Feature Interaction Filtering and Detection with Use Case Maps and LOTOS

Jameleddine Hassine

Thesis submitted to
the school of graduate studies and research
in partial fulfillment of
the requirements for the degree of

Master of Computer Science

Under the auspices of the Ottawa-Carleton
Institute of Computer Science

UNIVERSITE D'OTTAWA
UNIVERSITY OF OTTAWA

CARLETON UNIVERSITY

University of Ottawa,
Ottawa, Ontario, Canada
February 2001

Acknowledgment

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

First, I would like to express my deepest appreciation to my supervisor, Prof. Luigi Logrippo, for his encouragement, guidance and financial assistance through my graduate studies. His accuracy in reviewing the drafts of the thesis has greatly improved the contents and its presentation.

Second, I also owe special thanks to Dr. Masahide Nakamure for the inspiring and fruitful discussions we have had together.

I would also like to thank the members of University of Ottawa LOTOS group for all their advice and assistance. In particular, I would like to thank Daniel Amyot, Jacques Sincennes and Rossana M. Castro Andrade for their useful comments about some parts of the thesis.

Finally, I would like to express my gratitude for the financial support from the following sources: The Tunisian Government under the National scholarship program, Communications and Information Technology Ontario (CITO), Natural Sciences and Engineering Research Council of Canada (NSERC) and Nortel Networks.

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. However, these facilities are confronted with 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 describing telephony features at the requirements stage. This model is built using the Use Case Maps Notation (UCM). Based on this model, we propose a method to filter feature interactions at the requirements stage. This preliminary evaluation allows the detection process to focus on feature combinations where interactions are possible and therefore reduces the cost of the detection process.

In the second part of the thesis, a Feature Interaction Detection System is developed for detecting feature interactions between switch based and IN features. This method aims to detect interactions occurring at the abstract specification level and resulting in violation of feature properties. This technique in based on the Formal Description technique LOTOS and uses Abstract Data Types to detect those violations. Our method detects feature interaction by executing the system specification. The designer can reach those interaction points either by a step by step execution or using the goal oriented execution technique.

It is concluded that UCM and LOTOS are useful in specifying the telephony system with features and for detecting feature interactions at the abstract specification level.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction: Motivation and Background

1.1 Introduction

A hallmark of the introduction of digital technologies into the telephone network is the extension of the basic call service through switch-based software. Instead of only being able to carry out a basic conversation, where one party calls another and carries on a conversation until one party hangs up, a variety of different behaviours can now be supported during the course of a call.

Instead of only allowing a conversation between two parties, new behaviours such as forwarding calls, placing callers on hold, and blocking calls are realized through the use of telephony *features*. A single feature is defined as an extension/a modification of the basic service [10]. These features operate according to protocol rules defined between the users of the system and the network.

Due to market demands and high competition, the rapid development and deployment of features has become an important goal for telecom companies.

Different features of a system may be designed by different designers at different times. Because features are designed independently and validated in isolation, it is possible that their behaviour changes in certain feature combinations. When this occurs, it is said that there is “Feature Interaction” [67]. Extensive system testing is used during development to help ensure that features will function together properly when combined.

Feature interactions impact all phases of the software lifecycle. Timely mechanisms for resolving untoward feature interactions at all stages of the software lifecycle must be part of any new service development process.

Over the past several years, a large number of techniques to avoid, detect and solve feature interactions have been proposed in order to reduce the need for testing and therefore the time it takes to get new features to market [48][50][23]. These techniques act at all stages of the feature life cycle going from the requirement stage to the implementation stage.

1.2 The Feature Interaction problem

It is difficult to give a precise and complete definition of the “Feature Interaction” because of the huge variety of problems that could be classified under this heading. Meanwhile many formulations of the problem have been proposed and much research has been done in this area [63] [61] [53].

As a non-formal definition, we use the definition of [53] stating that there is “feature interaction”:

1. When a feature inhibits or subverts the expected behaviour of another feature (or another instance of the same feature).
2. When the joint execution of two features provokes a supplementary phenomenon that cannot occur during the processing of each of the features considered separately.

Note: This definition is not formal enough. We will give a more formal definition in Chapter 6.

Zygan-Maus [63] distinguishes two levels of challenges:

1. The service level challenges: Such challenges are independent of how the involved services are implemented. This means that they will persist independent of whether services are implemented in a switch, on an IN platform or in TINA (Telecommunication Information Networking Architecture) [49]. Service level Interactions are purely logical interactions.

2. The technical level challenges: Such challenges are dependent on implementation. When a new feature is being designed, feature functionality has to be mapped on network architecture and linked to the basic call processing. The feature implementation has to provide the functionality required of each particular network element in support of the new feature and has to minimize interaction prone impacts on existing feature implementations. At this level, feature designers should take into consideration the real network constraints such as network signalling restrictions, charging restrictions, network timing conflict, concurrent resources usage attempts…etc.

In this thesis we focus on the service level interactions. We wish to detect interactions when features are specified (at the design and integration level) before the implementation.

1.2.1 Example of feature interaction

Figure 1 illustrates an instance of such problem between two common features, namely Originating Call Screening (OCS) and Call Forwarding Busy Line (CFBL). OCS forbids the establishment of a call to phone numbers on a screening list, while CF forwards incoming calls to another phone number. The two features interact inappropriately when A, whose OCS screening list includes C, calls B. Since B, who subscribes to CFBL to C, is busy, the incoming call is forwarded to C.

However, C was on A's screening list, and therefore the connection should not have happened, as it violates assumptions related to OCS. So CFBL has inhibited the expected behaviour of OCS.

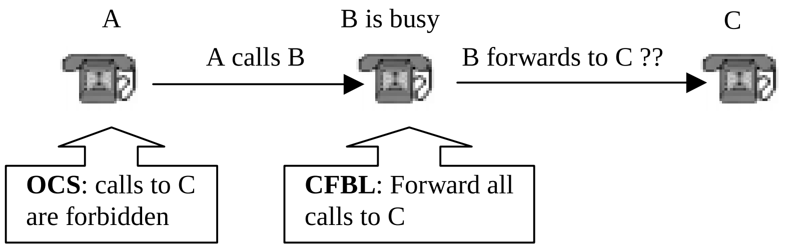

A
B is busy
C
A calls B
B forwards to C ??
OCS: calls to C are forbidden
CFBL: Forward all calls to C

Figure 1: Illustration of a telephony feature Interaction

Note: Feature interaction is necessary and inevitable in a feature-oriented specification, because so little can be accomplished by features that are completely independent. In fact, some interactions are expected by designers. However, which are and which are not is a matter of designer's judgment and outside of the scope of this thesis.

1.2.2 Addressing feature Interaction

The feature interaction problem can be addressed according to three approaches: Avoidance, Detection and Resolution [43].

- Avoidance:

  The objective of an avoidance approach is to define additional guidelines (constraints, service platform and service environments) to prevent the manifestation of unwanted interactions. This assumes that the causes of the interactions are known, which is not always the case.

  This approach can be adopted starting from the early phases of specification and design of features. An example of use of avoidance approach is the Wireless Intelligent Network (WIN) system, where the feature interaction problem is solved, at least in part, by giving pre-defined priorities to different features [10].

- Detection:

  Approaches for detection aim to determine whether or not a set of independently specified features can cause conflicts when they are composed. The detection analysis can be applied through the whole lifecycle of a feature [23][48][50], since the cause of interaction can be related to any phase of the feature lifecycle.

- Resolution:

  The objective of a resolution approach is to find solutions to interactions when they occur [11][50][60][68]. But before trying to solve the interactions that may occur between features, an accurate analysis of the undesired behaviour should be performed. Proposed techniques solve the interaction by:

  - Replacing the undesired behaviour by a reasonable one [11].

    Example: Consider the example of detecting the interactions between features CW (Call Waiting) and 3WC (Three Way Calling). We suppose that users A and B are in a phone conversation.

    Feature CW: If C calls A, the latter is informed by a CW-tone. A may generate a flash-hook signal to put B on hold and to be connected to C. A may switch between B and C with flash-hook signal.

    Feature 3WC: A can flash the hook to put B on hold, and then A can call C. While A and C are in a phone conversation and B is on hold A can flash the hook a second time to add B in the conversation. After that, if C hangs up or if A flashes the hook, then A and B are in a normal conversation.

    Interaction: Let us assume that C calls A and then the latter flashes the hook just before being informed by the CW-tone. The flash-hook signal must be interpreted by CW or by 3WC? In both cases, B is put on hold but A is connected to C only in the first case. We obtain undesirable state in the joint behaviour of CW and 3WC. This undesirable state consist on a non-deterministic state from which event “flash A” may lead to different states.

    Resolution: By removing this undesirable state and generating a behaviour corresponding to a mutual exclusion between these two features the interaction is resolved.

  - Build a negotiation protocol between the features involved in the interaction. This protocol consists of an exchange of necessary information to avoid the interaction [50][60][68].

Detection/Resolution during the service creation process are known as off-line detection/resolution while detection/resolution at run time are known as on-line detection/resolution.

In this thesis, the method proposed in Chapter 6 to detect feature interaction is an off-line process.

1.3 Contribution of the Thesis

The major contributions of this thesis is the development of a model for describing telephony features and of a method for detecting feature interactions at two stages: the requirements stage and the specification stage.

1.3.1 Contribution 1: Feature Interaction Filtering at the Requirements Stage

In Chapter 4, we present a model for describing Telephony features at the requirements stage. This model is built using the Use Case Maps Notation and allows us to describe many features. Based on this model, we propose a method to filter feature interactions at the requirements stage. This method offers a quick and rough evaluation of possible feature interactions before the Feature Interaction Detection Process (Proposed in Chapter 6). This preliminary evaluation allows the detection process to focus on feature combinations where interactions are possible and therefore reduces the cost of the detection process. Concrete examples are given in Chapter 4.

1.3.2 Contribution 2: Feature Interaction Detection Method

In Chapter 6, a Feature Interaction Detection System is developed for detecting feature interactions. This method aims to detect interactions occurring at the abstract specification level and resulting in violation of system integrity. This technique in based on the Formal Description technique LOTOS and uses Abstract Data Types to detect those violations. Combining features manually in the basic system could avoid some existing interactions and create new virtual interactions. Because of this, we chose to describe Features independently in our Feature Interaction Detection System. The Feature Interaction Detection System consists of three parts:

two feature specifications and a process called Feature Interaction Detector (FI Detector) for detecting the interactions.

Our method detects feature interaction by executing the feature Interaction detection system specification. The designer can reach those interaction points either by a step by step execution or using the goal oriented execution technique.

1.4 Organization of the Thesis

The six remaining chapters will cover the following issues:

Chapter 2: **Related work: Feature Interaction Detection Requirement and Specification stages**

We present a survey of related work on Feature Interaction Detection at both requirement and specification stages. We also introduce the techniques used to do such Feature Interaction Detection.

Chapter 3: Describing Requirements with Use Case Maps

In this chapter we present some existing requirement description techniques and we focus on the Use Case Maps notation that we are going to use in our work.

Chapter 4: Feature Interaction Filtering at Requirement Stage

First we introduce a model for describing Telephony features at the requirements stage. Based on this model, we propose a method to filter feature interactions at the same stage. Finally we present the results of the filtering on switch and IN based features.

Chapter 5: Specifying Features using LOTOS

In this chapter, we give an overview of the LOTOS specification language and its main operators. Our main objective in specifying the system model and features in LOTOS is to provide a specification that can be used for validating and detecting feature interactions.

Chapter 6: Feature Interaction Detection Method

We describe an improved formal definition of Feature Interaction and a Feature Interaction Detection System. Our technique aims to detect violation of system properties at the design stage, based on the Formal description Techniques (FDT) LOTOS and uses Abstract Data Types (ADT) to detect these violations.

Chapter 7: Conclusion and Future Work

Conclusion and future work are presented in this chapter.

Chapter 2

Related work:
Feature Interaction Detection at Requirement and Specification Stages

We present a survey of related work on Feature Interaction Detection on telephony systems at both requirement and specification stages.

Feature Interaction is a research area of some importance, and a number of papers are published every year on the subject. Six International Workshops have been held so far [1][2][3][4][5][6]. In this survey, we present only work that is in some way related to our approach.

2.1 Addressing Feature Interactions in the Requirements Stage

Requirement description plays an important role in the development of telecommunication systems. Early conflict detection can help prevent costly and time-consuming problem fixes during implementation [14][60].

- A. Gammelgaard and J. E. Kristensen [10] propose to let feature specifications be restrictions to the class of deterministic labelled transition system. A features specification consists of two parts:
  1. Network properties: they are formulas expressing static constraints. All states in a system must satisfy all given network properties.
  2. Declarative transition rules: Dynamic constraints are formalized by declarative transition rules. Such rules consist of a precondition, a trigger event, and a post-condition. If a state satisfies the precondition, then this state must be origin of a transition labelled by the trigger event, and the resulting state of the transition must satisfy the postcondition.

Network properties and the pre- and postconditions of declarative transition rules are formulated in a simple logic, which is a restriction of ordinary first order logic. A new feature can be added by replacing some rules of the core service by new rules and introducing new formulas.

Executing those rules by matching the postconditions with the preconditions allows finding inconsistencies. An interaction is detected when the system arrives to a state that doesn't satisfy the defined network properties.

By adding new features to the system, new rules and new predicates are defined. The matching process of postconditions and preconditions fails if there are missing rules. In our experience, we found that the detection process becomes very difficult if the set of rules describing the features is not complete.

- In [60] Buhr and al use the UCM (Use Case Maps) notation (see Section 3.2.2 for detailed introduction to UCM) to describe Telephony features. The proposed method generates tables from UCM behaviors and provides a framework for humans to add information that will enable executable prototypes to be generated. Features are modeled as competing rule engines and interactions are detected and resolved at run time by coordinating through a blackboard.

- Heisel and Souquieres [48] propose a method of *requirements elicitation* in order to detect feature interactions. The proposed approach is inspired by object oriented methods and gives guidance on how to identify, express, and systematically transform requirements into a formal specification. The proposed method uses *agendas* (a list of steps to be performed when carrying out some tasks in the context of software engineering) and first order logic to express and incorporate constraints into the system requirements. Two constraints are said *"Interaction Candidates for one another"* if they have common preconditions but incompatible postconditions. After expressing those constraints, an analysis of preconditions and postconditions determine which candidates could lead to an interaction. The incompatibility of postconditions takes place either in the state immediately following the state that is referred to by the postcondition or in a later state. Although this approach uses a simple logic like the one used by the previous approach, the interactions are expressed differently.

- Jonsson et al [17] propose a technique for hierarchically structuring requirements. Requirements are described as a structured hierarchy of predicates, capturing system properties, and shielded from concrete details of system implementation by one or several levels of abstract logical concepts. The set of abstract concepts, which represents a collection of abstract predicates, can be seen as forming a *vocabulary* of the application domain. This method uses a simple linear-time temporal logic using the defined predicates. Each requirement can be represented as an observer. These observers are added to the system, which is then subject to exhaustive state-space exploration. The exploration will detect when a requirement is violated, and as a side effect shows the sequence of events leading to the violation. Our method (Described in Chapter 4) employs Use Case Maps notation to describe requirements. UCM's elements are used to structure these requirements at different levels of abstraction. Then a filtering procedure is conducted to detect interaction.

- In a paper co-authored by the author of this thesis and Nakamura et al [50] propose a feature interaction filtering method at the requirement level. The method extensively utilizes the requirement notation method Use Case Maps (UCMs), which helps designers to visualize a global picture of call scenarios. In this framework, the addition of a feature is achieved by using the stub plug-in concept of UCMs. That is, a set of sub UCMs describing the feature's functionality are plugged into stubs of the basic call scenario in a *"root"* UCM. Thus, each feature is characterized by the stub configuration. The method proposes a pair wise composition of the features and gives one of the following verdicts: (a) FI occurs, (b) FI never occurs, (c) FI-prone.

In our work we use a similar model to describe features and to conduct the filtering. The relationship between our method and the method of [50] will be discussed in section 4.7.

- Aho et al [11] describe a language called Chisel for defining requirements. The authors claim that this language is unambiguous, applies to a variety of network technologies, and it has a sound basis for translation to commonly used formal software specification languages. Telephony features are then described in the Chisel language using the editing tool SCF3/Sculptor. Finally feature interaction detection is conducted based on an analysis of sequences of events for each feature. In our work we use features described in the Chisel notation to derive feature's Use Case Maps.

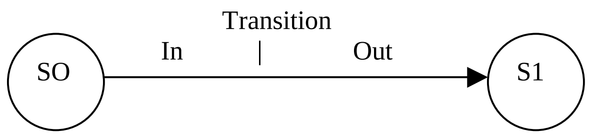
2.2 Formal Specification Methodologies for Telephony Systems

A formal description is a symbolic representation of a certain object in a given language. The language may use various kinds of symbols such as textual or graphical symbols. The description language uses strict rules for the construction of language expressions, the "formal syntax", and strict rules for the interpretation of well-formed language expressions, the "formal semantics". The main purpose of a formal description is to have unambiguous, clear and precise specification [64].

Various techniques have been developed for specifying telephony systems in a formal way. The main ones are: Finite State Machine Model (FSM), Petri Nets, LOTOS (Process Algebra) and SDL (Extended Finite State Machine EFSM).

2.2.1 Finite State Machines

Probably the earliest formal methods have been the ones based on the use of Finite State Machines “FSM” [30]. FSM is a transition model, where the behaviour of a given system is represented in terms of states and transitions. Each FSM is normally represented by a directed graph as outlined in figure 2 where a directed path represents an occurrence of an event which changes the state of the represented machine. The machine is in state S0 and when the event "In" happens, it changes state to S1 and the event "Out" happens.



Where S0 and S1 are states; "In" is an incoming event; and "Out" is an outgoing event

Figure 2: Finite State Machine

A state describes the current situation of the system, resulting from previous transitions, and at the same time it describes which transitions are possible in the future of the system. For example, in a telephony system, a user could be in *Dialing* state, in *Ringing* state, or in *Talking* state.

Telephony features described using FSM can be found in the Feature Interaction Contest 2000 [6][37].

The main shortcomings of the FSM model are:

1. The lack of a data model.
2. The lack of an architectural model
3. The lack of explicit representation of concurrency

In order to address these shortcomings, other models were defined, as described below.

2.2.2 Petri Nets

Petri Nets is a formal and graphically appealing language that is appropriate for modeling systems with concurrency. The Petri Net notation has been under development since the beginning of the sixties, where Carl Adam Petri defined the language [64]. It was the first time that a general theory for discrete parallel systems was formulated. The language is a generalization of automata theory, making it possible to express the concept of concurrently occurring events.

Petri Nets are abstract machines that are used to describe the behaviour of systems. They are represented by directed graphs 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. An example of a simple Petri Net is shown in figure 3.

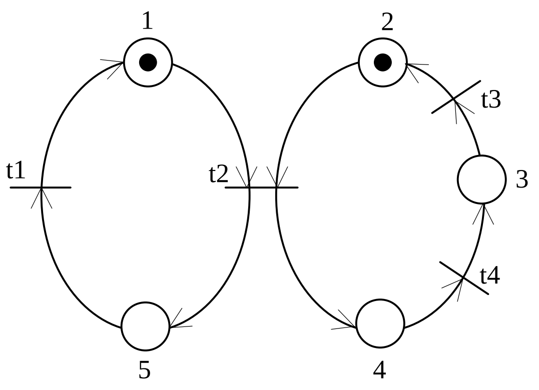


Figure 3: A simple Petri Net

When all the input places to a transition contain at least one token, the transition is enabled and may fire. When the transition fires, one token is removed from each input place and one token is added to each output place.

Yoeli and Barzilai [51] introduce the concept of extended Petri Nets (EPN) and use it to model the call processing operations in an automatic telephone exchange.

Two common problems with the FSM and Petri Nets are [38]: 1) the limited role assigned to data. Many features rely on data values and data structures for essential aspects of their functionalities, and so they are difficult to represent by Petri Nets. 2) The lack of process structure, which is very useful for design. Extended Finite State Machine (EFSM) methods such as SDL, remedy this situation.

2.2.3 LOTOS

We give an overview of the LOTOS specification language and its main operators in Chapter 5.

[44] Describes the Plain Old Telephone System (POTS) using LOTOS. Four different structural approaches could be adopted for specifying a telephone system:

- The resource-oriented style: In the resource-oriented style, the specification structure shows the architectural components of the design [45]. In [35], a formal specification of an IN network model was developed using the resource-oriented style.
- The state-oriented style: In the state oriented style there is an explicit reference to system states.

The resource-oriented style and the state-oriented style are implementation-oriented and suggest an implementation architecture.

- The constraint-oriented style: In the constraint-oriented style, one focuses on the composition of the requirements, expressed as behaviours [45].
- The monolithic style: In the Monolithic style, the specification is described as a tree of alternatives, i.e. expanded execution sequences are explicitly enumerated

The constraint-oriented style and the monolithic style are requirement-oriented.

The work presented in [21] and in [38] describes a new approach for specifying telephone systems using a mixture of the constraint-oriented style and the state-oriented style.

2.2.4 SDL

The Specification and Description Language (SDL) is an object-oriented, formal language defined by The International Telecommunications Union–Telecommunications Standardization Sector (ITU–T) (formerly Comité Consultatif International Telegraphique et Telephonique [CCITT]) as recommendation Z.100 [57]. The language is intended for the specification of complex, event-driven, real-time, and interactive applications involving many concurrent activities that communicate using discrete signals.

Just as the other formal languages, SDL covers different levels of abstraction, from a broad overview down to detailed design.

The basic theoretical model of an SDL system consists of a set of extended finite state machines (EFSMs) that run concurrently. These machines are independent of each other and communicate by means of discrete signals. SDL does not use any global data. It has two basic communication mechanisms: asynchronous signals (and optional signal parameters) and synchronous remote procedure calls. Both mechanisms can carry parameters to interchange and synchronize information between SDL processes and their environment.

Examples of specifying telephone system using SDL are presented in [53][34].

2.3 Addressing Feature Interaction at Specification Level using LOTOS

In this section we limit ourselves to reviewing work closely related to ours. We give a brief overview of the Feature Interaction detection methods using LOTOS.

- Boumezbeur and Logrippo [56] applied *the step by step execution* on the LOTOS specification of a telephone system. At each step of the step by step execution, the user chooses the next step to be taken among all possible actions that are offered at that point. This is useful for checking the conformance of a system defined informally to its formal description in LOTOS. In practice, this can be done by checking if test sequences that should be allowed according to the informal definition are also accepted by the formal specification; or checking if the test sequences obtained by executing the specification are included in the formal definition of the system; or by checking if test sequences that are not specified informally are not accepted by the formal specification. It is, however, a slow method and nowadays it is not used in practice.

- Stepien and Logrippo [19] proposed a method called "Backward Reasoning" to explore all the potential alternatives leading to feature interaction. The method involves specification of telephony 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 behaviour 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 behaviour of the system being specified. To prove that an action is ambiguous, backward reasoning for LOTOS is applied. It consists of a combination of forward and backward execution. Forward execution of the specification is applied to reach the action, then, using the resulting behaviour expression, backward execution is performed to find a different trace leading to the same action.

- In [20] a method for representing and verifying intentions in telephony features using abstract data types is presented. Feature intentions describe the intended behaviour of telephony features. The first step of the method is to specify a feature’s intentions using abstract data types. An intention is represented as an operation of Boolean result indicating whether a given combination of the basic data involved in an operation is allowed:

  Intention: Fid, partyRole, operation, Restriction_set -> Bool

  For example the origination call screening (OCS) prohibits any connection with a number that is in the screening list. This feature intention can be formulated as:

  Intention (Focs, called (N), connect, L) = N NotIn L;

  Where **Focs** identifies the feature originating call screening, **N** the phone number involved in a called role in operation **connect** and **L** is the restriction set that in this case is a screening list, **N NotIn L** is a Boolean expression that verifies whether the number N is in the screening list or not.

  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 expressed as Abstract Data Types operations which specify the intention's violation. The specification language considered is LOTOS. The second step consists in executing the 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.

Our FI Detection method presented in Chapter 6 is inspired by this idea of representing intentions in Telephony Features using Abstract Data Types.

- In [45], Faci and Logrippo developed a methodology for detecting feature interactions using LOTOS testing 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 features, such that each feature is able to execute all of its actions that are allowed in the context of POTS. Then, they reason about interactions in terms of the conformance relation studied in LOTOS testing theory, in the following way: an interaction exists between features if their integration does not conform to their composition.

- In [35], Kamoun and Logrippo developed a method for detecting feature interactions between IN services using the Goal Oriented method (Goal Oriented method is described in Chapter 6). The method detects interactions resulting in violation of features properties. It is based on formalization of feature's properties, derivation of goals satisfying the negation of these properties, and use of Goal Oriented Execution to detect traces satisfying these goals. A trace satisfying a goal shows that an interaction exists between the specified features by describing a scenario violating one of the properties of the introduced features.

Our Method uses also the Goal Oriented Execution method. We simplified the goal to be just a ‘‘VR” event (VR: Violation Report). We also use a global observer process called Feature Interaction Detector (FIDetector) to capture interactions regarding the following four issues: Connections, Billing, Signals and Display. This will be described in Chapter 6.

Chapter 3

Describing Requirements With Use Case Maps

In this chapter we present the different existing requirement description techniques and we focus on the Use Case Maps notation that we are going to use in our work.

3.1 Introduction

Emerging telecommunications services and features require standardization bodies (ANSI, ETSI, ISO, ITU, TIA, IETF, etc.) to describe and design increasingly complex functionalities, architectures, and protocols [24]. In the early stages of the design process, many features, services, and functionalities are described using informal operational descriptions, tables and visual notations such as Message Sequence Charts (MSCs) [33]. As these descriptions evolve, they quickly become error prone and difficult to manage. The need of precisely documenting all stages of the design process, which is very important in the industrial environment, becomes critical in the standardization process [25].

Following the practice in several standard groups, the development of each phase of a telecommunication standard is divided in three stages [25] (shown in figure 4): 1) service descriptions, 2) message sequence information, and 3) protocol and procedure specification.

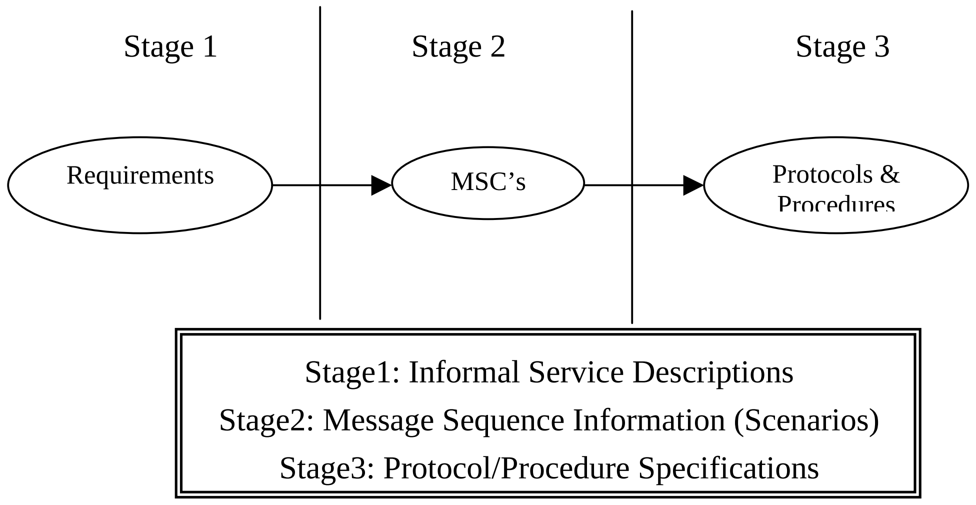

Stage 1
Stage 2
Stage 3
Requirements
MSC's
Protocols & Procedures
Stage1: Informal Service Descriptions
Stage2: Message Sequence Information (Scenarios)
Stage3: Protocol/Procedure Specifications

Figure 4: Three Stages Methodology

- In stage one, a description of a service should be given from the user's perspective. This stage describes what the service is supposed to do, not how it will do it.
- Stage two describes the capabilities and processes within the network. This is achieved by using sequences of messages between the different involved entities.
- The final stage produces the protocol specification.

3.2 Stage1: Requirement Description

During the past few years, a lot of research has been done in the area of Requirement description. Many models (Prototype model, Use Case Model, Organized by Roles Model, Organized by Classification) have been proposed for capturing the user requirements. The most commonly used model is use cases. The objective of this model is to capture the functional requirements from the user point of view. There are several reasons why use cases have become popular and universally adopted. According to [32] the two major reasons are: 1) they offer systematic and intuitive means of capturing functional requirements. 2) They drive the whole development process since most activities such as analysis, design, and test are performed starting from use cases.

Scenario-based approaches are now widely used in industry for the design of distributed systems. One of the main reasons is that scenarios describe top-level critical requirements that need to be fulfilled by the detailed design, and thereafter by implementation.

The following introduces the readers to the requirement description techniques that are used in this thesis: Chisel Diagrams and Use Case Maps. However the bulk of this chapter will consist of an introduction to Use Case Maps, which are extensively used in this thesis.

3.2.1 Chisel Diagram Notation

Chisel diagrams are a scenario-based approach that is used to describe requirements for communications services and features. The language Chisel is intended to reflect current practice for writing these requirements. This language could be applied to a variety of network technologies, and it has a sound basis for translation to commonly used formal software specification languages [7].

For illustration, a basic two-party POTS (Plain Old Telephony System) Chisel diagram is given in figure 5.

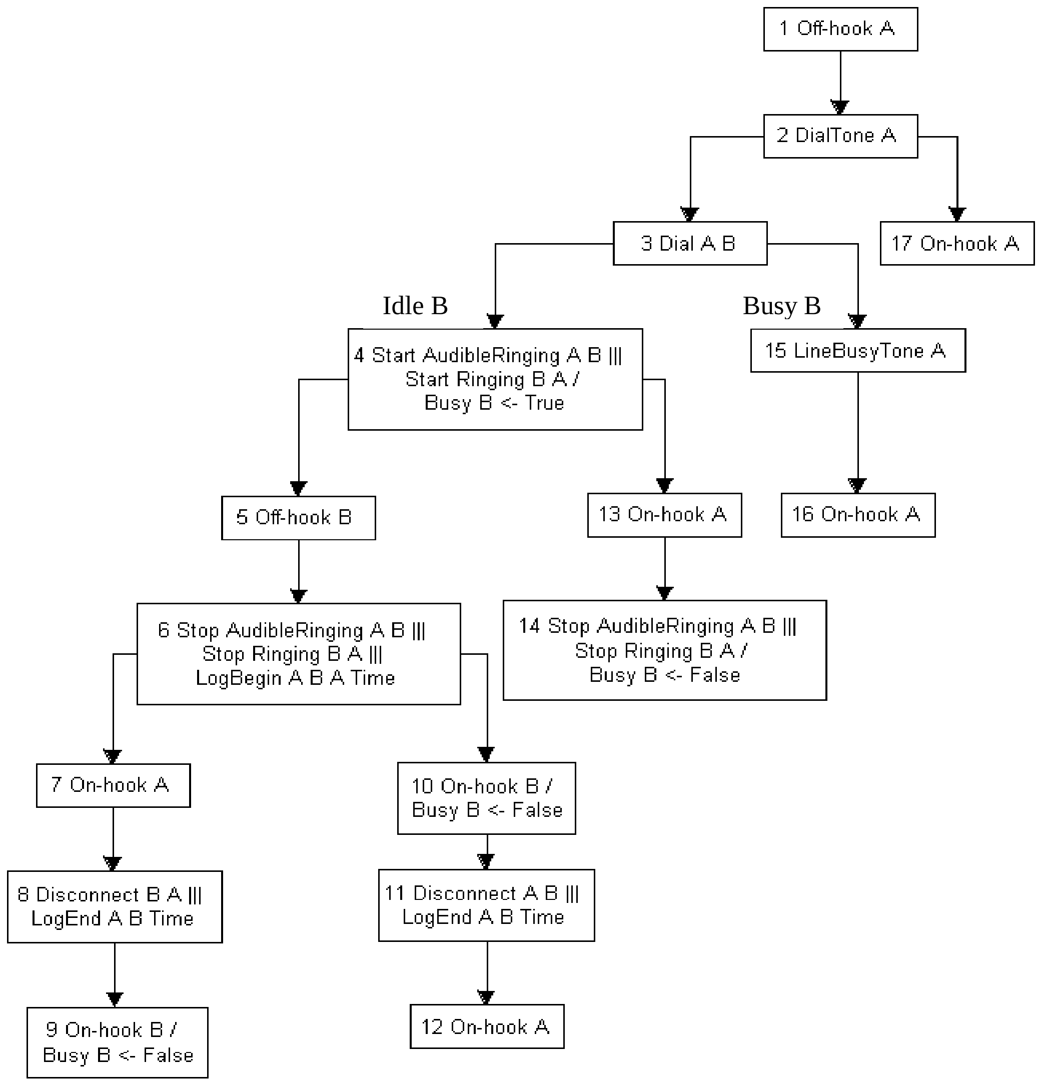

1 Off-hook A
2 DialTone A
3 Dial A B
17 On-hook A
Idle B
Busy B
4 Start AudibleRinging A B ||| Start Ringing B A / Busy B <- True
15 LineBusyTone A
5 Off-hook B
13 On-hook A
16 On-hook A
6 Stop AudibleRinging A B ||| Stop Ringing B A ||| LogBegin A B A Time
14 Stop AudibleRinging A B ||| Stop Ringing B A / Busy B <- False
7 On-hook A
10 On-hook B / Busy B <- False
8 Disconnect B A ||| LogEnd A B Time
11 Disconnect A B ||| LogEnd A B Time
9 On-hook B / Busy B <- False
12 On-hook A

Figure 5: Chisel Diagram for POTS

This Chisel diagram includes both telephones in a two-party call, and also some messages for the billing system.

A node (one of the boxes) in a chisel diagram contains a number, which uniquely identifies the node within the feature, and one or more events and variable assignments. The nodes are connected by directed edges (arrows in the diagrams). Multiple events in a node are separated by vertical bars (|||). A node containing multiple such events is equivalent to any possible sequence of those same events (i.e., A ||| B means {AB or BA}; A ||| B ||| C means {ABC or ACB or BAC or BCA or CAB or CBA}; and so forth).

This is a description of the main events involved in POTS:

- *Dial A B* means that the subscriber at address A dials the address B.
- *DialTone A* means that dial tone occurs at address A.
- *Start Ringing A B* means that alerting starts at address A for a call originated at address B.
- *Start AudibleRinging A B* means that the ring back tone is provided at address A while waiting for the user at address B to answer the call.
- *Stop Ringing A B* and *Stop AudibleRinging A B* mean to stop the ringing or tone occurring at address A in relation to a call to or from B.
- *LineBusyTone A* means that the telephone to which A is attempting a connection is busy.
- *Disconnect A B* informs A that B has disconnected a connection with A. (It is a signal from the switch to a user, signalling the user that a connected party has gone *On-hook*. The *On-hook* event is the signal from the user to the switch that the user is disconnecting)

Variables are used in conditions on edges, to define when an edge can be followed in constructing an event sequence from the diagram and to restrict possible interleavings of event sequences. A variable defines one or more sequences of events. For instance *Busy B* (Busy B <- True in figure 5, node 4) defines the set of event sequences having one of the following properties:

- An event sequence containing an *Off-hook B* not followed by *On-hook B*.
- An event sequence containing *Ringing B* not followed by *Disconnect B A*.

Note: All of the POTS event sequences start and end with Busy A = False (Idle A = True).

To define the value of a variable after an event an assignment statement can be included with the event to say that the variable takes on a new value after the event. The format of this is: <event> / <var> $\longleftarrow$ <value>.

Note: Value changes are shown in nodes (nodes 4,9,10 and 14 in figure 5).

A condition next to an edge means that to continue an event sequence by following that edge, the condition must be true at the end of the event sequence. C syntax is used in the conditions (~ for not, && for and, || for or).

3.2.2 Use Case Maps

3.2.2.1 Introduction

Like Chisel Diagrams, Use Case Maps (UCM) is a Scenario-based approach. It is a visual notation for representing use cases. It has been proposed by Buhr and Casselman [58].

The UCM notation is used to describe scenario paths in terms of causal relationships between responsibilities. UCM paths are wiggly lines that enable a person to visualize scenarios threading through a system without the scenarios actually being specified in any detailed way.

The notation is intended to be useful for requirement specification, design, testing, maintenance, adaptation, and evolution [69]. Already, UCMs have been used in a number of areas [69]:

- Requirements engineering and design of:
  - Real-time systems
  - Object-oriented systems
  - Telecommunication systems
  - Distributed systems
- Detection and avoidance of undesirable feature interactions
- Evaluation of architectural alternatives
- Functional testing

- Documentation of standards

  UCMs have raised a lot of interest in the software community, which led to the creation of a user group at the beginning of 1999, with more than one hundred members from all continents [69]. Currently, UCM standardization is underway within ITU-T.

3.2.2.2 UCM Notation Elements

In this section, we introduce the UCM notation. We limit ourselves here to the UCM elements that we are going to use in this thesis. Use Case Maps for Object-Oriented Systems [58] and Use Case Maps as Architectural Entities for complex systems [59 represent more complete tutorials on the UCM notation.

The core notation consists of only scenario paths and responsibilities along the paths.
The basic path notation addresses simple operators for causally linking responsibilities in sequence, as alternatives, and in parallel.

A UCM path may have any shape as long as it is continuous. It starts at a starting point (depicted by a filled circle) and ends at an end point (shown as a bar). Between the start and end points, the scenario path may perform some responsibilities along the path, which are depicted by crosses × with labels. Responsibilities are abstract activities that can be refined in terms of functions, tasks, procedures, events, and are identified only by their labels. Tracing a path from start to end is to represent a scenario as a causal sequence of events.
Note: Start points may have preconditions or triggering events attached, while responsibilities and end points can have post-conditions.

Figure 6 illustrates the basic elements of UCMs.

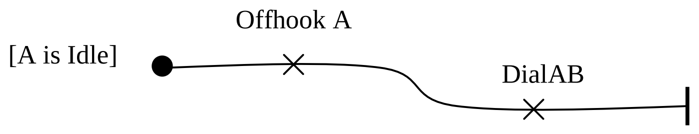


Figure 6: UCM Basic path

The responsibilities can be bound to components, which are the entities or objects composing the system. Figure 7 illustrates an UCM with three components: phone1, phone2, and a switch. We use the connection phase of a simplified telephone system in this figure because it is easy to understand. An initiator (phone A) tries to establish a connection with a Responder (phone B) via a simple switch. The scenario starts with a responsibility "OffHook A" where user A picks up the phone. This is the first activity that initiates the connection. Then "DialAB" is performed where user A dials the phone number of user B. We have now two alternatives (the OR-Fork is describes below) each one of which is associated with a pre-condition. For example user A receives a busy tone when the precondition is [B is busy].

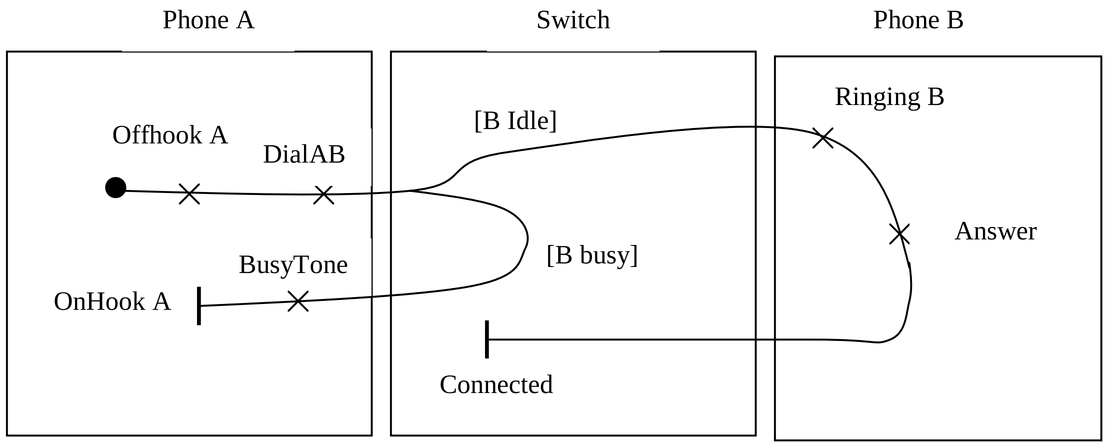


Figure 7: Bound UCM

Several paths can be composed by superimposing common parts and introducing forks and joins. There are two kinds of forks and Joins.

1. OR-Fork/Join: Depicted by branches on paths. They describe alternative scenario paths, which mean that one of the paths is selected to proceed at each branch.

- OR-Fork (Figure 8a): Splits a path into two (or more) alternatives. Alternatives may be guarded by conditions represented as labels between square brackets.
- OR-Join (Figure 8b): merges two (or more) overlapping paths.

2. AND-Fork/Join: Depicted by branches with bars, which describe concurrent scenario paths.

- AND-Fork (Figure 8c): Splits a path into two (or more) concurrent segments.
- AND-Join (Figure 8d): Synchronizes two (or more) paths together.

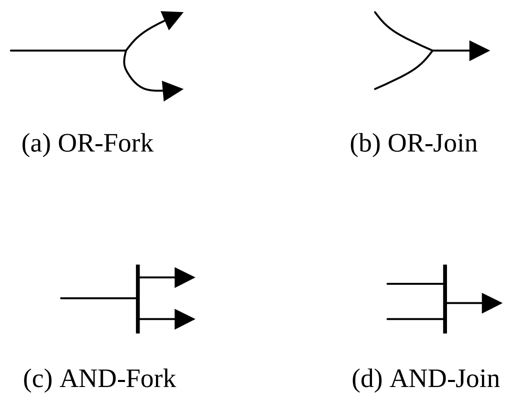


Figure 8: OR-Forks/Joins and AND-Forks/Joins

3.2.2.3 Advanced Notation Elements

More advanced operators can be used for structuring UCMs hierarchically and for representing exceptional scenarios and dynamic behaviour. When maps become too complex to be represented as one single UCM, a mechanism for defining and structuring sub-maps becomes necessary. A top-level UCM, referred to as root map, can include containers (called stubs) for sub-maps (called plug-ins). The stub plug-in concept allows UCMs to have a hierarchical path structure, to defer details, and to reuse the existing scenarios.
Stubs are of two kinds: Static Stubs and Dynamic stubs.

- **Static Stubs** (Figure 9): Represented as plain diamonds, they contain only one plug-in, hence enabling hierarchical decomposition of complex maps.

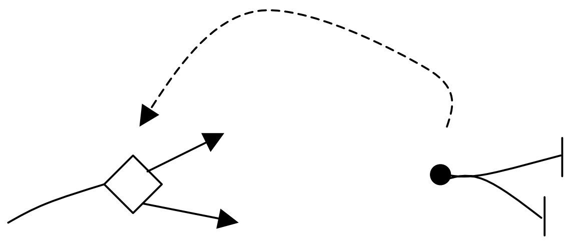


Figure 9: Static stubs have only one plug-in (sub-UCM)

- **Dynamic stubs** (Figure 10): represented as dashed diamonds, they may contain several plug-ins, whose selection can be determined at run-time according to a selection policy (often described with pre-conditions).

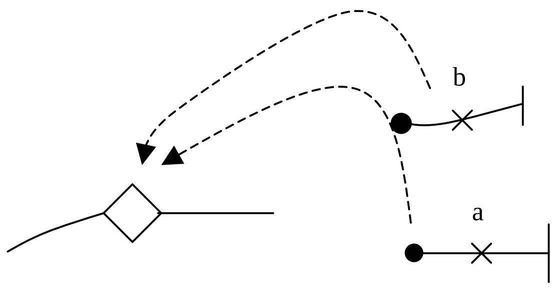


Figure 10: Dynamic stubs may have multiple plug-in

3.2.2.4 Philosophy of UCMs

The Use Case Maps notation aims to link behaviour and structure in an explicit and visual way. According to [59] UCM paths are first-class architectural entities that describe causal relationships between responsibilities, which are bound to underlying organizational structures of abstract components.

UCMs can be derived from informal requirements or from use cases if they are available. Responsibilities need to be stated or be inferred from these requirements. For illustration purpose, separate UCMs can be created for individual system functionalities, or even for individual scenarios. However, the strength of this notation mainly resides in the integration of scenarios.

It is important to clearly define the interface between the environment and the system under description. This interface will lead to the start points and end points of the UCMs paths, and it also corresponds to the messages exchanged between the system and its environment. These messages are further refined in models for detailed design (e.g. with Message sequence Charts, see Section 3.3).

3.2.2.5 UCM Tools

There currently exists only one tool that supports the UCM notation: The UCM Navigator (UCMNav). This tool is used for creation and maintenance of UCMs. UCMNav ensures the syntactical correctness of the UCMs manipulated, generates XML descriptions, exports UCMs in Encapsulated Postscript or Maker Interchange Format (for Adobe Framemaker) formats, and generates reports in PostScript.

3.3 Stage2: Message Sequence Chart

Message Sequence Charts (MSC) [33] is a graphical and textual language for the description and specification of the interactions between system components. The main area of application for Message Sequence Charts is the specification of the communication behaviour of distributed systems, like telecommunication switching systems. Message Sequence Charts may be used for requirement specification, simulation and validation, test-case specification and documentation of real-time systems. MSCs are often used in combination with SDL (Specification Description Language)[57].

Figure 11 describes the interaction between 3 components: C1, C2 and C3 via exchanging messages a, b, c and d.

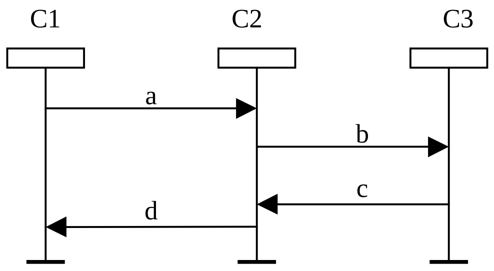


Figure 11: MSC Example

3.4 Benefits of UCMs and their relation with MSCs

These are some of Use Case Maps benefits:

- During early stages, UCM can be composed of paths where responsibilities are not allocated to any component. However, designers are likely to include architectural elements such as internal components. In this case the description of these components, their nature, and some relationships (e.g., components that include sub-components) are required. Communication links between components are usually not required, but they can be added.

- UCMs do not specify anything about details such as data transfers along paths, local data values at points along paths, and local decisions based on local data values.

- Use Case Maps are used to describe and integrate use case representing the requirements. UCMs give us the big picture at a high level of abstraction with using hiding mechanism.

- UCMs are intended to bridge the gap between requirements (use cases) and detailed design (MSCs for example), since they are expressed above the level of messages exchanged between components. More than one MSC may be derived for a single UCM. In figure 12, the paths in the UCM show the causal sequence abc in an abstract manner. Two possible implementations of this UCM are shown in the form of two MSCs.

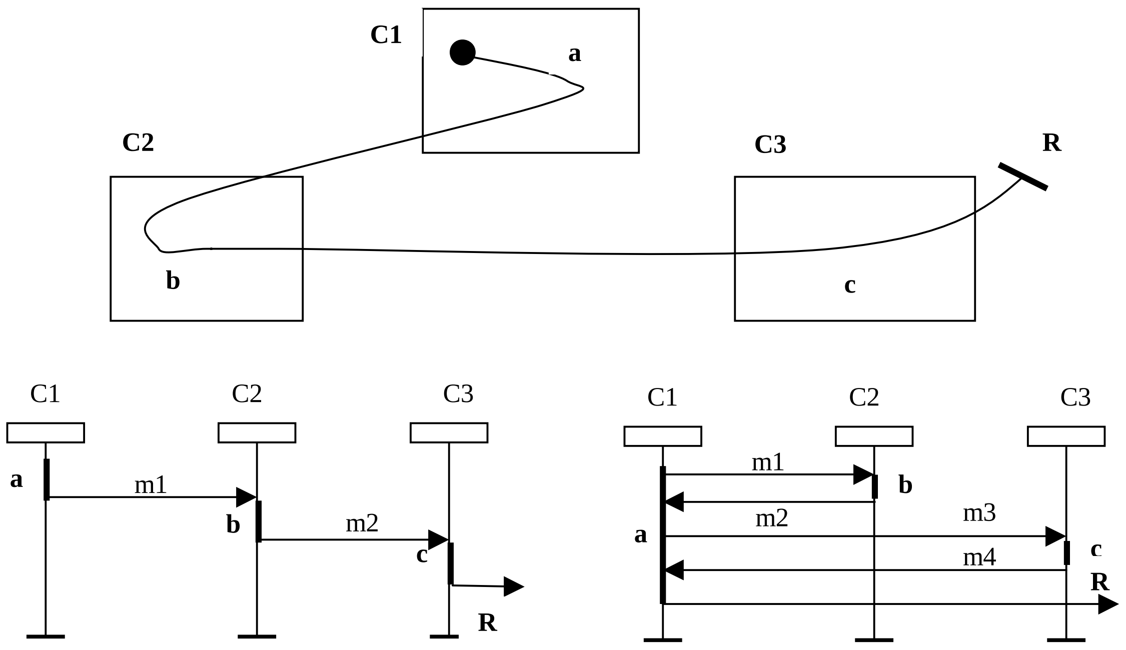


Figure 12: Causal Sequence of a UCM

- UCMs are not executable, but they can be manually translated to models that allow fast prototyping and validation. LOTOS, which will be introduced in the next chapter, is well suited for representing UCMs. Translation and execution of UCMs will be discussed in detail in Chapter 5.

- Test suites can be generated directly from UCM. The test cases generated from UCMs can be executed against the specification in order to prove consistency.

Chapter 4

Feature Interaction Filtering at Requirement Stage

4.1 Motivation

Communication Protocols are rules that are followed for orderly communication between two or more communicating parties. They are needed in order to ensure that the total system formed by the individual parties and their interaction is meaningful to all the parties concerned and performs the functions required. The basic functions of a protocol usually include: Connection, Disconnection, Access control (for security purposes), Addressing, Error control, Flow control and Synchronization. Figure 13 provides an abstract view of the relationship between network and users.

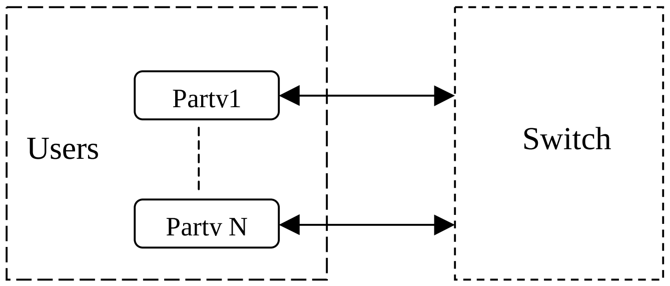


Figure 13: A reference model of a protocol system

The total communication protocol system is often divided into smaller ones, depending upon the stage that has been reached in the communication. For example, one protocol could be used to set up, prepare or establish communication, another could be used to ensure effective interchange of information after the connection has been established and a third protocol could be used to ensure the proper termination or closing down of the connection between the parties.

In practice this type of partitioning makes the total communication process easier to understand and enables modifications to various parts of the protocols to be made more simply and reliably.

4.2 Description of Services at Requirement stage

4.2.1 Service Decomposition

Intuitively, at least four main steps are needed to provide a service to an end user:

- Request the service
- Check for service availability and user authorization
- Provide the service
- Update the corresponding data and release the allocated resources

Each step may involve procedures to request resources, to set up the communication between network components, to access and update data…etc.

Starting from this simple intuitive decomposition we can decompose the Basic system service, the called Plain Old Telephone System (POTS), into four steps:

- ***Service request***: When a user wants to be served (e.g., wishes to make a call), the user indicates this desire to the network and receives an indication that this service can be provided. In the most familiar voice world this is accomplished through a user’s action of taking a telephone off-hook and through a network’s action of issuing a dial tone. The dial tone signals that the network is ready to provide a voice service, that is, that the access to the network has been granted.

- ***Checking the information***: The number dialed by the user could be invalid or out of service. In such a case an announcement will be played to the caller and a “busy-tone” signal is sent to him, otherwise the call procedure will continue.

- ***Provide the service***: This step is reached when the number dialled is valid and in service. Depending on the state of the called party, busy or idle, appropriate signals are sent to either parties and the service is provided. We consider that the busy tone signal sent to the caller, when the called party is busy, as a part of the service. During this step the billing process starts.

- ***Disconnection***: During this step the allocated network resources are released and the billing information is updated. Depending on who disconnects first appropriate signals are sent to either parties.

4.2.2 UCM Call Model

4.2.2.1 Introduction

Telephony features are usually complex and difficult to design and to implement. The specification of features written in a natural language (e.g. English) can be unclear or ambiguous and may be subject to different interpretation. As a result, independent implementations of the same feature may be incompatible. There is therefore, a need for a notation to help designers to understand and analyze these features.

Our main examples in this thesis are based on the feature requirements defined in the International Feature Interaction Detection contest held on the occasion of the Fifth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW98)[5]. The contest defined POTS and 12 switch–based and IN features. The requirements are described using the state-based language “Chisel” (see Section 3.2.1). Each feature is described with an end-to-end point of view and the different actions are not bound to network entities. Due to the nature of the state-based method, it is difficult to represent concurrent behaviours and feature addition is achieved by “gluing nodes” in the Chisel diagram, which results in difficulty to achieve global visualization in one picture. To cope with this problem we employ the requirement notation UCM.

4.2.2.2 UCM service description

To model a service (transactional, telephony...etc.) we use the service decomposition idea defined in 4.2.1. We build a “root UCM” (or simply root map) that specifies the scenario path structure commonly used by all services.

Figure 14 illustrates the service model described using UCM where each step is modeled as a “stub” containing a UCM sub map.

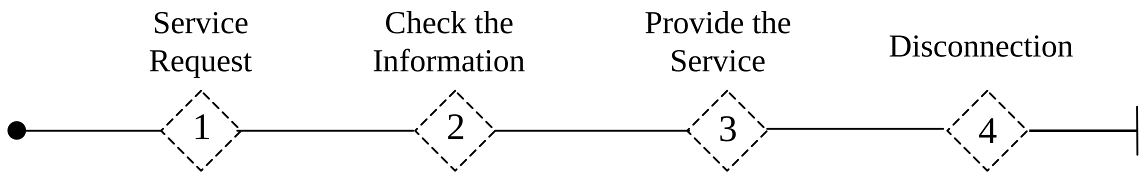


Figure 14: UCM service Model

4.2.2.3 UCM Call Model

The phone call model is derived from the UCM service model by defining the four UCM stubs corresponding to the four service stages.

The end points are not shown in the UCM service model. We choose to hide this information in the high level description and describe it at lower levels.

Figure 15 describes these four Stages for a phone call. This UCM is also called “Root Map”.

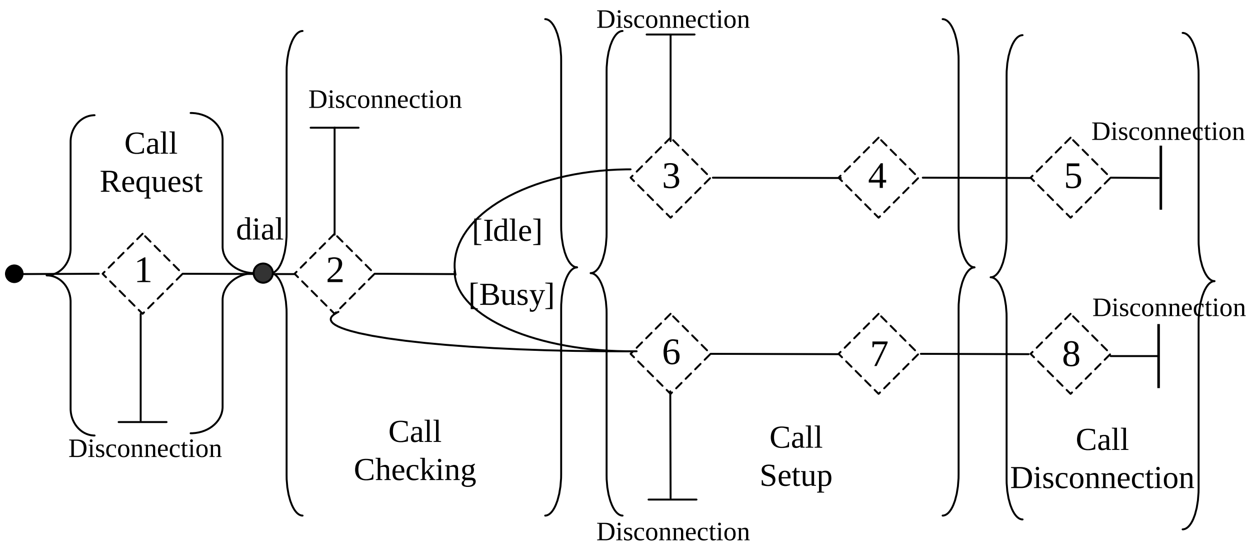


Figure 15: UCM Call Model (Root map)

Stub 1: Pre-Dial Stub
Stub 2: Post-Dial Stub
Stub 3: Idle Stub
Stub 4: Idle Setup Stub
Stub 5: Idle Disconnection Stub
Stub 6: Busy Stub
Stub 7: Busy Setup Stub
Stub 8: Busy Disconnection Stub

Note: A user can on-hook at any time before the disconnection stubs. In order to detect feature interactions we want to go as far as possible in the scenario, so we assume that no on-hook occurs before the disconnection stubs.

4.2.2.4 Stub Description

In our UCM model (Root map) we suppose that the user A is the caller, the user B is the called party (A will try to call B), C and D are parties introduced by some features.

❶ ***Call Request***

The Call request contains one stub:

- **Pre-Dial stub (Stub 1 in figure 15)**: During this step user A requests to be connected to another user. This stub contains the actions performed between “Off-hook A” (A is supposed Idle) and “dial AB” (If we suppose that A dials B’s number).

❷ ***Checking the call information***

All the checking is performed within one stub: the Post dial stub.

- **Post-Dial stub (Stub 2 in figure 15)**: This stub contains the actions performed after “dial AB” action. The checking consists of looking for the relevant user information, which could be related to a subscribed feature.
  The call could be blocked if the subscriber doesn’t meet the authorization rules defined by specific features. For example: When caller A is an OCS (Originating Call Screening) subscriber and B is in A’s screening list, the caller will receive an announcement message telling him that the call is denied. So the call is blocked.
  Note: The path between stub 2 and stub 6 is followed when the destination state (B in this case) is not relevant (we don't care whether B is idle or busy).

❸ ***Call Setup***

In this step the service is provided to the involved users. These four stubs cover all the possible scenarios based on whether the destination is idle or busy and on the features the users are subscribed to.

- **Idle Stub (Stub 3 in figure 15):** Contains the actions performed when the destination (B) is Idle. These actions occur before the establishment of the communication. The attempt to establish the call is successful but the call is not set up yet.

- **Idle Setup Stub (Stub 4 in figure 15):** Contains the actions performed after the destination (B) goes “Off-hook” and before the disconnection process (when one of the two parties decides to end the communication).

- **Busy Stub (Stub 6 in figure 15):** Contains the actions performed when the destination (B) is Busy. These actions occur before the establishment of the communication if any.

- **Busy Setup Stub (Stub 7 in figure 15):** Contains the actions performed after the destination (example: a third party C) goes “Off-hook” and before the disconnection process (when one of the two parties decides to end the communication).

❹ ***Disconnection***

- **Idle Disconnection Stub (Stub 5 in figure 15):** Contains the actions performed when one of the two parties involved decides to end the call.

Note: The caller A can hang up before the establishment of the call. In this case the disconnection procedure is described within the previous stubs.

- **Busy Disconnection Stub (Stub 8 in figure 15):** Contains the actions performed when one of the two parties (example: A or the third party C) involved decides to end the call.

4.2.2.5 Plug-ins of POTS

Figure 16 represents UCMs plug-ins for the basic call model, (or POTS- Plain Old Telephony System), described based on the first FI detection contest specifications.

There are six UCMs plug-ins in figure 16. Each one is identified by a name, e.g., Pre-Dial Plug-in. They are considered as default plug-ins.

1: Pre-Dial Plug-in (Default)

2: Post-Dial Plug-in (Default)

3: Idle Plug-in (Default)

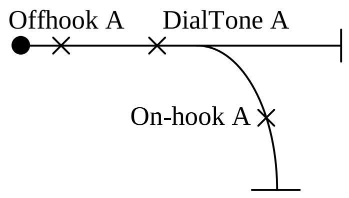
4: Idle Setup Plug-in (Default)

5: Disconnection Idle Plug-in (Default)

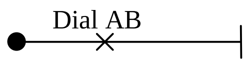
6: Busy Plug-in (Default)

7: None

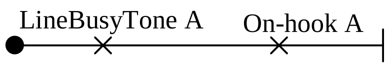
8: None



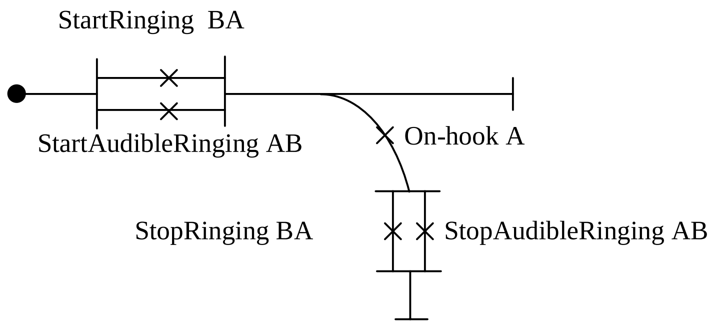
Stub1 :Pre-Dial plug-in



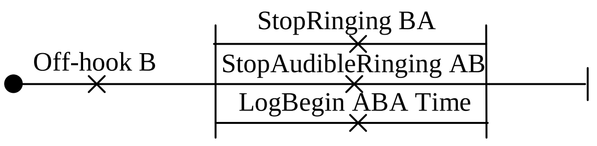
Stub 2: Post-Dial plug-in



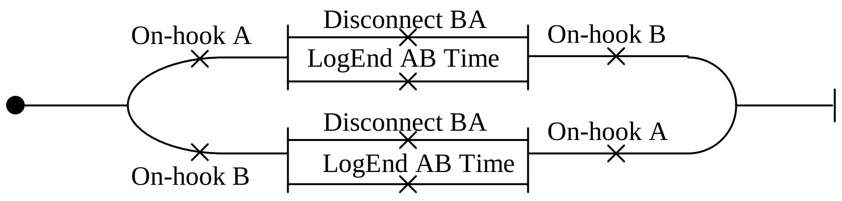
Stub 6: Busy plug-in



Stub 3: Idle plug-in



Stub 4: Idle Setup plug-in



Stub 5: DisconnectionIdle plug-in

Figure 16: Use Case Maps for Basic Call Model

The basic call model describes the basic actions for the establishment of a communication between two users A and B. First the user A goes pick up the phone (off-hook) then dials B’s number. If B is busy then a "LineBusyTone" signal is sent to A (Busy plug-in), otherwise the actions in the Idle plug-in are executed. The Idle plug-in starts with two actions taking place in either order: the phone at B's side starts ringing (StartRinging BA) and A receives an audible ringing (StartAudibleRinging AB) telling him that the phone on the called party is ringing. When the user B pick up the phone (off-hook B) to answer three actions took place at either order: the phone at B's side stops ringing (StopRinging BA), A stops receiving the audible ringing (StopAudibleRinging AB) and the billing procedure starts (LogBegin ABA Time) with respectively the caller, the called party, the charged party and the time of start billing as parameters.

4.3 Feature description using the UCM Model

In the 1998 Feature Interaction Contest [5], the feature requirements were described using the Chisel diagram notation.

We classify the features acting during the call establishment into two classes: *Originating features* and *terminating features*.

Features like TWC (Three Way Calling) and CW (Call Waiting), which need an already established communication between two users before they can perform their specific actions, cannot be classified into these two categories.

- **Originating features**

This class of features contains the ones that can be activated when the feature's subscriber tries to establish a call.

Among originating features we can mention OCS (see Section 1.2.1) and INTL (see Section 4.4.1).

Note: When describing an originating feature we suppose that the caller A is the feature's subscriber.

- **Terminating features**

This class of features contains the ones that can be activated when the feature's subscriber receives a call.

Among terminating features we can mention TCS (see Section 4.4.3), CFBL (see Section 4.4.5), INFB (see Section 4.4.4) and CND (see Section 4.4.2).

Note: When describing a terminating feature we suppose that the called party B is the feature's subscriber.

4.4 Feature Addition

Feature requirements described in the Chisel notation are represented by means of UCMs. The mapping from Chisel notation to UCM notation is straightforward because like UCMs the "Chisel diagrams" use an end-to-end model.

Table 1 gives some translation guidelines to translate Chisel notation to UCM notation:

Chisel Notation	UCM Notation
Event	Responsibility
Directed edges (arrows)	Scenario paths
Conditions	Guards
Interleaving operator |||	AND Fork

Table 1: Translation guidelines from Chisel to UCM

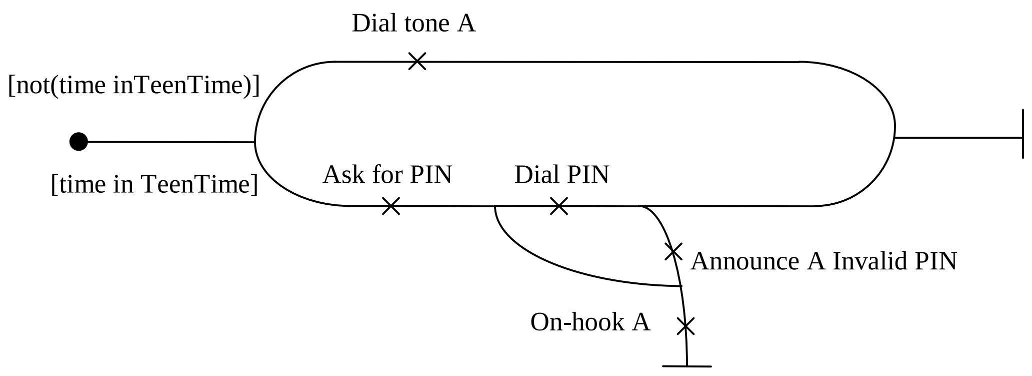
We consider in this thesis that the features are an Extension/Modification of the POTS because they use the stubs defined within the model. Adding features extends scenarios in the basic call model. In our framework, this is achieved in a simple way by using the stub plug-in concept of UCMs. Intuitively we only replace some default submaps with specific ones. The

UCMs obtained will be sets of submaps describing scenarios specific to the feature combined with the basic call model (Feature + BCM).

4.4.1 Originating feature: INTL (IN Teen Line)

Teen Line restricts outgoing calls based on the time of day (i.e., hours when homework should be the primary activity). This can be overridden on a per-call basis by anyone with the proper identity code. This is an IN feature.
Let us add INTL to the basic call. This is done by plugging INTL submap "INTL Pre-dial plug-in"(figure 17) into the Pre-dial stub of the root map in figure 15. This replaces the plug-in shown in figure 16.

Plug-ins for IN Teen Line:
1:INTL Pre-Dial plug-in
2:Default
3:Default
4:Default
5:Default
6:Default
7:None
8:None



Stub 1:INTL Pre-Dial plug-in

Figure 17: INTL plug-ins

When an INTL subscriber tries to place a call, two possible outcomes for his attempt are considered based on the time of the day and the correctness of his PIN.

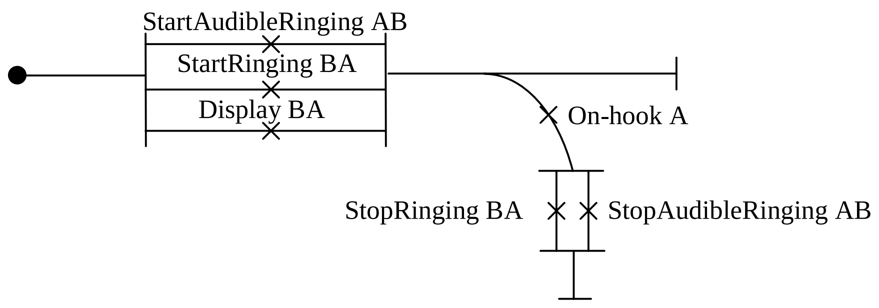
1) If the call is initiated outside the Teen time the call should continue normally
2) If the call is initiated during the Teen time then a message asking for a PIN is sent to the caller (Ask for PIN). If the caller dials the valid PIN then the call continues normally

otherwise an announcement is sent to the user (Announce A invalid PIN) telling him that the PIN entered was invalid. In this case the Call is blocked.

4.4.2 Terminating feature: CND (Calling Number Delivery)

CND is a feature that allows the called telephone to receive a calling party’s Directory Number (DN) and the date and time. In the on-hook state, in a real network, the delivery of this information occurs during the long silence between the first and second power ringing cycles.
For the purpose of the thesis, we assume the capability of delivering the number, and deliver it whenever an idle called party receives the Ringing event.

The addition of CND is done by plugging CND submap “Calling Number delivery Idle plug-in” (figure 18) into the Idle stub of the root map in figure 15. This replaces the plug-in shown in figure 16.
Plug-ins for Calling Number Delivery:
1:Default
2:Default
3:Calling Number delivery Idle plug-in
4:Default
5:Default
6:Default
7:None
8:None



Stub 3: Calling Number delivery Idle plug-in

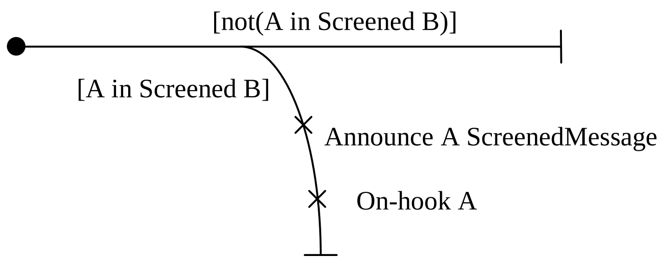
Figure 18: CND plug-ins

4.4.3 Terminating feature: TCS (Terminating Call Screening)

Terminating Call Screening restricts incoming calls. Calls from lines that appear on a screening list are redirected to a vague but polite message.

The addition of TCS is done by plugging the TCS submap “TCS Post-Dial plug-in” (figure 19) into the Post-Dial stub of the root map.

Plug-ins for Terminating Call Screening:
1:Default
2:TCS Post-Dial Plug-in
3:Default
4:Default
5:Default
6:Default
7:None
8:None



Stub 2: TCS Post-Dial Plug-in

Figure 19: TCS plug-ins

Two possible outcomes are considered based on the presence or the absence of A (the caller party) in the B's screened list:

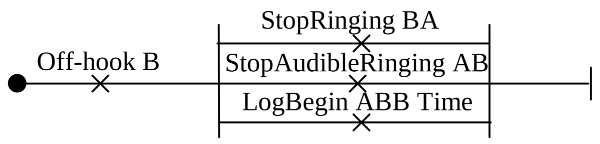
If A is in the B's screened list (condition: [A in Screened B]) then A receives an announcement telling him that is not allowed to call B, otherwise (condition: [not (A in Screened B)]) the call continues normally.

4.4.4 Terminating feature: INFB (IN Free Phone Billing)

The IN Freephone feature allows the subscriber to pay for incoming calls.

The addition of INFB is done by plugging INFB submap “INFB Idle setup plug-in” (figure 20) into the Idle Setup stub of the root map.

Plug-ins for IN Free phone Billing:
1:Default
2:Default
3:Default
4:INFB Idle setup plug-in
5:Default
6:Default
7:None
8:None



Stub 4: INFB Idle Setup plug-in

Figure 20: INFB plug-ins

B is the INFB subscriber. B is charged when answering incoming calls. This is defined by the third parameter, which is the charged party in the billing action (LogBegin AB**B** Time instead of the normal ABA).

4.4.5 Terminating feature: CFBL (Call Forwarding Busy Line)

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 subscriber’s originating service is not affected.

The addition of CFBL is done by plugging CFBL submaps “Busy CFBL” into the Busy stub, “Busy Setup CFBL” into the Busy Setup stub and the “Busy Disconnection CFBL” into the Busy disconnection stub.

Plug-ins for Call forwarding busy line:

1:Default

2:Default

3:Default

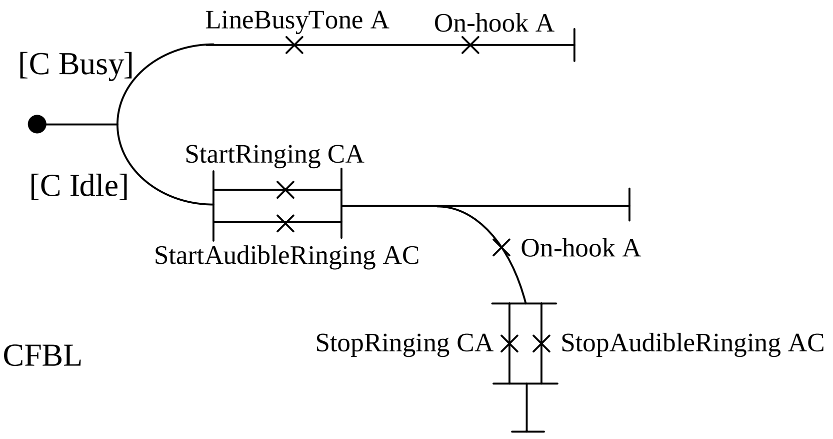
4:Default

5:Default

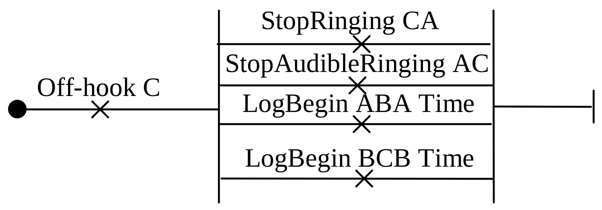
6:Busy CFBL

7:Busy Setup CFBL

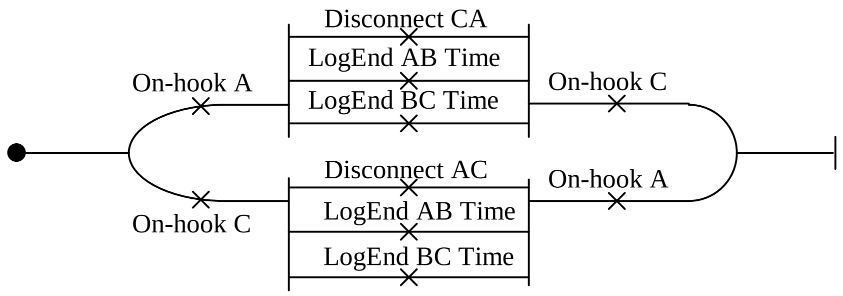
8:Busy Disconnection CFBL



Stub 6: Busy CFBL plug-in



Stub 7: Busy Setup plug-in



Stub 8: Busy Disconnection plug-in

Figure 21: CFBL plug-ins

In this scenario, we suppose that the originator is A, CFBL subscriber is B and the forward party is C. When the third party C is busy, the originator A gets a line busy tone signal. When C is idle the originator A gets an Audible Ringing signal while the destination C gets a

Ringing signal. Once the connection is established the originator A pays for “AB” leg while B pays for the forwarded leg of the call “BC”.

4.5 Feature Interaction Filtering Method

Feature Interaction detection algorithms need a significant amount of work due to the need of considering all possible combinations of behaviours. Thus, Feature Interaction detection can be expensive and even infeasible task [50]. Therefore, it would be helpful to have a method that can be used before feature interaction detection to estimate which feature combinations have a possibility of feature interaction.

The goal of feature interaction filtering is:

- To localize where interactions could take place (e.g. in which stub)
- To take out the interaction free scenarios from further analysis

4.5.1 Stub Configuration Vector

We can characterize features in terms of stub configuration vector, that is, information regarding which feature submap is plugged into which stub of the root map. In this section, we propose a vector representation called stub configuration vector (or simply SC-vector), to characterize features.

General Definition: A stub configuration vector (or simply SC-vector) is a vector of length n
F = [f_1,…, f_n], where f_i is the name of the plug-in of the i-th stub.

Note: For our UCM model n is equal to 8.

Example:
With A an INTL subscriber we have:
INTL=[INTL Pre-Dial plug-in, default, default, default, default, default, none, none]

Similarly, suppose that B is subscribes to TCS. Then,
TCS=[default, TCS Post-Dial plug-in, default, default, default, default, none, none]

4.5.2 Feature Composition

Once each individual feature is characterized by an SC-vector, we compose different configurations, in order to examine FI-Filtering between multiple features.

Composition Operators:

Suppose that f and g are two plug-ins plugged into the same stub in the root map (the proposed UCM model). Let *default* denote any default plug-in describing basic call scenarios. Let *ng* (stands for "no good") denote a special result not contained in the given plug-ins. Then, composition of *f* and *g*, denoted by $f \bullet g$, is defined as follows:

$$f \bullet g = g \bullet f = \begin{cases} f & (\text{if } f = g) & \text{(A1)} \\ f & (\text{if } g = default) & \text{(A2)} \\ f & (\text{if } g = none) & \text{(A3)} \\ ng & (\text{if } f \neq g) \text{ and } f,g \neq default & \text{(A4)} \end{cases}$$

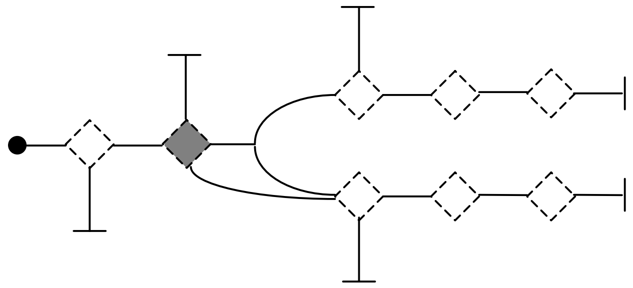
Figure 22: Composition Operator "."

The intuitive semantics of the composition is explained as follows: (A1) composition of the same submaps yields the same submap, (A2) a feature submap *f* can override a default map of basic call scenario, (A3) a feature submap *f* can override a missing submap ("none" in the SC-stub), (A4) two different feature submaps cannot be plugged into the same stub, since a non deterministic behaviour arises between f and g.

Now, we define the composition of SC-vectors:

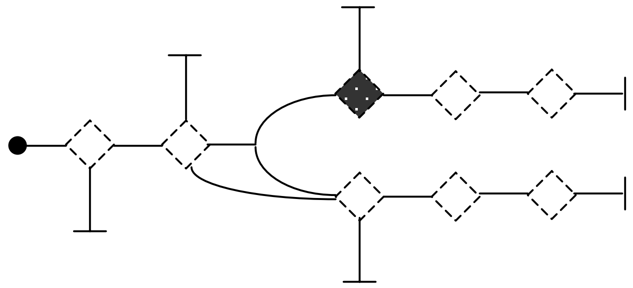
Let F = $[f_1,\dots,f_n]$ and G = $[g_1,\dots,g_n]$ be given SC-vectors. Then the composition of F and G, denotes by $F \oplus G$, is defined as $H = F \oplus G = [hi,\dots, hn]$ where hi = fi $\bullet$ gi for all i.

The composition of two SC-vectors is carried out by applying the "$\bullet$" operator to each pair of corresponding vector elements. Figure 23 illustrates an example of feature composition.



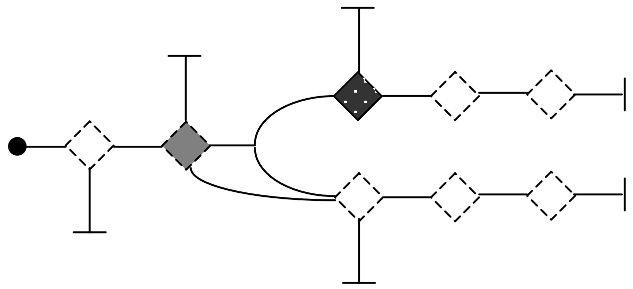
UCM for Feature 1(F1)

Feature1 = [default, F1 Post-Dial plug-in, default, default, default, default, none, none]



UCM for Feature 2(F2)

Feature2 = [default, default, F2 Idle plug-in, default, default, default, none, none]

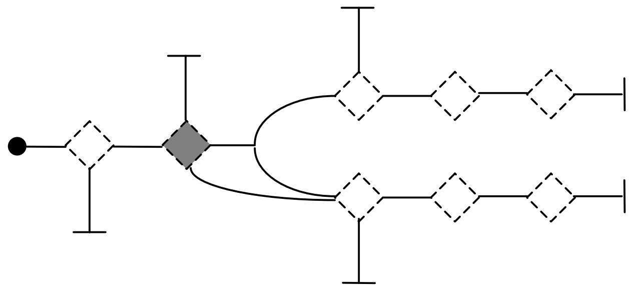


UCM for Feature 1 ⊕ Feature 2

Feature1 ⊕ Feature 2 = [default, F1 Post-Dial plug-in, F2 Idle plug-in, default, default, default, none, none]

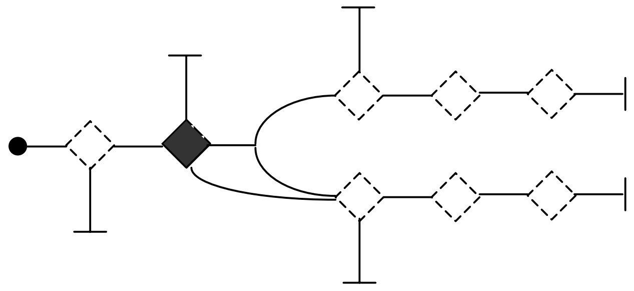
Figure 23: Feature Composition (1)

However the two following features, described in figure 24, cannot be combined because of a conflict in the Post-dial stub so they need further investigation to detect possible interactions.



UCM for Feature 1

Feature1 = [default, F1 Post-Dial plug-in, default, default, default, default, none, none]



UCM for Feature 2

Feature2 = [default, F2 Post-Dial plug-in, default, default, default, default, none, none]

Feature1 ⊕ Feature 2 = [default, **ng**, default, default, default, default, none, none]

Figure 24: Feature Composition (2)

Example:

Let us compose INTL with TCS:

INTL = [INTL Pre-Dial plug-in, default, default, default, default, default, none, none]

TCS = [default, TCS Post-Dial plug-in, default, default, default, default, none, none]

INTL ⊕ TCS = [INTL Pre-Dial plug-in, TCS Post-Dial plug-in, default, default, default, default, none, none]

So the two features can be combined without problems.

4.5.3 Feature Interaction targeted

We use the definition given in Chapter 1, which states that there is an interaction between features when the combined specification is inconsistent in some way, either due to specifying inconsistent state changes or inconsistent observable actions. The inconsistencies that we have addressed will be formalised by the two rules we give below.

4.5.4 Filtering rules

Let H = $F \oplus G$.

<u>Filtering Rule 1:</u>

There exists *ng* in H $\Rightarrow$ FI occurs (non determinism)

A *ng* entry appears in H iff the combination of the two features requires that a submap *fi* in F and a submap *gi* in G be plugged into a stub *i* simultaneously. If this is done, different scenarios are possible at the entry of the same stub, which causes non-determinism.

Figure 25 describes the inconsistency introduced by trying to put two different plug-ins into the same stub. The entry point represents state S1 from which we have two possibilities : execute X1 to get to state S2 or execute X2 to get to state S4.

- Inconsistent observable actions: Observing action X1 while we are expecting action X2 to occur and vice versa.
- Inconsistent state changes: Expecting to get to state S2 by executing X1 while we get to the state S4 by executing X2 and vice versa.

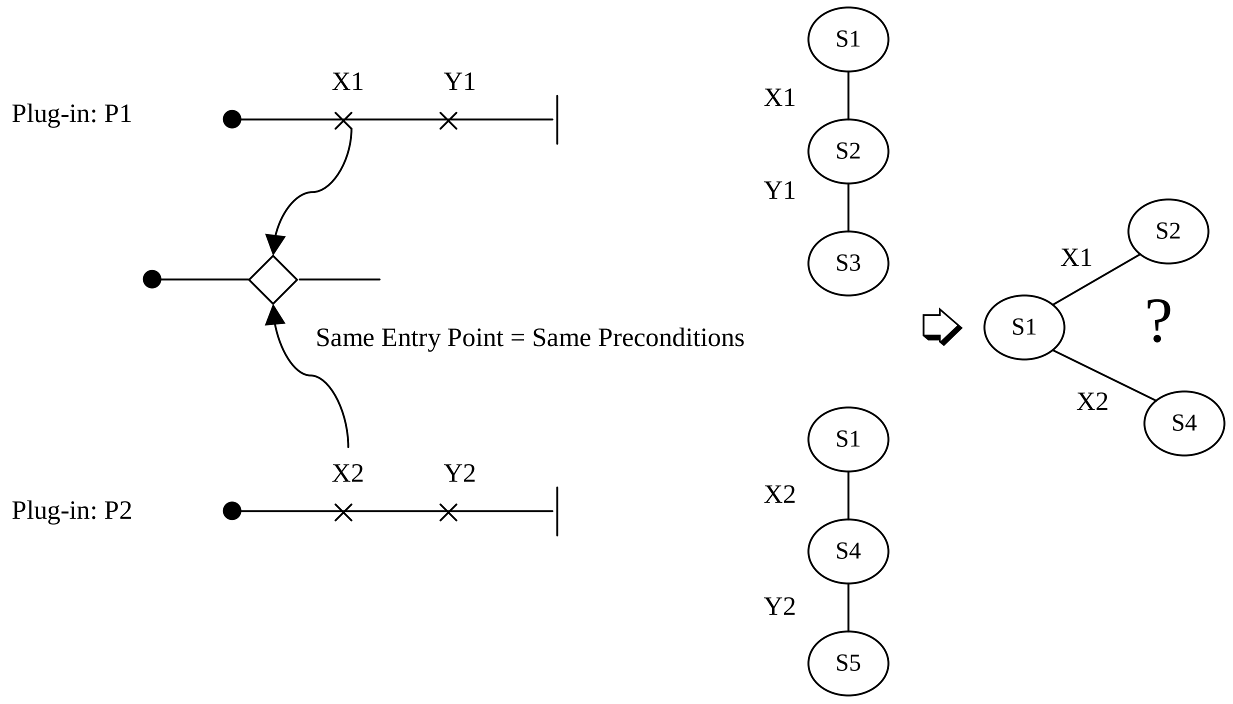


Figure 25: Non determinism (Interaction)

Filtering Rule 2:

(There is no *ng* in H) and (*f2* $\neq$ *default*) and ($\exists$ *gi* $\neq$ *default where* $6 \leq i \leq 8$) $\Rightarrow$ FI could occur (Inconsistent state changes)

The condition: ”There is no *ng* in H” is introduced not to include cases already treated in the Filtering rule1.

This rule derives from the way we have decomposed our services. As described in section 4.2.2.2 in stage 2, the network checks the information (authorization, user data…etc.) to determine whether the service should be provided or not (Stubs 1 and 2 represent the originating part of the call). This means that the service could be denied or blocked in this stage if the user doesn’t meet the required conditions. Stages 3 and 4 deal essentially with calls processing (Stubs 3 to 8 represents the terminating part of the call).

Let's analyse the preconditions of the filtering rule 2:

- *$f_2 \neq$ default:* The feature *F* has a specific plug-in (*f_2*) for the Post-Dial stub (which treats the checking part), this means that there is a possibility for the call not to take place.
- *$\exists\, g_i \neq$ default for $6 \leq i \leq 8$:* The feature *G* has a specific behaviour in stubs 6 or 7 or 8, this means that the call may be forwarded/routed to a third party. The third party has only the terminating part of the call described in stubs 6, 7 and 8.

Interaction: Since the call could be denied (blocked) in the originating part of the call (feature F, *$f2 \neq$ default*) and this part of the behaviour is missing in feature G for the third party, an interaction could occur.

Example: User A is an OCS subscriber (OCS has a specific Post-dial plug-in in stub 2. $OCS_2 \neq$ *default*). User B is a CFBL subscriber (CFBL has a specific plug-ins in stubs 6, 7 and 8. ($CFBL_i \neq$ *default* for $6 \leq i \leq 8$). When A calls B the call is forwarded to C and not blocked, which is a feature interaction.

4.5.5 Completeness

From the previous section we have seen that when one of the two filtering rules is applied there is a possible interaction. For completeness purpose let's analyse all remaining cases.

We distinguish three cases:

❶ Feature F has specific plug-ins (different from default) for at least one of the stubs 3, 4 and 5, feature G has also specific behaviour (different from default) for at least one of the stubs 3, 4 and 5, and $F \oplus G$ doesn't contain any "ng" (see figure 26): These two features act at distinct call levels of the same leg of the call (before, during or after the call establishment). Therefore their behaviour could be executed sequentially. So F and G are said to be "Interaction Free".

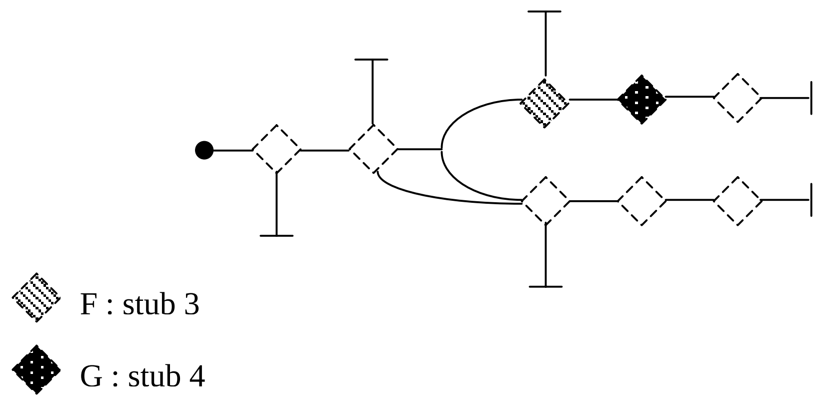


Figure 26: *F⊕ G* :Interaction Free Case 1

❷ Feature F has specific plug-ins (different from default) for at least one of the stubs 3, 4 and 5, feature G has specific behaviour (different from default) for at least one of the stubs 6, 7 and 8, and *F⊕ G* doesn't contain any "ng" (see figure 27): These two features have different preconditions: for feature F, the called party should be Idle whereas for feature G the called party should be in a busy state (or we don't care about the called state) and the call is forwarded to a third party. Therefore the two features are mutually exclusive (Only one of them is active at a time). So F and G are Interaction free.

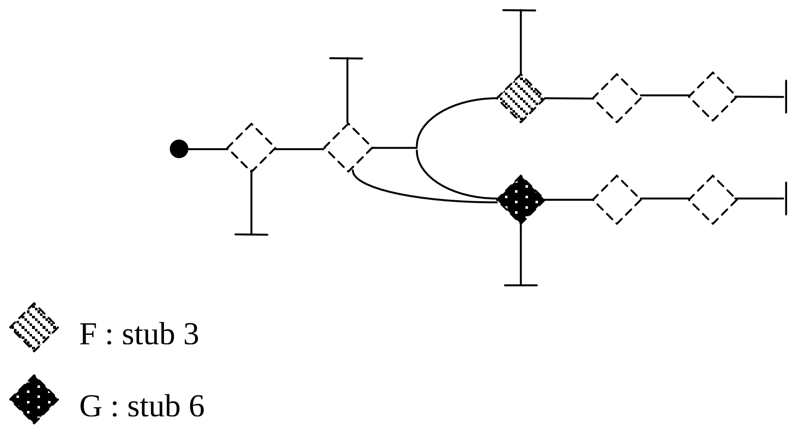


Figure 27: *F⊕ G* Interaction Free Case 2

❸ Feature F has a specific plug-in in stub 2 ($f_2 \neq default$), feature G has a specific behaviour (different from default) for at least one of the stubs 3, 4 and 5, and *F⊕ G* doesn't contain any "ng" (see figure 28): The model is designed such that the stub 2 precedes stubs 3, 4 and 5. The priority is given to feature G over F and we assume that in this case F and G are Interaction free.

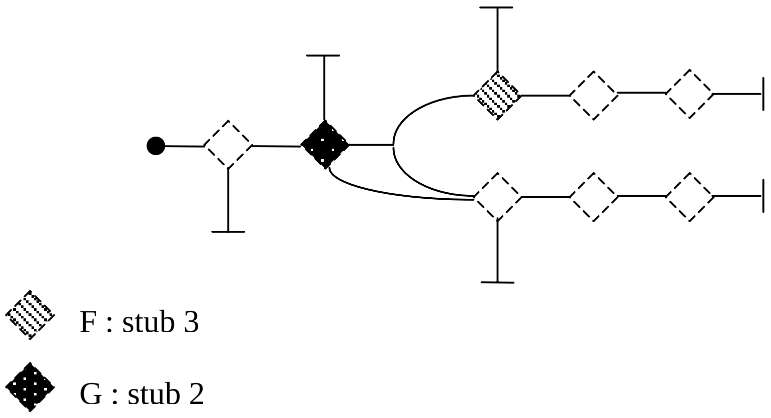


Figure 28: $F \oplus G$ Interaction Free Case 3

Notes:

- All the combinations involving the stub 1 where rules 1 and 2 are not applied are Interaction free because the call is not initiated yet at that level.
- The detection method presented in Chapter 6 doesn't consider any precedence assumptions.

4.5.6 Feature Interaction Filtering Method

Interaction between features depends on the way these features are assigned to subscribers. For example, an interaction may exist between feature F1 and feature F2 if F1 and F2 are assigned to the same subscriber. It is possible that there would be no interaction if they are assigned to different subscribers. Therefore, to detect interaction, we should look at all possible assignments of features to subscribers. In the case of two features, we should look at the system when both features are assigned to the same subscriber, and when the two features are assigned to different subscribers.

In our system, the assignment of features to subscribers is done statically, i.e. a feature is specified independently and assigned to a single subscriber.

Before looking for FI scenarios we choose to discard the irrelevant scenarios from the filtering process. This reduces the number of possible scenarios to be investigated.
So scenarios that should be discarded are those of the following types:

- A user subscribes to two features, one originating and one terminating. These scenarios are useless since a user cannot be a call originator and a call terminator at the same time.
- B or C subscribes to an Originating feature. These scenarios are useless since the originating feature is inactive for B and C during the scenario.

Filtering Method Input:
Feature Stub configuration Vectors: F = [*f1,…,fm*], G = [*g1,…,gm*]…etc.
Filtering Method Output:
Proceeding by a pair wise filtering the Output will be one of these three verdicts:

(1) Feature Interaction occurs

(2) Feature Interaction could occur

(3) F and G are Interaction Free

We provide a filtering routine to be used in the general filtering procedure presented below.
Filtering Routine:
❶ Make a composed vector $F\oplus G$

❷ If some "ng" elements exist in $F\oplus G$ then conclude that FI occurs (verdict 1) (From the Filtering Rule 1)

❸ If Filtering rule 2 holds then conclude that FI could occur (verdict 2) otherwise F and G are Interaction Free

Filtering procedure:

The procedure consists of three steps:

Step 1:

This step aims to treat scenarios where the same user subscribes to two features. The step 1 is divided into 2 sub steps:

- The feature SC vectors are provided
- Apply the Filtering Routine and get the verdict.

Step 2:

This step aims to treat scenarios where the two features are distributed between the caller and the called parties only (i.e. A and B).

The step 2 is also divided into 2 sub steps:

- The feature SC vectors are provided
- Apply the Filtering Routine and get the verdict.

Step 3:

This step treats scenarios where a third party C is the feature subscriber and B arranges to forward/route the call to C.

The step 3 is divided into 2 sub steps:

Step 3.1: SC Refinement:

C is a terminating party. So only the terminating part of the SC vector interests us.

To be able to detect the feature interactions we should look more in depth to the 6^{th}, 7^{th} and 8^{th} stubs for the feature that introduces the third party (i.e. CFBL, INCF, INFR…etc.). Figure 29 illustrates the refinement process.

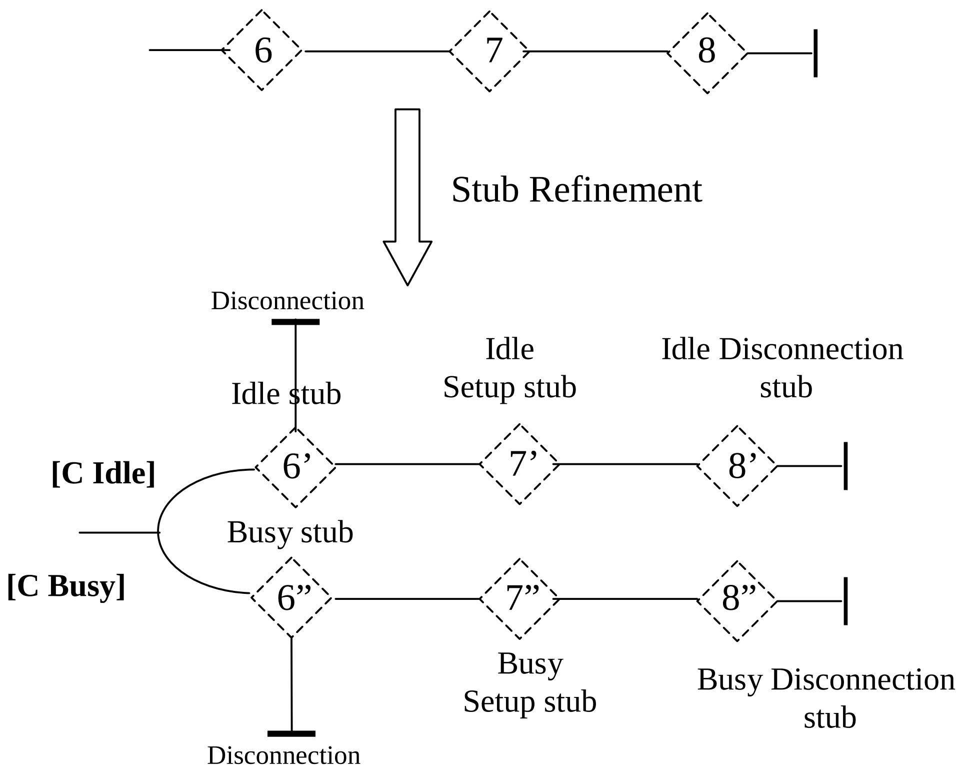


Figure 29: UCM Refinement

Stub Configuration sub-vector**:** A stub configuration sub-vector (or simply SC sub-vector) is a vector F’ = [*f ’1*,…, *f ’*n], where *f’* i is the name of the plug-in of the i-th stub.

Note: For our UCM model n is equal to 6 and an SC sub-vector corresponds to a vector:
F’= [6’ plug-in, 7’ plug-in, 8’ plug-in, 6” plug-in, 7” plug-in, 8” plug-in]

Step 3.2: Define the Stub Configuration sub-Vector (F’ and G’) for each feature

Note: The SC sub-vector will be the terminating part for the feature that introduces the third party. The feature that introduces the third party will be refined as follows:

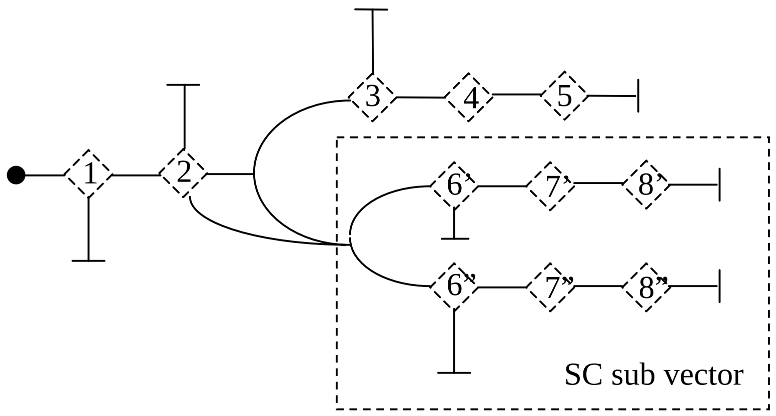


Figure 30: Feature causing the forward/Routing of the call (B)

The feature, to which the third party is subscribed, is be described as:

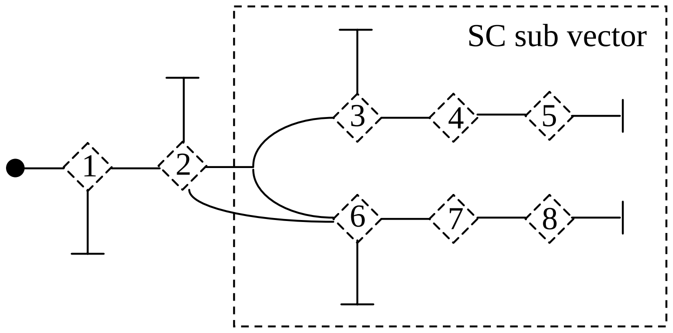


Figure 31: Feature to which C subscribes

- Make a composed SC sub-vector H'= [h'1,…,h'm] = $F' \oplus G'$ (that contains only 6 stubs)
- If some "ng" elements exist in H', conclude that (1) FI occurs (Filtering Rule 1)
- If the feature to which C subscribes has a specific behaviour for stub 2 (which is not described in the SC sub-vector) then conclude that (2) FI could occur (The filtering rule 2)
  Else F and G are (3) Interaction Free (In the cases where C is a feature subscriber)

4.6 Application

As an application of the proposed method we will illustrate the interactions happening between OCS (Originating feature), CND (Terminating Feature) and CFBL (Terminating

Feature). A is the caller; B the called and C is the third party. Table 2 illustrates some of the different possible distributions of the features between the users. Those that are not listed are of no interest.

	User A	**User B**	**User C**
1	OCS	CND	-
2	CND	OCS	-
3	OCS, CND	-	-
4	-	OCS, CND	-
5	OCS	CFBL	-
6	-	OCS, CFBL	-
7	-	CFBL	OCS
8	CND	CFBL	-
9	-	CND, CFBL	-
10	-	CFBL	CND
11	CND	CFBL	-
12	OCS	OCS	-
13	CND	CND	-
14	-	CFBL	CFBL

Table 2: Distribution of the features between users

Scenarios 3, 4 and 6 are useless because one of the users subscribes to both an originating and a terminating feature. Scenarios 2, 8, 11 and 13 are also useless since user A subscribes to a terminating feature. Scenarios 7, 12 are useless since B and C subscribe to an originating feature. This leaves scenarios 1, 5, 9, 10, 14, which are examined in detail below.

The two features are distributed between the caller and the called parties only:

- **Scenario 1:** OCS (A) and CND (B)

Stub #	1	2	3	4	5	6	7	8
CND	default	default	Calling Number delivery Idle plug-in	default	default	default	None	None
OCS	default	OCS Post-Dial Plug-in	Default	default	default	default	None	None
CND ⊕ OCS	default	OCS Post-Dial Plug-in	Calling Number delivery Idle plug-in	default	default	default	None	None

There is no *ng* in the resulting behaviour and filtering rule 2 does not hold.

No Interactions Detected.

- **Scenario 5:** OCS (A) and CFBL (B)

Stub #	1	2	3	4	5	6	7	8
OCS	default	OCS Post-Dial Plug-in	Calling Number delivery Idle plug-in	default	default	default	None	None
CFBL	default	default	default	default	default	Busy CFBL	Busy Setup CFBL	Busy Disconnection CFBL
OCS ⊕ CFBL	Default	OCS Post-Dial Plug-in	Calling Number delivery Idle plug-in	default	default	Busy CFBL	Busy Setup CFBL	Busy Disconnection CFBL

There is no *ng* in the resulting behaviour. However OCS affects the post-dial stub (stub 2) whereas CFBL affects the stubs 6, 7 and 8.

According to the filtering rule 2 a **FI could occur.** The interaction is that the call is not blocked when the call is forwarded to third party C, which is in the screening list of A.

The same user subscribes to two features:

- **Scenario 9:** CND (B) and CFBL (B)

Stub #	1	2	3	4	5	6	7	8
CND	default	default	Calling Number delivery Idle plug-in	default	default	default	None	None
CFBL	default	default	default	default	default	Busy CFBL	Busy Setup CFBL	Busy Disconnection CFBL
CND $\oplus$ CFBL	default	default	Calling Number delivery Idle plug-in	default	default	Busy CFBL	Busy Setup CFBL	Busy Disconnection CFBL

No *ng* in the resulting behaviour and filtering rule 2 does not hold.

No Interactions Detected.

A third party is the feature subscriber and B arranges to forward/route the call to C

- **Scenario 10:** <u>CFBL (B) & CND(C)</u>

CFBL = [default, default, default, default, default, Busy CFBL, Busy Setup CFBL, Busy Disconnection CFBL]

CFBL SC sub-vector:

CFBL_Refined = [Idle CFBL_Ref, Idle Setup CFBL_Ref, Idle Disconnection CFBL_Ref, default, none, none]

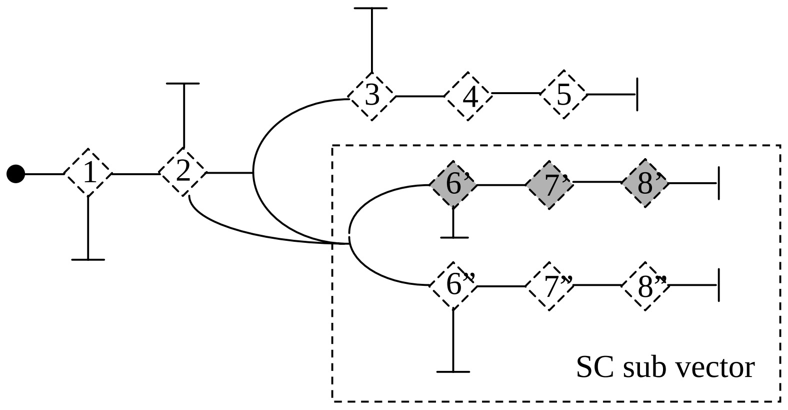


Figure 32: CFBL (B) SC sub vector

CND_Refined = [Calling Number delivery Idle, default, default, default, None, None]

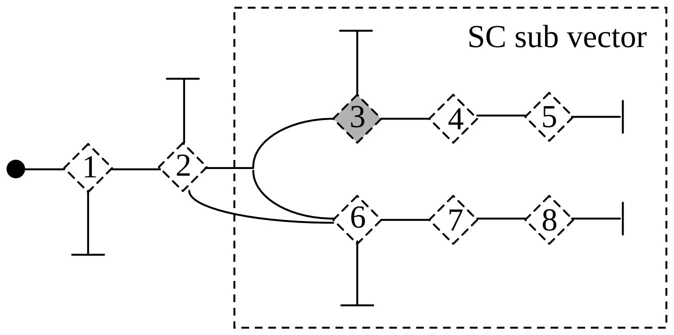


Figure 33: CND (C) SC sub vector

CFBL_Refined ⊕ CND_Refined = [**ng**, Idle Setup CFBL_Ref, Idle Disconnection CFBL_Ref, default, default, none, none]

There is *ng* in the resulting behaviour. **FI occurs.** Display Conflict: The number is not displayed at C.

- **Scenario 14:** CFBL (B) & CFBL(C)

The way CFBL is defined in the contest prevents the forwarding loop because CFBL as designed tests the forwarded-to line for busy and returns LineBusyTone if it is busy. CFBL is deactivated at C.

4.6.1 Results

We have prepared UCMs for the following nine features: Originating Call Screening (OCS), Terminating Call Screening (TCS), IN Free Routing (INFR), Call Forwarding Busy Line(CFBL), IN Teen Line (INTL), Call Number Delivery (CND), IN Freephone Billing (INFB), IN Call Forwarding (INCF) and IN Charge Call (INCC).

Table 3 shows the filtering results.

	OCS		**TCS**		**INFB**		**INCC**		**INTL**		**CND**		**CFBL**		**INCF**		**INFR**	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
OCS	-	-	-	1	-	3	1	-	3	-	-	3	-	2	-	2	-	2
TCS			-	-	3	-	-	1	-	3	3	-	3	2	3	2	3	2
INFB					-	-	-	1	-	3	3	-	3	1	3	1	3	1
INCC							-	-	1	-	-	3	-	2	-	2	-	2
INTL									-	-	-	3	-	3	-	3	-	3
CND											-	-	3	1	3	1	3	1
CFBL													-	3	1	1	1	1
INCF															-	1	1	1
INFR																	-	1

(a) : Same user (1) FI occurs

(b) : Different users (2) FI could occur

(3) FI never occur

(-) Useless scenarios: only one feature is active

Table 3: Filtering Results

Statistics & Discussion:

39 Scenarios (from 90 possible scenarios. These 90 scenarios represent all possible combinations of feature pairs among 2 and/or 3 different users) are discarded before the filtering process because they are useless scenarios. 51 scenarios remain to be investigated.

19 Scenarios (about 37 %) lead to a Feature interaction.

9 Scenarios (about 18%) need more investigation. (FI could occur)

23 Scenarios (about 45%) are safe scenarios (Interaction free).
The number of combinations with verdict (1) FI occurs and (3) FI never occur is 42, which is almost 82 % of all the combinations. That is, 82% of all scenarios can be filtered by the proposed method in an inexpensive way.

4.7 Comparison between our method and the method presented in [29]

The filtering methods presented in [50] and in this chapter have similarities and differences:

Similarities

- Both methods use the same UCM call model to describe features.
- The feature composition mechanism is slightly different: In [50] the feature composition operator operates on matrices while it operates on vectors in our method.
- They use the same filtering rule 1 to detect non-determinism.
- Neither method covers all possible features because they are based on the same UCM model.

Differences

- The filtering method described in this thesis is a “feature oriented method” since each feature is described as a UCM, which contains all involved users. However the filtering proposed in [50] is user oriented in the sense that the composition takes care of every single user scenario.
- Our filtering rule 2 derives from the way the service is decomposed in UCM and detects inconsistent state changes. While [50] detects the scenario changes after the feature composition. One of the theorems introduced in [50] states that if no user’s scenario is changed after feature composition there is no feature interaction.

4.8 Conclusion and Limitations

We have developed a method for feature interaction filtering that uses information that is available at the design stage of a telephone system.

This method allows the designer to localize where interactions could occur during a call and facilitates the further detection by taking out the interaction-free scenarios.

The method leads to certain results concerning existence or nonexistence of interactions, however in order to perform the filtering our structural model should be followed.

Another limitation of our method is that the model used for filtering does not cover all possible features. Features that handle more than one communication leg are not covered. Features like TWC (Three Way Calling) and CW (Call Waiting) deals with more than one communication leg. To be able to describe these features, a new model that describes more than one communication leg should be considered. As a sketch of solution, we propose a UCM model where each call leg (for instance AB between A and B, AC between AC) is described in a separate stub. The triggering event, which attempts to introduce a new call leg, would be described out of these call stubs representing involved call legs. We should be able to go from one call leg to another when the feature-triggering event is triggered. Figure 34 gives a sketch of what such model looks like, where stubs 1 and 2…etc describe the actions performed within each single call leg.

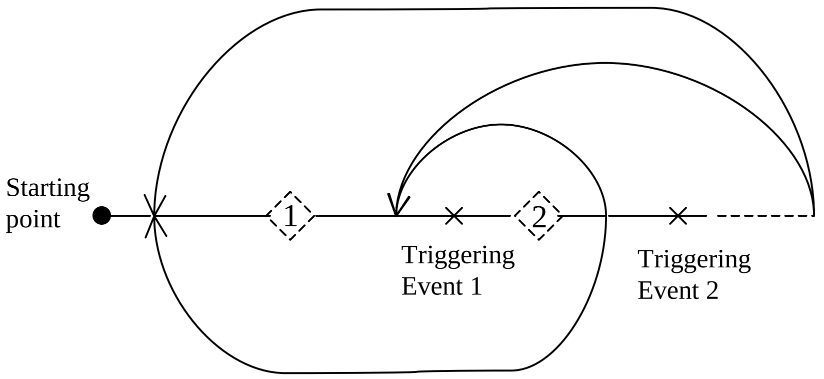


Figure 34: Sketch of multi-leg call Model

Chapter 5

Specifying Features Using LOTOS

In this chapter, we give an overview of the LOTOS specification language and of its main operators.

Our main objective in specifying the system model and features in LOTOS is to provide a specification that can be used for detecting feature interactions.

5.1 Introduction

LOTOS (Language of Temporal Ordering Specification) (ISO 8807,1989) was developed by the FDT experts of the working group ISO/TC97/SC21/WG1 during 80's. It is a specification language developed for the formal description of the various elements of the OSI (Open System Interconnection) architecture such as services and protocols. Nowadays, the LOTOS application area has been extended to cover some other domains such as hardware [42] and telephony [43][18].

The basic idea of LOTOS is that systems can be specified by defining the temporal relations among the interactions that constitute their externally observable behaviour (ISO 8807, 1989). LOTOS is made up of two components:

(i) A data type component, which is based on the formal theory of algebraic abstract data types ACT ONE (Ehrig and Mahr, 1985) [15]. It deals with the description of data structures and value expressions.

(ii) A control component, in which the external observable behaviour of the system is described. It is based on Milner's Calculus of Communicating Systems (CCS) [61], which includes the concepts of parallel processes that communicate through a synchronization mechanism. However the concept of multi-way synchronization is derived from Hoare's CSP [66].

A number of excellent LOTOS tutorials exist in the literature [41][11][66]; therefore, we limit ourselves to a very brief overview of the language and of its use in the context of our research.

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

5.2 LOTOS Abstract data types

In LOTOS, the representation of values, value expressions and data structures are derived from the algebraic specification method ACT ONE. The properties and operations of data are defined without any indication about how these data are represented and manipulated in memory. In addition, LOTOS provides features such as the use of a library of predefined data type, extensions and combinations of already existing specifications, parameterization and actualization of specifications, and renaming of specifications, in order to facilitate the specification of systems with a large number of operations, equations and complex data types.

A data type definition in LOTOS consists 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 define the domain name of the data. **opns** define the formats of operations on the data. **eqns** provide a means to define the semantics of operations.

Consider the following type definition of the users directory numbers:

```
type Number  (* define the type name *)
is Boolean, NaturalNumber (* list other sorts used to construct this data type *)
(* Signature *)
sorts number (* define the sort name *)
opns (* specify the format of operations *)
        null, A, B, C, D        :→ number (* Constants *)
        _eq_                    : number, number → Bool
        _ne_                    : number, number → Bool
        to_nat                  : number → Nat

        (* List of equations *)
        eqns    forall n1, n2:number

                ofsort nat
                        to_nat(A) = 0;
                        to_nat(B) = Succ(0);
                        to_nat(C) = Succ(Succ(0));
                        to_nat(D) = Succ(Succ(Succ(0)));
                        to_nat(none) = Succ(Succ(Succ(Succ(0))));

                ofsort Bool
                        n1 eq n2 = to_nat (n1) eq to_nat(n2);
                        n1 ne n2 = not (n1 eq n2) ;
 endtype
```

The signature of the type Number, identified by the keyword **sorts**, includes the sort number and the operations null, A, B, C, D, eq, ne and to_nat. The operations null, A, B, C, D result in five elements of the sort number, eq and ne define respectively the equality and inequality between variables of this type, and to_nat is the operation which maps a variable into a natural number (the type defining natural numbers is NaturalNumber and is already defined in the abstract data type library). This mapping is used to define the semantics of eq and ne operations, as described below.

The first five equations of sort nat define the images of the defined variables, null, A, B, C and D by the operation to_nat, then, to each variable of sort number corresponds a value of sort Nat (defined in the type NaturalNumber). The equation of sort Bool then defines the equality and inequality between two values of sort number. They are equal, respectively not equal, if their images by the operation to_nat are equal, respectively not equal. The operations eq and ne are already defined in the library type NaturalNumber.

5.3 The Control component

The behaviour component is the part of a specification that deals with the description of the system behaviour. In this part, a system is modeled as a collection of processes interacting with each other.

5.3.1 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 behaviour expression. A behaviour expression is built by combining LOTOS actions by means of operators and possibly instantiations of other processes.

The syntax of a process definition is of the form:

Process process_name [gate_list] (formal_parameter_list) : functionality
    < Behaviour expression>
endproc

In addition to the set of observable gates and to the behaviour 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 reusability.

5.3.2 LOTOS Action

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

Gate	Event1	Event2	Optional Predicate
G	? Get:Type	!Put	[Get <> 0]

As an example, consider the following two actions:

1. OffHook ?caller: number

The action occurs at gate "OffHook", which expects from the environment a value of sort number for the caller number.

2. DialTone !caller

The action occurs at gate "DialTone" and offers the value of the caller number (already assigned in the previous action) to the environment.

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

5.3.3 LOTOS Behaviour Expressions

The following are the basic behaviour expressions:

- *Inaction*: **stop**

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

- *Successful Termination*: **exit**

  It indicates a normal termination of a behaviour, i.e. a process has successfully performed all its actions. The keyword **exit** is also used in process definitions 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 a process cannot perform an **exit**, its the functionality is **noexit**.

- *Process Instantiation*: Process_Name[gate_list](actual parameter_list)

  The instantiation of a LOTOS process is equivalent to the invocation of a procedure in a programming language (such as Pascal). It can occur in the behaviour expression of other processes or in the behaviour expression of the process itself.

5.3.4 LOTOS Operators

It is possible to construct more complex expressions from those mentioned in section 5.3.3 by using LOTOS operators:

- *Action prefix operator:* a **;** B

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

For example, when a user (the caller) picks up the phone to make a call, she/he will get a tone. This can be expressed by a behaviour expression composed of two actions:

```
OffHook ?caller: number ;
DialTone !caller ;…
```

- *Choice Operator:* B1 [] B2

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

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

```
OffHook ? caller:number ; DialTone !caller;
        (Dial !caller ? called: number ; …
        [ ]
        OnHook !caller ; stop )
```

- *Enabling:* B1>> B2

  The enable operator >> has a similar function as the action prefix operator but is used to express sequential composition of two behaviour expressions. B1 has to terminate successfully (**exit**, see section 5.3.3) in order for B2 to be executed.

- *Disabling*: B1**[**>B2

  The disable operator **[>** is used to express situations where B1 can be interrupted by B2 during normal functioning. For example, a normal processing of a call could be interrupted at any point if the caller hangs up. This could be expressed by the behaviour expression:

```
(OffHook ? caller:number ;
 DialTone !caller ;
 Dial !caller ? called: number ;
  …
) [>   OnHook !caller ; …
```

- Guarded Behaviour: [P] → B

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

```
[called NotIn Busy] → StartRinging !called! caller ; …
```

- *Interleaving operator:* **B1 ||| B2**

  If B1 and B2 are in interleaving, they can perform their actions independently of each other. This operator expresses the concept of parallelism between behaviours where no synchronization is required. For example, a user dials a number, which is idle, then two actions in either order can take place: the phone rings at the called party (StartRinging) and the caller gets a Ringback tone (StartAudibleRinging).

```
OffHook ? caller:number ;
DialTone !caller ;
Dial !caller ? called: number ;
    (
    StartRinging !called! caller ;
    exit
            |||
    StartAudibleRinging !caller !called ;
    exit
    ) >>  ( Offhook !called ;…
```

- *Parallel Composition:* **B1|[g_1,…,g_n]| B2**

  The parallel composition of B1 and B2 on the gate list $g_1,\ldots,g_n$ expresses the fact that B1 and B2 behave independently, with the exception that they must synchronize on the gates $g_1,\ldots,g_n$, which means that processes B1 and B2 must participate in the execution of every action defined with a gate name g_i, $i \in \{1,\ldots,n\}$. Then the interleaving operator, explained above, can be defined as parallel composition on an empty gate list. Therefore ||| and || (see below) are special cases of this operator.

  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 list of events offered with the actions match, and the predicates (if any) are satisfied. The list of events of two actions "match" if the following conditions are satisfied:

  1) The numbers of events in the two actions match.
  2) An event in one action offers (!) the same value or accepts (?) a value of the same sort.

We will give now an example of use of this operator:
Consider the following two processes: CFBL_feature and INTL_feature, which represent the partial specification of the features: Call Forwarding Busy Line feature and IN Teen Line.

In order to detect the interactions between these features, we synchronize them on their common gates: OffHook, DialTone, Dial, OnHook,…etc. (the feature interaction detection method is discussed in Chapter 6)

```
Process CFBL_feature[Offhook,..] (B_State, …) :noexit:=
OffHook !A;
DialTone !A;
(
 Onhook !A; stop
   [ ]
  (
   Dial !A!B;…
  )
)
…
endproc
```

```
Process INTL_feature[Offhook,..] (B_State, T, …) :noexit:=
OffHook !A;
(
        Ask_For_PIN !A;
        Dial_PIN !A ? x:PIN;
 (
     [x eq Invalid_PIN] →      PlayAnnoucement!A;
                               Onhook! A; stop
                         [ ]
      [x eq Valid_PIN] →       DialTone !A;
                               Dial !A!B; …
                         )
)
…
endproc
```

Let us compose these two processes as follows:

The synchronization between these two processes is described as:

```
CFBL_feature[Offhook,..] (B_State,…)
|[OffHook,DialTone,OnHook,Dial]|
INTL_feature[Offhook,..] (B_State, …)
```

The processes will execute as follows. They start by synchronizing on gate "OffHook" offering the same value, which is A. The process CFBL_feature is now blocked waiting for the "DialTone" action from INTL_feature. However the actions "Ask_For_PIN !A", "Dial_PIN !A ?x:PIN", which are not in the synchronization list, are executed. Then depending on whether the predicate [x eq Valid_PIN] is true (in the case where the user enters a valid PIN:Personal Identification Number or not) the "DialTone" or the "OnHook" action are executed.

- *Full Synchronization:* B1 **||** B2

  The full synchronization of B1 and B2 is a parallel composition in which B1 and B2 must synchronize on all their gates. This is also a special case of the interleaving operator, where the set {g_1, …, g_n} is the set of all the gates of the two processes.

- *Hiding operator* : **hide** g_1, …, g_n **in** B
  Used to hide actions synchronizing on gates (g_1, …, g_n), which become internal (i.e. they become **i**) for the environment. Thus, hidden actions cannot synchronize with the environment.

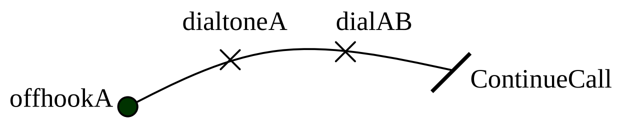
5.4 Mapping From UCM to LOTOS

We have chosen LOTOS in order to narrow the gap between the requirement notation Use Case Maps and the executable model. We are going to use the UCM description of the features discussed in Chapter 3 to obtain the corresponding LOTOS specification.

We propose a mapping between the UCM notation and the corresponding LOTOS operators. We don’t cover all the UCM components but only the ones we use in our UCMs. A method for translating UCM into LOTOS was outlined in [22]. However, a complete translation is difficult and is the subject of ongoing work. In this research, we used a manual translation based on the guidelines that follow, which are limited to a subset of the UCM notation.

Translation Guidelines:

- "Start points" and "end points" are usually represented by LOTOS gates.
- UCM responsibilities are also represented as gates, sometimes with additional message exchanges.
- LOTOS gates representing UCM responsibilities and channels that are not observable by users are hidden through the **hide** operator.
- **Sequence**: The sequence is a very common pattern that can be found in all UCMs. In following example, we consider a partial UCM containing three consecutive responsibilities. We obviously see that actions are directly mapped onto LOTOS gates and the sequence is mapped into the prefix operator "**;**".

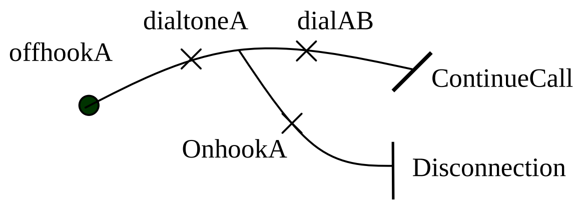


```
P :=      Offhook !A ;
        (
          dialtone !A;
          dial !A !B;
          continueCall; stop
        )
```

Figure 35: Sequence

- **OR-Fork**

  Suppose the user A has a choice between dialing a number and hanging up.

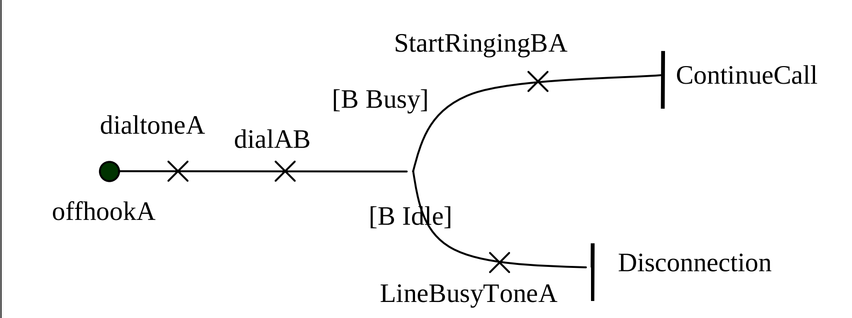


```
P :=      Offhook !A ;
       (
         dialtone !A;
            (
              dial !A !B;
              ContinueCall;  stop
                      []
              Onhook !A;
              Disconnection; stop
             )
        )
```

Figure 36: OR-Fork

Figure 36 presents such a (simplified) UCM. It shows two exclusive paths that will never join. The LOTOS choice operator ([]) is used in the interpretation of the OR-Fork.
The choice construct allows multiple alternatives (more than two options) and this is reflected in the LOTOS code accordingly. When an OR-Fork occurs, we choose between the continuation of the sub UCM (ContinueCall in figure 36) and the path segment (Disconnection).

We also have the possibility to add guards to the all segments. Figure 37 illustrates the case where the subsequent actions depends on the Called party's (B) state.

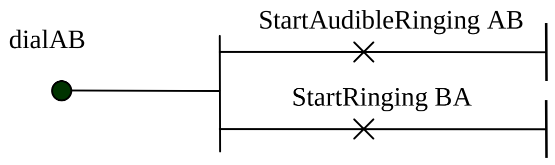


```
P :=       Offhook !A ;
        (
          dialtone !A;
          dial !A !B;
              (
               [B IsIn Busy] →
                  ContinueCall;
                  Stop
                         []
                 [B IsIn Idle] →
                   Disconnection;
                   Stop
                 )
          )
```

Figure 37: Guarded Behaviour

- **AND-Fork:**

Many actions could take place concurrently. In the next example (figure 38), when user A dials B's number, while B is Idle, two actions take place concurrently: "StartAudibleRinging AB" and "StartRinging BA". This is represented with two concurrent paths after an AND-Fork. The LOTOS interleaving operator is used here to represent that two tokens follow the two paths concurrently. The AND-Fork adds new concurrent path segments, and we may have more than two exiting paths (thus at least one new Path segment), without guards.

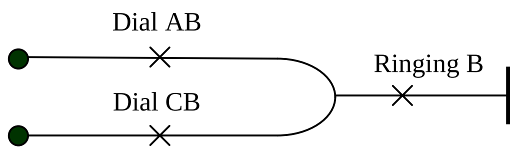


```
P :=      dial !A !B;
            ( StartAudibleRinging !A !B;
              stop
                    |||
              StartRinging !B !A;
              Stop
            )
```

Figure 38: AND-Fork

- **OR-Join**

An OR-Join merges two (or more) overlapping paths. These are two exclusive paths that will join. Figure 39 presents such a (simplified) UCM. Again the LOTOS choice operator ([]) is used in the interpretation of the OR-Join. The choice construct allows multiple alternatives (more than two options) and this is reflected in the LOTOS code accordingly.



```
P :=      (Dial !A !B;
            exit
                  []
           Dial !A!C;
             exit)
             >>  Ringing !B; stop
```

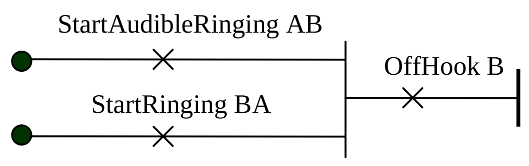
Figure 39: OR-Join

- **AND-Join**

Many actions could take place concurrently. For example, two actions take place concurrently: "StartAudibleRinging AB" and "StartRinging BA".

Again the LOTOS interleaving operator is used to represent that two tokens follow the two paths concurrently and synchronize on a further responsibility. The AND-join gives the possibility to add another path after concurrent responsibilities.

Figure 40 illustrates a case where the two events: "StartAudibleRinging AB" and "StartRinging BA" take place in either order and they are followed by the action "offhook B" (where the user picks the phone to answer the incoming call).



```
P :=      (  StartAudibleRinging !A !B;
              exit
                   |||
              StartRinging !B !A;
              exit
             ) >>  offhook !B; stop
```

Figure 40: AND-Join

- Abstract data types are used to represent databases, operations, and conditions (LOTOS guard expressions)
- UCM components are represented as processes synchronized on their shared channels/gates.
- Components with stubs have sub-processes, one for each stub. The plug-in is mapped to LOTOS according to the rules defined above.

Note: This mapping is done manually.

5.5 The LOTOS Specification

In this section, we describe the LOTOS specification of services that were given in the FI contest by describing their main abstract data types and the structure of their specification.

5.5.1 Design of the Abstract Data Types

In order to perform the detection and verification mechanisms, we need first to determine the categories of data and the rules associated with them. Telephone systems operate with a limited set of data and rules: phone numbers, signals, features names, and databases of various types (Messages, Signals, PIN Number, Billing, Display).

Our specification is at a high level of abstraction since we are only interested in the observable behaviour of the system and not in the implementation details. However, these could always be refined when the implementation details become relevant.

Table 4 describes the main data types that have been specified; some of the specified data were not used for feature specification but are still required to complete the feature interaction detection procedure. Those serving for the detection procedure will be discussed in detail in Chapter 6.

Data Type	Meaning
Number	Defines the telephone numbers of the users
Feature	Defines the list of the features described in the specification
PIN	Defines the possible Personal Identification Numbers required for some features
Message	Defines the messages that can be played to the users such as AskForPIN
State	Defines the state of the numbers: Ringing, linebusy…etc.
ConnectionRecord	Defines the allowed and the forbidden connections
ConnectionRecord_set	Defines the set of connections
BillingRecord	Defines the billing data
BillingRecord_set	Defines the set of billing data
SignalRecord	Defines signals received by users
SignalRecord_set	Defines the set of Signals
Violation	Defines the different kinds of violation (e.g. violation of connections, Inconsistency of signals, violation In Billing…etc.

Table 4: The main Abstract data Types

- **Type Feature**

The type Feature describes the operations OCS, TCS, CFBL...etc, eq, ne and h. The operations OCS, TCS, CFBL…etc result in nine elements of the sort feature, "eq" and "ne" define respectively the equality and inequality between two features, and "*h*" is the operation that maps a feature into a natural number.

```
type Feature is NaturalNumber
sorts Feature
opns
        OCS,            (*! constructor *)  (* Originating Call Screening *)
        TCS,            (*! constructor *)  (* Terminating Call Screening *)
        CFBL,           (*! constructor *)  (* Call Forwarding Busy Line  *)
        CND,            (*! constructor *)  (* Call Number Delivery *)
        INFB,           (*! constructor *)  (* IN Free Billing *)
        INTL,           (*! constructor *)  (* IN Teen Line *)
        INFR,           (*! constructor *)  (* IN Free Routing *)
        INCF,           (*! constructor *)  (* IN Call Forwarding *)
        INCC,           (*! constructor *)  (* IN Call Charging *)  :→ Feature

        _eq_, _ne_              : Feature,Feature → Bool
        h                       : Feature → Nat

eqns
        forall F1, F2: Feature
        ofsort Bool
                F1 eq F2 = h(F1) eq h(F2);
                F1 ne F2 = h(F1) ne h(F2);

        ofsort Nat
                h(OCS)  = 0;
                h(TCS)  = Succ(0);
                h(CFBL)= Succ(Succ(0));
                h(CND) = Succ(Succ(Succ(0)));
                h(INFB) = Succ(Succ(Succ(Succ(0))));
                h(INTL) = Succ(Succ(Succ(Succ(Succ(0)))));
                h(INFR) = Succ(Succ(Succ(Succ(Succ(Succ(0))))));
                h(INCF) = Succ(Succ(Succ(Succ(Succ(Succ(Succ(0)))))));
                h(INCC)= Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(0))))))));
endtype (*  Feature *)
```

Note: Operations h, _eq_ and _ne_ are similar to operations to_nat, _eq_ and _ne_ described in detail in section 5.2.

The LOTOS specification of the remaining data types is provided in the Appendix.

5.5.2 Feature Specification

The stubs are represented as processes in LOTOS and sequences of them are represented by using the LOTOS operator enable ">>" between them (with "accept" if data has to be transferred to the following process)

```
Process Feature1[…]:=
P[…] >> Q[…] >> Z[…]
```

Process P, respectively Q, should contain at least one "exit" to fire the process Q, respectively Z. In addition to the fact that the features synchronize on the "exit" operator they have to synchronize on their common gates and these gates could belong to different stubs. Considering this fact, we chose to flatten all the stubs of the UCM model. By doing this we solve the synchronization constraints and offer to the designer more flexibility to specify the features.

This is the LOTOS specification of Call Number Delivery obtained by a direct translation from the CND UCM defined in Chapter 4 section 4.4.2. Other LOTOS specifications of some telephony features are presented in the Appendix.

LOTOS specification of CND (Call Number Delivery):

Scenario where B subscribes to CND and A calls B:

```
process CND_feature[Offhook, DialTone, Onhook, Dial, StartRinging, StartAudibleRinging, StopRinging,
StopAudibleRinging, LineBusyTone, LogBegin, LogEnd, Display_number, SetLastDisplay, Disconnect,
Connection, Billing, Signal, Display](B_State: State) :noexit:=

Offhook ! A ;
Dialtone ! A ;
( onhook !A; stop
        []
Dial !A!B;
        (
        [B_State eq busy] →     LineBusyTone!A ;
                                Onhook!A ;stop
                [ ]
        [B_State eq idle] →     StartRinging !B!A ;
                                Display_number!B!A;
                                SetLastDisplay !B!A;
                                StartAudibleRinging !A !B;

                                ( Onhook !A;
                                StopRinging !B!A;
                                StopAudibleRinging !A!B;stop

                                        [ ]
                                Offhook !B;
                                StopRinging !B !A;
                                StopAudibleRinging !A !B;
                                LogBegin !A!B!A;
                                (
                                Onhook !A;
                                Disconnect !B!A;
                                LogEnd !A!B;
                                Onhook !B;
                                stop
                                        [ ]

                                Onhook!B;
                                Disconnect !A!B;
                                LogEnd !A!B;
```

```
                                        Onhook !A;
                                        stop
                                        )
                              )
                    )
)
endproc
```

Chapter 6

Feature Interaction Detection Method

In this section, we propose a Feature Interaction Detection Method at design stage, based on the use of LOTOS, that uses extensively Abstract Data Types (ADT) to detect system inconsistencies.

6.1 Formal Definition of Feature Interaction

Logical interaction between two or more features occurs when one or some of the requirements or assumptions, that must be satisfied in the network, is violated.

Therefore, we develop a method based on expressing the feature requirements as properties and we say that interactions occur when these properties becomes contradictory, introducing inconsistencies in the system description, which is its formal specification.

We use a modified formal definition of “Feature Interaction” introduced in [53]:

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

Let FP_1, FP_2, …, FP_n (Feature Properties) be n formulas expressing respectively the feature properties of F_1, F_2, …, F_n and let $N \models P$ denote that a system specification N satisfies formula P, i.e N is a model of P [53].

We say that there is an interaction between features F_1, F_2,…, F_n if:
$\forall i,\ 1 \leq i \leq n,\ S \oplus F_i \models FP_i$, but
$\neg\, (S \oplus F_1 \oplus F_2 \ldots \oplus F_i \models FP_1 \wedge FP_2 \wedge .. \wedge FP_n)$

We will usually consider the case where n=2 since most interactions reveal themselves in contexts where two features only are active [56].

Our feature interaction detection method doesn’t address all possible kinds of interactions between features. We limit ourselves to the following types of interaction:

- Connection violations
- Inconsistency of signals given to users
- Incorrectness of Billing
- Inconsistency in the display function

Example of interaction: Connection violation between OCS and CFBL
Consider the following scenario: User A is an OCS subscriber and user C is in his screening list. User B is a CFBL subscriber and C is the number to which the incoming calls are forwarded when busy.

OCS property is : PFocs : ¬Connection(A,C)
CFBL property is: PFcfbl : Connection(A,C)

These two properties could not coexist in the same system, which is a feature interaction.
How feature properties are derived is treated in detail further in this chapter.

6.2 Feature Interaction Detection Method

Our detection method will use feature descriptions from the user's point of view. This implies some very important advantages: Firstly, it can be applied at a very early stage of the design of new features. Secondly, the descriptions can be quickly created. Potentially, the feature specification will be created using UCMs. Each UCM will represent a single feature.

Combining the features and detecting possible interactions between them will be carried out automatically. Hence, the more complex task is not carried out by the designer but by some automatic detection mechanism and tools.
Figure 41 describes the feature interaction detection process.

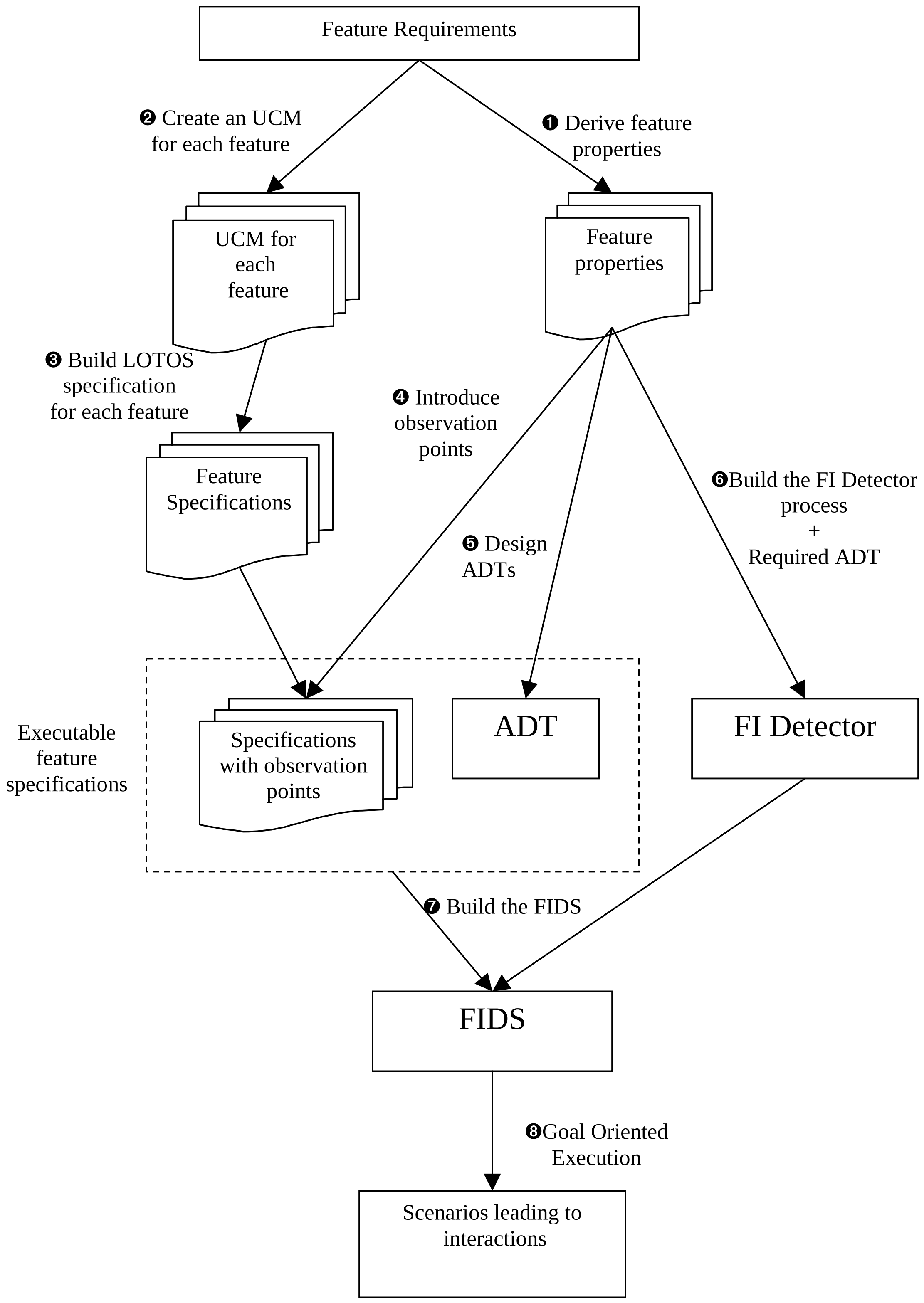

Feature Requirements
❷ Create an UCM for each feature
❶ Derive feature properties
UCM for each feature
Feature properties
❸ Build LOTOS specification for each feature
❹ Introduce observation points
Feature Specifications
❻Build the FI Detector process + Required ADT
❺ Design ADTs
Executable feature specifications
Specifications with observation points
ADT
FI Detector
❼ Build the FIDS
FIDS
❽Goal Oriented Execution
Scenarios leading to interactions

Figure 41: Feature Interaction Detection cycle

Given a set of feature requirements, described using natural language or visual notations such as “Chisel Diagrams”, feature properties are derived (step 1) and a UCM for each feature is created (step 2); These UCMs are then mapped to LOTOS to obtain a LOTOS specification for each feature (step 3). Step 4 consists of introducing observation points into the specification for detection purposes. In order to run the LOTOS specification we need to design abstract data types (step 5). The Feature Interaction Detector (FID) process is built in step 6.
By combining together these three pieces: Feature Specifications + ADT + FID (step 7), we obtain the Feature Interaction Detection System (FIDS).
The last step (step 8) consists in running the FIDS and results in generating scenarios leading to interactions using the Goal Oriented Execution Tool.

These steps will be explained in detail in the following sections.

6.2.1 Deriving Features Properties

We believe that how to derive the properties of features and how to represent them are the key issues of a good feature interaction detection method.

Nowadays, communication services and features are commonly described using an amalgam of informal operational and declarative descriptions, tables, and visual notations such as Use Case Maps and Message Sequence Charts (MSCs) [33]. However using an informal representation can lead to different understandings of a given feature. For example, the informal requirements 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 believe that the most clear and unambiguous manner to express feature properties is to use logic.

We choose to formalize a feature using declarative transition rules. Such rules consist of a precondition, a trigger event, and a post condition. The *pre* and *post* conditions of declarative transition rules are formulated in simple logic, which is a restriction of ordinary first order logic. The trigger event is also expressed in simple logic.

Declarative transition rule: Precondition $\xrightarrow{TriggerEvent}$ PostCondition

Features are described independently of other features. This means that the specification of a specific transition is done solely in consideration of the given feature. There is no consideration of potential interactions with another feature at this stage. This is important especially in the context of a multi-vendor environment where characteristics of a feature of a vendor may not be known to another vendor. Also, from a design point of view, this characteristic allows the feature provider to add new features without having to understand their behaviour in combination with other features.

The intented interpretations of the predicates should be clear. For instance Ringing (B,A) means that the telephone is ringing at B when A is calling B. The feature descriptions always depend on the basic service, to which actions like offhook(x), dial(x) and ringing(x) belong.

When a feature is activated its corresponding rule is fired. That is, the preconditions are met and the trigger event takes place. The post condition represents the resulting behaviour. We consider these post conditions as "**feature properties**".

As mentioned in the previous section, we are going to investigate only interactions dealing with connections establishment, Billing Records, Signal and Display. The derived feature's properties listed here are related to these four issues and are based on the best of our knowledge and on our practical experience with FI detection.

These are examples of feature rules and properties:

1. Call Number Delivery (CND) rule:

CND (B) $\xrightarrow{Ringing(B,A)}$ Display(B,A)

Where CND (B) means that B is a CND subscriber.

CND property is: Display (B,A), which means that A's number is displayed at B's side.

2. IN Freephone Billing (INFB) rule:

INFB (B) $\wedge$ Ringing (B,A) $\xrightarrow{offhook(B)}$ Billing (B,AB)

Where INFR (B) means that B is an INFB subscriber.

INFB property is: Billing (B, AB), which means that B is charged for the call leg between A and B.

3. Originating Call Screening (OCS) rule:

OCS (A) $\wedge$ InOCS_list(A,C) $\xrightarrow{dial(A,B)}$ $\neg$ Connection(A,C)

Where OCS (A) means that C is an OCS subscriber and InOCS_list(A,C) means that C is in the OCS screening list of A.

OCS property is: $\neg$Connection(A,C) which means that A and C should not be connected.

4. Call Forwarding Busy Line (CFBL) rule:

CFBL (B,C) $\wedge$ Busy (B) $\wedge$ $\neg$Busy (C) $\xrightarrow{dial(A,B)}$ Connection(A,C)

Where CFBL (B,C) means that B is a CFBL subscriber and C is the number to which the incoming calls are forwarded when he is busy. Busy(B) means that user B is busy and $\neg$Busy (C) means that user C is not busy.

CFBL property is: Connection(A,C) which means that A and C are connected.

6.2.2 UCM creation and Mapping to LOTOS

A UCM for each feature is created and translated to LOTOS using the rules introduced in Chapter 5. (Step 2 and 3 in figure 41)

6.2.3 Introducing observation points in the specification

In order to detect possible interactions we should provide the process FID (Feature Interaction Detector) with the relevant data during the execution of the specification. This data should be sent from the features to the FID. In order to do it, we introduce "**observation points**" in the specification of each feature. These observation points are LOTOS gates with parameters (data). Each gate is responsible of carrying data leading to a different kind of Feature Interaction.

We distinguish four observation points:

- ***Connection***: Used to communicate information about the allowed and forbidden connections between users. This data is used to detect connection violation.

- ***Signal***: Used to communicate the signals received by users. This data is used to detect signal conflicts. For example, a user should not receive two different signals at the same time.

- ***Billing***: Used to communicate billing information. This data is used to detect incorrectness of billing information. For example, we could not have two different users charged for the same leg: Billing (A, AB) and Billing (B, AB).

- ***Display***: Used to communicate the display information. This data is used to control display violation. Example: an absence of display on a CND subscriber.

The FID and the features should synchronize on these gates. These detection points are introduced according to the feature properties.

In the following we discuss how to introduce the observation points in the two cases:

1) When the feature behaviour is part of the basic service behaviour (POTS: described as default plug-ins in Chapter 4)
2) And when the feature behaviour modifies the basic service behaviour (Not Default plug-ins).

6.2.3.1 Information related to connection

Depending on the two cases mentioned above, we provide two different rules to insert observation points dealing with connection.

- Feature behaviour is part of the basic service (default plug-ins):

When the phone starts ringing at the called party it means that the connection attempt succeeded. So the rule will be: Insert in the specification a LOTOS action called "Connection" immediately after a "StartRinging" gate with the parameters: the name of the feature, the two users involved in the connection and a boolean value indicating whether the connection is allowed or not.

Example: Consider the following LOTOS sketch of a specification of a feature F that has default behaviour.

```
Offhook !A;
Dial !A !B;
StartRinging! A !B;
Connection ! F !A !B !True;
…
```

- Feature behaviour modifies the basic service behaviour:

The insertion of the connection gate depends on the presence of the predicate connection in the feature rule post condition. When the feature rule has the predicate connection as post condition, the action "connection" is inserted after a state where the precondition is satisfied and the trigger event is fired.

For example, consider the TCS rule:

TCS (B), InTCS_list(A,B) $\xrightarrow{dial(A,B)}$ ¬ Connection(A,B)

This is a sketch of TCS specification:

```
…
Dial !A!B;
[A IsIn TCS_list] →          Connection !TCS !A !B !false;
                              PlayAnnouncement !A !ScreenedMessageTCS;…
          [ ]
[A NotIn TCS_list] → …(* Default Behaviour *)
```

The connection gate is inserted after the "dial" event and at a state where A is in the screening list of B.

6.2.3.2 Information related to Billing

Depending on the two cases mentioned in section 6.2.3, we provide two different rules to insert observation points dealing with billing.

- Feature behaviour is part of the basic service (default plug-ins)

  The billing starts after the LogBegin action. The rule is to insert in the specification the LOTOS action: "Billing" immediately after a "LogBegin" gate with the name of the feature, the charged party and the two users involved in the connection leg.

Example: Consider the following LOTOS sketch of a specification of a feature F which has a “default idle setup” plug-in.

```
…
StartRinging! A !B;
Offhook !B;
LogBegin !A !B !A;
Billing !F !A !B !A;
…
```

- Feature behaviour modifies the basic service behaviour:

  The feature rule should have the predicate Billing as post condition. The rule is to insert the action "Billing" according to the feature rule i.e. we should look for a state where the precondition is satisfied and the trigger event is fired then we introduce the Billing gate.

For example consider INFB rule:

INFB (B) $\wedge$ Ringing (B,A) $\xrightarrow{offhook(B)}$ Billing (B,AB)

This is a sketch of the INFB specification:

```
…
StartRinging! A !B;
Offhook !B;
LogBegin !A !B !B;
Billing !A !B !B;
…
```

Note: Only the Billing parameters are modified with respect to the The INFB property.

6.2.3.3 Information related to Signals

For signals there is no distinction between default and specific behaviours.

Within the specification insert a LOTOS action: "Signal" immediately after the reception of a signal by a user. The "Signal" gate has the following parameters: The feature name, the signal, user who receives the signal and the other party involved in the signal.

Example: Consider the following LOTOS code from a specification of a feature F:

```
…
StartRinging! A !B;
Signal !F !StartRinging !A !B
…
```

6.2.3.4 Information related to Display

Depending on the two cases mentioned in section 6.2.3, we provide two different rules to insert observation points dealing with display.

- Feature behaviour is part of The basic service (default plug-ins):

Insert in the specification the LOTOS action: "Display" after a "StartRinging" gate with the following parameters: the name of the feature, the called party, the user initiating the call and a boolean value indicating whether there is a display or not. For the basic service, this boolean value is set up to false.

Example Consider the following LOTOS code from a specification of a feature F:

```
Offhook !A;
Dial !A !B;
StartRinging! A !B;
Display ! F !B !A !False;
…
```

- Feature behaviour modifies the basic service behaviour:

The feature rule should have the predicate “Display” as post condition. The rule is to insert the action "Display" according to the feature rule i.e. we should look for a state where the precondition is satisfied and the trigger event is fired then we introduce the Display gate.

For example consider CND rule:

CND (B) $\xrightarrow{Ringing(B,A)}$ Display(B,A)

This is a sketch of the CND specification:

```
…
StartRinging! A !B;
Display !CND !B !A !True;
…
```

6.2.4 Design of the Abstract Data Types

For implementation purposes we have chosen to use LOTOS ADTs to detect interactions. We thus define an operation "CausesViolation" having a Boolean value that indicates whether a violation occurs or not.

These are the abstract data types used to control the detection:

6.2.4.1 Storing Connection information

To store connection information, we defined two types: ConnectionRecord and ConnectionRecord_set.

- **ConnectionRecord**

A connection could take place between two users. The connection record is composed of: the feature name, the caller party number, the called party number and a boolean value to determine whether the connection is allowed or not.

The following is the LOTOS ADT description of ConnectionRecord type with interleaving comments:

type *ConnectionRecord is Number,Boolean*
sorts *ConnectionRecord*
opns *(* specify the format of operations *)*

"ct" is the constructor of the connectionRecord.

ct: Feature, Number, Number, Bool $\rightarrow$ *ConnectionRecord*

"num1" and "num2" are two operations used to extract the user numbers involved in the connection.

num1 : *ConnectionRecord* $\rightarrow$ *Number*
num2 : *ConnectionRecord* $\rightarrow$ *Number*

Bvalue is an operation for extracting the boolean value of a connectionRecord: if the connection is allowed then Bvalue returns true otherwise it returns false.

Bvalue : *ConnectionRecord* $\rightarrow$ *Bool*

eq, and *ne* are two operations used to compare two connectionRecord.

*_eq_, _ne_* : *ConnectionRecord, ConnectionRecord* $\rightarrow$ *Bool*

```
eqns  (* List of equations *)

        forall  N1, N2, N3, N4  :number,
                t1, t2          :ConnectionRecord,
                B1, B2          :Bool
                F, F1, F2       :Feature

        ofsort Number
                num1(ct(F,N1,N2,B1)) = N1;
                num2(ct(F,N1,N2,B1)) = N2;

        ofsort Bool
                Bvalue(ct(F,N1,N2,B1)) = B1;
        ct(F1,N1,N2,B1) eq ct(F2,N3,N4,B2) = ((F1 eq F2) and (N1 eq N3) and (N2 eq N4) and (B1 eq B2))
                                                                        or
                                                   ((F1 eq F2) and (N1 eq N4) and (N3 eq N2) and (B1 eq B2));

        ct(F1,N1,N2,B1) ne ct(F2,N3,N4,B2) = not( ct(F1,N1,N2,B1) eq ct(F2,N3,N4,B2));

endtype (*  ConnectionRecord  *)
```

- **ConnectionRecord_set**

  This set contains all the connections involved in the communication during the execution of the specification. Connections records could be inserted into the connectionRecord_set either during the initialization part, or during the execution of the specification.

The following is the LOTOS ADT description of ConnectionRecord_set type:

type *ConnectionRecord_set is ConnectionRecord*
sorts *ConnectionRecord_sets*
opns

This operation defines an empty set.

{} : $\rightarrow$ ConnectionRecord_sets

The operation “insert” inserts a new connection in the connection set

```
insert                          : ConnectionRecord,ConnectionRecord_sets → ConnectionRecord_sets
```

The operations *_eq_* and *_ne_* are used to compare two ConnectionRecord_set.

```
_eq_,  _ne_                     : ConnectionRecord_sets, ConnectionRecord_sets -> Bool
```

The operation “empty” checks whether the connection set is empty or not.

```
empty                           : ConnectionRecord_sets → Bool
```

The operation “isin” checks whether the connectionRecord is present in the connectionRecord_set or not.

```
isin                            : ConnectionRecord, ConnectionRecord_sets → Bool
```

The operation “CausesViolation_Connection” checks whether a new connection record is inconsistent with the existing data in the ConnectionRecord set.

```
CausesViolation_Connection          : ConnectionRecord, ConnectionRecord_sets → Bool
```

```
Eqns  (* List of Equations *)

        forall   t1,t2           : ConnectionRecord,
                 s               : ConnectionRecord_sets,
                 n1,n2,n3,n4     : Number,
                 b1,b2           : Bool
                 F, F1, F2       : Feature

ofsort ConnectionRecord_sets
        tail (insert(t1,s)) = s;

ofsort Bool
        {} eq {} = true;
        {} eq insert(t1,s) = false;
        insert (t1,s) eq {} = false;
        empty (s) = s eq {};
        isin (t1,{}) = false;
        t1 eq t2 => isin(t1,insert(t2,s)) = true;
        t1 ne t2 => isin(t1,insert(t2,s)) = isin(t1,s);
```

A new connectionRecord is not in conflict with an empty set of connectionRecord.

```
CausesViolation_Connection(ct(F, n1,n2,b1),{}) = false;
```

The operation "CausesViolation_Connection" checks if the insertion of a new connection record conflicts with existing constraints. A connection conflict occurs in two cases:

1. When one feature allows a connection, while another feature deny it. This is described by the following two conditions:

CausesViolation_Connection(ct(F1, n1,n2,b1),insert(ct(F2, n3,n4,b2),s)) =
((F1 ne F2) and (n1 eq n3) and (n2 eq n4) and (b1 ne b2))
or
((F1 ne F2) and (n1 eq n4) and (n2 eq n3) and (b1 ne b2))
or

For example: suppose that the user C is a terminating call screening subscriber and has A in his screening list. The connection set should contain the connection record (TCS, A, C, false). During the execution of the specification, if a connection is established between users A and C (for some reason) the system tries to add the connection record (F, A, C, true) to the connection set. The addition of such connection causes a conflict, which is reported to the user.

2. When two features cause the connection of one user to two different parties. This is described by the following two conditions:

((F1 ne F2) and (n1 eq n3) and (n2 ne n4) and (b1 eq true) and (b2 eq true))
or
((F1 ne F2) and (n1 ne n3) and (n2 eq n4) and (b1 eq true) and (b2 eq true))
or
CausesViolation_Connection(ct(F1, n1, n2, b1), s);

endtype

6.2.4.2 Controlling signals

We defined two types:SignalRecord and SignalRecord_set to store signals sent to the involved users.

- **SignalRecord**

The SignalRecord is composed of the name of the process (the feature) originating the signal, the user receiving the signal, the other party involved in the signal if any.

Example: (CND, StartAudibleRinging, A, B) and (OCS, LineBusyTone, A, none)

The following is a sketch of the LOTOS ADT description of SignalRecord type:

type SignalRecord is Feature, Number, State, Boolean
sorts SignalRecord
opns

“st” is the constructor of the signalRecord.

```
        st                     : Feature, State, Number, Number → SignalRecord
        ….
endtype (* SignalRecord *)
```

- **SignalRecord_set**

  This set controls the signals received by the involved users. It is updated during the execution of the specification.

The following is the LOTOS ADT description of SignalRecord_set type:

```
type SignalRecord_set is SignalRecord

sorts SignalRecord_sets
opns
          ….
```

The operation "CausesViolation_Signal" checks if the insertion of a new signalRecord conflicts with existing signals.

```
          CausesViolation_Signal           : SignalRecord, SignalRecord_sets → Bool

eqns

 forall   t1,t2               : SignalRecord,
          F1,F2               : Feature,
          S1,S2               : State,
          S                   : SignalRecord_sets,
          n1,n2,n3,n4         : Number
          b1,b2               :Bool

          ….
          ofsort Bool
```

A new signalRecord is not in conflict with an empty set of signalRecord

```
                    CausesViolation_Signal(st(F1,S1,n1,n2),{}) = false;


                    CausesViolation_Signal (st(F1,S1,n1,n2),insert(st(F2,S2,n3,n4),s)) =
                              ((F1 ne F2) and (S1 ne S2) and (n1 eq n3))
                                        or
                             ((F1 ne F2) and (S1 eq S2) and (n1 eq n3) and (n2 ne n4))
                                        or
                             CausesViolation_Signal(st(F1,S1,n1,n2),s);

endtype (* SignalRecord_set*)
```

The operation "CausesViolation_Signal" checks if the insertion of a new signalRecord conflicts with existing signals. We consider two possible inconsistencies between signals:

❶ A user receives two different signals

SignalRecord1: (F1, S1, A, B): Feature 1 generates a signal S1 to user A caused by user B.

SignalRecord2: (F2, S2, A, C): Feature 2 generates a signal S2 to user A caused by user C.

User A receives two different signals: S1 and S2 generated from two different features.

This inconsistency is detected in the condition: ((F1 ne F2) and (S1 ne S2) and (n1 eq n3))

Example:

SignalRecord 1: (F1, StartAudibleRinging, A, B)

SignalRecord 2: (F2, LineBusyTone, A, none)

The user A receives two different signals: "Start Audible Ringing" and "LineBusyTone" which denotes a feature interaction.

❷ A user receives the same signal but generated by two different users

SignalRecord 1: (F1, S, A, C)

SignalRecord 2: (F2, S, A, D)

The user A receives the same signal S but caused by different users.

This conflict is detected in the condition: ((F1 ne F2) and (S1 eq S2) and (n1 eq n3) and (n2 ne n4))

Example:

SignalRecord 1: (F1, StartAudibleRinging, A, C)

SignalRecord 2: (F2, StartAudibleRinging, A, D)

User A receives the same signal "Start Audible Ringing" but the involved users are different (C and D).

Note: The full description of types SignalRecord and SignalRecord_set is provided in Appendix.

6.2.4.3 Storing Billing information

To store Billing information, we defined two types: BillingRecord and BillingRecord_set.

- **BillingRecord**

For each connection leg there is a billing record containing the parties involved and the charged party. The BillingRecord is composed of the feature name, the caller party number, the called party number and the charged number.

The following is a sketch of the LOTOS ADT description of the BillingRecord type:

```
type BillingRecord is Feature, Number
sorts BillingRecord
opns
```

“bt” is the constructor of the BillingRecord.

```
        bt                      : Feature, Number, Number, Number → BillingRecord

….

endtype (*  BillingRecord  *)
```

- **BillingRecord_set**

This set contains the system billing records. The following is a sketch of the LOTOS ADT description of BillingRecord_set type.

```
type BillingRecord_set is BillingRecord
sorts BillingRecord_sets
opns
                ….
```

The operation "CausesViolation_Billing" checks if the insertion of a new Billing Record conflicts with existing Billing Records.

```
        CausesViolation_Billing  : BillingRecord, BillingRecord_sets → Bool

eqns

 forall  t1,t2 : BillingRecord,
         s: BillingRecord_sets,
         bts1,bts2: BillingRecord_sets,
         n1,n2,n3,n4,n5,n6:Number,
         b1,b2:Bool,
         F1,F2: Feature

         ….
         ofsort Bool
```

A new BillingRecord is not in conflict with an empty set of BillingRecord

```
                CausesViolation_Billing(bt(F1,n1,n2,n3),{}) = false;
```

The operation "CausesViolation_Billing" checks if the insertion of a new Billing Record conflicts with existing Billing Records. The conflict occurs when two users are billed for the same connection leg. For example (F1,A, B, A) and (F2, A, B, B) are two conflicting Billing records where A and B pay for the same leg.

```
CausesViolation_Billing(bt(F1,n1,n2,n3),insert(bt(F2,n4,n5,n6),s)) =
((F1 ne F2) and (n1 eq n4) and (n2 eq n5) and (n3 ne n6)) or
((F1 ne F2) and (n1 eq n5) and (n2 eq n4) and (n3 ne n6)) or
CausesViolation_Billing( bt(F1,n1,n2,n3),s);
```

endtype

Note: The full description of types BillingRecord and BillingRecord_set is provided in Appendix.

6.2.4.4 Display information

We defined two types: DisplayRecord and DisplayRecord_set.

- **DisplayRecord**

The DisplayRecord is composed of the feature name, the called party number, and a boolean value indicating whether the display occurred or not.

The following is the LOTOS ADT description of DisplayRecord type:

type DisplayRecord is Number,Boolean
sorts DisplayRecord
opns (* specify the format of operations *)

“bt” is the constructor of the BillingRecord

dt: Feature, Number, Bool $\rightarrow$ DisplayRecord

...
endtype (* DisplayRecord *)

- **DisplayRecord_set**

This set contains the display records. Display records could be inserted into the DisplayRecord_set either during the initialization part of the data types, or during the execution of the specification.

The following is the LOTOS ADT description of DisplayRecord_set type:

type DisplayRecord_set is DisplayRecord

```
sorts DisplayRecord_sets
opns


(* Test if a connection can cause violation within the connection set *)
        CausesViolation_Display  : DisplayRecord, DisplayRecord_sets → Bool

Eqns  (* List of Equations *)

        forall   t1,t2          : DisplayRecord,
                 s              : DisplayRecord_sets,
                 n1,n2,n3,n4    : Number,
                 b1,b2          : Bool
                 F1, F2         : Feature


ofsort Bool
        …
```

A new DisplayRecord is not in conflict with an empty set of DisplayRecord

```
        CausesViolation_Display (dt(F1, n1, n2, b1),{}) = false;
```

The operation "CausesViolation_Display" checks if the insertion of a new display record conflicts with existing constraints. This conflict happens when a user didn’t get a display where he is supposed to get one and vice versa. This is an example of two conflicting display records: (F1, A, B, True) and (F2, A, B, False).

```
        CausesViolation_Display (dt(F1, n1,n2,b1),insert(dt(F2, n3,n4,b2),s)) =
        ((F1 ne F2) and (n1 eq n3) and (n2 eq n4) and (b1 ne b2))
                or
        ((F ne F2) and (n1 eq n4) and (n2 eq n3) and (b1 ne b2))
                or
        CausesViolation_Display (dt(F1, n1,n2,b1),s);

endtype
```

6.2.5 Feature Interaction Detection System Architecture

As mentioned in Section 5.3.4, when several processes are combined together by means of the parallel composition "|[gates]|", if an action is in the list of gates in the operator then in order for that action to execute, all processes must participate simultaneously (synchronize) on that action. Further, each process can provide its own conditions for the actions to execute, and all such conditions must be true simultaneously in order for this to happen. Thus the control we want to obtain can be achieved gracefully by using an independent control process in parallel with the system specification. This process is “FI Detector” and will synchronize with the feature specification on the four defined detection points.

The top level of the behaviour part of our specification (shown in figure 42) consists of three processes: Process Feature1, Process Feature2 and Process FI Detector.
Feature 1 and Feature 2 are the LOTOS specifications of the two features involved, obtained by direct mapping from the UCMs of these two features. These two processes are composed in parallel and synchronize through their common gates, except those representing signals.

Note: We discarded gates representing signals from the set of synchronization gates to be able to execute those actions independently so we can detect signal conflicts.

The process “FI Detector” is the one responsible of detecting the feature interactions between the two features.

Features1 and Feature2 communicate with the “FIDetector” via specific synchronization gates. The two processes use these gates in order to send relevant data to FIDetector. The FIDetector checks for system consistency, notifies the environment if such interaction occur and re-instantiates itself to continue the FI detection. Figure 42 describes the top level of the specification.

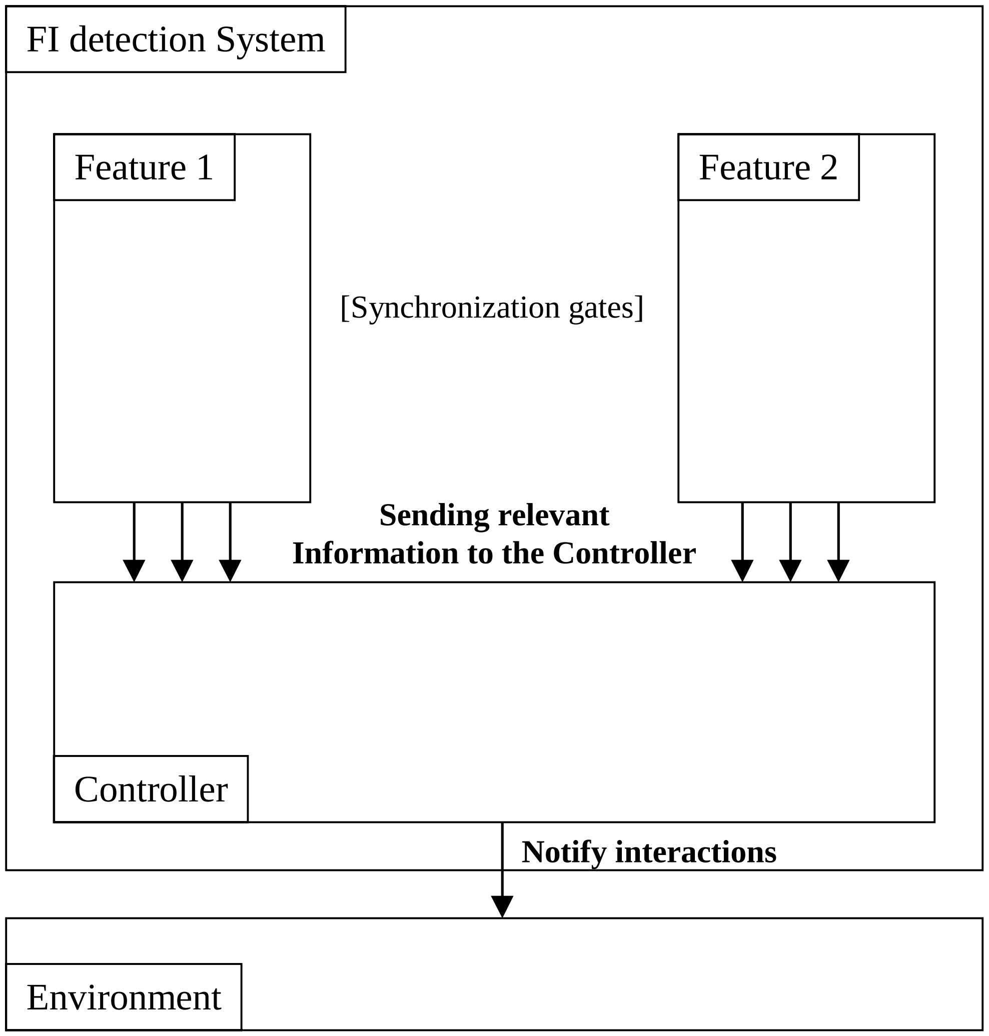


Figure 42: Graphical representation of the top-level of the specification

The specification below represents LOTOS top-level behaviour of the FI Detection System:

```
Specification FIDetection_System [offHook,dial,…..]: noexit:=
(* …Abstract data type definitions… *)
behavior
(
        (
        feature1 [Offhook, DialTone, Onhook, Dial, Connection, Billing, Signal…](B_State…)

                |[Offhook, DialTone, Onhook, Dial,…]|

        feature2 [Offhook, DialTone, Onhook, Dial, Connection, Billing, Signal] (B_State,…)
        )
                |[Connection, Billing, Signal, Display]|

        FIDetector[Connection, Billing, Signal,VR,…](Connection_set, Billing_set, Signal_set…)
)
```

6.2.6 Design of the process FI Detector

The role of the process FI Detector is to gather the relevant information from the features via the observation points and detect the interactions that could occur during the execution of the specification of the two features.

The FI Detector waits for actions to appear on the four observation gates discussed in Section 6.2.3. If an identical action has already been detected, nothing is done. If it is a new action, it checks whether a conflict has been created according to the method discussed in Section 6.2.4.

This is a sketch of the LOTOS code for process FI Detector that corresponds to the algorithm in figure 43. The LOTOS code describes only the detection of connection conflicts. Conflicts in signals, billing and display are similar to the connection conflict. Comments are interleaved with the code.

```
process FIDetector [Connection, Billing, Signal, Display, VR]
(Connection_set:ConnectionRecord_sets, Billing_set:BillingRecord_sets,
Signal_set:SignalRecord_sets, Display_set:DisplayRecord_sets) :noexit:=
```

Step 1: FI Detector is waiting for actions on gates: Connection, Signal, Billing and Display to appear.

```
(Connection ? f: Feature ? x:Number ? y:Number ?b:Bool;
```

Step 2: Data is received on the connection gate. We call "data" the parameters of actions. In this example above, these are the values of f, x, y and b.

Step 3: The operation "IsIn" checks if the data received is already in the database.

- Data already exists in the database: FI Detector re-instantiates itself and goes back to step 1 to wait for new data.

```
  [IsIn (ct (f, x, y, b), Connection_set)]-> FIDetector[Connection, Billing, Signal, Display, VR]
(Connection_set, Billing_set, Signal_set, Display_set)
                    []
```

- Data doesn't exist in the database: Check if the new data conflicts with existing data in the database. The operation "CausesViolation_Connection" discussed in Section 6.2.4.1 detects such violations.

```
   [not(IsIn(ct(f, x, y, b), Connection_set))]->
(        [CausesViolation_Connection(ct(f, x, y, b), Connection_set)] ->
```

If the new data is inconsistent with existing data, the violation is notified to the environment by executing the specific action VR. Then FI Detector re-instantiates itself and goes back to step 1 to wait for new data.

```
          VR! ViolationOfIntentions;
         FIDetector[Connection, Billing, Signal, Display, VR] (Connection_set, Billing_set, Signal_set, Display_set)
                   []
```

If the new data is consistent with existing data, the new data is inserted into the database and FI Detector is re-instantiated.

```
         [not(CausesViolation_Connection(ct(f, x, y, b), Connection_set))] ->
        FIDetector[Connection, Billing, Signal, Display, VR] (insert(ct(f, x, y, b), Connection_set), Billing_set,
Signal_set, Display_set)
     )
   )
)
[ ]
```

Same procedure for detecting Signal conflicts.

```
(Signal ? f: Feature ? x: signal ? y: Number ? z: signal;
…
[ ]
…
```

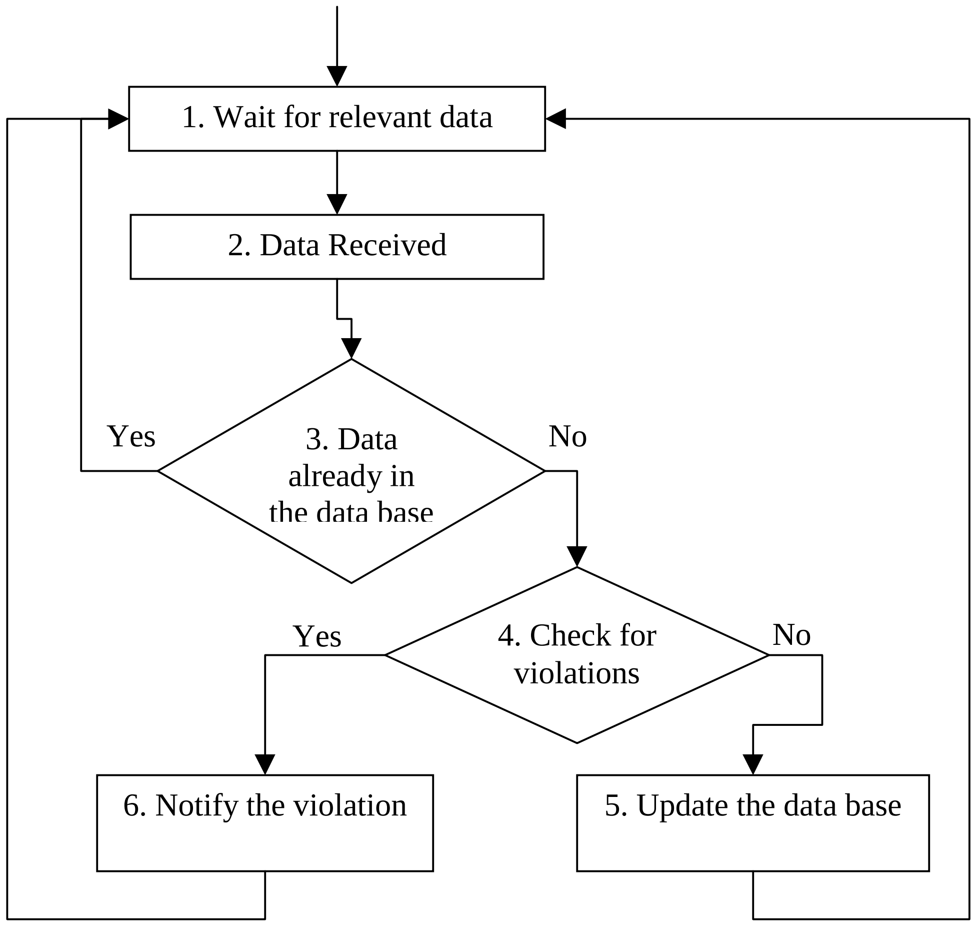


Figure 43: FI Detector procedure

6.3 Detection of Interactions

So far we have seen which interactions are considered in this work and how they emerge. Mechanisms for detection must now be considered. Our method works by executing a formal specification (which could be called also formal prototype or formal model) of the system with features. The ADTs included in the specification for this purpose will check execution to see whether a violation has occurred.

The ADT's represent a kind of a central call model monitor that captures every event associated with a call and verifies whether a violation has occurred or not.

In Chapter 4 we have filtered the interaction free scenarios from the relevant possible scenarios. However the verdict obtained was either "FI could occur" or "FI occurs".

The proposed mechanism is applied to investigate the cases having “FI could occur” as verdict and to characterize the interactions for those having “FI occurs” as verdict.

As seen in the filtering process two relevant cases should be considered:

- **Case 1**: Scenarios where the features are distributed between two users A and B.
- **Case 2**: Scenarios where a third party C is the feature subscriber and B arranges to forward/route the call to C.

For case 1 the UCM features are mapped directly to LOTOS. So we use the system as described above. However for case 2 only the terminating part (for user C) interests us. And to be able to detect the feature interactions we should map the UCM refinement proposed in the Chapter 4 into LOTOS. So we obtain a partial specification of the feature representing only the terminating part.

The system starts by executing the feature 1 (where A or B is subscribed to this feature) until the call is forwarded to C. At that time the specification describing the feature 2 (where C is subscribed) starts. This mechanism is shown on figure 44.

Note: Since the two features synchronize on their common gates this doesn't prevent the system from achieving its goal.

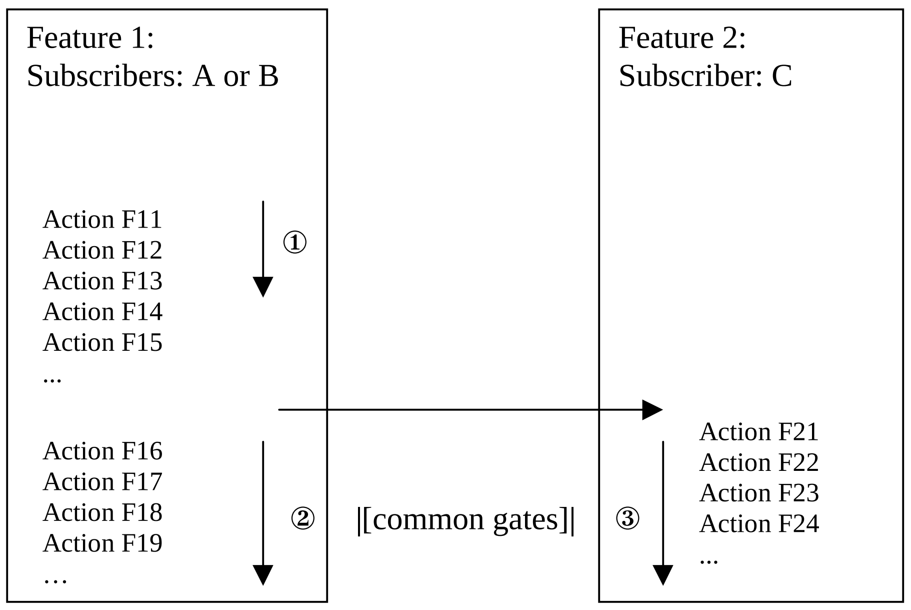


Figure 44: System with a third party as a feature subscriber

Example: Figure 45 illustrates such scenario between TCS and CFBL. B subscribes to CFBL and C subscribes to TCS. The CFBL process is executed first since TCS is blocked waiting for the action StartRinging_fwd to occur. A goes offhook and dials B's number. B is busy so the call is forwarded to C. Once the call is forwarded to C and the phone at C starts ringing the two processes start synchronizing on their common gates.

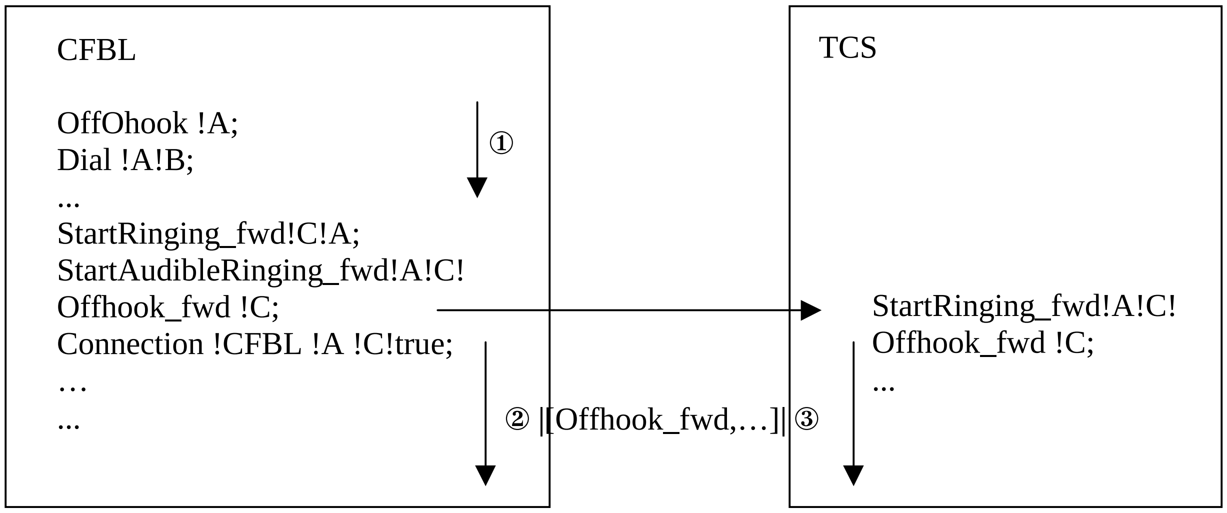


Figure 45: Example of a scenario involving TCS and CFBL

6.4 FI Scenario generation: Goal Oriented Execution (GOE)

In this step, the goal-oriented tool is applied to generate the traces leading to interactions. We call trace a sequence of observable actions that a LOTOS process can offer to the environment.

The method is based on the goal oriented execution tool developed within the LOTOS group of the University of Ottawa [46], [47]. Goal-oriented execution allows one to look for execution traces according to several properties. In this type of execution, the user specifies an action to be reached, usually an action that is not immediately derivable. The system then proceeds in a sort of selective eager execution, being able to select traces likely to reach the action. These traces can be found with the help of a static analysis of the behaviour expression. For example, if the behaviour expression is:

(a ; b ; stop ||| b ; c ; stop) [] c ; d ; f ; stop

and the user wants to be given an (or all) execution trace(s) reaching f, then the goal oriented execution algorithm is able to see that the left-hand side of the behaviour expression does not need to be expanded at all, because it does not contain action f. A considerable saving in computing time and space is obvious from the example.

Goal oriented execution can be used to find sequences of actions corresponding to certain criteria. The system proceeds to select traces that contain this sequence starting by the first action in the sequence. For example, if the behaviour expression is:

(a ; c ; stop [] a ; e ; c ; stop [] a ; stop)
and if the sequence of actions to be reached is: [a, c], then the possible traces that can be selected by the system are a ; c and a ; e ; c.

Events can be associated with actions in the sequence defining the goal to be reached. For example, if the sequence contains an action with gate name “a” and an offer value (!) x1, the selected trace must contain the action with gate a and with the offer of value x1. If the event associated with the action is an accept of a parameter (?), the system will instantiate all the possible values of that parameter.
An example of a goal to be reached is the following:
[a !x1 ?x2 , b~, c] \\ [e, f].

This goal is satisfied by all traces including a sequence of actions starting by an action with gate name a, with an offer of the value x1 and an acceptance of the value x2, leading to the action represented by gate c (without any event), and having as intermediate action an action with gate name b with an arbitrary event (~). Traces must not include actions with gates "e" and "f".

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

If some processes are instantiated recursively, leading to infinite LTS (Labelled Transition System: a notation where behaviours are represented by edges), GOE cannot guarantee the absence of a trace corresponding to the specified goal, because the tool limits the number of instantiations of processes.

Note that, after finding a trace, the tool will ask the user whether another one is desired.
To accelerate the search we can always guide it by adding some intermediate actions that we know must exist in a trace satisfying the specified goal. For example, if we are looking for a trace leading to an action specifying a connection establishment between two network users, we can add in the goal an action where a user dials a number, since it is evident that before an establishment of a connection, a user must dial a number. If we want to exclude the search from

some branches of the behaviour tree, we can add some specific gates and exclude them from the search. Then, the search process will not go in those branches where these specific gates are inserted.

In our work, we apply Goal Oriented Execution in order to obtain traces containing the gate "VR". Those traces lead to a feature interaction.

These traces (excluding observation gates) are used as test suites to test pairwise feature interaction within the implementation.

6.5 Application

In the following we study the interactions between OCS, CFBL and INFB. The LOTOS specifications are described in the Appendix.

6.5.1 Interactions between OCS and CFBL

This is an example of scenario leading to interactions between OCS and CFBL:
User A subscribes to OCS with C within his screening list. User B is a CFBL subscriber.
B is busy, A calls B so the call is forwarded to C.
The trace obtained with the goal-oriented tool is:
Data is received on the observation gate "connection" indicating that feature OCS doesn't allow connections between A and C.

```
Connection  ! OCS ! A !C ! false;
Offhook  ! A;  (* synchronization between CFBL and OCS *)
Dialtone  ! A;  (* synchronization between CFBL and OCS *)
Dial  ! A ! B;   (* synchronization between CFBL and OCS *)
LineBusyTone ! A; (* From OCS *)
```

A LineBusyTone signal is received on the observation gate "signal" indicating that this signal is received by A and is caused by OCS.

```
Signal ! OCS ! LineBusyTone ! A ! none;
detect_forward  ! C; (* from CFBL*)
```

Data is received on the observation point "connection" indicating that feature CFBL allows the connection between A and C.

```
Connection !CFBL! A !C !true;
```

The new connection information causes a conflict with the existing connection information. The conflict is reported to the environment.

```
VR ! violationofconnections;  (* from FIDetector *)
```

A StartRinging signal is received on the observation gate "signal" indicating that this signal is received by C where A is involved and is caused by CFBL.

```
StartRinging_fwd ! C !A; (* from CFBL *)
Signal ! CFBL !ringing !C !A  ;
```

A StartAudibleRinging signal is received on the observation gate "signal" indicating that this signal is received by A where C is involved and is caused by CFBL.

```
StartAudibleRinging_fwd !A ! C; (* from CFBL *)
Signal ! CFBL !audibleringing !A !C ;
```

The new signal information causes a conflict with the existing signal information. A is receiving two different signals: LineBusyTone from OCA and StartAudibleRinging from CFBL.

```
VR ! violationofsignals;  (* from FIDetector *)
…etc.
```

Two interactions are notified:

1. Violation of connection: Connection (CFBL, A, C, true) and Connection (OCS, A, C, false)
2. Inconsistency of signals: Signal (CFBL, audibleringing, A, C) and Signal (OCS, LineBusyTone, A, none)

6.5.2 Interactions between INFB and CFBL

This is an example of scenario leading to interactions between INFB and CFBL

User C subscribes to INFB and user B is a CFBL subscriber. A calls B, B is busy so the call is forwarded to C.

The trace obtained with the goal-oriented tool is:

```
Offhook  ! A;  (* From CFBL *)
Dialtone  ! A;  (* From CFBL *)
Dial  ! A ! B;   (* From CFBL *)
detect_forward  ! C; (* from CFBL*)
```

Data is received on the observation point "connection" indicating that feature CFBL allows the connection between A and C.

```
Connection !CFBL! A !C !true;
```

A StartRinging signal is received on the observation gate “signal” indicating that this signal is received by C where A is involved and is caused by CFBL.

```
StartRinging_fwd ! C !A;
Signal ! CFBL !ringing !C !A  ;
```

A StartRinging signal is received on the observation gate “signal” indicating that this signal is received by C where A is involved and is caused by INFB.

```
StartRinging_fwd ! C !A;
Signal ! INFB !ringing !C !A ;
…
```

A StartAudibleRinging signal is received on the observation gate “signal” indicating that this signal is received by A where C is involved and is caused by CFBL.

```
StartAudibleRinging_fwd !A ! C;
Signal ! CFBL !audibleringing !A !C ;
```

A StartAudibleRinging signal is received on the observation gate “signal” indicating that this signal is received by A where C is involved and is caused by INFB.

```
StartAudibleRinging_fwd !A ! C;
Signal ! INFB !audibleringing !A !C ;
```

Data is received on the observation point “connection” indicating that feature CFBL allows the connection between A and C.

```
Connection !CFBL !A !C !true;
…
```

Data is received on the observation point “connection” indicating that feature INFB allows the connection between A and C.

```
Connection !INFB !A !C !true;

Offhook_fwd !C;
StopRinging_fwd !C !A;
StopAudibleRinging_fwd !A !C;
```

Data is received on the observation point “billing” indicating that for feature CFBL A should pay for AB leg.

```
LogBegin_fwd !A!B!A;
Billing ! CFBL! A!B!A;
```

Data is received on the observation point “billing” indicating that for feature CFBL B should pay for BC leg.

```
LogBegin_fwd !B !C !B;
Billing ! CFBL! B ! C !B;
```

Data is received on the observation point “billing” indicating that for feature INFB B should pay for BC leg.

```
LogBegin !B!C!C;
Billing !INFB ! B !C! C;

VR ! violationofBilling;  (* from FIDetector *)
…etc.
```

One interaction is notified: Violation of Billing: Billing (CFBL, B, C, B) and Billing (INFB, B, C, C) are conflicting. B and C should not pay for the same leg BC.

6.5.3 Interactions between INFB and OCS

This is an example of scenario leading interactions between INFB and OCS.
User A subscribes to OCS with B within his screening list. User B is an INFB subscriber.
The trace obtained with the goal-oriented tool is:

```
Connection  ! OCS ! A !B ! false;
Offhook  ! A;  (* synchronization between INFB and OCS *)
Dialtone  ! A;  (* synchronization between INFB and OCS *)
Dial  ! A ! B;   (* synchronization between INFB and OCS *)

PlayAnnoucement !A !ScreenedMessageOCS ;
Signal ! OCS ! PlayAnnoucement ! A !none;

StartRinging! B !A; (* from INFB *)
Signal ! INFB !ringing !B !A  ;

StartAudibleRinging !A ! B; (* from INFB *)
(* Observation point: Signal *)
Signal ! INFB !audibleringing !A !B ;

VR ! violationofsignals;  (* from FIDetector *)
…etc.
```

Inconsistencies of signals: Signal (INFB, audibleringing, A, C) and Signal (OCS, LineBusyTone, A, none)

6.6 Results

We have investigated the interactions that could occur between the following eight features: Originating Call Screening (OCS), Terminating Call Screening (TCS), IN Free Routing (INFR), Call Forwarding Busy Line (CFBL), Call Number Delivery (CND), IN Freephone Billing (INFB), IN Call Forwarding (INCF) and IN Charge Call (INCC).
Table 5 shows the FI detection results.

	OCS	TCS	INFB	INCC	CND	CFBL	INCF	INFR
OCS	-	-	1	1	-	1, 2	1, 2	1, 2
TCS		-	1	1	4	1, 2	1, 2	1, 2
INFB			-	3	-	1, 3	1, 3	1,3
INCC				-	4	1	1	1
CND					-	4	4	4
CFBL						-	1, 2	1, 2
INCF							1, 2	1, 2
INFR								1, 2

(1) : Signal conflict
(2) : Connection conflict
(3) : Billing conflict
(4) : Display problem

Table 5: Feature Interaction Detection Results

Comparing our results with the benchmark Feature Interaction

The number and type of Feature Interactions detected are two basic factors when evaluating a Feature Interaction detection method. For this reason, the organizing committee of the Feature Interaction Contest [5] published a benchmark document [65], listing the FIs that they believed to exist among the feature to be studied in the contest. In this section, we evaluate our method by comparing the set of interactions detected by FIDS with the one provided in the benchmark.

Before presenting a detailed comparison, we should note that the contest specifications [5] were not specific concerning the composition of the features.

As mentioned in Section 6.2.5, we decided to compose features in parallel and synchronize them only on their common gates, except those representing signals. Thus, they can synchronize on signals in any order and the call process will not be affected if conflicting signals occur.

As mentioned in Section 6.1, we consider only four types of feature interaction: Connection violation, Inconsistency of signals given to users, Incorrectness of Billing, Inconsistency in the display function. The benchmark instead tries a more general classification: Feature Interactions are categorized into corresponding conflict/failure types such as Billing conflict, Call termination conflict, Forwarding conflict, Disconnect conflict, Number delivery failure (Number not displayed), PIN conflicts (over-ride PIN), Flash conflict.

Table 6 lists the mapping relationship from the benchmark FI types to our FI types.

Benchmark FI type	Our FI type
Billing conflict	Incorrectness of Billing
Call Termination conflict	Inconsistency of signals given to users
Flash conflict	Not addressed
Disconnect conflict	Inconsistency of signals given to users
Forwarding conflict	Inconsistency of signals given to users Connection violation
PIN conflicts (over-ride PIN)	Not addressed
Number delivery failure	Inconsistency in the display function

Table 6: The Mapping Table of FI Types

From the above mapping, we find that Flash conflict is not addressed in our method since features such as TWC (Three Way Calling) and CW (Call Waiting) are out of the scope of our method. However, most of benchmark FI types can be mapped to a corresponding FI type.

Interactions found under “forwarding conflict” type from benchmark FI types could be either an inconsistency of signals given to users or connection violation.

We failed to detect interactions of type “PIN conflicts” (found between INTL and INCC) because over-riding variables is not considered in our detection method.

The benchmark listed 38 interactions while we succeeded to detect only 36 interactions.
Not detected interactions are between:
1.CND-INCF: Scenario causing the interaction: User B subscribed to INCF(C), C subscribed to CND. A calls B and the call is forwarded to C.
Interaction: No number display at C.

2.CND-INFR: Scenario causing the interaction: User B subscribed to INFR(C), C subscribed to CND. A calls B and the call is forwarded to C.
Interaction: No number display at C.

We failed to detect these two interactions related to the display function because in our method we describe only the termination part of the call at user C and we assume that a forwarded call is similar to a normal call.

Table 7 gives a comparison in terms of number of interactions detected for each pair of features. For each feature pair we associate a pair (x, y) where: “x” is the number of feature interactions found with our method and “y” is the number of feature interactions found in the benchmark.

	TCS	INFB	INCC	CND	CFBL	INCF	INFR
TCS	-	(1,1)	(1,1)	(1,1)	(2,2)	(2,2)	(2,2)
INFB		-	(1,1)	-	(2,2)	(2,2)	(2,2)
INCC			-	(1,1)	(1,1)	(1,1)	(2,2)
CND				-	(1,1)	(1,2)	(1,2)
CFBL					-	(3,3)	(3,3)
INCF						(1,1)	(3,3)
INFR							(2,2)

Table 7: Comparison in terms of number of features interactions

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 we are using the parallel composition and we are introducing observation points to collect data, our method can tolerate any conflicts and the call process continues until all activated features finish. Thus, there is no wonder that our method can detect more than one feature interaction per scenario.

We detected interactions of type "Inconsistency of signals given to users" in some scenarios declared by the filtering method (in Chapter 4) as interaction free scenarios. This is due to:

- The way features are described using our UCM model presented in Section 4.2.2.3. The precedence assumption between stubs in our UCM model solves such interactions. Our FI detection process doesn't consider such assumptions.
- In our FI detection process features don't synchronize on signals.

6.7 Conclusion

The Feature interaction method presented is considered as a complementary step of the filtering process introduced in Chapter 4. Only scenarios with verdict "FI could occur" and "FI occurs" could be analyzed. The method generates also traces, using the goal oriented execution tool, leading to an interaction if one exists. These traces are used further to test the implementation.

The method we have presented uses extensively Abstract Data Types to detect feature interactions.

Chapter 7

Conclusion & Future Work

This thesis proposes a framework for feature Interaction detection.

7.1 Summary

This thesis consists of seven chapters. The background and motivation for our work is given in Chapter 1. In this chapter a list of contributions was also given.

Chapter 2 presents a survey of related work on the formal techniques that are used to specify telephony systems with their features. We also presented some of the Feature Interaction Detection methodologies using FDTs proposed in the literature.

Chapter 3 gives an overview of the existing requirement description techniques relevant to this thesis. We focus on the Use Case Maps notation.

Chapter 4 introduces a Use Case Maps model for describing telephony features at the requirement stage. This model, called also root map, allows us to integrate features, both switch based and IN ones, into the basic call model. Features like CFBL, INTL, INFB, OCS, TCS are used as examples to illustrate the feature integration mechanism. Based on this model, we propose a method to filter feature interactions at the requirement stage. This method allows the designer to localize where interactions could occur during a call and facilitates their further detection by taking out the interaction-free scenarios. Finally results of the filtering on examples of switch and IN based features were presented. The experimental evaluation shows that more than half of the feature combinations can be filtered (see Section 4.6.1).

As mentioned in Section 4.8, the method is limited to features that can be expressed with a certain structure, however it could be generalized.

Chapter 5 gives an overview of the LOTOS language and of its main operators. It also shows the use of LOTOS as a formal description Technique (FDT) in specifying the telephone system model and features. Our main objective in specifying the system model and features in LOTOS is to provide a specification that can be used for validating and detecting feature interactions.

In Chapter 6, a formal definition of Feature Interaction is provided and a FI Detection system (FIDS) is developed based upon the definition. Based on our experience with feature interaction, we have investigated four kinds of interactions:

- Connection violations
- Inconsistency of signals given to users
- Incorrectness of Billing
- Problems in the display function

Our technique is based on Abstract Data Types (ADT) to detect these violations.

The Feature interaction method presented is considered as a complementary step of the filtering process introduced in Chapter 4.

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 the four types of interactions mentioned above. It allows also the automatic generation of functional test cases that can be used to see whether an interaction exists in the implementation.

We have shown that telecommunication system designers can give precise description and can validate their design with respect to potential feature interaction problems before the implementation stage.

7.2 Future Work

The results of this thesis provide a basis for several future research directions.

As mentioned in Section 4.8, features such as Call Waiting and Three Way Calling are outside of the scope of the filtering method we propose. It would be useful to extend our UCM model to be able to describe these features and to investigate them since they can be in conflict with other features. The filtering quality of our method could also be enhanced using more information (e.g. connectivity of scenario paths and preconditions).

We should also acknowledge a conceptual problem that we could not address in this thesis. The rules we introduced in Chapter 4 to determine that a system is interaction free are not shown to be consistent with the formal definition of feature interactions we used in Chapter 6. In other words, it is possible that a combination of features declared to be interaction free according to the rules of Chapter 4 is in fact not interaction free according to the definition of Chapter 6. However, in all the examples we have analysed we have not found any example showing this possibility. We leave the study of this question to further work.

The current development of telecommunication technology is so intensive and goes in so many directions that probably nobody can really predict what the telecommunication market will look like five years from now. Looking at current developments, one of the things we can be certain is that new types of feature interaction are emerging. Complications arise since functionality tends to be more distributed. Many techniques for resolving interactions in the PSTN are no longer easily applied. Trying to solve these interactions using Use Case Maps and applying formal methods like LOTOS and SDL in this new area is certainly a worthwhile and challenging issue.

Appendix

Specification of the abstract data types

- **Type PIN: Personal Identification Number**

  The type PIN describes the different personal identification numbers

```
type PIN is Boolean
sorts  PIN
opns
        PIN_INTL             (*! constructor *),
        Invalid_PIN_INTL     (*! constructor *) : → PIN
        _eq_,
        _ne_                                       : PIN, PIN → Bool

eqns
        forall P1, P2:PIN
        ofsort Bool

                P1 eq P1 = true;
                P2 eq P2 = true;
                P1 eq P2 = false;
                P2 eq P1 = false;
                P1 ne P2 = not (P1 eq P2);

endtype  (*Type PIN*)
```

- Type Message

```
type Message is NaturalNumber
sorts Message
opns

        AskForPIN              (*! constructor *),
        ScreenedMessageOCS     (*! constructor *),
        EnterPhoneNumber       (*! constructor *) : → Message
        _eq_,
        _ne_                                        : Message, Message → Bool
        h                                           : Message → Nat

eqns
        forall M1, M2:Message
        ofsort Bool
                M1 eq M2 = h(M1) eq h(M2);
                M1 ne M2 = h(M1) ne h(M2);

        ofsort Nat
                h(AskForPIN) = 0;
                h(ScreenedMessageOCS) = Succ(0);
                h(EnterPhoneNumber) = Succ (Succ(0));

endtype  (*Type Message *)
```

- Type State

```
type State is NaturalNumber
sorts State
opns
        Idle,               (*! constructor *)
        Busy,               (*! constructor *)
        Ringing,            (*! constructor *)
        Audibleringing,     (*! constructor *),
        Linebusy,           (*! constructor *),
        Annoucement,        (*! constructor *),
        Talking             (*! constructor *)          : → State

        map                                             : State → Nat
        _ eq _,
        _ ne _                                          : State, State → Bool

eqns
        forall s1, s2: State
        ofsort Nat

                map (Idle)              =  0;
                map (Busy)              =  Succ(0);
                map (ringing)           =  Succ(Succ(0));
                map (audibleringing)    =  Succ(Succ(Succ(0)));
                map (linebusy)          =  Succ(Succ(Succ(Succ(0))));
                map (Annoucement)       =  Succ(Succ(Succ(Succ(Succ(0)))));
                map (talking)           =  Succ(Succ(Succ(Succ(Succ(Succ(0))))));

        ofsort Bool
                s1 eq s2 = map(s1) eq map (s2) ;
                s1 ne s2 = not (s1 eq s2);

endtype (* State *)
```

- Type SignalRecord

```
type SignalRecord is Feature, Number, State, Boolean
sorts SignalRecord
opns

        st                      : Feature, State, Number, Number → SignalRecord

        extract_feature         : SignalRecord → Feature
        extract_Number1         : SignalRecord → Number
        extract_Number2         : SignalRecord → Number
        extract_State           : SignalRecord → State
          _eq_, _ne_            : SignalRecord, SignalRecord → Bool
```

```
eqns

        forall  N1,N2,N3,N4     : number,
                F1,F2           : Feature,
                S1,S2           : State,
                B1,B2           : Bool

        ofsort Feature
                extract_feature(st(F1,S1,N1,N2)) = F1;

        ofsort Number
                extract_Number1(st(F1,S1,N1,N2)) = N1;
                extract_Number2(st(F1,S1,N1,N2)) = N2;

        ofsort State
                extract_State(st(F1,S1,N1,N2)) = S1;

        ofsort Bool
                st(F1,S1,N1,N2) eq st(F2,S2,N3,N4) =
                (F1 eq F2) and (S1 eq S2) and (N1 eq N3)and(N2 eq N4);

                st(F1,S1,N1,N2) ne st(F2,S2,N3,N4) =
                not(st(F1,S1,N1,N2) eq st(F2,S2,N3,N4));

endtype (*  SignalRecord  *)
```

- Type SignalRecord_set

```
type SignalRecord_set is SignalRecord
sorts SignalRecord_sets
opns

(* An empty set*)
        {}                              : → SignalRecord_sets

 (* Insert a new signalRecord in the signalRecord set *)
        insert                          : SignalRecord,SignalRecord_sets → SignalRecord_sets

        head                            : SignalRecord_sets → SignalRecord
        tail                            : SignalRecord_sets → SignalRecord_sets

(* Comparing signal sets *)
        _eq_, _ne_                      : SignalRecord_sets, SignalRecord_sets → Bool

(* Checks whether the signal set is empty *)
        empty                           : SignalRecord_sets → Bool

(* Cheks if a signalRecord is in the signalRecord set *)
        isin                            : SignalRecord, SignalRecord_sets → Bool

(* Test if a signalRecord can cause violation within the signalRecord set *)

        CausesViolation_Signal          : SignalRecord, SignalRecord_sets → Bool
```

```
eqns

 forall  t1,t2            : SignalRecord,
         F1,F2            : Feature,
         S1,S2            : State,
         S                : SignalRecord_sets,
         n1,n2,n3,n4      : Number
         b1,b2            :Bool

         ofsort SignalRecord_sets
                 tail(insert(t1,s))=s;


         ofsort SignalRecord
                 head(insert(t1,s))=t1;

         ofsort Bool
                 {} eq {} = true;
                 {} eq insert(t1,s) = false;
                 insert(t1,s) eq {} = false;
                 empty(s) = s eq {};
                 isin(t1,{}) = false;
                 t1 eq t2 => isin(t1,insert(t2,s)) = true;
                 t1 ne t2 => isin(t1,insert(t2,s)) = isin(t1,s);
                 CausesViolation_Signal(st(F1,S1,n1,n2),{}) = false;

                 CausesViolation_Signal (st(F1,S1,n1,n2),insert(st(F2,S2,n3,n4),s)) =
                         ((F1 ne F2) and (S1 eq S2) and (n1 eq n3) and (n2 ne n4))
                                 or
                         ((F1 ne F2) and (S1 ne S2) and (n1 eq n3))
                                 or
                         CausesViolation_Signal(st(F1,S1,n1,n2),s);

endtype (* SignalRecord_set*)
```

- Type BillingRecord

```
type BillingRecord is Feature, Number
sorts BillingRecord
opns

         bt                      : Feature, Number, Number, Number → BillingRecord
         extract_feature         : BillingRecord → Feature
         adr1                    : BillingRecord → Number
         adr2                    : BillingRecord → Number
         ChargedParty            : BillingRecord → Number
         _eq_,
         _ne_                    : BillingRecord, BillingRecord → Bool

eqns
         forall N1,N2,N3,N4,N5,N6        : Number,
         t1,t2                           : BillingRecord,
         B1,B2                           : Bool,
         F1,F2                           : Feature
```

```
        ofsort Feature
                extract_feature(bt(F1,N1,N2,N3)) = F1;

        ofsort Number
                adr1(bt(F1,N1,N2,N3)) = N1;
                adr2(bt(F1,N1,N2,N3)) = N2;
                ChargedParty(bt(F1,N1,N2,N3)) = N3;

        ofsort Bool
                bt(F1,N1,N2,N3) eq bt(F2,N4,N5,N6) =
                ((F1 eq F2)and(N1 eq N4) and (N2 eq N5) and (N3 eq N6)) or
                ((F1 eq F2)and(N1 eq N5) and (N2 eq N4) and (N3 eq N6)) ;

                bt(F1,N1,N2,N3) ne bt(F2,N4,N5,N6) =
                not (bt(F1,N1,N2,N3) eq bt(F2,N4,N5,N6));

endtype (*  BillingRecord  *)
```

- Type BillingRecord_set

```
type BillingRecord_set is BillingRecord
sorts BillingRecord_sets
opns
        {}      (*! constructor *):  → BillingRecord_sets
         insert (*! constructor *) : BillingRecord,BillingRecord_sets → BillingRecord_sets
         head   : BillingRecord_sets → BillingRecord
         tail   : BillingRecord_sets → BillingRecord_sets
         _eq_,
         _ne_   : BillingRecord_sets, BillingRecord_sets → Bool
         empty  : BillingRecord_sets → Bool
         eleof  : BillingRecord, BillingRecord_sets → Bool
         IsIn   : BillingRecord, BillingRecord_sets → Bool
         CausesViolation_Billing  : BillingRecord, BillingRecord_sets → Bool

eqns

  forall  t1,t2 : BillingRecord,
        s: BillingRecord_sets,
        bts1,bts2: BillingRecord_sets,
        n1,n2,n3,n4,n5,n6:Number,
        b1,b2:Bool,
        F1,F2: Feature


        ofsort BillingRecord_sets
                tail(insert(t1,s))=s;

        ofsort BillingRecord
                head(insert(t1,s))=t1;

        ofsort Bool
                {} eq {} = true;
                {} eq insert(t1,s) = false;
                bts1 ne bts2 = not(bts1 eq bts2);
```

```
                    insert(t1,s) eq {} = false;
                    empty(s) = s eq {};
                    eleof(t1,{}) = false;
                    t1 eq t2 => eleof(t1,insert(t2,s)) = true;
                    t1 ne t2 => eleof(t1,insert(t2,s)) = eleof(t1,s);
                    IsIn(t1,{}) = false;
                    t1 eq t2 => IsIn(t1,insert(t2,s)) = true;
                    t1 ne t2 => IsIn(t1,insert(t2,s)) = IsIn(t1,s);
                    CausesViolation_Billing(bt(F1,n1,n2,n3),{}) = false;
                    CausesViolation_Billing(bt(F1,n1,n2,n3),insert(bt(F2,n4,n5,n6),s)) =
                    ((F1 ne F2) and (n1 eq n4) and (n2 eq n5) and (n3 ne n6)) or
                    ((F1 ne F2) and (n1 eq n5) and (n2 eq n4) and (n3 ne n6)) or
                    CausesViolation_Billing( bt(F1,n1,n2,n3),s);

endtype
```

- Type DisplayRecord

```
type DisplayRecord is Number,Boolean
sorts DisplayRecord
opns  (* specify the format of operations *)

dt: Feature, Number, Number, Bool → DisplayRecord

(* num1 and num2 are 2 operations used to extract the user numbers involved in the connection *)

num1                : DisplayRecord → Number
num2                : DisplayRecord → Number

Bvalue              : DisplayRecord → Bool

(* eq, and ne are used to compare displayRecords *)

_eq_, _ne_          : DisplayRecord, DisplayRecord → Bool


eqns  (* List of equations *)

          forall    N1,N2,N3,N4       :number,
                    t1,t2             : DisplayRecord,
                    B1,B2             : Bool
                    F1, F2            : Feature
          ofsort Number
                    num1(dt(F,N1,N2,B1)) = N1;
                    num2(dt(F,N1,N2,B1)) = N2;

          ofsort Bool
                    Bvalue(dt(F,N1,N2,B1)) = B1;
                    dt(F,N1,N2,B1) eq dt(F,N3,N4,B2) = ((F1 eq F2) and (N1 eq N3) and (N2 eq N4) and (B1 eq B2))
                                                                  or
                                                  ((F1 eq F2) and  (N1 eq N4) and (N1 eq N2) and (B1 eq B2));

                    dt(N1,N2,B1) ne dt(N3,N4,B2) = not( dt(N1,N2,B1) eq dt(N3,N4,B2));

endtype (*  DisplayRecord  *)
```

- Type DisplayRecord_set

```
type DisplayRecord_set is DisplayRecord
sorts DisplayRecord_sets
opns

(* An empty set*)
        {}                              : → DisplayRecord_sets

(* Insert a new DisplayRecord in the DisplayRecord_set *)
        insert                          : DisplayRecord, DisplayRecord_sets → DisplayRecord_sets
        tail                            : DisplayRecord_sets -> DisplayRecord_sets

(* Comparing Display sets *)
        _eq_,  _ne_                     : DisplayRecord_sets, DisplayRecord_sets -> Bool

        empty                           : DisplayRecord_sets → Bool

(* Search for a connection in the display set *)
        isin                            : DisplayRecord, DisplayRecord_sets → Bool

(* Test if a connection can cause violation within the display set *)
        CausesViolation_Display  : DisplayRecord, DisplayRecord_sets → Bool

Eqns  (* List of Equations *)

        forall   t1,t2                  : DisplayRecord,
                 s                      : DisplayRecord_sets,
                 n1,n2,n3,n4            : Number,
                 b1,b2                  : Bool
                 F1, F2                 : Feature

ofsort DisplayRecord_sets
        tail (insert(t1,s)) = s;

ofsort Bool
        {} eq {} = true;
        {} eq insert(t1,s) = false;
        insert(t1,s) eq {} = false;
        empty(s) = s eq {};
        isin(t1,{}) = false;
        t1 eq t2 => isin(t1,insert(t2,s)) = true;
        t1 ne t2 => isin(t1,insert(t2,s)) = isin(t1,s);
        CausesViolation_Display (dt(F1, n1, n2, b1),{}) = false;

        CausesViolation_Display (dt(F1, n1,n2,b1),insert(dt(F2, n3,n4,b2),s)) =
        ((F1 ne F2) and (n1 eq n3) and (n2 eq n4) and (b1 ne b2))
                 or
        ((F ne F2) and (n1 eq n4) and (n2 eq n3) and (b1 ne b2))
                 or
        CausesViolation_Display (dt(F1, n1,n2,b1),s);

endtype
```

Some Feature Specifications

• OCS (Originating Call Screening)

process OCS_feature[Offhook, DialTone, Onhook, Dial, StartRinging, StartAudibleRinging, StopRinging, StopAudibleRinging, LineBusyTone, LogBegin, LogEnd, Disconnect, PlayAnnoucement, Connection, Billing, Signal, Display] (B_State: State, ocs_list: Number) :**noexit**:=

```
(* Observation point: Connection *)
Connection ! OCS !A !ocs_list ! false;

Offhook !A;
DialTone !A;
(
        Onhook !A; stop
        [ ]

        (  Dial !A!B;
                (
                [B eq ocs_list] → PlayAnnoucement !A !ScreenedMessageOCS ;
                                (* Observation point: Signal*)
                                Signal ! OCS ! PlayAnnoucement ! A !none;
                                Onhook !A;
                                stop

                        [ ]

                [B ne ocs_list] →
                        (
                        [B_State eq idle] →
                                        StartRinging !B!A ;
                                        (* Observation point: Signal*)
                                        Signal ! OCS ! ringing ! B !A;

                                        StartAudibleRinging !A !B;
                                        (* Observation point: Signal*)
                                        Signal ! OCS ! audibleringing ! A !B;

                                        ( Onhook !A;
                                        StopRinging !B!A;
                                        StopAudibleRinging !A!B;stop

                                                [ ]

                                        Offhook !B;
                                        (* Observation point: Connection*)
                                        Connection ! OCS !A !B !true ;

                                        StopRinging !B !A;
                                        StopAudibleRinging !A !B;
                                        LogBegin !A!B!A;

                                        (* Observation point : Billing*)
                                        Billing ! OCS !A !B !A ;
```

```
                                                    (
                                                    Onhook !A;
                                                    Disconnect !B!A;
                                                    LogEnd !A!B;
                                                    Onhook !B;
                                                    stop

                                                    [ ]

                                                    Onhook!B;
                                                    Disconnect !A!B;
                                                    LogEnd !A!B;
                                                    Onhook !A;
                                                    Stop
                                                    )
                                             )
                          [ ]
                   [B_State eq busy] →       LineBusyTone!A ;
                                             (* Observation point: Signal*)
                                             Signal ! OCS ! LineBusyTone ! B !A;
                                             Onhook!A ;
                                             stop
          )
        )
  )
)
endproc (* OCS Feature *)
```

• CFBL (Call Forwarding Busy Line)

```
process CFBL_feature [Offhook, DialTone, Onhook, Dial,StartRinging, StartAudibleRinging, StopRinging,
StopAudibleRinging, LineBusyTone, LogBegin, LogEnd, Disconnect, Detect_forward, LineBusyTone_fwd,
Onhook_fwd, Offhook_fwd, StartRinging_fwd, StartAudibleRinging_fwd, StopRinging_fwd,
StopAudibleRinging_fwd, LogBegin_fwd, LogEnd_fwd, Disconnect_fwd, Connection, Billing, Signal, Display]
(B_State:InitialState,fwd_number_State:InitialState, fwd_number:Number)
 :noexit:=

Offhook !A;
DialTone !A;
(
 Onhook !A; stop
   []
 (
   Dial !A!B;
    (
      [B_State eq idle] ->    StartRinging !B!A ;
                              (* Observation point: Signal *)
                             Signal ! CFBL !ringing !B !A ;

                             StartAudibleRinging !A !B;
                              (* Observation point: Signal *)
                              Signal ! CFBL !audibleringing !A !B ;
```

```
                    ( Onhook !A;
                    StopRinging !B!A;
                    StopAudibleRinging !A!B;stop

                    [ ]

                    Offhook !B;
                    (*Observation point*)
                    Connection !CFBL !A !B !true ;

                    StopRinging !B !A;
                    StopAudibleRinging !A !B;
                    LogBegin !A!B!A;

                    (* Observation point: Billing *)
                    Billing ! CFBL! A !B!A;
                    (
                    Onhook !A;
                    Disconnect !B!A;
                    LogEnd !A!B;
                    Onhook !B;
                    stop
                       []
                    Onhook!B;
                    Disconnect !A!B;
                    LogEnd !A!B;
                    Onhook !A;
                    stop
               )
              )

        [ ]

 [B_State eq busy]-> Detect_forward ! fwd_number;
       (
     [fwd_number_State eq Busy] -> LineBusyTone_fwd !A;

                        (* Observation point: Signal *)
                        Signal ! CFBL !linebusy !A !none;
                        onhook_fwd !A;
                        stop
        []
     [fwd_number_State eq Idle] ->

          (* Observation point: Connection*)
          Connection !CFBL! A !fwd_number !true;

          StartRinging_fwd ! fwd_number !A;
          (* Observation point *)
          Signal ! CFBL !ringing !fwd_number !A  ;

          StartAudibleRinging_fwd !A !fwd_number;
          (* Observation point *)
          Signal ! CFBL !audibleringing !A !fwd_number ;
```

```
                    ( onhook_fwd !A;
                      StopRinging_fwd ! fwd_number !A;
                      StopAudibleRinging_fwd !A !fwd_number;
                      stop
                          [ ]
                    Offhook_fwd ! fwd_number;
                     StopRinging_fwd ! fwd_number !A;
                    StopAudibleRinging_fwd !A !fwd_number;

                     LogBegin_fwd !A!B!A;
                      (* Observation point *)
                      Billing ! CFBL! A!B!A;

                     LogBegin_fwd !B!fwd_number!B;
                       (* Observation point*)
                       Billing ! CFBL! B ! fwd_number !B;

                        (
                                        onhook_fwd !A;
                                        Disconnect_fwd !fwd_number!A;
                                        LogEnd_fwd !A!B;
                                        LogEnd_fwd !B!fwd_number;
                                        onhook_fwd !fwd_number;stop
                                      [ ]
                                        onhook_fwd !fwd_number;
                                        Disconnect_fwd !A !fwd_number;
                                        LogEnd_fwd !A!B;
                                        LogEnd_fwd !B!fwd_number;
                                        onhook_fwd !A;stop
                                      )
                      )
                 )

    )
  )
)
```

endproc

- ## IN Free Billing (INFB)

process INFB_feature[Offhook, DialTone, Onhook, Dial,StartRinging, StartAudibleRinging, StopRinging, StopAudibleRinging, LineBusyTone, LogBegin, LogEnd, Disconnect, Connection, Billing, Signal, Display] (B_State:InitialState) :**noexit**:=

```
Offhook ! A ;
Dialtone ! A ;
( onhook !A; stop
[ ]
Dial !A!B;
   (
     [B_State eq busy]->     LineBusyTone!A ;
                             (* Observation point: Signal *)
                             Signal ! INFB !linebusy !A !none;
                             Onhook!A ;
```

```
                              stop
                                [ ]
  [B_State eq idle] ->        StartRinging !B!A ;
                              (* Observation point: Signal *)
                              Signal ! INFB !ringing !B !A ;

                              StartAudibleRinging !A !B;
                              (* Observation point: Signal *)
                              Signal ! INFB !audibleringing !A !B ;

                                ( Onhook !A;
                              StopRinging !B!A;
                              StopAudibleRinging !A!B;
                              stop

                               [ ]

                              Offhook !B;
                              (*Observation point: Connection*)
                              Connection !INFB !A !B !true;

                              StopRinging !B !A;
                              StopAudibleRinging !A !B;

                              LogBegin !A!B!B;
                              (* Observation point*)
                              Billing !INFB ! A !B! B;
                          (
                           Onhook !A;
                           Disconnect !B!A;
                           LogEnd !A!B;
                           Onhook !B;
                            stop
                              [ ]
                           Onhook!B;
                           Disconnect !A!B;
                           LogEnd !A!B;   (*Add The Time*)
                           Onhook !A;
                           stop

                          )
                       )
  )
)
endproc
```

Bibliography

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

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

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

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

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

[6] 6th International Workshop on Feature Interactions in Telecom. Software Systems, Scotland, May, 2000.

[7] A. Aho, S. Gallagher, N. Griffeth, C. Scheel and D. Swayne. Sculptor with Chisel: Requirements Engineering for Communications services. In: K. Kimbler and W. Bouma (eds.), Fifth international workshop on Feature Interactions in Telecommunications Software Systems, IOS Press 1998, 45-63

[8] A. Dardenne, A. V. Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. Science of Computer Programming XX:3-50, 1993.

[9] A. Grinberg. Seamless Networks: Interoperating Wireless and Wireline Networks. Addison-Wesley, 1996

[10] A. Gammelgaard and J. E. Kristensen. Interaction Detection, a logical approach. In: L.G. Bouma and H. Velthuijsen (eds.) Feature Interactions in Telecommunications Systems. IOS Press, 1994 (Proc. of the 2nd International Workshop on Feature Interactions in Telecommunications Systems, Amsterdam) 178-196.

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

[12] A. Khoumsi. Detection and Resolution of Interactions between Services of telephone Networks. In: Feature Interactions in Telecommunications and Distributed systems IV, P. Dini et al. (Eds.) IOS Press 1997, 78-92

[13] A. Miga. Application of Use Case Maps to System Design with Tool Support. M.Eng. Thesis, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1998. http://www.UseCaseMaps.org/UseCaseMaps/ucmnav

[14] A. P. Felty and K. S. Namjoshi. Feature Specification and Automatic Conflict Detection. In: M. Calder and E. Magill, Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems, IOS Press, Glasgow 2000, 274-289

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

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

[17] B. Jonsson, T. Margaria, G. Naeser, J. Nystrom and B. Steffen. Incremental Requirement Specification for Evolving Systems. In: M. Calder and E. Magill, Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems, IOS Press, Glasgow 2000, 145-162

[18] B. Stepien, L. Logrippo. Status-Oriented Telephone Service Specification. IN: T.Rus and C.Rattray (eds.) Theories and Experiences for Real-Time System Developpement. AMAST Series in computing, Vol2, world Scientific, 1994, pages 265-286

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

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

[21] 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

[22] D. Amyot. Formalization of Timethreads Using LOTOS. MSc Thesis, 1994, University of Ottawa, Ottawa, Canada.

[23] D. Amyot, L. Charfi, N. Gorse, T. Gray, L. Logrippo, J. Sincennes, B. Stepien T. Ware. Feature Description and Feature Interaction Analysis with Use Case Maps and LOTOS. In: M. Calder and E. Magill, Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems, IOS Press, Glasgow 2000, 274-289

[24] D. Amyot, R. Andrade. Description of Wireless Intelligent Network Services with Use Case Maps. 17o. Brazilian Symposium on Computer Networks proceedings. May 1999.

[25] D. Amyot, R. Andrade, L. Logrippo, J. Sincennes, and Z. Yi. Formal Methods for Mobility Standards. In the IEEE 1999 Emerging Technologies Symposium: Wireless Communications

and Systems proceedings, Richardson (TX) April 1999. Editor: Traci King, Samsung Telecommunications America. Publisher: Steve Bootman, Hitachi Telecom

[26] F. Bordeleau, A Systematic and Traceable Progression from Scenario Models to Communicating Hierarchical State Machines. PhD Thesis, 1999. Carleton University, Ottawa Canada.

[27] G. Booch, Object-Oriented Design with Applications. The Benjamin Cummings Publ. Co., 1991

[28] G. Booch, I. Jacobson, J. Rumbaugh. Unified Modeling Language for Real-time systems design. Documentation set version 2.0, 1997.

[29] G. Bruns, P. Mataga, I. Sutherland. Features as Service transformers. In: K. Kimbler and W. Bouma (eds.), Fifth international workshop on Feature Interactions in Telecommunications Software Systems, IOS Press, 1998, 85-97

[30] G. V. Bochmann. Finite State Description of Communication Protocols. In: Computer Networks, Vol. 2 (1978), 361-372.

[31] I. Jacobson. Object-Oriented Software Engineering. Addison-Wesley, 1992

[32] I. Jacobson, G.Booch, J.Rumbaugh. The Unified software Development Process. Addison-Wesley, 1999.

[33] ITU, Recommendation Z. 120: Message Sequence Chart (MSC). Geneva, 1996

[34] J. Bergstra, W. Bouma. Models for Feature Descriptions and Interactions. In: Feature Interactions in Telecommunications and Distributed systems IV, P. Dini et al. (Eds.) IOS Press 1997

[35] 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.

[36] J. Lennox, H. Schulzrinne. Feature Interaction in Internet Telephony. In: M. Calder and E. Magill, Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems, IOS Press, Glasgow 2000, 38-50

[37] J. P. Gibson. Feature Requirements Models: Understanding Interactions. In: Feature Interactions in Telecommunications and Distributed systems IV, P. Dini et al. (Eds.) IOS Press 1997, 46-60

[38] J. Q. Fu. Feature Interaction Detection in a Telephony Network Integrated with Switch based Features and IN Features. Master's Thesis, University of Ottawa, 2000.

[39] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. Object Oriented Modeling and Design. Prentice-Hall, 1991.

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

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

[42] M. Faci and L. Logrippo. Specifying Hardware in LOTOS. Proceedings of the IFIP WG10.2. 11th international Conference on computer Hardware Description Languages and their Applications, Ottawa, Canada, April 1993, 305-312

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

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

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

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

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

[48] M. Heisel and J. Souquieres. A Heuristic Approach to detect Feature Interactions in Requirements. In: K. Kimbler and W. Bouma (eds.), Fifth international workshop on Feature Interactions in Telecommunications Software Systems, IOS Press, 1998, 165-171

[49] M. Kolberg and E. H. Magill. Service and Feature Interactions in TINA. In: K. Kimbler and W. Bouma (eds.), Fifth international workshop on Feature Interactions in Telecommunications Software Systems, IOS Press 1998, 78-84

[50] M. Nakamura, J. Hassine, L. Logrippo. Feature Interaction Filtering with Use Case Maps at Requirements Stage. In: M. Calder and E. Magill, Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems, IOS Press, Glasgow 2000, 163-178

[51] M. Yoeli and Z. Barzilai. Behavioural Descriptions of Communication Switching Systems Using Extended Petri Nets. In: Digital Processes, Vol. 3, 4, 307-320. 1977.

[52] N. D. Griffeth and H. Velthuijsen, The negotiating agents approach to run-time feature interaction resolution, In: Feature Interactions in Telecommunications Systems, L.G. Bouma and H. Velthuijsen (eds.), IOS Press, Amsterdam, 1994.

[53] 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

[54] P. Zave. Classification of research efforts in requirements engineering. ACM Computing Surveys XXIX (4): 315-321, December 1997

[55] P. Zave and M. Jackson. *Four dark corners of requirements engineering.* ACM Trans. Software Eng. and Methodology, 6(1):1-30, Jan. 1997. 85

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

[57] Recommendation Z100. Specification and Description Language SDL. CCITT SGX, Contribution com X-R 15-E, 1987

[58] R. J. A. Buhr, R. S. Casselman. Use Case Maps for Object-Oriented Systems. Prentice-Hall 1995, USA.

[59] R. J. A. Buhr. Use Case Maps as Architectural Entities for complex systems. In: Transactions on Software Engineering, IEEE, December 1998, pp. 1131-1155

[60] R. J. A. Buhr, D. Amyot, M. Elammari, D. Quesnel, T. Gray, and S. Mankovski, Feature-Interaction Visualization and Resolution in an Agent Environment. In: K. Kimbler and

W. Bouma (eds.), Fifth international workshop on Feature Interactions in Telecommunications Software Systems, IOS Press, 1998, 135-149

[61] R. J. Hall. Feature Combination and Interaction Detection via Foreground/Background Models. In: K. Kimbler and W. Bouma (eds.), Fifth international workshop on Feature Interactions in Telecommunications Software Systems, IOS Press, 1998, 232-246

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

[63] R. Zygan-Maus. Feature Interaction Management for Public Switched Networks. In: K. Kimbler and W. Bouma (eds.), Fifth international workshop on Feature Interactions in Telecommunications Software Systems, IOS Press 1998, 32-44

[64] R. Lai and A. Jirachiefpattana. Communication Protocol Specification And Verification. Kluwer Academic Publishers, 1998

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

[66] T. Bolognesi and Brinksma. Introduction To The ISO Specification Language LOTOS. In: The Formal Description Technique LOTOS, P.H.J van Eijk, C.A. Vissers and M. Diaz (Eds.). Elsevier Science Publishers B.V, North-Holland, 1989

[67] T. F. LaPorta, D. Lee, Y.-J. Lin, and M. Yannakakis. Protocol Feature Interactions. In S. Budkowski, A. Cavalli, and E. Najm, editors, Proc. Formal Description Techniques And Protocol Specification, Testing and Verification, FORTE XI/PSTV XVIII'98, 3-6 November, Paris, France, pages 59--74, 1998.

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

[69] Use Case Maps Web Page and UCM User Group, 1999. http://www.UseCaseMaps.org

[70] 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

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