the LTS tree of INTL. The first transitions of INTL involve reading the current time, geting the TeenTime period of A from the user status database and checking if it is in the TeenTime period. If it is, INTL sends a *trigger,* to the SCP with the trigger type (ORIGINATION_ATTEMPT), the subscriber's address (A), the caller's address (A) and the time just collected. Otherwise, a dialtone is given to user A and INTL returns to BCP at PIC_2, giving the caller A a dialtone. After receiving INTL's trigger message, the SCP responds to askPIN from the caller A. INTL announces to A a prompting message to dial the PIN. Then, INTL sends a "*resource"* message to the SCP with the number P dialed by A. If P is the valid TeenPIN, the SCP responds to continue the call, a dialtone is given to user A and INTL is returned back to BCP at PIC_2. Otherwise, A will receive an announcement that an invalid PIN was given and the call is blocked by the SCP's "RES_DISCONNECT" response. (see *§3.7.2 Switch/SCP* for definitions of Trigger, Response and Resource.) In Fig3.8, transitions between states from 3 to 10 are interactions between the SCP and the user through the switch.

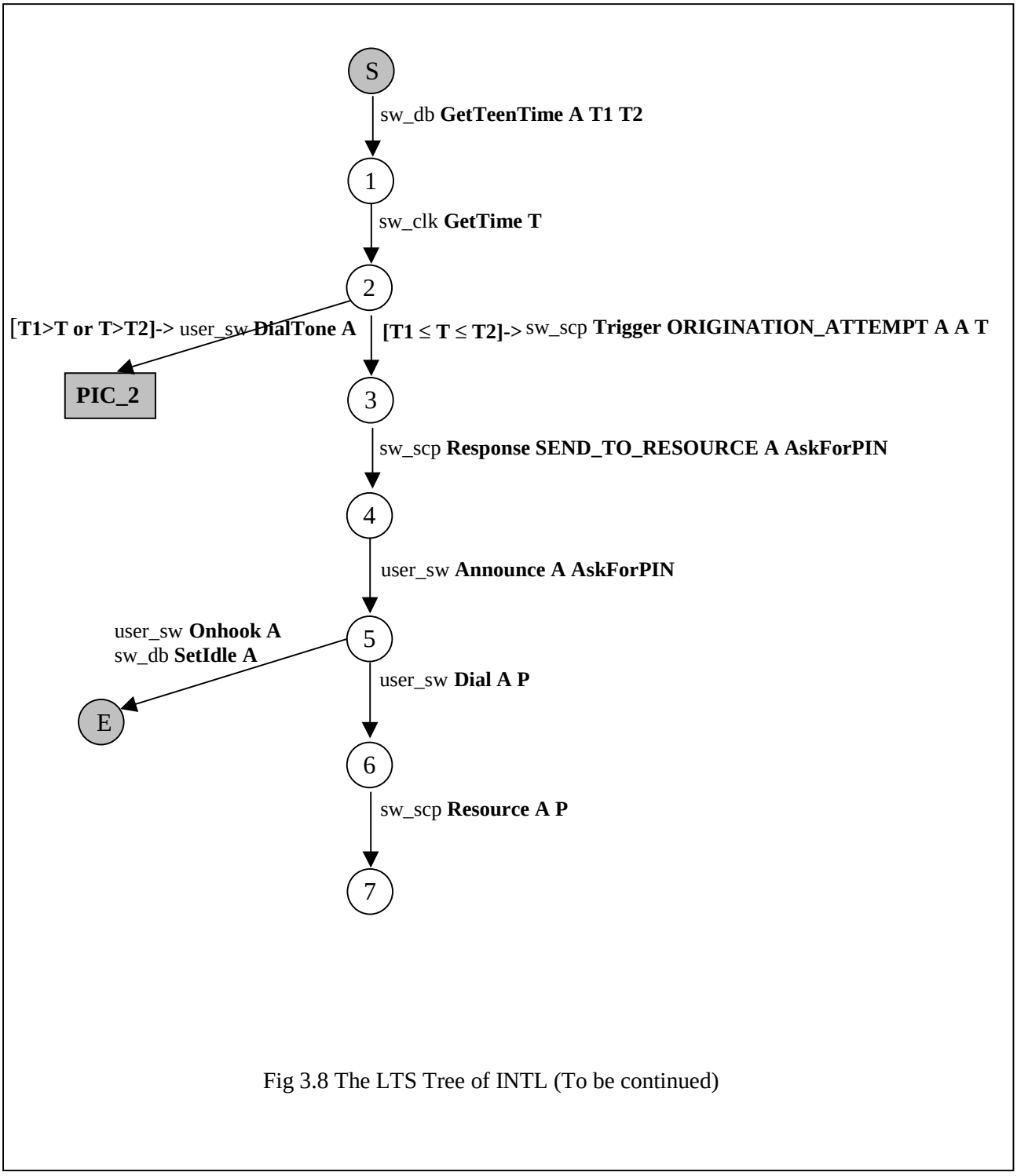


Fig 3.8 The LTS Tree of INTL (To be continued)

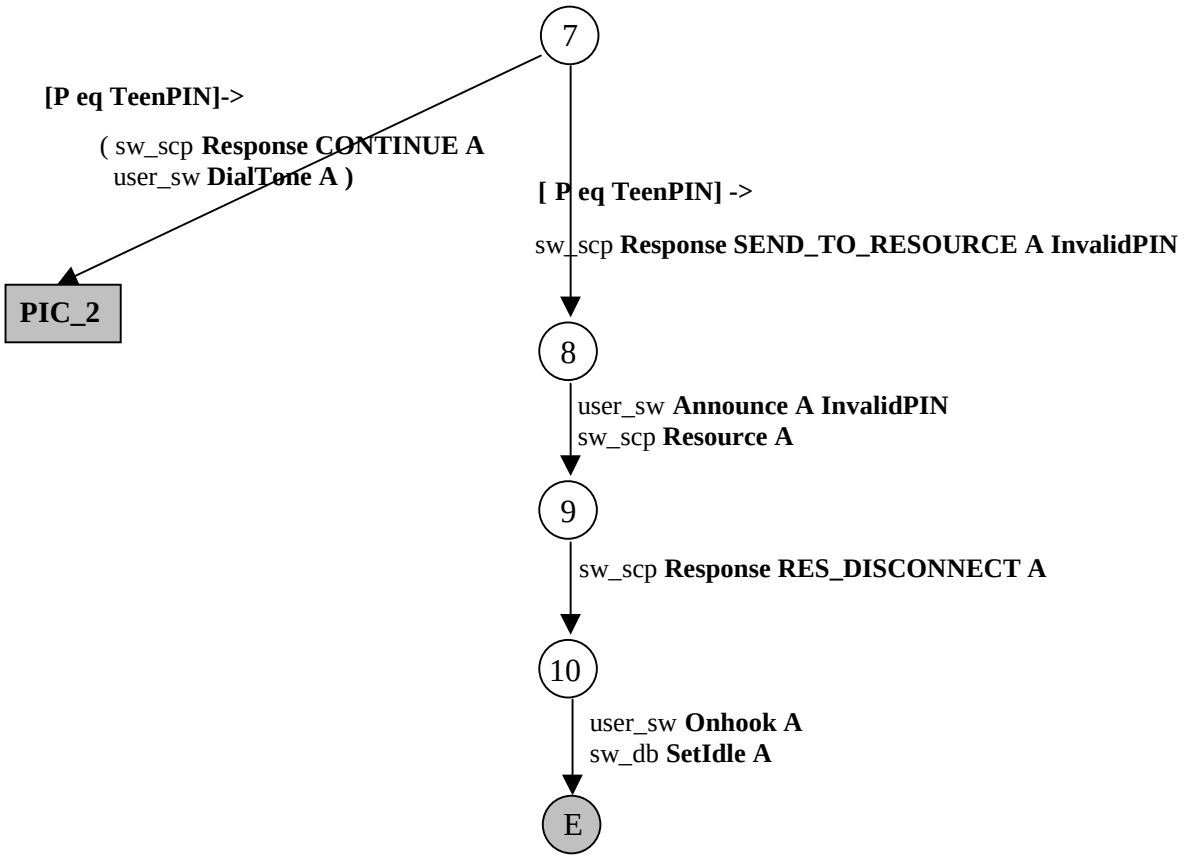


Fig 3.8 The LTS Tree of INTL (Continued)

- Signals:

  Signals of INTL feature occur at four interfaces.

  1) Signals at *user_sw*. Comparing with BCP, INTL has only one new signal “Announce”. The “Announce” signal has two parameters. The first one refers to the receiver of the signal and the second parameter is the message to be announced, such as prompting the user to input a PIN number (“AskForPIN”) or informing the user that an invalid PIN number is input (“InvalidPIN”).

2) Signal at *sw_clk*, "GetTime", is the same as in BCP.

3) New Signal at *sw_db* is "GetTeenTime". The "GetTeenTime" signal queries the "TeenTime" period from the user status database. It has three parameters: the first one, A, indicates the subscriber's name; the last two parameters T1, T2 take the starting and ending time of the "TeenTime" period.

4) Signals at *sw_scp.* As mentioned above, two kinds of signals, "Trigger" and "Resource", are sent to the SCP from INTL. The SCP responds to "Trigger" and "Resource" signals with the signal named "Response".

- Possible Exits

INTL has four exits: 1) The current time is not in TeenTime period, A dialtone is given to user A and INTL returns to PIC_2. 2) The caller A onhooks after being announced the prompting message to dial the PIN. 3) The caller inputs a valid PIN and the SCP responds to continue the call. A dialtone is given to caller A and INTL returns to PIC_2. 4) The caller inputs an invalid PIN and the SCP responds to disconnect the call. The caller A onhooks. Only in the first and the third cases, A is allowed to originate a call.

3.6.2 INFB

INFB enables the subscriber to pay for incoming calls.

The LTS of INFB is shown in Fig 3.9.

When the callee B, who subscribes to INFB, is dialed by the caller A, INFB is activated by FAP from PIC_3 and takes the place of BCP to control the call process. The first transitions of INFB is to read the current time from the clock and send a *trigger* message to the SCP with the

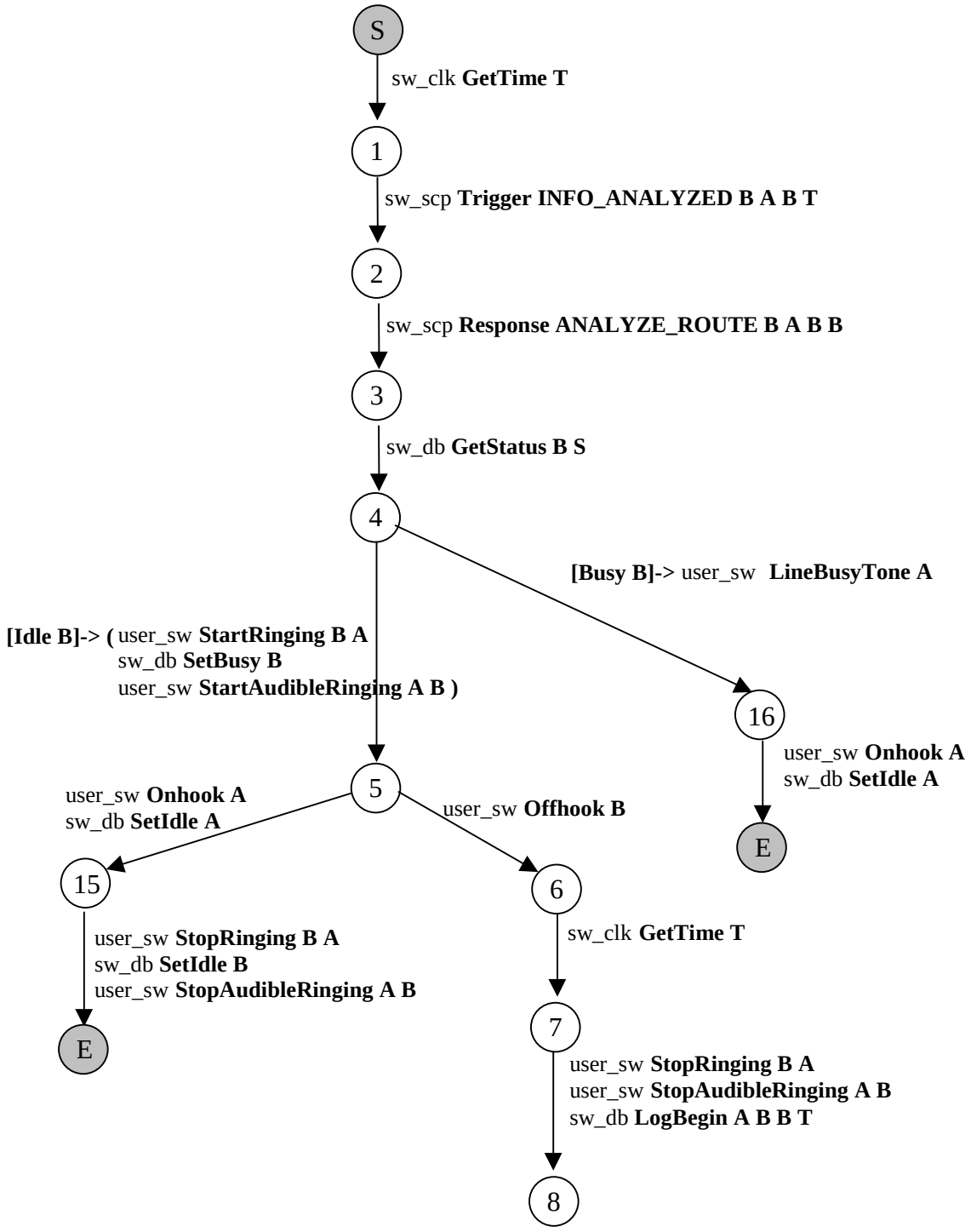


Fig 3.9 The LTS Tree of INFB (To be continued)

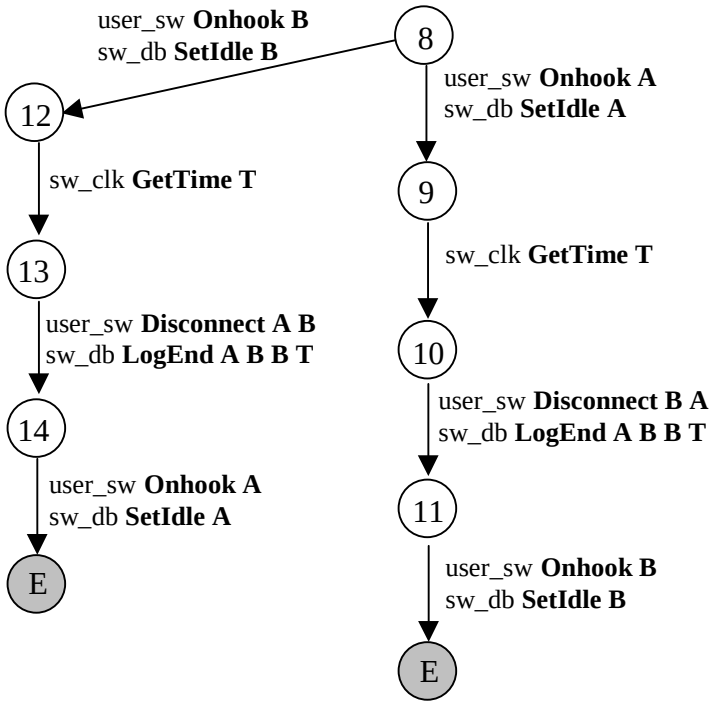


Fig 3.9 The FSM graph of INFB (Continued)

trigger type (INFO_ANALYZED), the subscriber's address (B), the caller's address (A), the callee's address (B) and the current time. After receiving INFB's trigger message, the SCP sends back an "ANALYZE_ROUTE" response to INFB, indicating that B should be the payer of the call from A. In this feature, this is the only part where the SCP is involved. Then, INFB becomes very similar to BCP. It checks B's status and if it is busy, a LineBusyTone is given to A, otherwise the call is connected and rings B. After B offhooks, the "LogBegin" signal logs the beginning time and charges the call to B, as specified in the SCP's response. When A (or B) finishes talking, A (or B) onhooks. The "LogEnd" signal logs the ending time of the call. At the same time a "Disconnect" signal is sent to B (or A) and B (or A) onhooks.

Like INTL, INFB has to consult the SCP and follows the SCP's instructions to charge the call.

- Signals:

Transitions of the INFB's LTS are signals occurring at four interfaces.

1) Signals at *user_sw* in INFB are the same as those in BCP

2) Signal at *sw_clk* is the same as that in BCP.

3) Signal at *sw_db* is the same as that in BCP.

4) Signals at *sw_scp*. INFB has one "Trigger" signal, *INFO_ANALYZED* and one "Response" signal, *ANALYZE_ROUTE*.

   "INFO_ANALYZED" trigger has four parameters. The first parameter indicates the subscriber's address (B), the second one specifies the caller's address (A), the third one takes the callee's address (B) and the fourth one holds the current time (T).

   "ANALYZE_ROUTE" is the response of the SCP to the trigger "INFO_ANALYZED". It has four parameters. The first parameter indicates the subscriber's address (B), the second one describes the caller's address (A), the third one specifies the callee's address (B) and the fourth one designates the payer of the call (B).

- Possible Exits

INFB has four exits: 1) The caller A onhooks because the callee B is busy. 2) The caller A onhooks when B is rung. 3) The caller A onhooks first after talking to B. 4) B onhooks first

after talking to A. Only in the last two cases, the connections between A and B are successfully established.

3.6.3 CFBL

CFBL, a switch-based feature, allows a subscriber to redirect incoming calls when it is busy. The subscriber pays for the forwarded part of the call. For example, if B has CFBL and B is busy when A calls, the call is forwarded to C given that C is the forwarded address. After the connection is established, the call is separated into two parts and charged in the following way: A pays for the part from A to B and B pays for the forwarded part from B to C.

Fig 3.10 gives the LTS tree of CFBL.

When caller A dials callee B who subscribes to CFBL, CFBL is activated by FAP at PIC_3. The first transitions of CFBL are to check the status of both B and C. 1) If B is idle, the call process returns to BCP at PIC_5. In this case, the call to B will be proceeded normally. 2) If both B and C are busy, a LineBusyTone is given to A and A onhooks. 3) If B is busy and C is idle, the call is forwarded to C. After C offhooks, there are two “LogBegin” signals of which

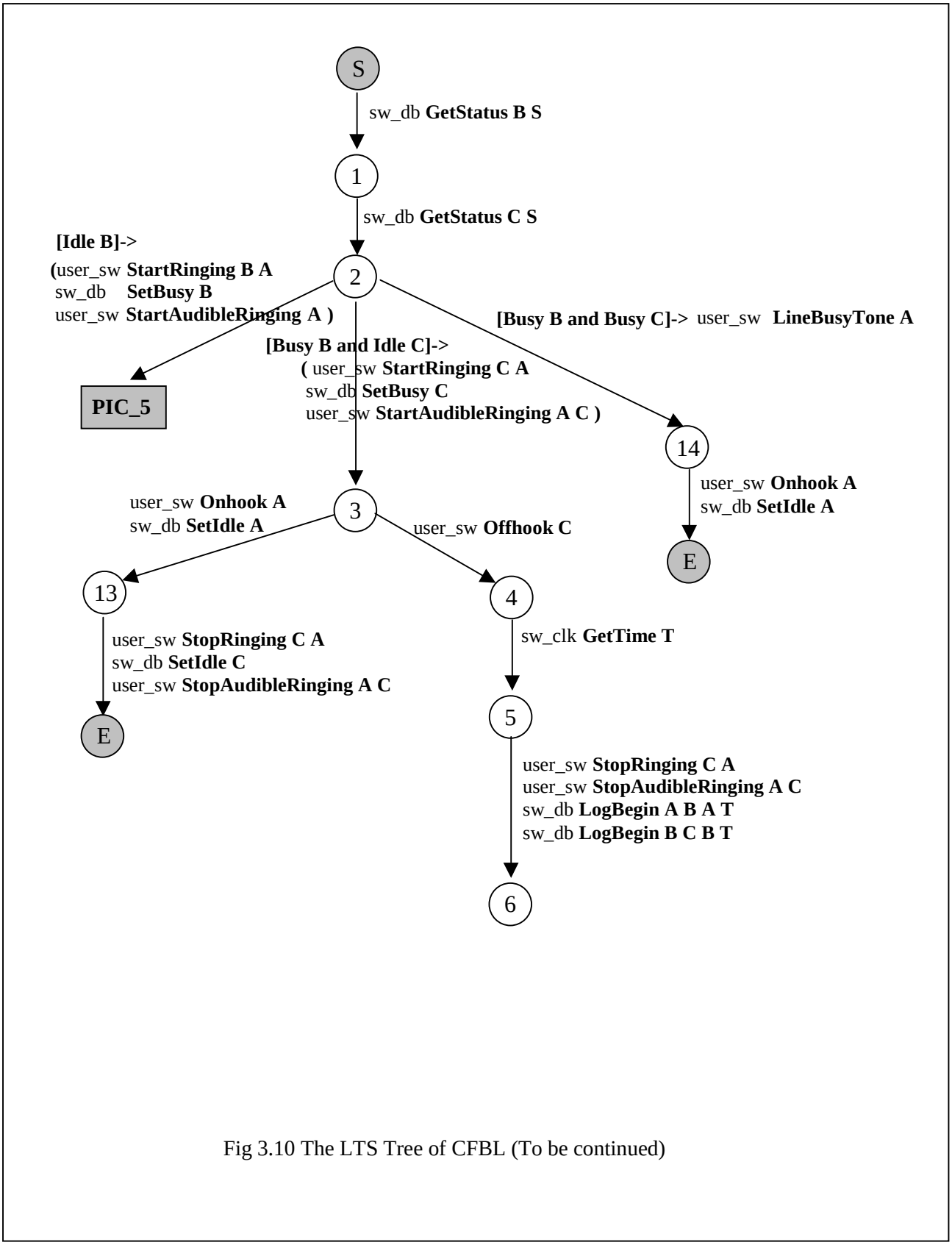


Fig 3.10 The LTS Tree of CFBL (To be continued)

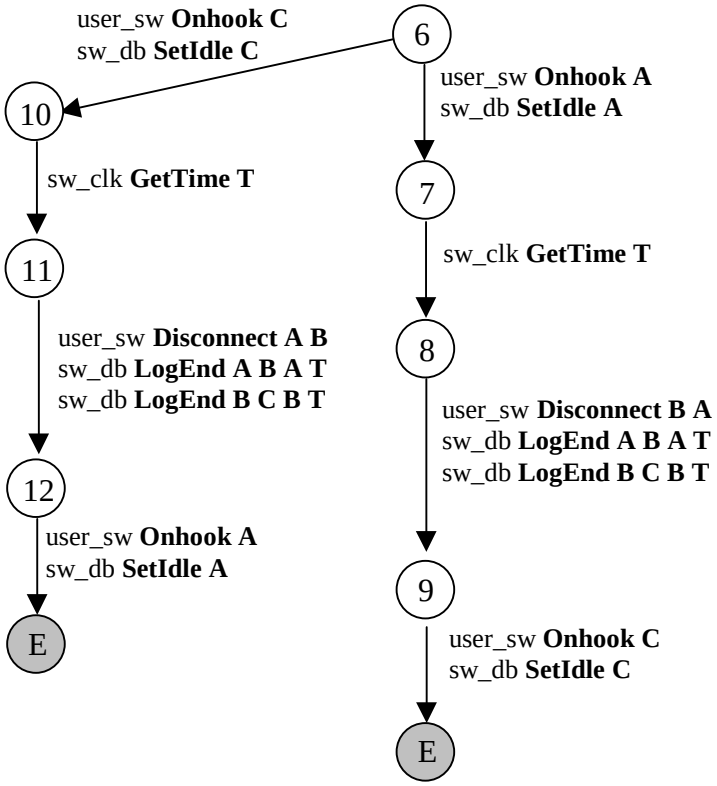


Fig 3.10 The LTS Tree of CFBL (Continued)

one charges the A-B part of the call to A and the other charges the forwarded B-C part to B. When A (or C) finishes talking, A (or C) onhooks. Two “LogEnd” signals log the ending time of the call to each part of the call. At the same time, a “Disconnect” signal is sent to C (or A) and C (or A) onhooks.

- Signals

Transitions of the CFBL’s FSM are for the same set of signals as for BCP.

- Possible Exits

CFBL has five exits: 1) Caller A onhooks because both B and C are busy. 2) B is not busy when A calls, thus the call process returns to PIC_5. 3) Caller A onhooks when C is rung, without waiting for C's answer. 4) Caller A onhooks first after talking to C. 5) C onhooks first after talking to A.

3.6.4 TWC

Three Way Calling is a switch-based feature that allows the connection of three parties in a single conversation.

Fig 3.11 illustrates the main part of the LTS tree of TWC. Since TWC is a very complex feature that contains 60 states, we hide the details of some unimportant branches, where no three-way connection is established, using blocks with dashed line. Details of these blocks can be found in [GBGT98].

Three Way Calling is activated by FAP from PIC_8 when subscriber A has connected to callee B. To connect the third party C, subscriber A temporarily suspends conversation with B, flashhooks and dials C. A's "Threeway" flag is set to be true. 1) If C is busy, A gets the LineBusyTone and flashhooks again. The call process returns to PIC_8. 2) If C is idle, A gets connected to C after C offhooks. Then, A flashhooks again to make B join the conversation between A and C and a three-way connection of A B C is established. Then, 1) If B (or C) finishes talking and onhooks, A gets the "disconnect" signal from B (or C), and A's "Threeway" flag is set to be false. The call process of A and B (or A and C) returns to PIC_8. 2) If A flashhooks, A's "Threeway" flag is set to be false, C gets the "disconnect" signal from A, C onhooks. 3) If A onhooks, A's "Threeway" flag is set to be false. Both B and C get the "disconnect" signal from A and onhook.

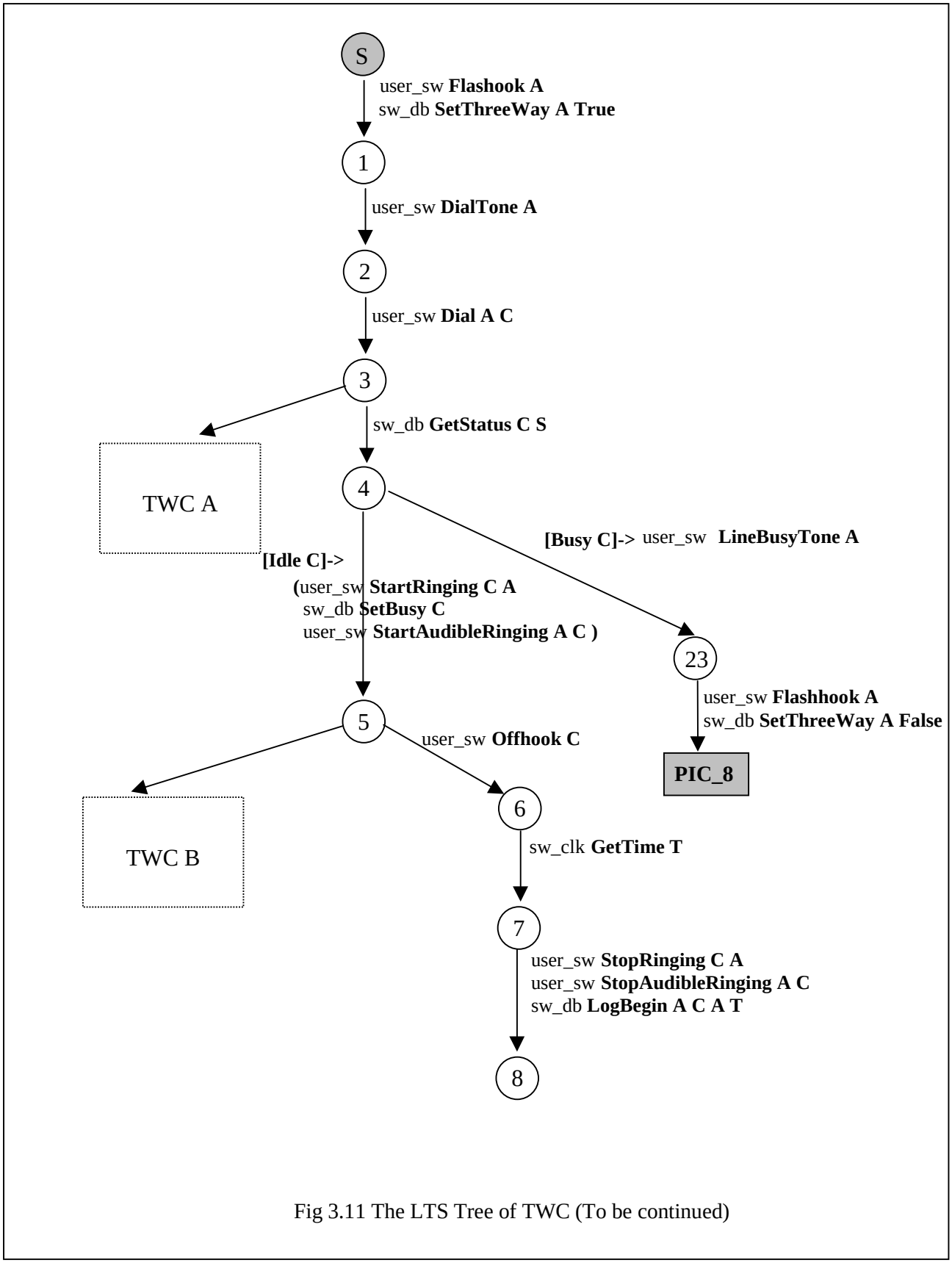


Fig 3.11 The LTS Tree of TWC (To be continued)

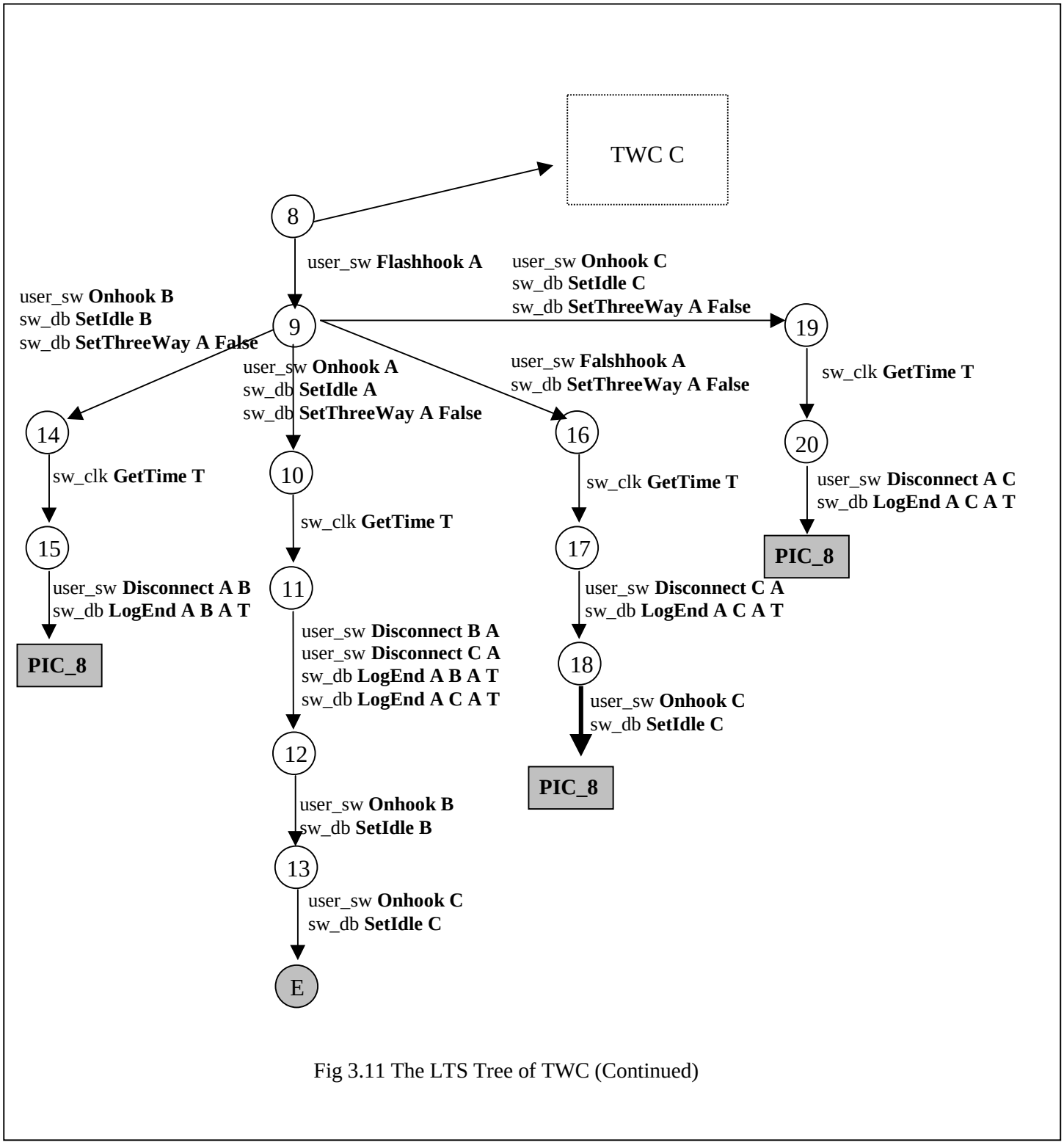


Fig 3.11 The LTS Tree of TWC (Continued)

- Signals

Transitions of the TWC's FSM are for the same set of signals as for the BCP except a new signal at *sw_db,* "SetThreeWay". "SetThreeWay" has two parameters. The first parameter indicates the user's address and the second parameter specifies the value to be set. For example, "SetThreeWay A True" means to set the "ThreeWay" flag of A to be "True".

- Possible Exits

TWC has 19 possible exits. Four of them are where a three way connection is successfully established: 1) Caller A onhooks to terminate the three-way-connection among A, B and C. 2) C onhooks and A gets the "disconnect" signal from C. The three-way connection among A, B and C becomes the two-way connection between A and B. 3) B onhooks and A gets the "disconnect" signal from B. The three-way connection among A, B and C becomes the two-way connection between A and C. 4) A flashhooks and C gets "disconnect" signal from A. The three-way connection among A, B and C becomes the two-way connection between A and B. Except the first one, all other exits may make TWC a loop since they bring it back to the same PIC where TWC is activated, hence the user can invoke TWC again.

3.7 Interface definition

In this section, we describe the complete set of signals defined in our system model. To describe signals, we use the following notation: the name of the signal is followed by the name and type of parameters, Signal-name X_1: ParameterType, X_2: ParameterType, …, X_n: ParameterType.

3.7.1 User/Switch *(user_sw)*

User to Switch (signals are sent from the users to the switch):

- Offhook X: Address (User with phone number X offhooks)
- Onhook X: Address (User with phone number X onhooks)
- Dial X: Address Y: Address (User at address X dials address Y)
- Flashhook X: Address (User at address X flashhooks. *Flash X* is equivalent to an *Onhook X* immediately followed by an *Offhook X*, unless a feature uses it otherwise. We assume that end-users have a *Flash* button)

Switch to User (Signals are sent from the switch to the user):

- DialTone X: Address (A dialTone is given to user X. DialTone means that the switch has approved the user to make an outgoing call. DialTone stops automatically when the user dials or hangs up)
- LineBusyTone X: Address (A lineBusyTone is given to user X. LineBusyTone is a negative signal for a call establishment attempt. LineBusyTone stops when the user onhooks or flashhooks)
- StartAudibleRinging X: Address Y: Address (The ringback tone is provided at address X while waiting for user Y to answer the call. AudibleRinging is a positive signal for a call establishment attempt.)
- StopAudibleRinging X: Address Y: Address (The ringback tone at address X from Y is disabled.)

- StartRinging X: Address Y: Address (Ringing starts at address X for a call originated at address Y.)
- StopRinging X: Address Y: Address (Ringing at address X from Y is disabled)
- Disconnect X: Address Y: Address (The switch informs user X that Y has disconnected a connection with X. User X should either hang up or flashhook after receiving the disconnect signal)
- Announce X: Address M: Message (An announcement M is played at address X)
- Start CallWaitingTone X:Address Y:Address (A special signal given to user X indicating that Y is trying to reach him/her)
- Display X:Address M:Message (It uses a display screen on telephone at address X to display the message M concerning the call)

3.7.2 Switch/SCP

The Bellcore AIN document GR-1298-CORE has been a reference for this interface, but the contest committee decided to use a simplified version of the message parameters. Messages sent from the switch to the SCP are of two kinds, “Trigger” and “Resource”. Messages sending from the SCP back to the switch are of one kind, “Response”.

- Trigger

  A general format of “Trigger” is:

  Trigger *Trigger_type subscriber’s address [parameter1, …]*

When SCP receives the trigger message, the corresponding routine for that trigger type is invoked. Besides the subscriber address, parameters may include information such as the calling party address, the called party address, and the time, etc.

- Resource

A general format of Resource is:

Resource *Subscriber's address, Parameter1,[Parameter2…]*

Resource responds to the SEND_TO_RESOURCE message from SCP, which is caused by a trigger. Besides subscriber's address, parameters in a resource message may include data collected from users, e.g. a PIN number

SCP to Switch (messages are sent from the SCP to the switch)

- Response

Response *ResponseType Subscriber address [parameter1, parameter2…]*

Different ResponseTypes indicate different instructions given from SCP to process the call listed as follows:

Response *ANALYZE_ROUTE S: Address A: Address B: Address C: Address* means to route a call from A to B and charge the call to C. S is the subscriber address.

Response *CONTINUE S: Address A: Address B: Address* means to continue processing the call from A to B using BCP. S is the subscriber address.

Response *SEND_TO _RESOURCE S: Address A: Address M: Message* means to play the message M at address A and collect the input data (if any)

Response *RES_DISCONNECTS: Address A: Address* means to terminate the processing of calls from A .

3.7.3 Switch to DBAPI

Billing signals:

- LogBegin X: Address Y: Address P: Address T: Time (DBAPI starts to charge P for a call from X to Y by opening a new billing record and logging the beginning time. T is the time when the called party offhooks.)
- LogEnd X: Address Y: Address P: Address T: Time (DBAPI stops charging the call from X to Y by logging the ending time T and closing the record.)

User status inquiry and setting signal:

- GetSatus X: Address S:Status (The switch queries the status information of user X.)
- GetSubscribingFeatures X:Address S:SubscribedFeatureSet (The switch queries the subscribing information of user X.)
- GetTeenTime X:Address T1:Time T2:Time (The switch queries the TeenTime period defined by user X)
- SetIdle X: Address (The status of X is set to be “idle”)
- SetBusy X: Address (The status of X is set to be “busy”)
- SetThreeWay X: Address B: Bool (The *“ThreeWay”* of X is set with the Boolean value B)

3.7.4 Switch to Clock

- GetTime T: Time (The switch queries the current time from the clock)

3.7.5 SCP to DBAPI

- Get TeenPIN X: Address P: PIN (The SCP queries the TeenPIN number of user X, which is stored in the user status database)

Chapter 4 LOTOS Specification of the System Model

In this chapter, we describe the LOTOS formal specification of our system model and of some features (BCP, INTL, INFB, CFBL and TWC) as defined in the functional plane of the system model. Our main objective in specifying the system model and features in LOTOS is to provide a specification that can be used as a test-bed for specifying, validating and detecting FIs. In the LOTOS specification, only the external behavior of the system is captured, that is, describing *what* the system does for the user, not *how* it does it (black-box specification).

Before introducing the details of the LOTOS specification of our model, we give an overview of the LOTOS specification language and of its main operators by describing some examples in the context of the telephony networks.

4.1 An Overview of LOTOS

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 [ISO8807]. Nowadays, LOTOS applications have been extended to cover some other domains such as hardware [FaLS97] and telephony [FaLS91], [StLo93].

A LOTOS specification consists of two parts, *data* part and *control* part. The control part defines the external observable behavior of the system that is described. It is based on Milner's Calculus of Communicating Systems (CCS) [Miln89] and Hoare's (CSP) [Hoar85]. The data part defines all the data types and value expressions needed to specify the behavior of the system. It is

based on the formal theory of algebraic abstract data types ACT-ONE [EhMa85]. A number of excellent LOTOS tutorials can be found in the literature [BoBr87]. Therefore, we limit ourselves to a brief overview of the language and its use in the context of our research.

All key words of LOTOS used in this thesis are highlighted in bold.

4.1.1 LOTOS Abstract Data Types

LOTOS adopts ACT-ONE, an algebraic abstract data type language, to define data types. ACT-ONE defines abstractly data operations without reference to implementation details.

A data *type* definition in LOTOS consists of a definition of a *signature* and possibly of a list of *eqns* (equations). A signature of a type is a definition of its *sorts* and *opns* (operations). Sorts defines the domain name of the data. Opns defines the formats of operations on the data. Eqns provide a means to define the semantics of operations.

LOTOS Data Types can be built hierarchically by using *is*. That is, one data type can be a collection of other data types. This constitutes an inheritance mechanism of a simple kind.

Consider the following type definition of the bill item in the billing database:

type TypeBillItem (*define the type name*)
is TypeAddress,TypeTime (*list other sorts used to construct this data type*)
(* Signature *)
sorts BillItem (*define the sort name*)
opns (*specify the format of operations*)
    Item(* Constructor *):Address(* Charged *), Address(* Caller *), Address(* Callee *), Time(* LogBegin *), Time(* LogEnd *) -> BillItem
    setLogEndTime:Time,BillItem -> BillItem
    getCaller,getCallee : BillItem -> Address
(* List of equations *)

```
eqns
forall a1,a2,a3,a4,a5,a6:Address, t1,t2,t3:Time
ofsort BillItem (*specify the return type of the operations list below*)
        setLogEndTime(t3,Item(a5,a1,a2,t1,t2))= Item(a5,a1,a2,t1,t3);
ofsort Address (* these are query functions *)
        getCaller(item(a1,a2,a3,t1,t2))=a2;
        getCallee(item(a1,a2,a3,t1,t2))=a3;
endtype (* TypeBillItem *)
```

Type *TypeBillItem* defines the billing items stored in the billing database. The format of "BillItem" is: Item(Payer, Caller, Callee, StartTime, EndTime). The "payer", "caller" and "callee" are of type *Address*. "StartTime" and "EndTime" are of type *Time*. "BillItem" has four operations: 1)"Item" is the constructor operation building a new "BillItem"; 2)"setLogEndItem" is a setting operation to set the "EndTime" of the "BillItem"; 3) "getCaller" is a query operation that returns the "Caller" address of the "BillItem"; 4)"getCallee" is a query operation that returns the "Callee" address of the "BillItem".

4.1.2 The Control Part

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

4.1.2.1 LOTOS Process

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 synchronization occurs, 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] (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 *parameter_list*. This set represents the set of parameters through which values can be passed to the process from outside. The parameterization of a process also enables its reusability.

4.1.2.2 LOTOS Action

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

An example of action is:

user_sw ***!*** *Offhook* ***?*** *caller:Address*

Offhook is of sort *UserSignal* that defines a set of all possible signals occuring at gate user_sw. When the action happens, it will obtain a value of sort *Address* from the environment for the *caller.*

Actions are considered to be atomic in the sense that they occur instantaneously, without consuming time. Generally speaking, actions in LOTOS are always executed by synchronization with the environment. However, there is a special type of actions in LOTOS, the *internal action,*

which is represented by *"i"*. It can be executed independently by the process and it is unobservable to the environment.

4.1.2.3 LOTOS Behavior Expressions

- *Inaction:* ***stop***

  It represents a deadlock, i.e. No more actions can be executed.

- *Successful Termination:* ***exit***

  It indicates a normal termination of the behavior, i.e. a process has successfully performed all its actions.

  The key word "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 [gate_list] (initial_value_list)*

  The instantiation of a LOTOS process is equivalent to the invocation of a procedure in a programming language (such as Pascal). Parameters of the process listed in "parameter_list" are initialized by the values given in "initial_value_list".

  Process Instantiation can occur either in the behavior expression of other processes or in the behavior expression of the process itself.

4.1.2.4 LOTOS Operators

- *Action Prefix Operator: a* ***;*** *B*

The *action prefix* operator, represented as a semi-colon ";", expresses sequential composition of action *a* and behavior expression *B*. It is used to sequentially order actions. For example, *user_sw !Dialtone !A; user_sw !Dial !A !B* denotes that caller A must get the dialtone before dialing the callee's number B.

- *Choice Operator:* B_1 **[]** B_2

The choice operator "[]" is used to express a choice between two alternatives, B_1 and B_2. Consider the following scenario as an example: after dialing the callee's number, the caller may 1) either get the linebusytone from the switch (if the callee is busy) 2) or get the audibleringing indicating that the call is connected and the callee is ringing. 3) or change his/her mind of making the call and hang up. This is expressed by the behavior expression listed below.

```
user_sw   ! LineBusyTone   ! A
[]
user_sw   ! StartAudibleRinging   ! A   ! B
[]
user_sw   ! Onhook   ! A
```

- *Enabling Operator:* B_1 >> B_2

The enabling operator ">>" has a similar function as the action prefix operator. The difference between them is that the action prefix operator ";" expresses the sequential composition of an action and a behavior expression; the enabling operator ">>" expresses the sequential composition of two behavior expressions. B_2 is executed if and only if B_1 is successfully terminated (exit).

- *Disabling Operator:* 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 onhooks. This could be expressed by the behavior expression as follows.

```
    ( user_sw   ! DailTone   ! A;
      user_sw   ! Dial   ! A   ! B;
  …
) [> user_sw   ! Onhook  ! A;…
```

- *Interleaving Operator: B_1 ||| B_2*

We say that B_1 and B_2 interleave if they can perform their actions independently of each other. The interleaving operator "|||" expresses the concept of parallelism between behaviors where no synchronization is required. For example, three users A, B and C in the network behave independently of each other. If we use process "User" to describe one user's behavior, the relationship between user A, B and C can be represented as follows.

```
USER [user_sw] (A)
 |||
USER [user_sw] (B)
|||
USER [user_sw] (C)
```

- *Parallel Composition Operator: B_1 |[g_1, …, g_n]| B_2*

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

Synchronization of processes on a gate g_i, $i \in \{1, \ldots,n\}$ occurs, if each process provides an action with a gate name g_i, the lists of events offered by the actions match, and the predicates (if any) are satisfied. The lists of events of two actions "match" if the following conditions are satisfied:

1) The numbers of events of the two actions match.

2) If an event in one action offers (!) a value, then the "matching" event in another action, should either offer (!) the same value or accept (?) a value of the same sort.

Consider the following example where two processes USER and BCP synchronize on the gate "user_sw".

```
( USER [user_sw](A)
   |||
  USER [user_sw] (B)
   |||
   USER [user_sw] (C)
where
process USER [user_sw] ( X: Address ): noexit :=
( user_sw !Offhook !X;
  user_sw !DialTone !X;
  …
  )
 endproc (* USER*)
)
|[user_sw]|
( hide sw_clk, sw_db in SWITCH [user_sw, sw_clk, sw_scp, sw_db]
 where
process SWITCH [user_sw, sw_clk, sw_scp, sw_db]: noexit :=
 BCP[user_sw, sw_clk, sw_scp, sw_db]
endproc (* SWITCH *)
process BCP [user_sw, sw_clk, sw_scp, sw_db]: noexit :=
( user_sw !Offhook ?Caller:Address;
  user_sw !DialTone !Caller;
  …
```

```
        )
    endproc (* BCP *)
)
```

Process USER stands for a user of the telephony network. It takes a parameter that holds the user's address, i.e. the process that simulates user A is *USER[user_sw](A).* The three users above are independent of each other. However, they all have to synchronize with the switch at gate "user_sw". Process SWITCH consists of only one process called BCP. Process USER actually synchronizes with process BCP at gate "user_sw". The first action of BCP is *user_sw !Offhook ?Caller:Address,* so it synchronizes with the first action of USER, *user_sw !Offhook !X*. In other words, the following two actions synchronize at the very beginning:

user_sw !Offhook ?Caller:Address offered by USER

user_sw !Offhook !X offered by BCP

As a result of synchronization, Caller acquires the value of X, which contains the address of the calling user.

- *Full Synchronization Operator: B_1 || B_2*

The full synchronization of B1 and B2 is a parallel composition in which B1 and B2 must synchronize on all their gates.

- *Hiding Operator:* ***hide*** $g_1, \ldots, g_n$ ***in*** B

The hiding operator "hide in" is used to hide actions synchronizing on gates (g_1,…g_n) within the process. These actions become internal actions (***i***) to the environment. As mentioned above, these internal actions cannot synchronize with the environment. In the previous example, gates *sw_db, sw_clk* are hidden within process SWITCH from the environment.

- *Guarded Behavior:* ***[P]*** $\rightarrow B$

The behavior expression B can be executed if and only if the predicate P is true; otherwise it equals to *stop*. For example, the callee can be rung only if it is not busy. Otherwise, a linebusytone should give back to the caller. The following behavior expression represents such scenario.

sw_db ! GetStatus ! Callee ? S: Status;
([busy(S)] $\rightarrow$ *user_sw !LineBusyTone !Caller*
[]
[not busy(S)] $\rightarrow$ *user_sw !StartRinging !Callee !Caller;*
…
)

4.1.3 Expansion

A basic concept in process algebraic languages is *expansion*. Any LOTOS behavior expression can be rewritten as an equivalent expression containing only choice, action prefix, and **stop** (although this expression could be infinite) [Miln89]. An expanded LOTOS specification represents directly the labeled transition system (LTS) of the system in consideration (LTS is a Finite-State Machine whose transitions are labeled with actions, more details can be found in §3.2). Each alternative path in an expanded specification, or each branch in an LTS, represents explicitly a possible sequence of actions in the system. Sequences of visible actions are called *traces*. Internal actions (see § 4.1.2.2) such as *i* or hidden actions (see § 4.1.2.4) usually are not included in traces, although sometime they are shown for completeness.

4.1.4 LOTOS Supported Tool: CADP

CADP (CAESAR/ALDEBARAN Development Package) is a toolbox for protocol engineering. CADP is jointly developed by the VASY action at INRIA Rhone-Alpes / DYADE and

the Verimag laboratory. It is dedicated to the efficient compilation, simulation, formal verification, and testing of descriptions written in the ISO language LOTOS [Fern96]. The CADP toolbox contains 1) two compilers (CAESAR and CAESAR.ADT) which translate LOTOS descriptions into C code which can be used for simulation, verification and testing purposes and 2) a set of applications (OPEN/CAESAR) which provides user extended functionalities such as interactive simulation, trace-searching tool, model checking, etc.

- CAESAR

CAESAR is a compiler that translates the control part of a LOTOS specification into either a C program (to be executed or simulated) or into an LTS (to be verified using bisimulation tools and/or temporal logic evaluators).

The CAESAR translation algorithms proceed in several steps. First the LOTOS description is translated into a simplified process algebra called SUBLOTOS. Then an intermediate Petri Net model is generated, which provides a compact, structured and user-readable representation of both the control and data flow. Eventually the LTS is produced by performing reachability analysis on the Petri net.

CAESAR accepts full LOTOS with the following restriction as regards the control part: process recursion is not allowed on the left and right hand sides of |[...]|, nor on the left hand side of >> and [>. Despite these restrictions, the subset of LOTOS handled by CAESAR is large and usually sufficient for real-life needs. The current version of CAESAR allows the generation of large LTSs (some million states) within a reasonable lapse of time.

The most recent version of CAESAR provides functionality called EXEC/CAESAR for C code generation. This C code interfaces with the real world, and can be embedded in applications. This allows rapid prototyping directly from the LOTOS specification.

- CAESAR.ADT

CAESAR.ADT is a compiler that translates the abstract data part of LOTOS specifications into libraries of C types and functions.

Each LOTOS sort is translated into an equivalent C type and each LOTOS operation is translated into an equivalent C function (or macro-definition). CAESAR.ADT also generates C functions for comparing and printing abstract data type values, as well as iterators for sorts having finite domain.

- OPEN/CAESAR

OPEN/CAESAR is an extensible, language-independent environment that allows user-defined programs for simulation, execution, verification (partial, on-the-fly, etc.), and test case generation to be developed in a simple and modular way. Various modules are involved in the OPEN/CAESAR framework. However, only two of them are used in our work:

- Caesar.Simulator, an interactive simulator.

  Caesar.Simulator provides an interactive environment where a user can execute the specification in a step-by-step way. The GUI has two parts: one displaying the traces of actions that have been executed and the other listing all available actions that could be executed next. The executed action traces are initially empty and the list of next available actions includes all possible actions to be executed at the beginning. After the user selects one action to execute,

that action is performed and added to the executed action traces and the next available action list is refreshed.

- Caesar.Exhibitor, a trace-searching tool.

  Caesar.Exhibitor provides a searching environment where users specify the patterns of traces using predicates and keywords. The tool executes the C program generated by Caesar and Caesar.ADT. Traces matching the given patterns are output. The user could choose whether the searching algorithm should be breadth-first or depth-first, and also can choose to find all occurrences or just the first one.

  Patterns could reflect complex semantics by using various predicates. However, the pattern we used in our work is very simple: only one predicate "~" and two keywords: <until> <deadlock>. ~ means "no". <until>"ActionA" refers to all traces leading to ActionA. <deadlock> refers to a state where no action can be further executed. See *§5.6.5 FI Hunter* for examples.

4.2 Specification Styles of Telephony Systems

Vissers, Scollo, van Sinderen and Brinksma [ViSV88] [VSVB91] identify four main styles for writing LOTOS specifications. They are *the monolithic style, the state-oriented style, the constraint-oriented style* and *the resource-oriented style*. Each style has its own uses in telephony system specifications and they can be mixed in one specification to meet different requirements.

- The monolithic style gives explicitly all possible sequences of actions allowed by a specification. The main operator is the choice operator "[]", and the specification is

shown as a tree of choices. Therefore, this style is useful for debugging the specification and generating test sequences.

- In the state-oriented style, explicit system states are identified, e.g. by using state variables. Using the state-oriented style may lead to increased readability of the specification in cases where the informal specification uses the state concept, as is quite common for telephone devices. It may also lead to LOTOS specifications that can be implemented directly.

- The constraint-oriented style focuses on event sequencing and logical constraints as seen from the external interaction points. It is useful for implementation-independent specifications [Turn87]

- In the resource-oriented style, the processes are chosen in such a way as to represent resources, which means implementation modules. This style is useful for implementation specification.

In our specification, we used a mixture of the resource-oriented style and state-oriented style. The observable behavior of the system is described as a composition of separate resources which functionalities are well defined, and these resources may be specified using any style. The resource-oriented style is used to preserve the architectural model of the system at the specification level and the state-oriented style is used to specify features (BCP, INTL, INFB, CFBL, TWC) that are defined as LTSs.

4.3 LOTOS Specification of the system model

In this section, we are going to describe the LOTOS specification of the system and its features by describing the Abstract Data Types (ADT), the architecture of the specification and the different processes of which it is composed.

4.3.1 Abstract Data Types

In our specification of the system model, ADTs are built in a hierarchical way, by using the inheritance mechanism described in §4.1.1.

The basic level are standard ADTs: *Boolean* and *Natural Numbers*, which are provided by the standard LOTOS ADT library.

- The value of a “Boolean” type variable is either “True” or “False”, so we call “True” and “False” *constructors* of Boolean. A couple of logic operators are also defined as equations in Boolean, such as “and”, “or”, “not” etc.
- We limit the domain of “Natural Number” to be [0 .. 20] because specifications with infinite ADTs cannot be fully expanded. Operators defined in “Natural Number” that are used by second level ADT are the comparing operators “=”, “<”, “>” and the increasing operator “*inc*”.

The second level ADTs are enumerations, whose elements can be mapped to corresponding “Natural Number”, so that they can inherit the comparing functionality of “Natural Number”. “AddressType”, “SignalType”, “FeatureType”, “MessageType”, “TriggerName” and “ResponseType” are second level ADTs. Fig 4.1 uses “FeatureType” as an example to show how second level ADTs are built. The keyword “is” in the first row indicates the inheritance relationship

between Natural Number and FeatureType. Then we define the mapping function "h" between Features and Natural Numbers so that the equivalence comparison between two features becomes comparing the corresponding natural numbers, as indicated in equation "f1 eq f2 = h(f1) eq h(f2)". "eq" in the LHS of the equation is the equivalence operator of the "FeatureType" and "eq" in the RHS of the equation is the equivalence operator of the natural number.

```
type FeatureType is NaturalNumber

 sorts FeatureType

 opns
        INTL, CFBL, INFB, TWC :->FeatureType
        h: FeatureType->Nat
        _eq_, _ne_: FeatureType, FeatureType->Bool

 eqns for all f1,f2:FeatureType
    ofsort Nat
        h(INTL)=1;
        h(CFBL)=2;
        h(INFB)=3;
        h(TWC)=4;

    ofsort Bool
        f1 eq f2 = h (f1) eq h (f2);
        f1 ne f2=  h (f1) ne h (f2 );

endtype (* FeatureType *)
```

Fig 4.1 An Example of Second Level ADTs

The third level ADT defines sets. "SubscribedFeatures", which is a set of features, is the third level ADT. Basic set manipulation operators such as "*e* eleof *S*" are defined in "SubscribedFeatures". "eleof" returns a Boolean value *True* if *e* is in set *S*. For example, the expression "INTL eleof {INTL, CFBL}" is true because INTL is an element of {INTL, CFBL}.

The fourth level ADT are record ADTs, which represent fixed-length records. “BillItem” and “Status” are fourth level ADT.

- “BillItem” is a billing record data type, which stores all the necessary information to charge a single call. The format of “BillItem” is (a3, a1, a2, t1, t2), where *a3* is the address of the paying party, *a1* is the address of the caller, *a2* is the address of the callee, *t1* is the time when charging starts and *t2* is the time when charging stops.

- The “Status” record stores the user’s status information (busy or idle) and subscribing information (a set of features subscribed by the user). The format of “Status” is (b, p, t1, t2, a, s), where *b* is a boolean variable indicating whether the subscriber is busy or not; *p, t1, t2* are variables of INTL, respectively “TeenPIN”, “TeenTime1” and “TeenTime2” (see *§3.2.3 INTL* for details); *a* is a variable of CFBL, “BLForward”, which stores the forwarded address to be used when the subscriber is busy; *s* is a set that stores subscribed features of the user.

The fifth level ADT is a multiple record ADT. “UserStatus” is a fifth level ADT and consists of an “address” ADT and a “Status” record ADT. The format of “UserStatus” is *(a, S)*, where *a* is the subscriber’s address and *S* is the corresponding status information.

The sixth level ADTs are database ADTs which simulate two databases: “TheUser” and “TheBill”. “TheUser” database stores status information of all users in the telephony network. Records in “TheUser” database are of sort “UserStatus”. Records in “TheBill” database are of sort “BillItem”. Basic database operations are defined on each database such as “add”, “inquire” and “set” data etc. Fig 4.2 depicts the ADT hierarchy pyramid.

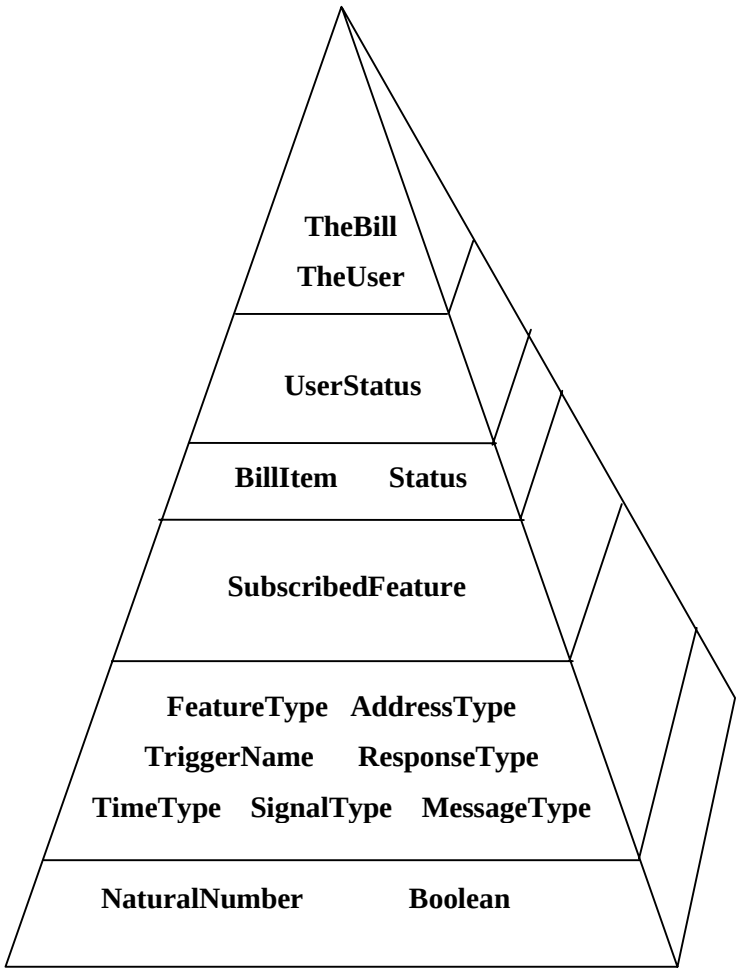


Fig 4.2 ADT Hierarchy Pyramid

4.3.2 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 simple descriptions where no further decomposition is possible.

The structure of the LOTOS specification corresponds to the system structure defined in Chapter 3. In the LOTOS specification, components and interfaces between them, which are described in Fig. 3.1, are simulated by corresponding processes and gates with the same names. Fig. 4.3 gives a graphical representation of the top level of our system model specification and the corresponding LOTOS top level specification is given in Fig 4.4.

The control part of the specification has only one process *SYSTEM*, which consists of five processes: *USERS, SWITCH, SCP, CLOCK* and *DBAPI*. First, the SWITCH synchronizes with the CLOCK at gate *sw_clk*. Second, the USERS synchronizes with the SWITCH and the CLOCK at gate *user_sw*. Then, the SCP synchronizes with the SWITCH, the CLOCK and the USERS at gate *sw_scp*. Last, the DBAPI synchronizes with the SWITCH, the CLOCK, the USERS and the SCP at gates *sw_db* and *scp_db*.

- The USER has only one gate *user_sw*, so it can only interact with the SWITCH.
- The CLOCK has one gate *sw_clk* through which it can only communicate with the SWITCH.
- The SWITCH has four gates, *user_sw*, *sw_scp*, *sw_clk* and *sw_db*, through which it can synchronize with the USERS, the SCP, the CLOCK and the DBAPI respectively.

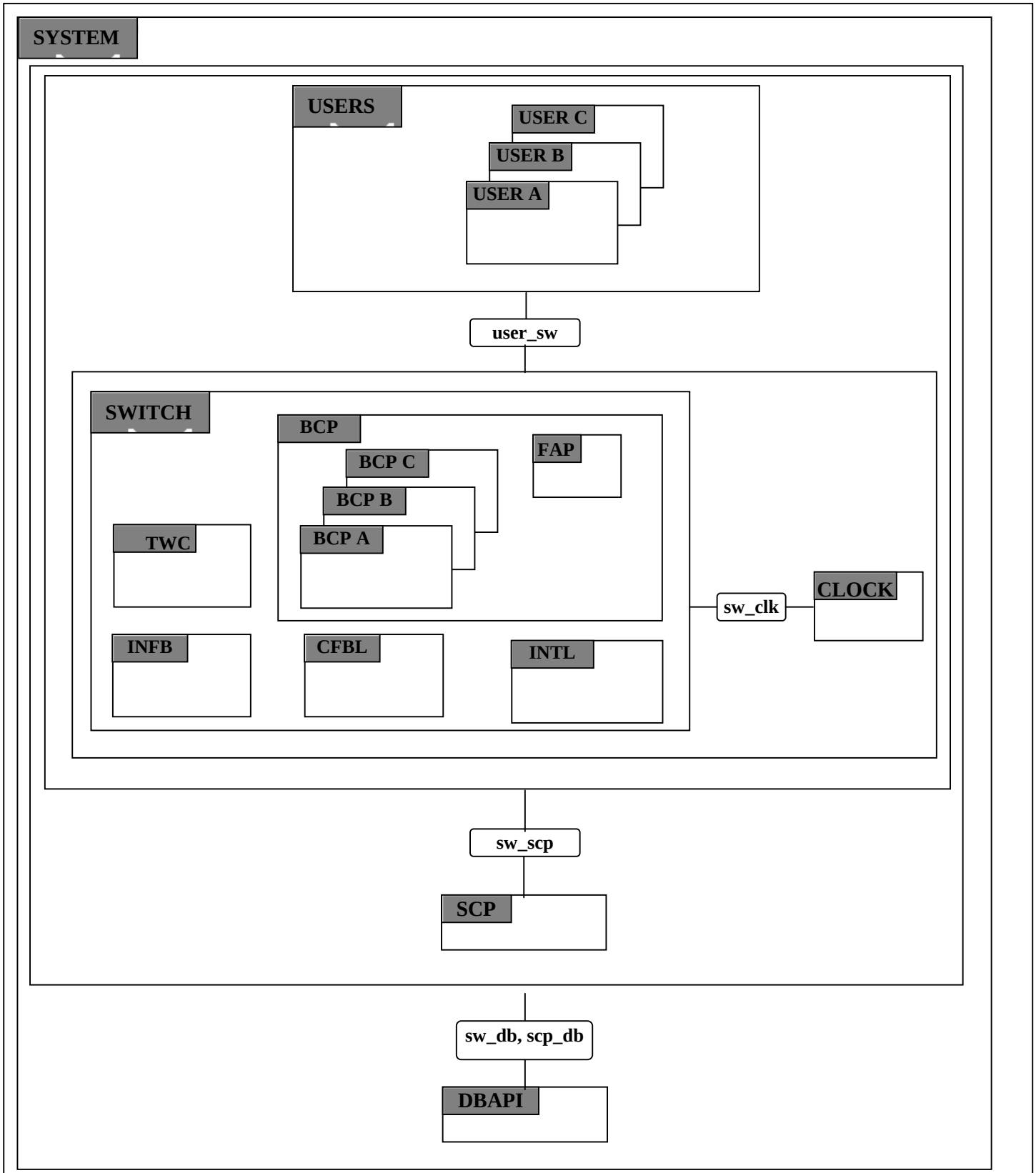


Fig 4.3 Graphical Representation of the Top Levels of the Specification

```
specification SystemModel [user_sw, sw_scp, sw_db, sw_clk, scp_db]: noexit
        …
        (* Data Part *)
        …

behaviour
        SYSTEM [user_sw, sw_scp, sw_db, sw_clk, scp_db]
        where
        process SYSTEM [user_sw, sw_scp, sw_db, sw_clk, scp_db]: noexit :=
                …
                (* Initialization Part *)
                …
                    (
                        ( USERS [user_sw]
                          |[user_sw]|
                                ( SWITCH [user_sw, sw_scp, sw_db, sw_clk]
                                  |[sw_clk]|
                                  CLOCK [sw_clk] (Initial Time)
                                )
                        )
                      |[sw_scp]|
                      SCP [sw_scp, scp_db]
                    )
                  |[sw_db, scp_db]|
                  DBAPI [sw_db, scp_db] (Initial Data)


        endproc (* SYSTEM*)

endspec (* SystemModel *)
```

Fig 4.4 Top-level LOTOS Specification

- The SCP can synchronize with the SWITCH and the DBAPI at gate *sw_scp* and *scp_db*, respectively.

- The DBAPI has two gates *sw_db* and *scp_db*, through which it can communicate with the SWITCH and the SCP.

Process CLOCK is instantiated with the initial time, *Time(1).* Process DBAPI is instantiated with the initial data of *TheUser* and *TheBill.* (For more details about the initial data of TheUser and TheBill, see *§5.2 Test Scenario Design*)

4.3.3 Process USER & USERS

USER is a very simple entity that accepts any valid signal and does nothing with them. As shown in Fig 4.5, process USERS consists of three users A B C. Since each user is independent of the others, we use the interleaving operator "|||" to compose them. Each user, A, B and C, is instantiated from process *USER* with address A, B and C respectively. Process USER synchronizes with any signal that comes from the SWITCH through gate *user_sw.* After synchronization on one signal, a new instance of user with the same address will be generated to synchronize on successive signals. In order to catch as many FI sequences generated from the switch as possible, no constraint is put on the order of user sequential behaviors.

Fig 4.5 gives the LOTOS specification of process *USERS* and process *USER.*

For this process, as well of as for similar processes below, note that the specification could have been stuctured in order to make possible to have an arbitrary number of users, by using recursive instantiation. However, in practice this would have complicated the simulation process.

```
process USERS [user_sw]: noexit :=
     USER [USER_sw] (A)
     |||
     USER [USER_sw] (B)
     |||
     USER[USER_sw] (C)

endproc (* USERs *)


process USER [user_sw] (Ad: Address) : noexit :=
  ( user_sw   ? e:Signal   ! Ad;
    USER [User_sw] (Ad)
   )
    []
   ( user_sw   ? e:Signal   ! Ad   ? Dest:Address;
     USER [User_sw] (Ad)
   )
    []
  ( user_sw   ! StartRinging   ! Ad   ? Orig:Address;
     USER [User_sw] (Ad)
  )
    []
  ( user_sw   ? e:Signal   ! Ad   ? p:Nat;
    USER [User_sw] (Ad)
   )
    []
  ( user_sw   ? e:Signal   ! Ad   ? M:MessageType;
    USER [User_sw] (Ad)
  )

endproc (* USER *)
```

Fig 4.5 LOTOS Specification of Process USER and USERS

4.3.4 Process CLOCK

Process CLOCK takes one parameter, *T*, which holds the current reading of the CLOCK. The Initial time is Time(0). When the SWITCH reads the time via "GetTime" signal, the CLOCK sends the current time, T, to the SWITCH and instantiates a new CLOCK with the reading increased by one. The specification of the CLOCK is given in Fig 4.6. Note that this process does not attempt to simulate real time, however it is sufficient for our purpose.

```
process CLOCK [sw_clk] (T: Time): noexit :=

    sw_clk !GetTime !T;
    CLOCK [sw_clk] ( inc (T) )

endproc (* CLOCK *)
```

Fig 4.6 LOTOS Specification of Process CLOCK

4.3.5 Process SWITCH

The switch controls the whole call process. Three Basic Call Processes (BCP) instantiated with user's addresses control the call process originated from A, B and C respectively. Due to the mutual independence of the users, three BCPs are also independent of each other and composed using the interleaving operator "|||". The specification of process *SWITCH* is shown in Fig 4.7.

```
process SWITCH [user_sw, sw_scp, sw_db, sw_clk]: noexit :=

        BCP [user_sw, sw_scp, sw_db, sw_clk] (A)
        |||
        BCP [user_sw, sw_scp, sw_db, sw_clk] (B)
        |||
        BCP [user_sw, sw_scp, sw_db, sw_clk] (C)

        FAP [user_sw, sw_scp, sw_db, sw_clk ] (F: Feature, Ad_A, Ad_B: Address)

        (* Integrated features *)
        INTL [user_sw, sw_scp, sw_db, sw_clk ] (Ad_A: Address)
        …
        CFBL [user_sw, sw_scp, sw_db, sw_clk ] (Ad_A: Address, Ad_B:Address)
        …
        INFB [user_sw, sw_scp, sw_db, sw_clk ] (Ad_A: Address, Ad_B:Address)
        …
        TWC [user_sw, sw_scp, sw_db, sw_clk ] (Ad_A: Address, Ad_B:Address, Ad_C:Address)
        …

endproc (* SWITCH *)
```

Fig 4.7 LOTOS Specification of Process SWITCH

4.3.5.1 BCP

The BCP process controls a general call process (see §3.3). It is like a backbone. Other new features are integrated into BCP and get activated from BCP.

The BCP is specified in the state-oriented style. The mapping rules from the LTS (see *Fig 4.2 LTS Tree of BCP*) to LOTOS processes (see *Fig 4.8 LOTOS PIC Processes*) are as follows:

- States

  Eighteen numbered states of the LTS are mapped to 16 LOTOS processes with the same name. The starting state *S* is mapped into the process *BCP*. The Ending state *E* is mapped into a *stop* action in the last process.

```
process BCP [user_sw, sw_scp, sw_db, sw_clk] (Ad_A: Address): noexit :=
        user_sw   ! OffHook    ? Ad_A: Address ;
        sw_db   ! SetBusy    !Ad_A ;
        FAP [user_sw, sw_scp, sw_db, sw_clk] (INTL, Ad_A)
endproc (* BCP *)

process PIC_1 [user_sw, sw_scp, sw_db, sw_clk] (Ad_A: Address): noexit :=
        user_sw   ! DialTone    ! Ad_A: Address ;
        PIC_2 [user_sw, sw_scp, sw_db, sw_clk] ( Ad_A)
endproc (* PIC_1 *)

process PIC_2 [user_sw, sw_scp, sw_db, sw_clk] (Ad_A: Address): noexit :=
        user_sw   ! Dial   ! Ad_A: Address   ? Ad_B: Address ;
        PIC_3 [user_sw, sw_scp, sw_db, sw_clk] ( Ad_A, Ad_B)
endproc (* PIC_2 *)

process PIC_3 [user_sw, sw_scp, sw_db, sw_clk] (Ad_A, Ad_B: Address): noexit :=
        user_sw   ! Onhook   ! Ad_A: Address ;
        sw_db   ! SetIdle   ! Ad_A;
        stop
        []
        (  FAP [user_sw, sw_scp, sw_db, sw_clk] ( CFBL, Ad_A, Ad_B)
             |||
           FAP [user_sw, sw_scp, sw_db, sw_clk] ( INFB, Ad_A, Ad_B)
             |||
           … )
        []
        sw_db  ! Get Status  !B ?S:Status;
        PIC_4 [user_sw, sw_scp, sw_db, sw_clk] (Ad_A, Ad_B, S)
endproc (* PIC_3 *)

process PIC_4 [user_sw, sw_scp, sw_db, sw_clk] (Ad_A, Ad_B: Address, S:Status): noexit :=
        ([Busy(S)]->
            PIC_16 [user_sw, sw_scp, sw_db, sw_clk] (Ad_A))
        []
        ([Idle(S)]->
            PIC_5 [user_sw, sw_scp, sw_db, sw_clk] (Ad_A, Ad_B))
endproc (* PIC_4 *)

    …
```

Fig 4.8 LOTOS PIC Processes (partial)

- Transitions

Transitions are mapped into actions in corresponding state process. For example, In LTS, after executing the transition “DialTone A”, the system moves from state PIC_1 to state PIC_2. Thus, in *process* PIC_1, after the action *user_sw !DialTone !Ad_A, process* PIC_2 is instantiated and all associated parameters, i.e. the caller and callee’s address, are passed to it.

- POI

Unlike normal PICs which instantiate another PIC process at the end, at each POIs, (PICs where the features is activated), POIs call the FAP process, which detects subscribed features and activates them if there are any. If two features have the same POI, such as PIC_3, then two FAP processes are instantiated for the two features respectively. Since features are independent of each other, these two FAP processes are interleaved.

4.3.5.2 Feature Activation Process (FAP)

At the POI of each feature, PIC_1 (INTL), PIC_3 (CFBL, INFB) and PIC_8 (TWC), BCP calls the *Feature Activation Process* (FAP) to activate each feature.

FAP takes three parameters:

- *F*, indicating which feature is going to be activated;

- *Ad_A*, holds the caller’s address;

- *Ad_B* holds the callee’s address (If any)

```
process FAP [ user_sw, sw_scp, sw_db, sw_clk ] (F: Feature, Ad_A, Ad_B: Address): noexit :=
    ( [F eq INTL]->
                sw_db !GetStatus !Ad_A  ?S:Status;
                ( [eleof (INTL, S)]->
                        INTL [ user_sw, sw_scp, sw_db, sw_clk] ( Ad_A)
                  []
                  [not (eleof (INTL, S))]->
                        PIC_1[ user_sw, sw_scp, sw_db, sw_clk] ( Ad_A)) )
  []
  ( [F eq CFBL]->
                sw_db !GetStatus !Ad_B  ?S:Status;
                ( [eleof (CFBL, S)]->
                        CFBL [ user_sw, sw_scp, sw_db, sw_clk] ( Ad_A, Ad_B)
                  []
                  [not (eleof (CFBL, S))]->
                        PIC_4[ user_sw, sw_scp, sw_db, sw_clk] ( Ad_A, Ad_B, S)) )
  []
  ( [F eq INFB]->
                sw_db !GetStatus !Ad_B  ?S:Status;
                ( [eleof (INFB, S)]->
                        INFB [ user_sw, sw_scp, sw_db, sw_clk] ( Ad_A, Ad_B)
                  []
                  [not (eleof (INFB, S))]->
                        PIC_4[ user_sw, sw_scp, sw_db, sw_clk] ( Ad_A, Ad_B, S)) )
  []
  ( [F eq TWC]->
                !GetStatus !Ad_A  ?SA:Status;
                !GetStatus !Ad_B  ?SB:Status;
                ( [eleof (TWC, SA)]->
                        TWC [ user_sw, sw_scp, sw_db, sw_clk] ( Ad_A, Ad_B, Ad_A)
                  []
                 [eleof (TWC, SB)]->
                        TWC [ user_sw, sw_scp, sw_db, sw_clk] ( Ad_A, Ad_B, Ad_B)
                  []
                  [not (eleof (TWC, SA) and eleof (TWC,SB))]->
                        PIC_8[ user_sw, sw_scp, sw_db, sw_clk] ( Ad_A, Ad_B)) )

endproc (* ActivateFeatures3 *)
```

Fig 4.9 Feature Activation Process

FAP inquires about the status information of the caller or callee's or both (depending on different features). Then the FAP checks if the feature to be activated has been subscribed or not. If it has, the corresponding feature process is called. Otherwise, the call process returns to BCP. Fig 4.9 gives the LOTOS specification of FAP.

FAP uses "*eleof*", an operation defined on ADT "*Status*", to check if a user subscribes to a specific feature, i.e. expression "*eleof (INTL, Status)*" is TRUE if *INTL* is an element of "*Status*". A feature is activated by generating an instance of the corresponding feature process with specific parameters, i.e. addresses of the caller and the callee.

4.3.5.3 Features

The LOTOS processes of the features are obtained from the LTSs of those features in the same way as BCP and PIC process mapped from the LTS of BCP. (Refer to *§3.2.3 INTL, §3.2.4 CFBL, §3.2.5 INFB, §3.2.6 TWC, § 4.3.5.2 BCP* for the details of the LTSs and the mapping rules)

4.3.6 Process SCP

In our system model, the SCP includes five IN feature specifications. Every IN feature has a unique trigger name, i.e. INFB's trigger name is INFO_ANAYZED, and INTL's trigger name is ORIGINATION_ATTEMPT, so that the SCP can know which trigger message was sent from which feature by checking their trigger names. Responses to different features are composed together using the choice operator "[]" in the SCP. A new instance of the SCP is generated when the processing of the feature finishes.

Fig 4.10 illustrates the LOTOS specification of the SCP.

```
process SCP [sw_scp, scp_db] : noexit :=
  …
 (* INTL Feature *)
  sw_scp  ! Trigger  ? Trig:TriggerName  ? Ad_S:Address  ? Ad_A:Address  ? T:Time;
  (  [Trig eq ORIGINATION_ATTEMPT]->
            scp_db   ! getStatusReq   ! Ad_S;
            scp_db   ! getStatusRes   ? S:Status;
            sw_scp   ! Response   ! SEND_TO_RESOURCE   ! Ad_S   ! AskForPIN;
            sw_scp   ! Resource   ? Ad_S:Address   ? P:Nat;
            (  [P eq GetTeenPIN(S)]->
                        sw_scp   ! Response  ! CONTINUE  ! Ad_S;
                        SCP [sw_scp, scp_db]
              []
              [P ne GetTeenPIN(S)]-> sw_scp  ! Response   ! SEND_TO_RESOURCE   ! Ad_S   ! InvalidPIN;
                        sw_scp  ! Resource  ? Ad_S:Address;
                        sw_scp  ! Response  ! RES_DISCONNECT   ! Ad_S;
                        SCP [sw_scp, scp_db]
             )
   )
  []
 (* INFB & INFR Feature *)
 sw_scp  !Trigger   ?Trig:TriggerName   ?Ad_S:Address   ?Ad_A:Address  ?Ad_B:Address  ?T:Time;

    [Trig eq INFO_ANALYZED]->
         sw_scp  ! Response  ! ANALYZED_ROUTE   ! Ad_S   !Ad_A   !Ad_B   !Ad_B;
         SCP [sw_scp, scp_db]

endproc (* SCP *)
```

Fig 4.10 LOTOS Process of SCP (partial)

4.3.7 Process DBAPI

Process DBAPI is an interface of the user status database (TheUser) and the billing database (TheBill), which are represented using ADTs.

Its main functions are: 1) processing the queries of “GetStatus”, “GetSubscribedFeatures” from the switch and the query of “GetTeenPIN” from the SCP and outputing the corresponding

replies using query operations of the ADTs and 2) following the setting instructions of “SetBusy”, “SetIdle”, “SetThreeWay”, “LogBegin” and “LogEnd” to set or construct records (“LogBegin”) using setting or constructor operations of the ADTs.

```
process DBAPI [sw_db,scp_db](TheStatus:UserStatusSet,TheBill:BillSet) : noexit :=
 ( ( sw_db ! GetStatus ? Ad_A:Address ! GetStatus(GetUserStatus(Ad_A,TheStatus));
      DBAPI [sw_db,scp_db] (TheStatus,TheBill) )
   []
  ( sw_db !SetIdle ? Ad_A:Address;
    DBAPI [sw_db,scp_db]
    (SetUserStatus(Ad_A,Idle(GetStatus(GetUserStatus(Ad_A,TheStatus))),TheStatus),TheBill) )
  []
   ( sw_db! SetBusy ? Ad_A:Address;
     DBAPI [sw_db,scp_db]
     (SetUserStatus(Ad_A,Busy(GetStatus(GetUserStatus(Ad_A,TheStatus))),TheStatus),TheBill) )
  []
  ( sw_db! SetThreeWay ? Ad_A:Address ?b:Bool;
     DBAPI [sw_db,scp_db]
    (SetUserStatus(Ad_A,SetThreeWay(b,GetStatus(GetUserStatus(Ad_A,TheStatus))),TheStatus),TheBill) )
 []
  ( sw_db! LogBegin ?Ad_A:Address ?Ad_B:Address ?Ad_C:Address ?T:Time;
    DBAPI [sw_db,scp_db]
   (TheStatus,LogLogbegin(Ad_C,Ad_A,Ad_B,T,TheBill)) )
 []
  ( sw_db ! LogEnd ?Ad_A:Address ?Ad_B:Address ?T:Time;
    DBAPI [sw_db,scp_db]
   (TheStatus,LogLogEnd(Ad_A,Ad_B,T,TheBill)) )
 []
  ( sw_db !GetSubscribedFeatures ? Ad_A:Address !GetFeatures(GetUserStatus(Ad_A,TheStatus));
     DBAPI [sw_db,scp_db]
     (TheStatus,TheBill) )
 []
  ( scp_db !GetTeenPIN ? Ad_A:Address !GetTeenPIN(GetUserStatus(Ad_A,TheStatus));
     DBAPI [sw_db,scp_db]
     (TheStatus,TheBill) ) )
 []
  ( sw_db !GetTeenTime ? Ad_A:Address !GetTeenTime(GetUserStatus(Ad_A,TheStatus));
     DBAPI [sw_db,scp_db]
     (TheStatus,TheBill) ) )
  []
  …

endproc (*DBAPI*)
```

Fig 4.11 LOTOS Process of DBAPI (partial)

Chapter 5. Feature Interaction Detection System

In Chapter 1, the FI problem is explained in a general way. In this chapter, we give FI a precise and formal definition and explain how a FI Detection System (FIDS) is developed according to this formal definition. Since it would be very long to cover all 12 features, we will use four representative features, INTL, CFBL, INFB, TWC as examples to show how feature properties are derived and how FIs are detected by FIDS. The full results of our FI analysis on all contest features are reported in [FHLS98].

5.1 Classification of FI

During the feature development process, a feature is defined at several different levels of abstraction, from a high level view to implementation code. Therefore, FIs can occur at all these levels. In [BDCG89], FIs occurring at the level of abstract specification are called *logical feature interactions*, those occurring when the feature specification is mapped onto a network architecture are called *network feature interactions* and those occurring when the feature software is mapped onto an execution environment are called *implementation feature interactions*.

Clearly, FI detection must be done as early as possible, otherwise FIs will propagate through the whole feature development process. Since we are dealing with formal specification of features, which abstracts from design and implementation details, the FIs that we detect here are logical feature interactions.

5.2 Formal definition of FI

Many definitions of Feature Interaction are either too inclusive or too exclusive. For example, Cameron et al. [CGDN93] understood feature interactions "*to be all interactions that interfere with the desired operation of the feature*". Here, the "*desired operation*" of a feature is an imprecise notion, which might have different meanings to subscribers, to designers, and to people who made the specifications.

P. Combes et. al [CoPi94] and W. Bouma [BoZu92] formalize the above "desired operation" to be properties of features, and address the FI problem as violations of these properties when a new feature is introduced into the network. However, they concentrate only on the violation of features' properties and miss the FI cases where the system properties are violated, while the features' properties hold.

We improve the definition of FI by adding system properties to the set of properties that must be checked after the introduction of a new feature. The definition then becomes the following one.

Let S be an executable specification of a basic telephony system (POTS), and let F_1, F_2, …, F_n, be specifications of n features.

We use $S \oplus F_1 \oplus F_2 \oplus \ldots \oplus F_i$ to denote the system obtained by integrating i features, $1 \leq i \leq n$, to the basic telephony system (POTS).

Let SP (System Property) be logical formulae expressing the properties of the basic telephony system (POTS), FP_1, FP_2, …, FP_n (Feature Property) be n formulas

expressing respectively the feature properties of F_1, F_2, …, F_n, and let N |= 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 :

$\forall$ i, $1 \leq i \leq n$, $S \oplus F_i \models SP \wedge FP_i$, but

$\neg (S \oplus F_1 \oplus F_2 \oplus \ldots \oplus F_i \models SP \wedge FP_1 \wedge FP_2 \wedge \ldots \wedge FP_n)$

Examples of SP and FP will be given in §5.4.2 and §5.4.3 respectively.

5.3 Two Phases of FI Detection Process

According to the FI definition of the previous section, our FI detection process is divided into two phases:

1) *Validation phase* to validate that every feature works well individually after having been integrated into BCP, that is, both the feature property and system properties hold. Thus, the first part of the FI definition, $S \oplus F_i \models SP \wedge FP_i$ is checked.

2) *Detection phase* to detect any undesirable effect caused when two or more features work together, that is, to detect if any feature property or system property is violated. Thus, the latter part of the FI definition, $\neg (S \oplus F_1 \oplus F_2 \oplus \ldots \oplus F_i \models SP \wedge FP_1 \wedge FP_2 \wedge \ldots \wedge FP_n)$, is checked.

5.3.1 Validation Phase

The validation phase has two stages:

- First, using a LOTOS tool called *CAESAR.Simulator* to validate the consistency between the LTS specification and the LOTOS executable specification in a step-by-step fashion. That is, it is checked that our LOTOS specification has all and only those traces specified in its LTS tree (described in Chapter 3.)

- Then, the FI Detection System (FIDS) is used to verify that both the system properties and the feature properties hold when there is only one feature activated during a call process. FIDS, given the name of a feature, activates the feature during a call process and checks the presence of the feature's properties. We will explain how FIDS works in *§ 5.5 FI Detection System*.

5.3.2 Detection Phase

In the detection phase, we use FIDS to detect FI pair-wise, that is, two features will be activated during a call process. Then, the feature properties, together with the system properties, will be checked by analyzing the billing data and monitoring conflicting signals.

Although FIDS is used in both the detection phase and the validation phase, the differences between them are listed below.

1) Only one feature is activated in the validation phase while two features are activated in the detection phase.

2) The goal of the validation phase is to find defects in the specification, and then to fix them. Therefore, the faults found in the validation phase are not FIs.

3) The goal of the detection phase is to find interactions between features when they are activated and to report them. So, any abnormality found in this phase is a symptom of FI.

5.4 Deriving the Properties of Features

How to derive the properties of features and how to represent them are the biggest challenges of FI detection since a feature's property is usually defined informally using natural language and people may have different understandings of a given feature. For example, the informal description of feature INFB is *"The IN Freephone Billing(INFB) feature allows the subscriber to pay for incoming calls."* When deriving the properties of a feature from such definitions, divergences could occur in understanding the exact scope of "incoming calls". Is a forwarded call an "incoming" call? If it is, should the subscriber of INFB pay for the whole call or only for the forwarded part of the call? We experienced the same interpretation problems during the property derivation process. Since some features such as INFB are so new that little research has been done on them, we could not compare our work with any reference concerning a "standard" explanation of them. Therefore, the derived feature's properties listed here are based on the best of our knowledge and on our practical experience with FI detection.

Beside the problem of interpretation, how to establish the necessity and the completeness of the derived feature property set is another big challenge. If the derived

feature property set is not a minimum set of all necessary properties, then much extra work may have to be done to validate those unnecessary properties, or even "false" FI might be detected. We call these "false" FIs because in such cases, only the unnecessary properties are violated while other necessary properties are all well preserved.

On the other hand, a derived property set should be complete. Otherwise, some FI may not be found due to the incompleteness of the set.

However, given the fact that feature properties normally are provided in a semi-formal notation, completeness and necessity cannot be checked formally and depend on judgement.

Furthermore, the completeness and the necessity of a property set are system-dependent. That is to say, we cannot derive a feature's property set without considering the system and the specific activation mechanism of the integrated features.

Thus, before discussing derived feature properties, let us briefly describe the feature integration and activation mechanism in our system model.

5.4.1 Feature Composition

In our system model, features are represented using LOTOS processes. All new features are integrated into the BCP, a basic call control feature, via FAPs at corresponding PICs. One feature's activation will not affect the activation of other features. Therefore, if two features are integrated into the same PIC, their FAPs are mutually independent of each other. We use the interleaving LOTOS operator "|||" to describe the mutual independence between FAPs.

Inappropriate feature composition may "solve" or "invent" some "FIs". For example, feature activation might result in some unintended priorities if the features are not properly composed. If the priorities are given correctly, we will miss the FI because it has already been solved. If the priorities are not given correctly, we may get FIs with misleading symptoms, e.g. one feature's activation "disables" another feature's activation. So, we use the interleaving operator "|||" to preserve the mutual independence of the feature activations.

5.4.2 System Properties

In our system model, the system properties are derived as follows:

- *Absence of deadlock.*

  That is, at any time, the telephony system has at least one event to occur next.

- *Valid billing records.*

  A billing record, (c, a, b, t1, t2) is *valid* if

  1) a, b, c are in the registered network address set. (caller *a*, callee *b* and payer *c* are all valid registered network addresses)

  2) $a \neq b$ (the caller *a* and the callee *b* should not be the same address)

  3) $t1,t2 \neq 0 \wedge t1 < t2$ (the call starting time *t1* and the call ending time *t2* are set and *t1* is earlier than *t2*)

- *Correctness of the billing database.*

The billing database is *correct* if all calls occuring in the system have one and only one corresponding billing record stored in the billing database.

- *Compatiblity of successive signals given to the user*.

In our system model, three types of audible signals are given to the user during the call establishment process:

- *AudibleRinging*

AudibleRinging is a positive signal to the caller because it means that the call is connected to the callee and the callee is being rung.

- *LineBusyTone*

- *Announcement of ScreenedMessage (INTL)*

- *Disconnect*

LineBusyTone, *announcement of ScreenedMessage* and *Disconnect* are negative signals to the caller because the call connection is blocked /terminated in such cases. A LineBusyTone is generated because the callee is busy and it has no CFBL feature or it has the CFBL feature but the forwarded address is also busy. The SceenedMessage is played to the caller when the caller attempts to originate a call during the TeenTime period but fails to input the correct TeenPIN, thus the call is blocked. A disconnect signal is given to the user when the other party hangs up, thus the call is terminated.

We say that two signals are "*compatible*" if they have the same meaning to the user. So, negative signals are compatible with each other because they have the same

meaning to the user. Negative signals are incompatible with positive signals because they have conflicting meanings to the user. Positive signals are compatible with each other if they correspond to the same destination that is being rung and incompatible if different destinations are being rung.

Table 5.1 depicts the compatible relations among LineBusyTone, AudibleRinging, Disconnect and the announcement of ScreenedMessage.

Table 5.1 Compatible Relations of Signals Given to User

	LineBusy Tone	**Screened Message**	**Disconnect**	**AudibleRining from A**	**AudibleRining From B**
LineBusy Tone	Compatible				
Screened Message	Compatible	Compatible			
Disconnect	Compatible	Compatible	Compatible		
Audible Ringing from A	Incompatible	Incompatible	Incompatible	Compatible	
Audible Ringing from B	Incompatible	Incompatible	Incompatible	Incompatible	Compatible

5.4.3 Feature Property

The telephony features that we discuss here are marketable services [FaLS97]. The subscribers who buy the services know nothing about the implementation details of either the system or the feature. To them, the telephony network is like a black box. They interact with it through the telephones (through gate “user_sw” in our model) and periodically pay the bill for the services (in our model, all billing records are created by the billing actions occurring at gate “sw_db”). Therefore, the feature properties can be

described from the user's point of view. That is, the feature properties can be mapped into specific restrictions on billing actions and/or user's behavior traces. For example, INFB can be mapped as follows: "for all billing records where the subscriber is the callee, the payer should be the subscriber too." It is important to note that all features that were considered in this study could be characterized by one property only.

The feature property validation process is effectively simplified by adopting such feature property representation. Instead of checking the entire trace of a call process to validate the feature properties, we only need to examine the billing records and user behavior traces to detect FI.

Before specifying the properties of features, let us define some basic concepts. For the description of billing records, refer to *§4.3.1 Abstract Data Types.*

- *Forwarded Call*

  If there exist two billing records, *(p1, a, b, t1, t2), (p2, c, d, t3, t4),* which have the same LogBegin time and LogEnd time, *t1=t3, t2=t4,* and the caller of one record is the callee of the other record, *b=c*, we say that there is a *forwarded call* from *a* to *d* through *b(c)*. *a* is the *originating party* of the forwarded call. *d* is the *terminating party* of the call.

- *Next Forwarded Address*

  In the above example, *d* is the *next forwarded address* after *b*.

- *Direct call*

We say the call is a *direct* call if 1) it is not a *forwarded call*, 2) the *originating party* of a direct call is the caller and 3) the *terminating party* of a direct call is the callee.

5.4.3.1 Derived Property of INTL

The informal requirement description of INTL is "*INTL restricts outgoing calls based on the time of day. This can be overridden on a per-call basis by anyone with the proper identity code.*"

The property of INTL derived from the above informal specification is:

If user X subscribes to INTL and defines that the TeenTime period is from T_1 to T_2 and the TeenPIN is P, then if X originates any call (direct or forwarded), during the TeenTime period, a valid TeenPIN P must have been input by X.

In FIDS, the property of INTL is validated in the following way:

1) Checking all billing records whose LogBegin time is within the TeenTime period of *X* to see whether *X* is the originating party or not. If *X* does originate a call during the TeenTime period, turn to step 2).

2) Checking the user's behavior traces to see if signal "user_sw !Dial !X !P" (P is equal to the TeenPIN) occurs before. If it does, the property of INTL holds. Otherwise, the property is violated.

5.4.3.2 Derived Property of INFB

The informal requirement description of INFB is "*INFB allows the subscriber to pay for all incoming calls.*"

The property of INFB derived from the above informal specification is:

If user X subscribes to INFB, then X pays for all incoming calls.

In FIDS, the property of INFB is validated as follows:

Checking all billing records where *X* is the callee to see whether the payer is also *X* or not. If it is, the property of INFB holds, otherwise, the property is violated.

5.4.3.3 Derived Property of CFBL

The informal requirement description of CFBL is "*with the CFBL feature, all calls to the subscribing line are redirected to a predetermined number when the line is busy. The subscriber pays any charges for the forwarded call from his station to the new destination.*"

The property of CFBL is derived as follows:

If user X subscribes to CFBL, then all incoming calls made to X when X is busy must be forwarded to a third party predefined by X.

In FIDS, the property of CFBL is validated in two steps:

1) Checking if X is initially set to be busy in the testing scenario. If it is, turn to step 2).

2) Checking all billing records where *X* is the callee to see if the call is a forwarded call and the next forwarded address is the predefined party. If it is, the property of CFBL holds and otherwise it is violated.

5.4.3.4 Derived Property of TWC

The informal requirement description of TWC is *"TWC allows the connection of three parties in a single conversation."*

Every successful connection has a corresponding billing record in TheBill database, which consists of five parts: Payer, Caller, Callee, LogBeginTime, and LogEndtime. Thus, if a three-way connection is established, there must be two billing records such that 1) the TWC subscriber is either the caller or callee in one call (the TWC subscriber must first be engaged in one call before it can initiate a second one) 2) the TWC subscriber is caller in the other call (the second call must be initiated by the TWC subscriber) 3) their logging time periods are overlapped (the second call must be established during the first call's connection.).

The property of TWC derived from the above informal specification is:

If user X subscribes to TWC and TWC is activated, then there are two billing records that 1) in one call X is the originating party and in the other call X is either the originating party or the terminating party 2) the LogBegin time of the first call is within the log time of the second call.

Unlike the previous three policy features whose activation condition is predefined (e.g., IN Teen Time for INTL, the subscriber's busy time for CFBL or no extra activation condition for INFB except the registration to the feature), in the case of TWC, it is the subscriber who decides whether the feature is to be activated or not during a call process. In FIDS, we assure the activation of TWC by making the system synchronized with a

specific test scenario where the TWC subscriber A flashhooks and dials the third party C when talking to B (see detailed description in *§5.6.1 Scenario Designer: Test Scenario Generation*).

In FIDS, the property of TWC is validated as follows:

1) Checking the whole billing history, find all the billing records where X is the originating party.

2) For each above billing record, check all the billing records whose log time is overlapped and see if X is either the originating party or the terminating party. If it is, the TWC property is preserved. Otherwise, the TWC property is violated.

5.5 Feature Interaction Detection System

In the previous sections, we have discussed the definition of FI and the derived system properties and feature properties. In this section, we introduce an FI Detection System (FIDS) using the above method to detect FI.

The input of FIDS is a collection of feature names whose properties are going to be validated. The output are traces that violate either the system properties or the feature properties or both, reported using the Message Sequence Charts (MSC) format with a brief description of the symptoms.

In the validation phase, only one feature is input into FIDS, where it gets activated and validated. Property violations found in this phase are not FI but design defects of the

feature. In the detection phase, the input of FIDS is a collection of two or more features to be considered.

All of the input features are activated during one call process via synchronization between the system and a pre-designed test scenario. Their activations are interleaved. The system property checking is done during execution by a global monitoring process, WatchDog, which raises an error flag when system property violations are detected, e.g conflicting signals given to user or incorrect billing actions. The feature property is validated by the "Property Checker", a component of FIDS which checks the feature property by analyzing a snapshot of the billing database, taken at the end of the scenario by the WatchDog process, together with user's behavior trace if necessary.

As illustrated in Fig 5.1, FIDS consists of five parts*: Scenario Designer, Integrator, FI Hunter, Property Checker and MSC translator*. Below, we give a brief description of each part and in the next section *§ 5.6 FI detection between INTL and CFBL, INFB, TWC*, we illustrate in detail how FIDS works, using four features INTL, CFBL, INFB, and TWC as examples.

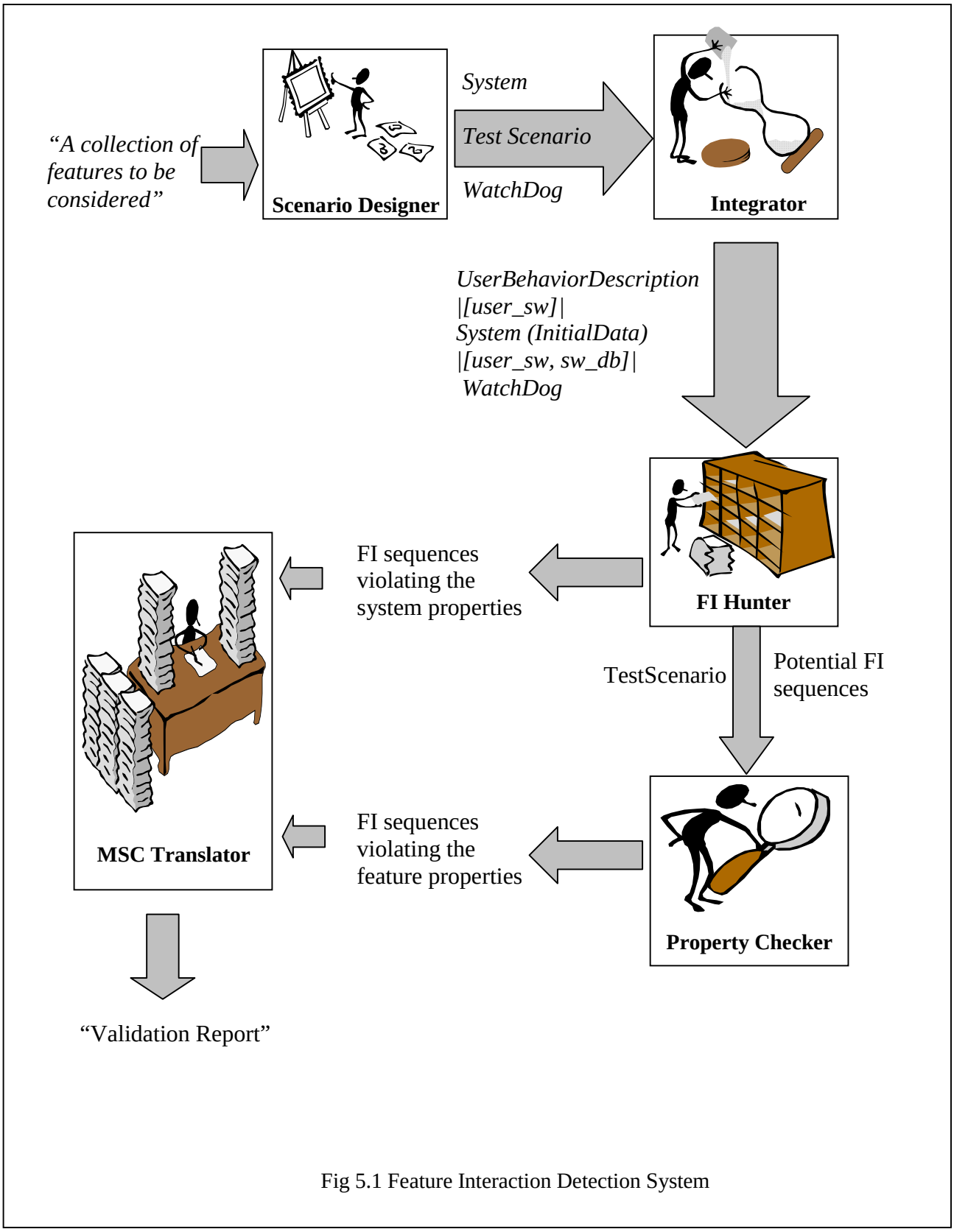


Fig 5.1 Feature Interaction Detection System

- Scenario Designer

Scenario Designer is the first step of FIDS. It takes the names of features to be tested and designs specific test scenarios for them. A test scenario consists of two parts: 1) the *initial data* that indicates the subscribing data and status of *A*, *B* and *C* 2) the *user behavior description* that describes events that must occur at the user side to activate the input features during a call process. For each pair of features to be considered, the Scenario Designer will design 4 test scenarios where the user behavior descriptions are the same and the initial data cover all 4 possible combinations of B and C's initial busy/idle status, (A should be always idle at the beginning since A is the caller). By synchronizing the system with such test scenarios, we could reduce the size of the expanded specification without losing any possible FIs, since the user behavior description only restricts the system behavior until all features are activated. Only one FI type could possibly occur during this period, that is, one feature's activation is inhibited due to other activated features. If such FI happens, the system will deadlock since it cannot synchronize with the test scenario any more, since the latter is designed in a way to assure that all features are activated. This deadlock can be detected by the *FI Hunter* later while searching for FI traces.

- Integrator

Integrator of FIDS takes the test scenario generated by Scenario Designer, initializes the user status and billing database as specified in the initial data and lets the system synchronize with the users behavior description at gate “usr_sw” and with a global monitoring process “WatchDog” at gate “usr_sw” and “sw_db”. “WatchDog” is general to the system and monitors the same events for all pairs of features:

1) Incorrect billing actions

2) Conflicting signals

3) End of scenario reached

In addition to this run-time analysis, each time the end of a scenario is reached, WatchDog also saves data records to be analyzed in a post-test analysis performed by Property Checker.

- FI Hunter

The third step of FIDS is “FI Hunter”. In “FI Hunter”, 1) the new integrated system is translated to a C program which can be further executed or simulated using *Caesar* and *Caesar.ADT* 2) the trace-matching tool, *Caesar.exhibitor*, will execute the generated C program and filter out all traces leading to either “error” flags or a “finish” flags raised by “WatchDog”. Traces where WatchDog raises the “error” flags are FI traces violating the system properties. Traces where WatchDog raises the “finish” flag are *potential FI traces* that need to be further

analyzed by the "Property Checker." We call these "potential FI traces" because they might become real FI traces if the feature properties are found to be violated in the subsequent analysis of the billing data.

- Property Checker

For each potential FI trace generated from FI Hunter, Property Checker examines the final billing data saved by WatchDog, and checks if there is any violation of the properties of the activated features. If there is, the trace becomes a FI trace and will be output by the "Translator", along with a brief diagnostic.

- MSC Translator

The last step, the "Translator" takes all detected FI traces generated from either the "FI Hunter" or the "Property Checker", translates them into the format of Message Sequence Charts (MSC) and generates a final validation report.

5.6 FI detection between INTL and CFBL, INFB, TWC

In our model, FI are detected pair-wise. Thus, to detect the FI between INTL and CFBL, INFB, TWC, the input of FIDS is {INTL, CFBL}, {INTL, INFB}, {INTL, TWC}, respectively.

5.6.1 Scenario Designer: Test Scenario Generation

The Test scenario process describes the testing environment, which interacts with the telephony network system so that all input features can be activated during a single call process from caller *A* to callee *B*.

The Test Scenario process consists of two parts: initial data part and user behavior description part.

- Initial data part

The initial data part specifies the initial data stored in the billing database "TheBill" and user status database "TheUser". The billing database is initially an empty database and grows along with the execution of the system. Unlike the billing database, the size of the user status database will not change once it is initialized. The user status database consists of three user status records, carrying *busy/idle* status and subscription information for each user. The initial *busy/idle* status of A is set to be *idle* to originate a call. B and C can be either *idle* or *busy* at the beginning. Therefore, FI needs to be analyzed with four initial scenarios: 1) both B and C are busy; 2) both B and C are idle; 3) B is busy while C is idle; 4) B is idle while C is busy.

- User behavior description part

In our model, we define 1) *A* to be the caller of the call process and to subscribe to features affecting outgoing calls, e.g. INTL; 2) *B* to be the callee and to subscribe to features affecting incoming calls, such as INFB and CFBL 3) *C* to be the third party of features involving three users, e.g. CFBL or TWC. TWC can be subscribed by either the caller or the callee but in our model, we let the caller, *A*, subscribe to TWC.

The user behavior description part specifies a call origination process from *A* to *B.* Different features are activated by different scenarios.

The basic scenario is "*A* offhooks; *A* gets DialTone; *A* dials *B*". INFB, CFBL can be triggered in this scenario. Some features such as INTL and TWC need more specific

actions occurring on part of the user's behavior: The scenario for INTL is "*A* offhooks; *A* gets an announcement of AskForPIN; *A* dials the valid PIN, *P*; *A* dials *B*". The scenario needed by TWC is "*A* offhooks; *A* gets DialTone; *A* dials *B*; Ringing tone at B, audibleRinging tone at A; *B* offhooks; AudibleRinging tone at *A* stops; Ringing tone at *B* stops; *A* flashhooks; *A* dials *C*".

If input features have different scenarios to be activated, the "Scenario Designer" will combine corresponding scenarios into a comprehensive one so that all features can be triggered within it. For example, if input features are INTL and CFBL, the combined scenario is "*A* offhooks; *A* gets an announcement of AskForPIN; *A* dials the valid PIN, *P*; *A* dials *B*".

Fig 5.2 illustrates one of four LOTOS test scenarios for INTL and CFBL when B is busy and C is idle. The initial data part consists of five sentences "let". The first three assignment sentences define the status of users A B C. User A is initially idle and subscribes to INTL. B is initially busy and subscribes to CFBL. The forwarded address (a parameter of CFBL that indicates the next forwarded address while the subscriber is busy) of B is C. C is initially idle and subscribes to CFBL. The fourth "let" sentence defines the user status database "TheUser" which is composed of above three user status

```
(* Initial Data Part *)

let Status_A: Status = Status ( false, 9, Time(0), Time(20), Undefined, insert(5, {}) ) in

let Status_B: Status = Status ( true, 0, Time(0), Time(0), C, insert ( 1, {})) in

let Status_C: Status = Status ( false, 7, Time(0), Time(20), {}, false, Undefined, insert ( 1, {} ) ) in

let InitSet: UserStatusSet = insertStatus ( CreateUserStatus ( A, Status_A ),
                             insertStatus ( CreateUserStatus ( B, Status_B ),
                              insertStatus ( CreateUserStatus (C, Status_C ),
                              {} of UserStatusSet ) ) ) in
let InitBill:BillSet = {} in


(* User Scenario Part *)

process TestScenario [user_sw, ot] : noexit:=
  user_sw    ! OffHook    ! A;
  user_sw    ! Announce ! A !AskForPIN;
  user_sw    ! Dial   ! A   ! P;
  user_sw    ! DialTone   ! A;
  user_sw    ! Dial    ! A    ! B;
  (
    Users[user_sw]
    [>
    ot    ! Finish;
  )
 stop

endproc (* Test_Scenario *)
```

Fig 5.2 Test Scenario for INTL and CFBL

records. The fifth “let” sentence states that the initial billing database is empty. The user scenario part describes the combined user scenario of INTL and CFBL. Since the “TestScenario” process specifies the signal occurring at the user side, it synchronizes

with the telephony system through “user_sw” gate. When all activated features finish, the “WatchDog” process will send a signal “finish” on gate “ot” to terminate the execution of the “TestScenario”.

5.6.2 Watch Dog

Unlike the “TestScenario” which needs to be tailored for different features, the global monitoring process, the “WatchDog”, does not need to change for different features. Besides the system property violation monitoring, it is also responsible for monitoring the end of scenario reached (all activated features finish execution) and for taking a snapshot of the billing database when the call process finishes. The snapshot of the billing database will be further analyzed by the “Property Checker” to see if the activated feature properties are violated or not.

The “WatchDog” monitors every billing action and signal given to users by synchronizing with the telephony network system at gate “user_sw” and “sw_db”. When conflicting signals going to the user are detected, the “WatchDog” reports an error message “ConflictingSignals” at “err” gate; when an invalid billing action is detected, the error message reported is “InvalidBilling”.

Fig 5.3 lists part of the WatchDog process.

```
process WatchDog[user_sw, sw_db,err,ot]: noexit:=

(*When it detects a StartAudibleRinging, WatchDog monitors the next signals given
    to  that user, and raises an "error" message if it is LineBusyTone,
   ScreenedMessage, Disconnect or AudibleRing from another user*)
 (user_sw !StartAudibleRinging ?Ad ?Dest1:Address;
         (user_sw !LineBusyTone !Ad;
          err !ConfictingSignals;
          WatchDog [user_sw, sw_db,err,ot]
          []
          user_sw !ScreenedMessage !Ad;
          err !ConfictingSignals;
          WatchDog [user_sw, sw_db,err,ot]
          []
          user_sw !Disconnect !Ad;
          err !ConfictingSignals;
          WatchDog [user_sw, sw_db,err,ot]
          []
          user_sw !StartAudibleRining !Ad ?Dest2:Address
          ( [Dest1 ne Dest2]->
            err !ConfictingSignals;
            WatchDog [user_sw, sw_db,err,ot]
             []
             [Dest1 eq Dest2]->
             WatchDog [user_sw, sw_db,err,ot]
          )
          []
          user_sw !StopAudibleRinging !Ad !Dest;
          WatchDog [user_sw, sw_db,err,ot]
         ….
         )
 )
(*When it detects LineBusyTone, WatchDog monitors the next signals given
    to  that user, and raises an "error" message if it is StartAudibleRinging*)
 []
 ( user_sw !LineBusyTone ?Ad ?Dest:Address;
         (user_sw !StartAudibleRinging !Ad;
          err !ConflictingSignals;
          WatchDog [user_sw, sw_db,err,ot]
          []
          user_sw !Onhook !Ad;
          WatchDog [user_sw, sw_db,err,ot]
         …
         )
 )
 []
….

endproc (* Watch_Dog *)
```

Fig 5.3 The WatchDog Process (partial)

5.6.4 Integrator

The Integrator composes the “Test Scenario” and the “WatchDog” into the telephony system in the following way: the initial data part of the “Test Scenario” replaces the initialization part of the telephony system. The “TestScenario” process of “Test Scenario” is selectively synchronized with the system at gate “user_sw”. “WatchDog” monitors billing actions and signals going to users and the end of scenario reached. It is partially synchronized with the system at gate “sw_db”, “user_sw”. Fig 5.4 illustrates the new system integrated with TestScenario and WatchDog.

5.6.5 FI Hunter

FI hunter uses 1) Caesar and Caesar.ADT to compile the new system integrated with TestScenario and WatchDog process into a C program, and 2) the trace-searching tool *Caesar.Exhibitor* to filter out all FI traces where WatchDog raises a “error” or “finish” message by executing the generated C program.

Four types of traces are detected by FI hunter:

1) FI traces leading to deadlock before the call process is completed.

   The pattern specified for this type of traces is: ~“*ot !Finish”*<*until*> <*deadlock*>. The *goal event* is <*deadlock*>, which is a *Caesar.Exhibitor* keyword representing the deadlock state of the system. The *condition* of this goal is ~“*ot !Finish”*, which means “no ‘ot !Finish’ event occurs before reaching the goal event”. <until> is a keyword separating the condition and the goal.

```
specification SystemModel [user_sw, sw_scp]: noexit
        …
        (* Data Part *)
        …

behaviour
        SYSTEM [user_sw, sw_scp]
        |[user_sw]|
        WatchDog [user_sw, sw_db, err]

        where
        process SYSTEM [user_sw, sw_scp]: noexit :=
                …
                (* Initialization Part *)
                …
                hide, scp_db, sw_clk, sw_db in
                 (
                    (
                        (       ( TestScenario[user_sw]
                                  |[user_sw]|
                                  USERS [user_sw]
                                 )
                                  |[user_sw]|
                                 ( SWITCH [user_sw, sw_scp, sw_db, sw_clk]
                                  |[sw_clk]|
                                  CLOCK [sw_clk] (Initial Time)
                                 )
                                  |[user_sw, sw_db]|
                                 WatchDog [user_sw, sw_db, err, ot]
                        )
                      |[sw_scp]|
                      SCP [sw_scp, scp_db]
                    )
                  |[sw_db,scp_db]|
                  DBAPI [sw_db, scp_db] (Initial Data)
                )

        endproc (* SYSTEM*)

endspec (* SystemModel *)
```

Fig 5.4 System Integrated with TestScenario and WatchDog

2) FI traces leading to conflicting signals to user.

Since the "WatchDog" process will report an error message "ConflictingSignals" at gate "err" when catching conflicting signals give to users, the searching goal for this type of FI traces is: <*until*> *"err !ConflictingSignals"*. The *goal event* is *"err !ConflictingSignals"*. No *condition* is required in this goal.

3) FI traces leading to invalid billing actions.

Since the "WatchDog" process will report an error message "InvalidBilling" at gate "err" when the invalid billing actions are detected, the searching goal for this type of FI traces is: <*until*> *"err !InvalidBilling"*. The *goal event* is *"err !InvalidBilling"*. No *condition* is required in this goal.

4) Potential FI traces.

Potential FI traces are those traces reflecting the entire scenario. When all activated features finish at the end of the scenario, the "WatchDog" process will take a snapshot of the billing database and raise the "Finish" signal at gate 'ot". Therefore, the searching goal for potential FI traces should be: ~ "err !*" <*until*> *"ot !Finish"*. The *goal event* is *"ot !Finish"*. A condition for this goal is that no "err" flag has been raised before.

The following is the example of the FI hunter output:

Test features: CFBL and INFB

Test Scenario: B subscribes to CFBL and INFB; The forwarded address of CFBL is C; B is initially **BUSY**. C subscribes to INFB; C is initially idle.

Output of FI hunter:

- FI traces leading to deadlock: None

- FI traces leading to conflicting signals:

```
<initial state>
"USER_SW !OFFHOOK !A"
"i" (SW_DB [971])
"USER_SW !DIALTONE !A"
"USER_SW !DIAL !A !B"
"i" (SW_DB [971])
"i" (SW_DB [971])
"i" (SW_DB [971])
"i" (SW_DB [971])
/* INFB gives caller A a linebusytone since B is busy */
"USER_SW !LINEBUSYTONE !A"
"i" (SW_DB [971])
"i" (SW_DB [971])
"i" (SW_DB [971])
"i" (SW_DB [971])
/* CFBL forwards the call to C and gives back to caller A an audibleringing tone
when rings C */
"USER_SW !STARTAUDIBLERINGING !A !C"
/* Error flag raised because linebusytone and audibleringing are conflicting
successive signals given to user A  */
"ERR !CONFLICTINGSIGNALS"
<goal state>
```

- FI traces leading to invalid billings: NONE

- Potential FI traces: None

Another example of FI hunter output for the same pair of features but with different initial states of the callee is as follows:

Test features: CFBL and INFB

Test Scenario: B subscribes to CFBL and INFB; The forwarded address of CFBL is C; B is initially **IDLE**.

Output of FI hunter:

- FI traces leading to deadlock: None
- FI traces leading to conflicting signals: NONE
- FI traces leading to invalid billings: None
- Potential FI traces:

```
<initial state>
"USER_SW !OFFHOOK !A"
"i" (SW_DB [971])
"USER_SW !DIALTONE !A"
"USER_SW !DIAL !A !B"
"i" (SW_DB [971])
"i" (SW_DB [971])
"i" (SW_CLK [971])
"i" (SW_CLK [971])
"i" (SW_DB [971])
"i" (SW_DB [971])
"i" (SW_DB [971])
"SW_SCP !TRIGGER !INFO_ANALYZED !B !A !B !TIME (2)"
"i" (SW_DB [971])
"i" (SW_DB [971])
"i" (SW_DB [971])
"SW_SCP !RESPONSE !ANALYZE_ROUTE !B !A !B !B"
"i" (SW_DB [971])
```

/*Since B is idle, CFBL processes the call normally */

```
"USER_SW !STARTAUDIBLERINGING !A !B"
"USER_SW !STARTRINGING !B !A"
"USER_SW !STARTRINGING !B !A"
"i" (SW_DB [971])
"i" (exit)
"USER_SW !OFFHOOK !B"
"USER_SW !STOPRINGING !B !A"
"USER_SW !STOPAUDIBLERINGING !A !B"
"USER_SW !STARTAUDIBLERINGING !A !B"
"i" (SW_DB [971])
"i" (SW_CLK [971])
```

/* CFBL charges the call to caller A*/

```
"SW_DB !LOGBEGIN !A !B !B !TIME (3)"
"i" (exit)
```

/* INFB connects the call to B*/

```
"i" (SW_DB [971])
```

```
"i" (exit)
"USER_SW !OFFHOOK !B"
"USER_SW !STOPAUDIBLERINGING !A !B"
"USER_SW !STOPRINGING !B !A"
"i" (SW_DB [971])
"i" (SW_CLK [971])
```

/* INFB charges the call to B */

```
"SW_DB !LOGBEGIN !A !B !A !TIME (4)"
"i" (exit)
"USER_SW !ONHOOK !B"
"i" (SW_DB [971])
"i" (SW_CLK [971])
"USER_SW !DISCONNECT !A !B"

"SW_DB !LOGEND !A !B !TIME (5)"
"i" (exit)
"USER_SW !DISCONNECT !A !B"
"SW_DB !LOGEND !A !B !TIME (6)"

"USER_SW !ONHOOK !A"
"i" (SW_DB [971])
"USER_SW !ONHOOK !A"
"i" (SW_DB [971])
"i" (exit)
"i" (SW_DB [971])
"i" (SW_DB [971])
"i" (SW_DB [971])
```

/* When the WatchDog detects that the call process is completed, it takes a snapshot of the billing database at that moment and sends a "finish" signal at gate "ot" to stop the whole system. Note that two billing records are generated here, since CFBL and INFB were executed in parallel. One of them billed B from time(4) to time(5). The other billed A from time(3) to time(6). Only the first record is correct. The WatchDog process is unable to detect this FI, however further analysis done by the Property Checker will detect it. Two records have different start and ending times because two features read the clock separately. */

```
"OT !COMPLETED !INSERT (ITEM (B, A, B, TIME (3), TIME(5)), INSERT (ITEM (A, A, B, TIME (4), TIME (6)), {})) "
"OT !FINISH"
<goal state>
```

5.6.6 Property Checker

"Property Checker" consists of two parts, the *main checking routine* and the *property checking routines*.

The main checking routine analyzes the test scenario and invokes the corresponding property checking routines to validate the property presence in the final status of the billing database (the snapshot taken by the "WatchDog"), which is stored at the second -to-last event in the potential traces. The property of the feature in the detected pair is not always checked by the "Property Checker". For example, if the test scenario is CFBL&INFB (Busy B) (the detected pair is CFBL and INFB and B is initially busy), the "Property Checker" will check both the properties of CFBL and INFB. However, for the same pair, if the initial state of B is idle, only the property of INFB is checked because CFBL processes the call as a normal call if subscriber B is idle when the call comes.

Every derived feature property described in § 5.4.3 has a corresponding property checking routine in the "Property Checker". The property validating routine takes one parameter passed from the checking routine, *the subscriber's address*, and validates the presence of the property by examining every record in the billing database.

5.6.6.1 INTL

If the user subscribes to INTL, the main checking routine will call the INTL property checking routine.

The INTL property checking routine examines the billing records generated during a given TeenTime period and counts the billing records where the subscriber is the originating party (See §5.4.3 for the definition of the originating party). Note that since features are executed in parallel, one call may have been charged more than once. INTL property checking routine only counts those billing records reflecting different calls. If the number of such billing records is 0, it returns to the main checking routine.

Otherwise, the INTL property will check how many times the TeenPIN has been input. If the number of input TeenPINs is no less than the number of billing records where the subscriber is the originating part, then the property holds. Otherwise the INTL property is violated and this FI is written into the analysis report.

The following is a snapshot taken when INTL and TWC feature finish execution:

```
"OT !COMPLETED !INSERT (ITEM (A, A, B, TIME (5), TIME(7)), INSERT (ITEM (A, A, C, TIME (6), TIME (7)), {}))
"
```

Since A subscribes to INTL and there are two billing records where A are the originating parties, INTL checking routine will check the traces backwards seeing if at least two TeenPINs have been input. However, since the call from A to C is a second call of the three-way calling among A, B and C, INTL feature is bypassed and no TeenPIN is required for the second call, INTL can find only one TeenPIN. Thus, the INTL property is violated and this FI trace is written into the analysis report.

5.6.6.2 INFB

If the user subscribes to INFB, the main checking routine will call the INFB property checking routine.

The INFB property checking routine examines the billing records where the subscriber is the callee and sees if the payer is also the subscriber. If it is, then it returns to the main checking routine. Otherwise, the INFB property is violated and the FI is recorded into the analysis report.

The following is a snapshot taken when INFB and CFBL finish execution:

```
   "OT  !COMPLETED  !INSERT  (ITEM  (B,  A,  B,  TIME  (4),
TIME(5)), INSERT (ITEM (A, A, B, TIME (3), TIME (6)), {}))
"
```

The second record from time(3) to time (6) is not correct since B is the callee but not the payer. Thus, INFB property is violated and this FI is written into the analysis report.

5.6.6.3 CFBL

If the user subscribes to the CFBL, the main checking routine will further check if the subscriber is initially set to busy when the call comes. If it is, the CFBL property checking routine is called. Otherwise, the main checking rountine continues to check the next subscribed feature.

The CFBL property checking routine examines all billing records where the subscriber is the callee and searches for the corresponding forwarded part, which is another record with the same starting and ending time and where the subscriber is the caller. If found, then it continues with the next subscribed feature. Otherwise, the CFBL property is violated and the FI is recorded into the analysis report.

The following is a snapshot taken when INFB and CFBL finish execution:

```
   "OT  !COMPLETED  !INSERT  (ITEM  (A,  A,  B,  TIME  (2),
TIME(3)), INSERT (ITEM (B, B, C, TIME (2), TIME (3)), {}))
"
```

Since the subscriber B is initially set to busy in the test scenario, the CFBL property checking routine is called. The CFBL checking routine finds that there is an incoming call to B from A from time(2) to time(3), so it searches for the corresponding forwarded part. The latter is another record which has the same time period and is for a

call from B to the predefined forwarding address C. The search is successful, so the CFBL property holds.

5.6.6.4 TWC

If the user subscribes to TWC in test scenario, the main checking routine will call the TWC property checking routine.

The TWC property checking routine examines every billing record where the subscriber is the originating party and searches for records whose starting time fits into any other records where the subscriber is either the originating party or terminating party. (See § 5.4.3 for the definition of the originating party and the terminating party) If found, the TWC property checking routine returns to the main checking routine. Otherwise, the TWC property is violated and the FI is recorded into the analysis report.

Consider the following example given in §5.6.6.1.

```
"OT !COMPLETED !INSERT (ITEM (A, A, B, TIME (5), TIME(7)),
INSERT (ITEM (A, A, C, TIME (6), TIME (7)), {})) "
```

Since A subscribes to TWC, TWC property checking routine will search an occurrence of a three-way connection by examine the billing records. First, it finds the second record where A is the originating party from time(6) to time(7), then it searches for another record where A is either the originating part or the terminating party and the talking time period covers time(6). The search is successful. Thus, TWC property holds.

5.6.7 MSC Translator

Message Sequence Charts are a well-known technique for the description and specification of scenarios in distributed systems with asynchronous communication,

especially telecommunication systems. They are also a standard language recommended by the International Telecommunication Union (ITU) [ITU-T96]. The MSC language consists of both a graphical and a textual syntax. It describes both system structure (i.e. components) and behavior (i.e. messages exchanged). Message Sequence Charts can be used as an overview language of services offered by distributed entities, as a requirement statements for SDL specifications, for simulation and validation, for the selection and specification of test cases, for formal specification of communication, and for interface specification.

To enhance the readability of our FI detection report, in the last step of FIDS, we transform the FI traces from LOTOS traces to a more easily understood MSC format.

Note that 53 MSCs were generated by FIDS to illustrate 150 FI that are found during the contest [FHLS98]. Fig 5.5 shows an example of MSC generated from CFBL&INFB (idle B) FI traces. (See §5.6.5 for the corresponding LOTOS FI traces). The network entities are represented using boxes on the top and extending lines under them. The signals (messages) sent between these entities are described using labeled arrowhead lines. The direction of the arrow indicates the sending direction of the message. The label above the arrowhead line is the name of the message and the bracket characters under the arrowhead line are the parameters passed in the message. For clarity, user A B C are listed as independent network entities in the MSC and parameters indicating the signals is from /to which user is omitted. For example, “user_sw !Dial !A !B” is mapped into an “Dial” message with parameter “B” passing from user A to the switch. “user_sw !StartRinging !B !A” is mapped into a “StartRinging” message with parameter “A” passing from the switch to user B. “sw_scp !Trigger !INFO_ANALYZED

!B !A !B !Time (2)" is mapped into a "Trigger" message sent from the switch to the SCP with parameters "INFO_ANALYZED", "B" "A" "B" "Time(2)"

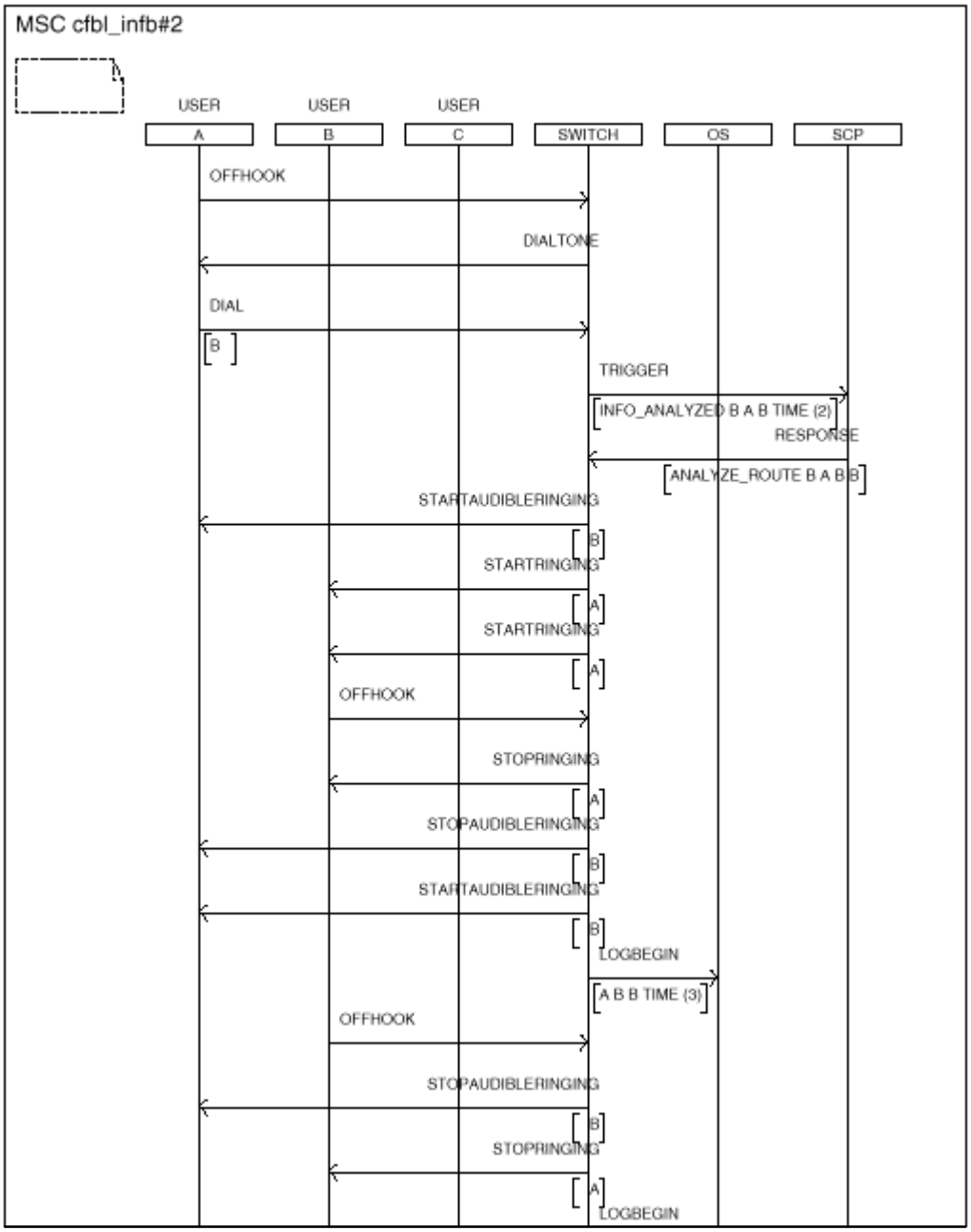


Fig 5.5 An Example of MSC (to be continued)

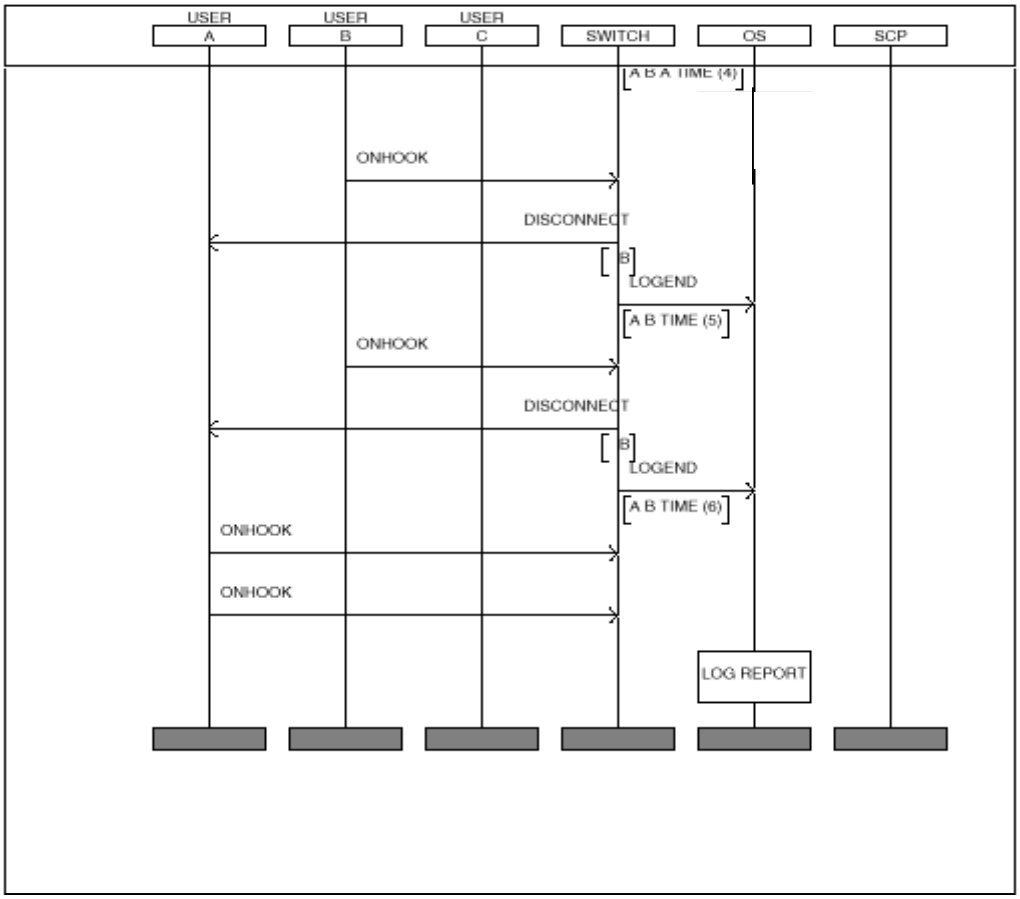


Fig 5.5 An Example of MSC (continued)

5.7 FIDS evaluation - Comparing our result with the benchmark FI

The number and type of FIs detected and the efficiency of the tool are two basic factors when evaluating an FI detection tool. For this reason, the organizing committee of the Feature Interaction Contest (see *§1.3 Feature Interaction Contest*), published a benchmark document [BGGO99], listing the FIs that they believed to exist among the feature to be studied in the contest. In this section, we evaluate our tool by comparing the set of interactions detected by FIDS with the one provided in the benchmark.

Before presenting a detailed comparison, we should note two architectural issues that determine what and how many FIs would be generated. As mentioned in *§5.4.1 Feature Compositions*, the contest specifications were not specific concerning the composition of the features. We decided to use an interleaving composition method, i.e. features can execute in parallel (LOTOS operator "|||") and do not define any specific behavior patterns on the user and the billing database side, so that they can synchronize on signals in any order and the call process will not be affected if conflicting signals occur. The advantage of designing such a robust system is that since FIs occurring under the same testing scenario are more or less related, a series of FIs can give us more clues than a single FI when analyzing and fixing FIs.

As to the benchmark, its authors did not mention assumptions on the system architecture, it is possible to infer from the FI scenario descriptions in [BGGO99] that 1) the features are executed in parallel, 2) the call process terminates when any conflict occurs.

The number and types of FI detected are easy to compare. However, the measurement of efficiency is more complicated. Due to the fact that different FI tools use different FI detecting methodologies and different implementation languages and the processing time highly depends on the hardware and software used, we choose the number of testing scenarios used per FI to calculate the efficiency. In this way, we can concentrate more on the methodology itself by excluding the implementation details.

Concerning the execution time, we limit ourselves to saying that this varied from few seconds to 24 hours, on a low-end Sparc machines, depending on the complexity of the feature involved.

5.7.1 Comparison Based on FI Types

As mentioned in §5.4.2 and §5.4.3, we clarify FIs according to the feature and system properties that they violate: 12 feature properties and 3 system properties. Thus, we have 12 feature property violation FI types and 3 system property violation FI types, *Deadlock, Incorrect billing,* and *incompatible successive signals given to user*. The benchmark instead tries a more general classification: FIs are categorized into corresponding conflict/failure types such as *Billing conflict, Call termination conflict, Forwarding conflict, Disconnect conflict, Feature inhibition (Feature fails to activate), Number delivery failure (Number not displayed), PIN conflicts (over-ride PIN), Flash conflict.* In our view, they don't quite succeed, as pointed out below.

Table 5.2 lists the mapping relationship from the benchmark FI types to the FIDS FI types.

Table 5.2 The Mapping Table of FI Types

Benchmark FI Type	FIDS FI Type
Billing conflict	Incorrect billing
Call termination conflict	Incompatible successive signals to user
Flash conflict.	TWC/CW feature property violation
Disconnect conflict	Incompatible successive signals to user
Forwarding conflict	Incompatible successive signals to user (Audibleringing from different resource)
PIN conflicts (over-ride PIN)	INTL/CC feature property violation
Number delivery failure (Number not displayed)	CND feature property violation
Feature inhibition (Feature fails to activate)	Feature property violation

From the above comparison, we find that all benchmark FI types can be mapped to a corresponding FIDS FI type. Thus, theoretically speaking, FIDS can detect all benchmark FI. However, on the other hand, not all FIDS FI types can find a suitable benchmark FI mapping. For example, in FIDS, the feature property violation check is done to all features, but in benchmark FI detection, only some features properties, i.e. only the feature properties of CND, INTL, CC, TWC, CW are partially checked. The well-known FI between CFBL and TCS (Calls forwarded by CFBL bypass the incoming call screening of TCS) is not mentioned in the benchmark paper and can not be mapped to any of their types because no failure or conflict occurs in this case and only the feature property of TCS is violated (numbers in the screened list reach the subscriber anyway).

5.7.2 Comparison Based on the Number of FI Detected

Since FIDS can detect more types of FI than the benchmark FI, there is no surprise that FIDS detects more FIs, 150, than the benchmark FI, which detects only 99. Detailed comparisons of FI detected for each pair of features are listed in Appendix. However, there are two kinds of benchmark FIs that are not detected by FIDS:

- FI between feature and itself

  According to our FI definition, FIs occur only among 2 or more than 2 integrated features. Any undesirable effects (interaction) between the feature and itself, which maybe due to recursive execution or multi-user simultaneously execution, are not considered as FIs but as design defects of the feature itself. Note this is another issue for discrepancy between our findings and those of the benchmark, because the latter lists such undesirable effects as FIs.

- FIs involving four users

  Due to limited resource, the test scenario of FIDS is restricted to have only 3 users or less. Thus, FIs involving 4 users, i.e. FIs between CW and TWC features, cannot be detected by FIDS. This is because currently FIDS uses Caesar.Exhibitor as its trace searching tool, which does the trace searching on a fully pre-expanded behavior tree. Since the users' behaviors in the system are interleaved with each other, the size of the expanded tree is growing exponentially when incrementing the number of users: if the number of users is more than 3, the expanded behavior tree will exceed the maximum size that Caesar.exhibitor can handle. This problem can be solved by using other techniques, however this is left for further research.

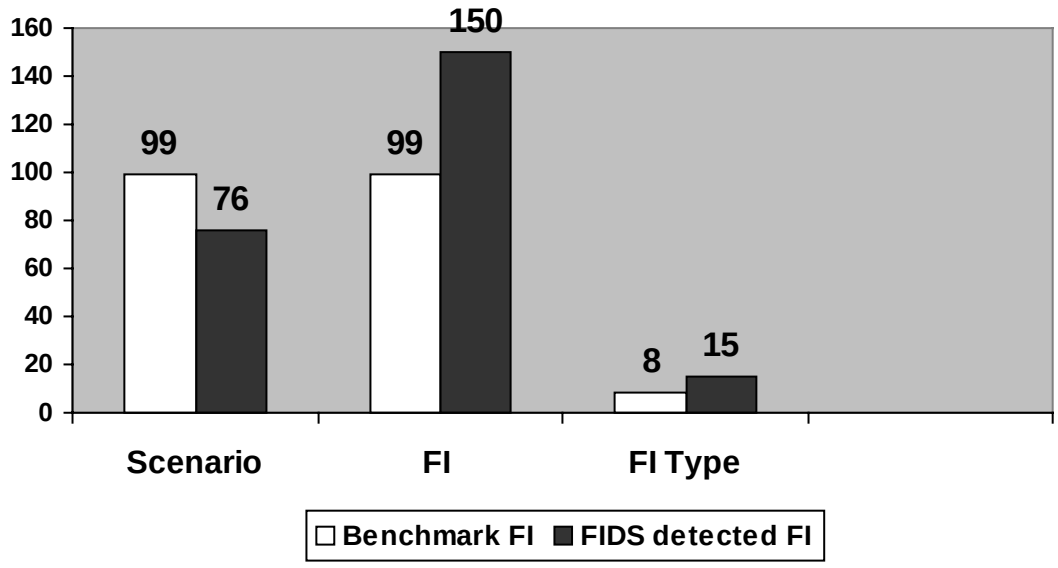
In total, the benchmark includes 23 FIs that are not detected by FIDS. Among those, 12 FIs are between a feature and itself and 11 FIs involve feature TWC or feature CW or both, and use at least four users.

5.7.3 Comparison Based on Testing Scenario Used Per FI

According to FI traces described in the benchmark paper, the call process will be terminated when it encounters the first FI. Thus, only one FI can be detected per scenario. However, since no specific behavior patterns are defined on the user or the billing database side, FIDS can tolerate any conflicting signals and the call process continues until all activated features finish. Thus, there is no wonder that FIDS can detect more than one FIs per scenario.

Chart 5.1 summarizes the comparison between FIDS and the benchmark FI.

Ch. 5.1 Comparison between FIDS Result and the Benchmark FI



The white columns represent the benchmark FI results and the black column represents the FIDS results. Each benchmark FI corresponds to one testing scenario, thus, 99 testing scenarios generate 99 benchmark FI of 8 types (see Table 5.1). FIDS uses 76 testing scenarios and detects 150 FI of 15 types (3 system property violation types + 12 feature property violation types).

Chapter 6. Conclusions and Future Work

Although much progress has been made on accelerating the development and introduction of new telephony features (for example, the Intelligent Network concept) [Viss95], the feature interaction problem [BDCG89] remains one major obstacle for the rapid development and introduction of new features into modern telecommunications systems. This thesis describes a model, based on a formal approach, for specifying a telephony system integrated with both switch-based features and IN features, together with an implemented feature interaction detection systems. Our system rated among the best world wide in a recently held international context (see *§1.3 Feature Interaction Contest*)

6.1 Summary

The background and motivation for our work is given in Chapter 1. This chapter also includes a list of contributions.

Chapter 2 presents a survey of related work on the formalisms that are used to specify telephony systems and of FI detection methodologies using FDTs.

Chapter 3 gives an overview of the Basic Call Process, a classification of features, and presents the concepts of feature integration and activation. It describes the design of a telephony system model integrated with both switch-based features and IN features, finite or infinite. Four features, INTL, CFBL, INFB and TWC, are used as examples to illustrate the feature integration and activation mechanism.

Chapter 4 shows the use of LOTOS as a Formal Description Technique (FDT) in specifying the telephony system model and features. First, it gives a brief overview of the LOTOS language by describing its main operators and some examples in the context of telephony network systems. Then, it discusses four main styles of writing LOTOS specifications of telecommunication systems. They are *the monolithic style, the state-oriented style, the constraint-oriented style and the resource-oriented style*. Each style has its own uses in telephony system specifications and styles can be mixed in one specification to meet different requirements. In our system model, since the observable behavior of the system is described as a composition of separate resources whose functionality is well defined, we chose a mixture of resource-oriented style and state-oriented style: the resource-oriented style is used to reflect the architectural model of the system at the specification level, and the state-oriented style is used to specify features (BCP, INTL, INFB, CFBL, TWC) that are defined by LTSs.

In Chapter 5, a formal definition of Feature Interaction is provided and an FI Detection System (FIDS) is developed based upon the definition. FIDS deals with the detection of logical interactions which occur when some of the requirements or assumptions (the properties of the system and the features), that must be satisfied when a feature is introduced separately in the network, are violated. Our FI definition improves on the traditional one given by P. Combes et. al [CoPi94] and W. Bouma [BoZu92] by adding system properties into the set of properties that must be checked. This includes the correctness of billing and the consistency of successive signals given to user.

FIDS consists of five parts: *Scenario Designer*, which takes the names of features to be considered and designs specific test scenario for them; *Integrator*, which integrates

the test scenarios generated by the Scenario Designer, and the WatchDog process, that monitors the system property violation, into the system specification; *FI Hunter*, which can find FI sequences violating the system properties and potential FI sequences that will be further analyzed; *Property Checker*, which examines the potential FI sequences generated from the FI hunter to check the property of the activated features and filters out the FI sequences violating the feature property; and *Translator* which translates the FI sequences generated from the FI Hunter and the Property Checker into the format of Message Sequence Charts (MSC) and compiles the final FI report. An evaluation of FIDS with respect to the Feature Interaction contest benchmark is given at the end of Chapter 5 in terms of detected FI type, FI number and test scenarios used. The discussion shows that FIDS can detect 7 more FI types and 51 more FIs than the benchmark by using 23 fewer testing scenarios. On the negative side, 23 benchmark FIs were not detected by FIDS and the reasons for this are also discussed in Chapter 5.

The methodology presented in this thesis does not give a general solution to the feature interaction problem but a partial solution limited to the detection of logical interactions at the specification level. Detecting feature interactions at the specification level contributes significantly to speed up the design phase and to the correctness of the design. We have shown that telecommunication system designers can give precise descriptions and validate their designs with respect to potential feature interaction problems before the implementation stage.

6.2 Future Work

The results of this thesis provide a basis for several future research directions. As new telecommunication features emerge, the need to provide a sound and flexible architecture becomes even greater. We believe that the model we present here for specifying telecommunications features and for the formalization of the notion of interactions provides a good starting point for defining such architecture. Still, there are many ways by which other contributions can improve and complement our model.

6.2.1 Goal-Oriented Exploration

As mentioned before, the trace-searching tool of FIDS, Caesar.Exhibitor, needs a fully pre-expanded specification to do the trace-searching. Because of the very large global state space generated, this greatly limits the size of the telephony network, the number of the end-users it can have and the number of features that can be introduced. Therefore, FIDS cannot detect those FIs that involve more than 3 users or complicated features such as CW, TWC, although theoretically they could be handled, see *§5.7 FIDS evaluation.* One solution to this problem could be using "on-the-fly" state exploration techniques [Pele96], which do not require saving the whole state space. Unfortunately, however, these techniques require more complicated algorithms.

Another solution to this problem could use Goal-Oriented Exploration methodology. Haj-Hussein et al. [HaLS93] define a new type of inference rules which are capable of generating traces of actions leading to pre-selected actions in the specification. Unlike Caesar.Exhibitor, which needs a full expansion for searching, the goal-oriented

exploration tool expands a small part of the behavior tree at a time. However, appropriate tools for this techniques are not available yet.

6.2.2 Enrichment of the system property set

As mentioned above, to establish the completeness and necessity of the derived property set is a big challenge of FI detection. No reference so far provides a systematic way for deriving the property set nor for proving its completeness and necessity.

Deriving system properties is even more difficult than deriving feature properties. Unlike feature properties, which express expectations of marketable services well known by both sellers and buyers, the system properties are an iceberg of various assumptions made about the network, where the underwater part is noticed only when violated. Work needs to be done in this area.

In our simplified telephony network model, only the basic signals, i.e. signals given to user and billing signals, are considered and investigated. However, in a real system, there are more advanced signals, i.e. signals used for routing and roaming, which need to be analyzed and added into the system property set.

Appendix. Comparison between FIDS Result and the Benchmark FI

CFBL Related FIs

Feature Pair	Scenairos		Feature Interactions		FI Types	
CFBL-CFBL	1	*	0	*	0	*
CFBL-CND	1	2	1	3	1	3
CFBL-INFB	2	2	2	5	2	3
CFBL-INFR	3	4	3	11	1	3
CFBL-INTL	0	0	0	0	0	0
CFBL-TCS	2	2	2	4	1	2
CFBL-TWC	2	*	2	*	2	*
CFBL-INCF	3	3	3	8	1	3
CFBL-CW	1	1	1	1	1	1
CFBL-INCC	1	1	1	1	1	1
CFBL-RC	0	1	0	1	0	1
CFBL-CELL	2	1	2	3	1	1
Total	18	17	17	37	11	18

CND Related FIs

Feature Pair	Scenairos		Feature Interactions		FI Types	
CND-CND	0	*	0	*	0	*
CND-INFB	1	1	1	1	1	1
CND-INFR	2	3	2	9	1	4
CND-INTL	0	0	0	0	0	0
CND-TCS	1	1	1	2	1	2
CND-TWC	1	1	1	1	1	1
CND-INCF	2	3	2	9	1	3
CND-CW	1	1	1	1	1	1
CND-INCC	1	1	1	1	1	1
CND-RC	1	1	1	1	1	1
CND-CELL	0	0	0	0	0	0
Total	10	12	10	25	8	14

INFB Related FIs

Feature Pair	Scenairos		Feature Interactions		FI Types	
INFB-INFB	0	*	0	*	0	*
INFB-INFR	2	4	2	12	2	5
INFB-INTL	0	0	0	0	0	0
INFB-TCS	1	2	1	4	1	4
INFB-TWC	2	1	2	1	1	1
INFB-INCF	2	3	2	10	2	4
INFB-CW	1	1	1	1	1	1
INFB-INCC	1	1	1	1	1	1
INFB-RC	1	1	1	1	1	1
INFB-CELL	2	1	2	2	1	1
Total	12	14	12	32	10	18

INFR Related FIs

Feature Pair	Scenairos		Feature Interactions		FI Types	
INFR-INFR	2	*	2	*	1	*
INFR-INTL	0	0	0	0	0	0
INFR-TCS	2	3	2	8	1	4
INFR-TWC	3	1	3	1	2	1
INFR-INCF	3	2	3	8	1	4
INFR-CW	1	1	1	1	1	1
INFR-INCC	2	1	2	1	1	1
INFR-RC	1	2	1	2	1	1
INFR-CELL	2	1	2	3	1	1
Total	16	11	16	24	9	13

INTL Related FIs

Feature Pair	Scenairos		Feature Interactions		FI Types	
INTL-INTL	0	*	0	*	0	*
INTL-TCS	0	0	0	0	0	0
INTL-TWC	1	1	1	1	1	1
INTL-INCF	0	0	0	0	0	0
INTL-CW	0	0	0	0	0	0
INTL-INCC	1	0	1	0	1	0
INTL-RC	1	2	1	2	1	1
INTL-CELL	0	0	0	0	0	0
Total	3	3	3	3	3	2

TCS Related FIs

Feature Pair	Scenairos		Feature Interactions		FI Types	
TCS-TCS	0	*	0	*	0	*
TCS-TWC	1	1	1	1	1	1
TCS-INCF	2	2	2	5	1	4
TCS-CW	1	1	1	1	1	1
TCS-INCC	2	1	2	1	1	1
TCS-RC	1	2	1	2	1	1
TCS-CELL	0	0	0	0	0	0
Total	7	7	7	10	5	8

TWC Related FIs

Feature Pair	Scenairos		Feature Interactions		FI Types	
TWC-TWC	1	*	1	*	1	*
TWC-INCF	3	1	3	1	2	1
TWC-CW	9	*	9	*	3	*
TWC-INCC	4	0	4	0	1	0
TWC-RC	1	1	1	1	1	1
TWC-CELL	4	1	4	3	2	2
Total	22	3	22	5	10	4

INCF Related FIs

Feature Pair	Scenairos		Feature Interactions		FI Types	
INCF-INCF	1	*	1	*	1	*
INCF-CW	1	1	1	1	1	1
INCF-INCC	2	2	2	2	1	1
INCF-RC	1	2	1	2	1	1
INCF-CELL	2	1	2	4	1	2
Total	7	6	7	9	5	5

CW Related FIs

Feature Pair	Scenairos		Feature Interactions		FI Types	
CW-CW	0	*	0	*	0	*
CW-INCC	1	0	1	0	1	0
CW-RC	1	1	1	1	1	1
CW-CELL	2	1	2	2	1	1
Total	4	2	4	3	3	2

INCC Related FIs

Feature Pair	Scenairos		Feature Interactions		FI Types	
INCC-INCC	0	*	0	*	0	*
INCC-RC	0	0	0	0	0	0
INCC-CELL	0	1	0	2	0	1
Total	0	1	0	2	0	1

RC Related FIs

Feature Pair	Scenairos		Feature Interactions		FI Types	
RC-RC	0	*	0	*	0	*
RC-CELL	0	0	0	0	0	0
Total	0	0	0	0	0	0

CELL Related FIs

Feature Pair	Scenairos		Feature Interactions		FI Types	
CELL-CELL	0	*	0	*	0	*
Total	0	0	0	0	0	0

Bibliography

[1st FITS.92] 1st International Workshop on Feature Interactions in Telecommunication Software Systems, Florida, December, 1992.

[2nd FITS.94] 2nd International Workshop on Feature Interactions in Telecommunications Software Systems, Netherlands, May, 1995.

[3rd FITS.95] 3rd International Workshop on Feature Interactions in Telecom. Software Systems, Japan, 1995.

[4th FITS.97] 4th International Workshop on Feature Interactions in Telecom. Software Systems, Montreal, June, 1997.

[5th FITS.97] 5th International Workshop on Feature Interactions in Telecom. Software Systems, Sweden, September, 1998.

[BAEQ98] R.J.A. Buhr, D.Amyot, M. Elammari, D. Quesnel, et al., Feature-Interaction Visualization and Resolution in an Agent Environment, *Feature Interactions in Telecommunications and Software Systems V,* eds. K. Kimbler and L.G. Bouma, IOS Press 1998.

[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

[BeHo89] F. Belina and D. Hogrefe, The CCITT Specification and Description Language SDL, *Computer Networks and ISDN Systems*, Vol.16, 1989

[Bern95] K. Bernhard. *Digital Telephony and Network Integration*. Van Nostrand Reinhold, 1995

[BGGO99] R. Blumenthal, N. Griffeth, J. C. Gregoire and T. Ohta, *Feature Interaction Contest Interaction Descriptions,* 1999

[BoZu92] W. Bouma and H. Zuidweg, *Formal Analysis of Feature Interactions by Model Checking.* PTT research, the Netherlands, December, 1992

[BoLo93] R. Boumezbeur and L. Logrippo, Specifying Telephone Systems in LOTOS. *IEEE Communications Magazine,* August, 1993

[BORZ98] L. du Bousquet, F. Ouabdesselam, J.-L. Richier and N. Zuanon, Incremental Feature Validation: a Synchronous Point of View. *Feature Interactions in Telecommunications and Software Systems V,* eds. K. Kimbler and L.G. Bouma, IOS Press 1998.

[CaVe93] J. Cameron and H. Velthuijsen, Feature Interactions in Telecommunications Systems. *IEEE Communications Magazine,* August, 1993

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

[CGLN93] E.J. Cameron, N. Griffeth, Y.J. Linand, M.E. Nilson, W.K. Schnure, and H. Velthuijsen. A feature-interaction benchmark for IN and beyond. *IEEE Communications Magazine* 31, 1993.

[Chen94] K.E. Cheng, Towards a Formal Model for Incremental Service Specification and Interaction Management Support. *Feature Interactions in Telecommunications Systems*. eds. L.G. Bouma and H. Velthuijsen, IOS Press 1994.

[CoPi94] P. Combes and S. Pickin, Formalization of a User View of Network and Services for Feature Interaction Detection. *Feature Interactions in Telecommunications Systems.* eds. L.G. Bouma and H. Velthuijsen, IOS Press 1994.

[FaLS90] M. Faci, L. Logrippo and B. Stepien, Formal Specifications of Telephone Systems in LOTOS. *Protocol Specification, Testing, and Verification IX*, eds. E. Brinksma, G. Scolo, and C. Vissers, 1990

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

[FaLS97] M. Faci, L. Logrippo and B. Stepien, Structural Models for Specifying Telephone Systems. *Computer Networks and ISDN Systems*, 1997

[Fern96] Fernandz. J.-C. et al., CADP (CAESAT/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. *Proceedings of the 8th Conference on Computer-Aided Verification Vol. 1102 of Lecture Notes in Computer Science*, eds. R. Alur and T. Henzinger, 1996

[FHLS98] Q. Fu, P. Harnois, L. Logrippo, J. Sincennes, *1998 Feature Interaction Contest*, Technical Report 98-08, TSERG, University of Ottawa, 1998

[FHLS99] Q. Fu, P. Harnois, L. Logrippo, J. Sincennes, Feature Interaction Detection: a LOTOS-based approach, to appear in *Computer Networks*, 1999

[Grin97] A. Grinberg, *Seamless Networks*, McGraw-Hill, 1997

[GTGB98] N. Griffeth, O. Tadashi, J.-C. Gregoire and R. Blumenthal, First Feature Interaction Detection Contest. *Feature Interactions in Telecommunications and Software Systems V,* eds. K. Kimbler and L.G. Bouma, IOS Press 1998.

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

[IrEr97] Y. Iraqi and M. Erradi, An Experiment for the Processing of Feature Interactions within an Object-Oriented Environment. *Feature Interactions in Telecommunication Networks IV,* eds P. Dini, R. Boutaba and L. Logrippo, IOS Press 1997.

[ISO8807] ISO, IS8807. Informal Processing Systems – Open Systems Interconnection – LOTOS: A Formal Description Technique Based on the Temporal Ordering of Observational Behavior, May 1989.

[ITU-T96] International Organization for Standardization – Telecommunication Standardization Sector (ITU-T) Series Z: Programming Languages, Message Sequence Charts , 1996

[KaLo98] J. Kamoun, L. Logrippo, Goal-Oriented Feature Interaction Detection in the Intelligent Network Model. *Feature Interactions in Telecommunications and Software Systems V,* eds. K. Kimbler and L.G. Bouma, IOS Press 1998.

[Kawa71] H. Kawashima et al. Functional Specification of Call Processing by State Transition Diagram. *IEEE Trans. on Comm.* 19, 5, October 1971

[Kell94] B.Kelly et al., Feature Interaction Detection Using SDL Models, *Proceedings of IEEE GlobeCom*, 1994

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

[Miln89] R. Milner, *Communication and Concurrency*, Prentice Hall, 1989

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

[NaKK97] M. Nakamura, Y. Kakuda and T. Kikuno, Petri-Net Based Detection Method for Non-Deterministic Feature Interactions and its Experimental Evaluation. *Feature Interactions in Telecommunication Networks IV*, eds P. Dini, R. Boutaba and L. Logrippo, IOS Press 1997.

[NBGO98] N. Griffeth, R. Blumenthal, J.-C. Gregoire and T. Ohta, Feature Interaction Detection Contest. *Feature Interactions in Telecommunications and Software Systems V,* eds. K. Kimbler and L.G. Bouma, IOS Press 1998.

[Pele96] D. Peled, Combining partial order reductions with on-the-fly model checking, *Formal Methods in System Design 8*, 1996

[Pete77] J. Peterson, Petri-Nets, *ACM Computing Surveys*, September, 1977

[SteL95] B. Stepien and L. Logrippo. Representing and Verifying Intentions in Telephony Features Using Abstract Data Types. *Feature Interactions in Telecommunications III*, eds. K. Cheng and T. Ohta, IOS Press 1995

[StLo93] B. Stepien and L. Logrippo. Status-Oriented Telephone Service Specification. Theories and Experiences for Real-Time System Development. *AMAST Series in Computing*, Vol.2 World Scientific, 1994

[StLo95] B. Stepien and L. Logrippo, Feature Interaction Detection Using Backword Reasoning with LOTOS. *Protocol Specification, Testing and Verification XIV*, ed. S. Vuong, 1995

[Thom97] M. Thomas, Modeling and Analyzing User Views of Telecommunications Services. *Feature Interactions in Telecommunication Networks IV*, eds P. Dini, R. Boutaba and L. Logrippo, IOS Press 1997.

[Thor94] J. Thorner. *Intelligent Network*. Artech House, 1994

[Turn93] K. J. Turner (Ed.), *Using Formal Description Techniques*, Wiley 1993

[Viss95] J. Visser. *Tutorial on Intelligent Network Basics*. IEEE IN'95 Workshop, Ottawa, May 1995

[WhCh81] V. Whitis and W. Chiang, A State Machine Development Method for Call Processing Software. In *Proceedings of the IEEE Electro 81*, IEEE Press, Washington D.C., April 1981

[YaBa79] M. Yoeli and Z. Barzilai, Behavioral Descriptions of Communications Switching System Using Extended Petri-nets, *Digital Processes 3, 4* 1997

[Zave93] P. Zave, Feature Interactions and Formal Specifications in Telecommunications, *IEEE Computer*, August, 1993

List of Acronyms

A

ADT	Abstract Data Type, §4.1.1

B

BCP	Basic Call Process, §3.3

C

CC	Charge Call, §3.1
CCITT	Committee Consulative Interactional de Telephonie et Telegraphie, §2.1.3
CCSS	Common Channel Signaling System, §1.1
CELL	Cellular, §3.1
CFBL	Call Forward on Busy Line, §3.6.3
CFD	Computational Fluid Dynamics, §2
CND	Call Number Delivery, §3.1
CW	Call Waiting, §3.1

D

DBAPI	Database Application Interface, §3.1

F

FAP	Feature Activation Process, §3.4
FDTs	Formal Description Techniques, §1.2
FI	Feature Interaction, §5.2
FIDS	Feature Interaction Detection System, §5.5
FPP	Feature Property, §5.2
FSM	Finite State Machine, §2.1.1

I

IMAG	Informatique et de Mathematiques Appliquees de Grenoble, §1.3

IN	Intelligent Network, §1.1
INCF	IN Call Forwarding, §3.1
INFB	IN Free Billing, §3.6.2
INFR	IN Free Routing, §3.1
INTL	IN Teen Line, §3.6.1
ISO	International Organization for Standardization, §4.1
ITU	International Telecommunication Union, §2.1.3

L

LOTOS	Language Of Temporal Ordering Specification, §4.1
LTS	Label Transition System, §3.2

M

MSC	Message Sequence Charts, §5.6.7

O

OSI	Opening System Interconnection, §4.1

P

PIC	Points In Call, §3.3
POI	Point of Initialization, §3.4
POR	Point of Return, §3.4
POTS	Plain Old Telephony System, §1.1

R

RC	Return Call, §3.1

S

SCP	Service Control Point, §3.5.2
SDP	Service Data Point, §3.5.2
SDL	Specification and Description Language, §2.1.3

SP	System Property, §5.2

T

TCS	Terminating Call Screening, §3.1
TWC	Three Way Calling, §3.6.4

V

VCC	Virtual Central Control, §2.1.2
VDC	Virtual Dial Control, §2.1.2
VSC	Virtual Station Control, §2.1.2
VSS	Virtual Station Subsystem, §2.1.2

W

WIN	Wireless Intelligent Network, §1.2