

Formal Modeling and Test Generation Automation with Use Case Maps and LOTOS

Leila Charfi

Thesis submitted to the
School of Information Technology and Engineering
in partial fulfillment of
the requirements for the degree of

Master of Computer Science

under the auspices of the
Ottawa-Carleton Institute for Computer Science

University of Ottawa
Ottawa, Ontario, Canada

March, 2001

© Leila Charfi, Ottawa, Canada, 2001

To my parents

Abstract

This thesis addresses the problem of formal modelling and test generation, from system requirements represented in the form of Use Case Maps. In the first part of our thesis, we present an existent development methodology based on Use Case Maps for the design of the requirements and on LOTOS and SDL for the formal modeling of telecommunication systems. We follow this methodology for the formal specification and validation of a telephony system using LOTOS. In the second part of the thesis, we develop a method for the automatic generation of LOTOS scenarios from Use Case Maps called `Ucm2LotosTests`. The obtained scenarios can be used for the verification of the LOTOS specification built from the same Use Case Maps and for conformance testing purposes at the implementation stage. Finally, we propose a development methodology based on Use Case Maps for the design of the requirements and on LOTOS for the formal modeling of the system. In addition, this methodology offers a fast test generation process; it proposes the use of `Ucm2LotosTests` for the automatic generation of LOTOS scenarios from requirements in UCM and of TGV for the automatic generation of TTCN test suites from LOTOS. The methodology is illustrated with a case study which is a telephony system providing the basic call feature.

Acknowledgments

I would like to express my deepest gratitude to my supervisor Professor Luigi Logrippo who provided guidance and support during this research, and improved the contents of this thesis and its presentation.

I would also like to thank the members of the LOTOS research group for their useful discussions and ideas; in particular, I would like to thank Nicolas Gorse for his collaborative work during the achievement of the Mitel project, and Jacques Sincennes, Daniel Amyot and Rossana Andrade for their precious advice and suggestions.

I am also very thankful to Andrew Miga of Carleton University for his help on the reuse of the UcmNav tool and the test generation automation performed during this thesis. Dr Hubert Garavel of INRIA also helped us considerably with the use of the Caesar tool during a week's visit to our research group.

My deep thanks go to my parents and Mondher Ben Miled for their continuous support and encouragement during the production of this thesis.

Finally, I would like to express my gratitude to the following funding sources: The University Mission of Tunisia in North America, Mitel Corporation, Communication and Information Technology Ontario, and the Natural Sciences and Engineering Research Council of Canada.

Contents

1	Introduction	1
1.1	Introduction and Motivation	1
1.2	Contributions of the Thesis	2
1.3	Organization of the Thesis	3
2	Review of Basic Methods and Notations	4
2.1	Introduction	4
2.2	Overview of Use Case Maps	5
2.3	Overview of LOTOS	7
2.3.1	The Abstract Data Types Part	8
2.3.2	The Control Part	9
2.3.3	Specification Styles of Telephony Systems	12
2.4	Message Sequence Chart	12
2.5	Specification and Description Language	14
2.6	Tree and Tabular Combined Notation	14
2.7	Conclusion	16
3	Overview of Applicable Testing Theory and Existing Testing Tools	17
3.1	Introduction	17
3.2	Software Testing Techniques	18
3.3	Software Testing Phases	19
3.4	Testing LOTOS Specifications	20
3.4.1	Verification of LOTOS Specifications	21
3.4.2	Validation of LOTOS Specifications	22
3.5	Some Existing Test Generation Tools from Specifications	23
3.5.1	Structure of a Test Suite	23
3.5.2	Test Suite Generation with TGV	24
3.5.3	Test Generation and Execution with TorX	25
3.6	Conclusion	25
4	LOTOS Specification of Telephony Features for a New PBX	26
4.1	Introduction	26
4.2	Presentation of the Project	26
4.3	General Overview of the Development Methodology	27
4.4	Presentation of the Telephony Features	30

4.5	Presentation of the Requirements	31
4.6	UCM to LOTOS Transformation	33
4.6.1	Analysis of the Requirements	33
4.6.2	Graph Representation of the Features	33
4.6.3	UCM to LOTOS Mapping	35
4.6.4	Specification of a Simplified Basic Call	36
4.6.5	Specification of the Other Features	40
4.7	LOTOS Specification of the Telecommunication System	40
4.7.1	Abstract Data Types	40
4.7.2	Behavior of the Specification	41
4.7.3	Process DEB	43
4.7.4	Process Database	44
4.8	Verification of the LOTOS Specification	45
4.8.1	Test Scenarios	45
4.9	Analysis of the Results	48
4.9.1	Testing Results	48
4.9.2	Cross-Validation	48
4.10	Conclusion	49
5	Ucm2LotosTests: Automatic LOTOS Test Generation From Use Case Maps	50
5.1	Motivation	50
5.2	Desired Behavior of the Ucm2LotosTests Functionality	51
5.3	Internal Representation of the UCMs in UCMNav	51
5.4	Design of the Ucm2LotosTests Functionality	52
5.4.1	UCM Route Generation	52
5.4.2	UCM Routes to LOTOS Scenarios Transformation	59
5.5	Example of Test Generation with Ucm2LotosTests	62
5.6	Usefulness of Ucm2LotosTests	62
5.7	Automatic Test Generation Issues	63
5.8	Conclusion	66
6	Proposition of a Development Methodology based on Fast Test Generation	68
6.1	Introduction	68
6.2	Development Methodology based on UCMs and LOTOS	68
6.3	Advantages of the Methodology	70
6.4	Case Study	71
6.4.1	Introduction	71
6.4.2	Requirements	72
6.4.3	LOTOS Specification	72
6.4.4	LOTOS Test Generation	73
6.4.5	Testing the LOTOS specification	73
6.4.6	TTCN Test Generation	74
6.5	Limitations of the methodology	75

6.6	Conclusion	76
7	Conclusion	77
7.1	Contributions of the Thesis	77
7.2	Future Work	78
A	Case Study Details	80
A.1	Presentation of the Requirements	80
A.2	LOTOS Specification	82
A.3	LOTOS scenarios generated with Ucm2LotosTests	97
A.4	TTCN test case example generated with TGV	102

List of Figures

2.1	A simple UCM	6
2.2	Some Use Case Maps notation	7
2.3	A Message Sequence Chart and a corresponding LOTOS trace	13
3.1	A LOTOS specification and its corresponding LTS	21
3.2	Test generation using TGV	24
3.3	On-the-fly testing with TorX	25
4.1	The software development methodology approach	28
4.2	System architecture	32
4.3	Hierarchical relationship between stubs of the BC map	34
4.4	Example of UCM to LOTOS transformation using the mapping rules	36
4.5	Process of flattening stubs	37
4.6	BC map flattened into a simplified BC map without stubs	38
4.7	Graphical representation of the top-level specification	42
4.8	Message Sequence Chart of Scenario1	46
4.9	Message Sequence Chart of Scenario2	47
4.10	Message Sequence Chart of Scenario3	48
5.1	Example of UCM	51
5.2	UCMNav internal representation of the UCM of Figure 5.1	52
5.3	Routes of a UCM	53
5.4	Route generation with or_fork	54
5.5	Route generation with or_join	54
5.6	Route generation with or_fork followed by or_join	54
5.7	Route generation with and_fork	55
5.8	Route generation with and_join	55
5.9	Route generation with and_fork followed by and_join	55
5.10	Route generation for the static stub	56
5.11	Route generation for the dynamic stub	56
5.12	Route generation for the loop	56
5.13	Route generation for the waiting place	57
5.14	Route generation for the timed waiting place	57
5.15	Mapping UCM to LOTOS for the default LOTOS scenarios generation	60
5.16	Example of UCM route to LOTOS scenario generation using the default LOTOS scenarios generation	60

5.17	UCM to LOTOS elements transformation using a mapping M entered by the user	61
5.18	Mapping M entered by the user and used for the generation of the LOTOS scenario of Figure 5.19	62
5.19	Example of UCM route to LOTOS scenario generation using the mapping M of Figure 5.18	62
5.20	Possible mapping of the UCM elements of Figure 2.1	63
5.21	Automatic generation of 3 possible scenarios in LOTOS from a UCM	64
5.22	Possible mapping of the UCM elements of Figure 5.21 for white-box testing	65
5.23	Automatic generation of rejected scenarios	66
6.1	New software development methodology approach based on fast test generation	69
6.2	Root map of the Basic Call map	72
6.3	Scenario corresponding to a successful connection ending with caller hanging up first	73
A.1	Root map of the Basic Call map	80
A.2	Root map of the Basic Call map	81
A.3	Answer submap	81
A.4	Hangup originating submap	82
A.5	Hangup terminating submap	82
A.6	UCM to LOTOS mapping	83

Chapter 1

Introduction

1.1 Introduction and Motivation

In the past few years, telephony systems have undergone big improvements and changes. Increasingly complex features and increasing flexibility of use are offered to the customer. Telecommunication companies are constantly improving their systems by making their networks more reliable, adding new features and giving to the customer more control and freedom. Nevertheless, product reliability and time to market continue to be a source of concern.

The ISO (International Organization of Standardization) and the ITU-T (International Telecommunication Union International - Telecommunication Standardization Sector) have proposed a number of *Formal Description Techniques* (FDTs) for the *specification* of complex and reliable communicating systems. The main objective of FDTs is to allow the production of unambiguous specifications and to provide a well-defined basis for validation of the design and for conformance testing of implementations against the specification. In the telecommunication industry, processes using FDTs such as LOTOS, SDL and Estelle have proven their usefulness for increasing the reliability of systems, decreasing their cost, and making the design process more efficient.

In the development of telecommunication systems, we distinguish two stages:

- The *requirements stage* which represents the general behavior requirements for the system. Use Case Maps (UCMs) are the main notation for this purpose.
- The *formal modeling stage*, also called *specification stage* which provides a specification of the system using FDTs. The technique we use is LOTOS; however, other techniques to do this are SDL, Estelle, etc.

This thesis has several motivations. The first motivation is to demonstrate, once again, the usefulness of FDTs for the specification of telecommunication systems. This thesis is part of a joint project between industrial and academic research groups. This project focuses on industrial-strength tools to rapidly develop high quality telecommunications software. Our effort in this project is concentrated towards accelerating the software lifecycle, having as the main goal the fast validation of the telecommunication system under design using LOTOS and its tools.

Another concern that we address in this thesis is test suite generation from the design of a system. Some test generation tools based on formal specifications exist today. For instance, some tools generate from LOTOS tests test cases that are executable by implementations. The LOTOS tests are generally built manually. Our motivation in this thesis is to obtain these LOTOS tests by automatic generation from requirements. Since UCM is a semi-formal notation that represents scenario-based use cases and since it is already supported by a tool (UcmNav), it could be the basis for test suite generation from requirements.

1.2 Contributions of the Thesis

In the context of telecommunication systems specification using formal methods, this thesis presents three contributions:

First contribution: Design of a LOTOS model for a new Mitel PBX (Feasibility study and case study)

In chapter 4, we first present a development methodology for telecommunication systems essentially based on UCMs at requirements stage and LOTOS and SDL for formal modeling stage. Second, we describe the use of the development methodology presented for the LOTOS specification of a Private Branch eXchange (PBX) with 7 telephony features. The specification was manually generated from requirements in UCM form. After specifying the system in LOTOS, some LOTOS scenarios describing the behavior of the system were manually generated from the requirements and were used for validation purposes.

Second contribution: Automatic LOTOS scenario generation from UCMs (Requirement engineering methodology)

In chapter 5, we propose and implement **Ucm2LotosTests**, a method for the automatic LOTOS scenario generation from UCMs. The set of generated scenarios is used for validation at the formal modeling stage and for conformance testing purposes at the implementation stage. The set of generated scenarios is such that each UCM path is covered at least once.

Third contribution: Proposition of a new development methodology with fast test suite generation

We propose in chapter 6 a slightly different development methodology from the one used in chapter 4. This new methodology proposes the use of the UCMs notation at the requirement stage, the use of the formal method LOTOS at the formal modeling stage, and the use of new testing tools for validation and test suite generation purposes. The method **Ucm2LotosTests** is proposed for the automatic generation of LOTOS scenarios from UCMs. The tool **TGV** is proposed for the Tree and Tabular Combined Notation (TTCN) test suite generation from LOTOS scenarios. Thus, using **Ucm2LotosTests** and **TGV**, this development methodology proposes a fast test generation methodology that ensures coverage of all control flow described in the UCM.

1.3 Organization of the Thesis

The six remaining chapters will cover the following issues:

Chapter 2: Review of Basic Methods and Notations

We give an overview of the Use Case Maps notation and of the LOTOS specification language by describing their main operators and by giving examples in the context of telephony systems. The other formal methods used in the fast Spec-to-Test project are : MSC, TTCN and SDL. They are briefly presented.

Chapter 3: Overview of Applicable Testing Theory Concepts and Existing Testing Tools

We review briefly some related testing methods and phases from the literature. Then some existing tools for test suite generation from LOTOS are presented.

Chapter 4: LOTOS Specification of Telephony Features for a New PBX

This chapter presents the first contribution of the thesis. We describe the specification of a LOTOS model from UCMs representing the architecture of a new PBX.

Chapter 5: Ucm2LotosTests: Automatic LOTOS Test Generation From Use Case Maps

This chapter presents the second contribution of the thesis. It proposes an example of automatic LOTOS scenario generation from UCMs. It is shown that this method assures full coverage of the UCM paths.

Chapter 6: Proposition of a Development Methodology based on UCMs and LOTOS

In this chapter, we propose a telecommunication system development methodology based essentially on using UCMs for the requirements specification and on LOTOS for the design. The advantages of this methodology are listed, particularly the fast and automatic test suite generation based on the second contribution of the thesis. Some limitations are also presented.

Chapter 7: Conclusion

This last chapter concludes the thesis; it reviews the contributions with their benefits and limitations. Then some future work is proposed.

Chapter 2

Review of Basic Methods and Notations

This chapter introduces basic methods and notations for feature design, specification and validation of telecommunication systems: The UCM notation used at the requirements stage, the LOTOS language used at the formal modeling stage, the MSCs used to express test scenarios and TTCN, the test suite notation.

2.1 Introduction

Different methodologies for the design and specification of telephony systems are being used by research and industrial groups. The methodology adopted in this thesis is based on the use of the semiformal notation *Use Case Maps* (UCMs) [Bur96] [Bur98] at the requirements stage, and the *Language Of Temporal Ordering Specification* (LOTOS) [BB87] [ISO89] at the formal modeling stage.

UCMs is a recent notation, but it has already been adopted by some design groups to capture requirements for telecommunication systems.

LOTOS has been used for years for the specification and validation of telephony systems ([FLS90], [FLS91]), hardware systems ([KVZ99]) and for the detection of telephony feature interactions ([FL94], [Fac95], [Kam96], [SL95], [StL95], [Tuo96], [Tur98]). LOTOS-based tools are numerous. They verify the LOTOS specification and generate tests for the purpose of verifying the conformance of the implementation against the specification.

The use of LOTOS for the formal semantics of UCMs has been thoroughly studied in recent years. Other formal methods, such as Petri nets and event structures, can be considered as options [Amy94].

The combination of UCMs and LOTOS has been used to describe and validate telephony systems and network protocols such as distributed systems [ALB99], Wireless Mobile ATM networks [And00], General Packet Radio Services (GPRS) [AHL98], and telephony features of a new PBX architecture [ACG00]. This thesis is within the framework of this latter work.

The following sections present an overview of the UCM notation, the LOTOS language, the *Message Sequence Chart* (MSC) notation used to specify scenarios, and the *Tree and Tabular Combined Notation* (TTCN) used to specify test suites. The *Specification and Description Language* (SDL) is briefly presented.

2.2 Overview of Use Case Maps

Use Case Maps [Bur96] [Bur98] is a scenario based notation for gathering systems requirements. It describes *causal relationships* between activities of different *components* of the system. Activities are represented by *responsibilities* which are generic and can represent internal actions, tasks to perform, messages to send or receive and so on. Components are also generic and can represent software entities (objects, processes, databases, servers, functional entities, network entities, etc.) as well as non-software entities (users, actors, processors, etc.). The causal relationships represent sequencing of activities, activities triggering post-conditions or activities triggered by preconditions.

UCMs can represent either an abstract view of a system, or a more detailed one. It is up to the UCMs designers to define the amount of detail provided in the UCMs, in terms of actions performed and message exchanges.

The complete UCM notation elements are described in [Bur96] [Bur98]. In this thesis, we will only introduce the ones that we used. In this section we present a simple UCM. More complex ones are presented in chapter 4.

A UCM is represented by a top-level map, referred as a *root map* (it is represented in Figure 2.1 by the `simplifiedCallConnection` map), and possible *sub-maps*. Sub-maps of a map are referred to by stubs somewhere defined in this map. A stub is used to hide an internal behavior inside the behavior of a current map. Two kinds of stubs are defined:

- *static stub* represented as a plain diamond. It refers to a sub-map.
- *dynamic stub* represented as a dashed diamond. It may refer to one of a set of sub-maps. Each sub-map is referred to by a *plug-in*.

Each map includes one or several UCM *paths* that start with a *start point* and end with an *end point*. A start point represents a precondition that triggers the path following it. An end point represents a post-condition or a resulting effect from the actions performed in the path.

The UCM in Figure 2.1 represents a simplified call connection between two users interfaced by the components `Phone1` and `Phone2`. The component `Switch` represents the telephony network. This UCM is initiated in `Phone1` by the start point `offHook`. A path following this start point includes the responsibility `dial` in component `Phone1` and by the static stub `checkCalleesStatus` in component `Switch`. Inside this stub, the `checkIdle` responsibility checks if the callee is idle or busy. The *or-fork* splits the path into two. Only one of them can be chosen; in fact, a user can either be idle or busy but not in both states at the same time. The busy case is represented by the `[busy]` *condition* and the `Busy` end point of the stub, followed by the `busyTone` end point of the root map. The idle case is

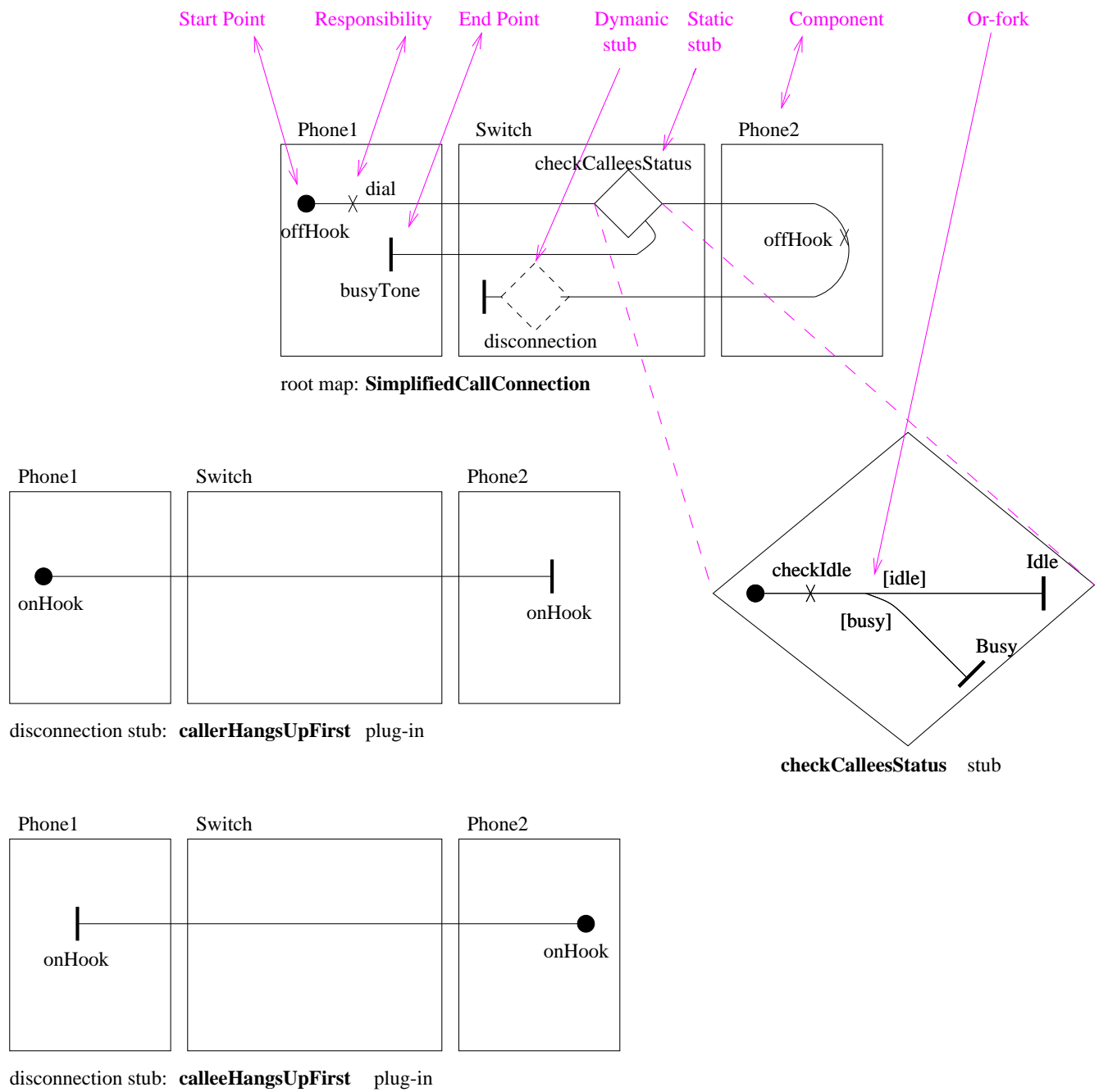


Figure 2.1: A simple UCM

represented by the `[idle]` condition and the `Idle` end point of the stub, followed by the responsibility `offHook` and the *dynamic stub disconnection* of the root map.

This example introduces the basic UCM notation. The next section gives the definitions of the other UCM notation used in this thesis.

Other UCM Notation Elements

In the previous section, we presented some *UCM elements*: responsibility, start point, end point, static stub, dynamic stub, `or_fork`, `or_fork` condition and component. In this section, we present additional UCM elements that were used in our project.









UCM element	Description	UCM element	Description
or join 	Indicates that several alternative paths join in a single path.	put value 	Put value in database
and fork 	Indicates that a single path is split into many concurrent paths	get value 	Get value from database
and join 	Indicates that several concurrent paths synchronize into a single path	waiting place 	Synchronization point for a scenario that pauses until a triggering event is received
loop 	Indicates that a path <code>p</code> is split into two paths. One of them goes back to <code>p</code>	timed waiting place 	The path <code>waiting_path</code> can continue on the path <code>continuation</code> if a timeout does not occur. Otherwise, it continues on the <code>timeout_path</code>

Figure 2.2: Some Use Case Maps notation

The UCMNav Tool

There currently exists only one tool that supports the UCM notation: the UCM Navigator (*UCMNav*) [Mig98]. It creates and modifies UCMs, ensures their syntactical correctness and generates their XML descriptions.

A new functionality of UCMNav was developed in this thesis and added to the tool. It deals with the LOTOS scenario generation from a UCM. This work constitutes one of the contributions of the thesis. It is detailed in chapter 5.

2.3 Overview of LOTOS

The Language Of Temporal Ordering Specification [ISO89] [BB87] is an FDT developed within the ISO as a formal specification language for the purpose of describing and specifying the different elements of OSI (Open System Interconnection) architecture such as services and protocols. Nowadays, its use has been extended to other domains such as the design of distributed systems.

LOTOS is a language for formal specification and formal modeling systems. By 'formal specification' we mean that it specifies the behavior of a system with a sound semantic basis.

By 'formal modeling' we mean that a LOTOS specification is executable, thus it constitutes an 'abstract model' of the system.

A number of LOTOS tutorials exist in the literature ([LFH92], [BB87], [Toc89]). Therefore, we limit ourselves to a very brief overview of the language and of its use in the context of our research.

A LOTOS specification consists of two main parts: the Abstract Data Types (ADT) part and the Control part, as follows:

```
specification systemName[gate1, ..., gateN]: <exit-behavior>

    (* Abstract Data Type part: data types and value expressions *)

behavior

    (* Control Part: system behavior *)

endspec
```

2.3.1 The Abstract Data Types Part

The ADT part defines the data types and value expressions needed to specify the behavior of a system. It is based on the formal theory of algebraic abstract data types ACT-ONE [EM85]. The most commonly used predefined libraries are *Boolean* and *NaturalNumber*. The library *Boolean* defines the constants *true* and *false* and defines the *not* operation that complements a Boolean value. The *NaturalNumber* library defines positive numbers (including zero).

For instance, if we need to create a data type that represents different tones in a phone such as the signals: *dialTone*, *ringTone* and *busyTone*, we would have an ADT as follows:

```
(* ADT *)
library Boolean endlib
type Tone is Boolean
  sorts Tone
  opns dialTone, (* dial tone *)
        ringTone, (* phone is ringing *)
        busyTone (* busy tone *)
        : -> Tone
  _ == _ : Tone, Tone -> Bool
  _ <> _ : Tone, Tone -> Bool

  eqns forall x, y: Tone ofsort Bool
        x == x = true;
        dialTone == ringTone = false;
        dialTone == busyTone = false;
        ringTone == dialTone = false;
        ringTone == busyTone = false;
        busyTone == dialTone = false;
        busyTone == ringTone = false;
endtype
(* end ADT *)
```

There are some extended capabilities for specifying abstract data types, such as extension, combination, conditional equations and renaming. But we won't cover them in our tutorial.

2.3.2 The Control Part

The Control Part is the part of the specification that describes the internal behavior of the system. It is defined by a *behavior expression* followed by possible *process* definitions. A behavior expression is built by *combining* LOTOS *actions* by means of operators and possibly instantiations of processes. By composition we mean sequence, choice or *parallelism*. A possible structure of the control part of a LOTOS specification is:

```
behavior
  (* General behavior of the system *)
  <behavior expression> (* involving P1 and P2 *)
where

  (* Processes definition *)
  process P1 [<gateList>](<valueList>):<exit-behavior>:=
    <behavior expression>
  endproc

  process P2 [<gateList>](<valueList>):<exit-behavior>:=
    <behavior expression>
  endproc
```

In this definition, <gateList> lists the *gates* through which the process will *synchronize*. The notion of synchronization is introduced in the section defining the general parallel composition operator.

The specification of <exit-behavior> could either be **exit** to indicate that the process terminates successfully, or **noexit** to indicate that the process cannot terminate. A further extension of the <exit-behavior> allows for passing values to the next process using <valueList>.

LOTOS Processes

A LOTOS process describes the behavior of a physical or logical entity in the system or a function. It appears as a *black-box* to its environment. By black-box we mean that the process internal behavior is hidden to the environment. The encapsulation provided by the process concept makes this part of the language highly suitable for specifying communicating objects in a telecommunication system. A process is also defined by a behavior expression.

LOTOS Gates

A process interacts with its environment by means of synchronization at common points called *gates*. LOTOS gates may be used to model logical or physical interfaces between a system and its environment. Values, specified by the ADT, may be passed and received at these gates during synchronization.

LOTOS Actions

The basic units in a behavior expression are *actions*. They are atomic, instantaneous and synchronous behaviors. Each action is associated with a gate, namely the gate at which the event occurs.

Two types of actions exist in LOTOS. There are actions that need to synchronize with the environment of the process in order to be executed; and there are *internal* actions, that a process can execute independently. These are unobservable to the environment and are represented by the internal action *i* (also called τ in process algebra).

The action denotation is structured, consisting of a gate name and an optional list of data fields (called *experiments*). A possible structure of a LOTOS action with experiments is:

```
gateName !valueSent ?valueReceived: valueType
```

Some examples of use of gates and experiments are shown in the section defining the general parallel composition operator.

Choice Operator: []

To indicate that alternative sequences of events are possible, LOTOS provides the choice operator: []. This latter is placed between two or more behavior expressions, and indicates that either (or any) of the specified behaviors is possible. For example, the expression:

```
phone_to_user !ringTone; RingBehavior [] phone_to_user !busyTone; BusyBehavior
```

can either synchronize with `phone_to_user !ringTone` then behave like `RingBehavior`, or synchronize with `phone_to_user !busyTone` then behave like `BusyBehavior`.

Parallel Composition Operators

Behavior expressions may be composed in parallel in three different ways. The simplest way is through the *interleave* operator (`|||`), which indicates that the processes continue independently and must agree to exit before the composition can exit. The parallel composition `||` requires strict synchronization on all non-hidden gates. The general parallel composition operator `|[<gateList>]|` requires that the processes synchronize on all the gates of the `<gateList>`.

General Parallel Composition Operator: |[<gateList>]|

An event occurs only if all processes that participate in it are ready for it. When an event takes place, all the processes involved in that event synchronize and have a common view of the event. This synchronization is interpreted as communication between these processes, and permits values to be passed between them.

As shown in the action definition, LOTOS allows value passing during synchronization on a gate. An action can offer a value `!valueSent`, or accept a value `?valueReceived`. If two processes synchronize on a gate, and both offer a value, it must be the same value in

order for the synchronization to occur. And if one offers a value and the other is waiting for a value of the same type, then this value is passed from the first process to the second.

Example of synchronization on gates:

```

Process exampleSynchro[synchroGate]: exit
  Process P1 [synchroGate]
    |[synchroGate]|
  Process P2 [synchroGate]
endproc

process P1[synchroGate]: exit
  synchroGate ?val: ID !message;
  ...
endproc

process P2[synchroGate]: exit
  synchroGate !1 ?msg: msgType;
  ...
endproc

```

In this example, processes P1 and P2 are sharing a gate: `synchroGate`. Each of them wants to execute an action with this gate. If the value `1` is defined as a possible value of sort `ID` and the value `message` as a possible value of sort `msgType`, then the two processes will execute together the action `synchroGate !1 !message`;

Hiding Operator

`hide <gateList> in Behavior` is used to hide actions on gates in `<gateList>`, which become internal for the environment. Thus these actions cannot synchronize with the environment.

Guarded Behavior

Once values have been passed in a behavior expression, we can impose conditions on further execution using the *guard* construct. For example, in the LOTOS fragment below, the value accepted at a gate `phone_to_user` must be `ringTone` for the `RingBehavior` to be executed, and `busyTone` for the `BusyBehavior` to be executed.

```

phone_to_user? t: tone;
(
  [t == ringTone] -> RingBehavior;
  []
  [t == busyTone] -> BusyBehavior;
)

```

Enable and Disable Operators: `>>` and `[>`

Processes may be composed sequentially using the enable operator (`>>`). Given 2 processes, P1 and P2, `P1 >> P2` indicates that if and when P1 terminates with an `exit`, process P2 will start.

The disable operator ($[>]$) indicates that one process may interrupt another. For example, $P1 [> P2$ indicates that process $P2$ may interrupt $P1$ at any point while $P1$ is executing or before $P1$ starts. The interrupt can only occur if the first action of process $P2$ is enabled. If the interrupt does occur, no further execution of $P1$ happens, and the execution continues with the actions of $P2$. If $P1$ terminates unsuccessfully, $P2$ may execute, while if $P1$ terminates successfully, the actions of process $P2$ cannot be executed anymore.

LOTOS provides a number of other constructs that will not be presented here since they are not used in this thesis.

2.3.3 Specification Styles of Telephony Systems

Visser, Scollo, v. Sinderen and Brinskma [VSS91] identify four major styles of specification. The first two styles, called monolithic and constraint-oriented styles, are intended for the early stages of design, while the state-oriented and resource-oriented styles are intended for later stages of design.

Monolithic style is characterized by the absence of hidden actions and the lack of parallel operators. The behavior of the specification is a choice ($[\square]$) between sequences of actions. Therefore, specifications written in this style could be very long and hard to read. Thus, this style is mostly used for debugging and test generation purposes.

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

State-oriented style where explicit system states are identified, e.g. by using state variables. Using this style may lead to increased readability of the specification in cases where the informal specification uses the state concept. It may lead to LOTOS specifications that can be implemented directly.

Resource-oriented style where the system is described by processes that represent different resources (or system components). The resources interact among themselves through interfaces, each resource being defined by a temporal ordering of both internal and external interactions. Interactions among internal modules are hidden. This style allows modularity and parallel structures. Therefore, it is useful for implementation specification.

In our LOTOS specification (parts are shown in chapter 4), we used the resource-oriented style in order to preserve the architectural model of the UCMs.

2.4 Message Sequence Chart

Message Sequence Chart [ITU96-2] is an ITU-T standard that is used to show sequences of messages interchanged between system components and their environments. We call these

scenarios. Designers create MSCs to specify the system behavior or to specify the test scenarios. These test scenarios are used for validation of the specification or during the implementation stage. In our project, MSCs have been essentially used to express scenarios used to validate the LOTOS specification. Lotos2msc and Msc2lotos converters [Ste00] have been recently developed in order to generate LOTOS traces (definition in section 3.4) directly from MSCs and vice versa. Both were used in the Fast Spec-to-Test project (details in section 4.3). MSCs were manually generated from UCMs in the project. MSCs can be created, edited and viewed by means of the MSC editor of the Tau toolset from Telelogic.

Lotos2msc Converter

In order to be able to convert a LOTOS trace into an MSC using Lotos2msc, a special format of LOTOS actions representing messages exchange has to be adopted:

```
<orig_to_dest> !<instanceOrig> !<instanceDest> !<message>
```

This is best explained by an example. A LOTOS action representing a message `offHook` going from the entity `user1` of type `user` to the entity `phone1` of type `phone` has to be expressed as follows:

```
user_to_phone !user1 !phone1 !offHook;
```

It is expressed in the MSC by an arrow labeled `offHook` and going from the entity `user1` to the entity `phone1`. Figure 2.3 shows an example of an MSC and its corresponding LOTOS trace.

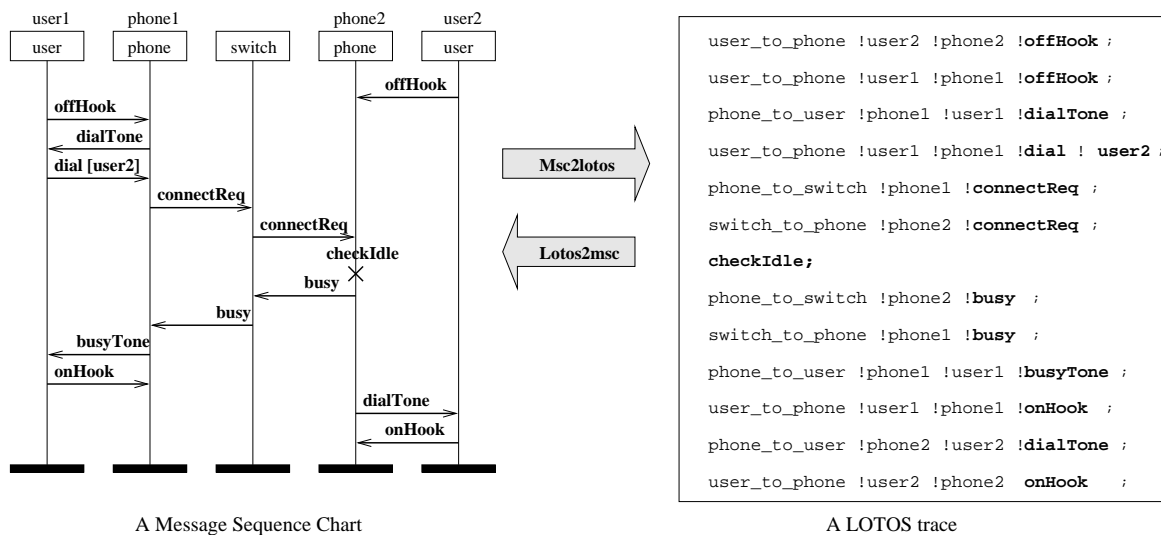


Figure 2.3: A Message Sequence Chart and a corresponding LOTOS trace

Note that it is not necessary to write an instance name when there is only one instance of a given entity. In fact, in the example of Figure 2.3, since there is only one instance of the entity `switch`, no instance number is carried in messages involving the `switch`.

2.5 Specification and Description Language

Specification and Description Language (SDL) [ITU96-1] is an FDT designed for reactive, concurrent, real-time, distributed, and heterogeneous system [AAL99]. It is used to describe both the behavior and the structure of systems, from a high description level down to a detailed design level. The behavior of a system is described by extended finite state machines represented by processes. Processes work concurrently and communicate asynchronously with each other by sending and receiving discrete messages called *signals*. Signals are also the means by which SDL processes communicate with the environment.

A problem with the SDL language is that it enforces rigid system boundaries in the form of processes and blocks. Although these are useful to represent system architecture, they may cause difficulties in the early design stages when the system architecture is not quite clear. LOTOS structure, which consists of only processes, is more flexible.

2.6 Tree and Tabular Combined Notation

Tree and Tabular Combined Notation (TTCN) [ISO92] is the standardized notation for specifying test suites that was recommended by ISO/IEC 9646. TTCN is well known within the telecommunications industry since it is widely used for conformance testing. Therefore, in the second part of our thesis, we focused our interest on TTCN test suite generation tools and on the possible automation of the test generation process (chapters 3 and 5). We will give a brief overview of TTCN. A more complete tutorial can be found in [KW91]

A TTCN description specifies a whole test suite (a general definition of a test suite is given in section 3.5.1). It consists of:

- a *test suite overview*: it is mainly a contents list of the test suite,
- a *declaration part*: it contains the definitions of all the message components that comprise the test suite: variables, timers, Points of Control and Observations (PCOs), and test components,
- a *constraint part*: it consists of conditions on message parameters, i.e. default values or value ranges which should be tested, and
- a *dynamic part*: it defines the *test cases* of a test suite in terms of trees of behavior.

As indicated by the name 'Tree and Tabular Combined Notation', a TTCN test suite is a collection of different tables. They are elements of the dynamic part.

Dynamic Part

A test case is generally composed of the following components:

- **Test purpose**: It describes the objective of the test case (expected behavior, verification goal, etc).

- **Test preamble:** It contains the necessary steps to bring the Specification Under Tests (SUT) into the desired starting state.
- **Test body:** It defines the test steps needed to achieve the test purpose.
- **Test postamble:** Used to put the SUT into a stable state after a test body is executed.
- **Test Verdict:** During test execution on a system, a test suite is carried out, resulting in a verdict FAIL, (PASS), PASS or INCONCLUSIVE. FAIL occurs when the test does not conform to the specification. (PASS) occurs when the purpose is reached but a postamble is needed to go to the initial state, this could lead to a FAIL. The result is PASS when the objective is reached and the system is in the initial state. Finally, it is INCONCLUSIVE when the test case purpose cannot be achieved.

Test Case Dynamic Behaviour					
Test Case Name : scenario1					
Group : End2EndCall/					
Purpose : To test that if user2 is busy then user1 gets the busy					
: tone when he tries to call user2.					
Default :					
Comments :					
Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		user2 !offHook	scenario1_001		
2		user1 !offHook	scenario1_001		
3		user1 ?dialTone	scenario1_002		
4		user1 !dial	scenario1_003		
5		user1 ?busyTone	scenario1_004		
6		user1 !onHook	scenario1_005		
7		user2 ?dialTone	scenario1_002		
8		user2 !onHook	scenario1_005	PASS	
9		user2 ?dialTone	scenario1_002		
10		user1 ?dialTone	scenario1_002		
11		user1 !dial	scenario1_003		
12		user1 ?busyTone	scenario1_004		
13		user1 !onHook	scenario1_006		
14		user2 !onHook	scenario1_006	PASS	
15		user2 !onHook	scenario1_006		
16		user1 !onHook	scenario1_006	PASS	
17		user2 !onHook	scenario1_006		
18		user1 ?dialTone	scenario1_002		
19		user1 !dial	scenario1_003		
20		user1 ?busyTone	scenario1_004		
21		user1 !onHook	scenario1_006	PASS	
22		user1 ?callInProgress	scenario1_007	(INCONC)	

A TTCN test suite is automatically generated from the composition of a formal specification S (written for instance in LOTOS or in SDL) and a test T (generally presented in MSC

form). If the test T is a sequence, an alternative or a composition between sequences and alternatives of actions, then the TTCN test suite is a bigger tree that includes T . The added actions in a TTCN test suites are actions that can or cannot be executed on the specification S . A verdict associated with these actions provides this information.

The most recent version of the TTCN language, TTCN-3 [GWW00], supports several notations that are equivalent. One of them is a textual notation (TTCN-MP for TTCN Machine Processible form). The test case presented above is an example of the dynamic part of a TTCN test suite in MP format. It is part of a test suite that was generated from an SDL specification and the MSC of Figure 2.3. A branch of the test case with a verdict **PASS** corresponds to the sequence of message exchanges of the MSC. The other branches of the test case correspond to other alternatives of message exchanges. Their verdicts show whether or not the message sequencing is allowed in the SDL specification.

2.7 Conclusion

The chapter presented an overview of UCMs, LOTOS, MSC, SDL and TTCN. UCMs are suitable for capturing requirements of distributed systems. LOTOS is a formal and unambiguous specification. In addition, LOTOS operators allows easily to specify a behavior of distributed and communicating systems. A development methodology based on UCMs for the description of the requirements and LOTOS for the specification of the system is adopted in this thesis for the specification of telephony features of a PBX. It is detailed in chapter 4. MSCs are a graphical representation of scenarios. They are adopted in this thesis for the description of the generated scenarios. Finally, TTCN is suitable for writing test suites. In the development methodologies that we present in the thesis, this notation is adopted to write test suites that will be executed on implementations.

Chapter 3

Overview of Applicable Testing Theory and Existing Testing Tools

In this chapter, we present an overview of software testing techniques and phases and two software testing tools for LOTOS models: TGV and TorX.

3.1 Introduction

System failure in any industry can be very costly. One way to increase system reliability is to perform software testing in combination with appropriate techniques to solve the identified problems. One of the testing principles announced by Myers [Mye79] is:

Testing is the process of executing a program or system with the intent of finding errors.

Testing is usually carried out in a special environment by means of experimentation. It can never be exhaustive for any realistic system of a very large or infinite number of allowed behaviors since the time and effort that can be spent on it is always limited by practical and economical considerations. As a result, as Pressman [Pre97] states (following a well-known statement by Dijkstra):

Testing cannot show the absence of defects, it can only show that software errors are present.

A thorough review of testing theory would be very long. We are limiting ourselves to reviewing the concepts that are used in the thesis. We will discuss some software testing techniques and phases. Then we will present some existing testing tools for LOTOS.

3.2 Software Testing Techniques

There are many ways to conduct software testing, but the most common techniques rely on the following concepts [Pre97]:

1. **Test Case Design** Test cases should test the program by using inputs that could be correct or incorrect, producing outputs that will reveal possible errors. Variables should be tested using all possible values (for small ranges) or typical and out-of-bound values (for larger ranges). They should also be tested using valid and invalid types and conditions. Arithmetical and logical comparisons should be examined as well, again using both correct and incorrect parameters.
2. **White-Box Testing** It is a test case design technique that relies on intimate knowledge of the code. It determines all possible paths in a module¹ and tests all logical expressions.

Using white-box testing, the tester can guarantee that all independent paths within a module have been covered at least once; examine all logical decisions on their *true* and *false* sides; execute all loops at their boundaries and within their operational bounds; and exercise internal data structures to assure their validity [Pre97].

3. **Basis Path Testing** It is a white-box technique first proposed by McCabe [McC76]. It allows the design of sets of test cases that examine each possible path in a program by executing every branch in the program at least once during testing. It should be monitored by coverage measurement.
4. **Control Structure Testing** It is a white-box technique that examines each possible path through the program by executing each statement at least once. It includes basis path testing (presented above), condition testing, branch testing, domain testing, data flow testing and loop testing. These notions will not be explained in this thesis but we understand by their names that they test the execution paths for every structure of a program.
5. **Black-Box Testing** Unlike white-box testing, black-box testing focuses on the overall functionality of the software. That is why it is the chosen technique for designing test cases used for *functional testing* (definition in section 3.3). This technique allows the functional testing to uncover faults like incorrect or missing functions, errors in any of the interfaces, errors in data structures or databases and errors related to performance and program initialization or termination.
The disadvantage of black-box testing is that in case of failure of a test, it is very hard to find the errors since it is impossible to locate them by looking at the test.
6. **Grey-Box Testing** Between white-box and black-box testing is grey-box testing. This technique does not test the internal behavior of the system entities and does not test the external behavior of the system but focuses on the interactions between the entities.

¹the term module is used to describe a logical division of the functionality of the system which contains a set of strongly coupled functions and procedures.

3.3 Software Testing Phases

In order to construct a proper and thorough set of tests, the testing phases mentioned below are generally performed in the order in which they are described. They use one or more software testing techniques described in section 3.2.

Humphrey [Hum90] identifies the following software testing phases:

1. **Unit Testing** It is a process of testing the individual subprograms, subroutines, or procedures in a program. That is, rather than initially testing the program as a whole, testing focuses first on the smaller building blocks of the program.
2. **Integration Testing** It focuses on testing multiple modules working together.
3. **Functional Testing** It is a testing process that is black-box in nature. Its goal is to take a user's view of the system, and check that customer requirements have been met. In particular, the actions and reactions of the system are considered from the user's viewpoint, to ensure that the system behaves as the user expects [PrW99]. In functional testing, different classes of test scenarios are defined: *Primary scenarios* which test the expected behavior of the system (success paths); *Secondary scenarios* which test its unexpected behavior (fail paths, race conditions, etc.); *Low yield* scenarios which describe situations and actions which are generally understood and are most likely to pass; And *high yield scenarios* which are scenarios that are not well documented, and therefore are not well understood and are most likely to fail [MaP01]. These definitions are not developed further in this thesis.
4. **Regression Testing** When fixing an error or adding a new functionality in an existing program, the code is modified. The modifications could be minor (adding, deleting or modifying few lines of code), or major (adding, deleting or modifying modules of the program). In both cases, the specification of the program may or may not be changed. The objectives of regression testing are to insure that the modified parts of the software system still satisfy their original unmodified requirements and that the previous functionality of the software, which should not be affected by the modifications, has indeed not been affected. Only those parts that are affected by the modification need to be retested.
5. **System Testing** It is the final stage of the testing process. This type of test involves examination of all the software components, of all the hardware components and of any interfaces. It is designed to reveal bugs that cannot be attributed to individual components. It concerns issues and behaviors that can only be exposed by testing the entire integrated system or a major part of it.
6. **Acceptance Testing** It is the process of comparing the program to its initial requirements and to the current needs of its end users.

3.4 Testing LOTOS Specifications

Several tools for testing a LOTOS specification are available today. In this section we present some testing techniques offered by the majority of the existing tools. We describe how they are performed on specifications using the tool LOLA (LOtos LABoratory) [QPF88]. This tool was used in this research to test our LOTOS specifications.

With relation to testing, a distinction is often drawn between the terms *verification* and *validation*:

- Verification is used to describe any checking of a design that includes the internal behavior of the system. Another definition from Boehm [Boe81] is:

Are we building the product right ?

- Validation applies only to confirm that end-to-end behavior requirements are met. The definition from Boehm [Boe81] is:

Are we building the right product ?

Accordingly, we distinguish between verification and validation aspects in our testing methodology. The semantics of LOTOS is defined in terms of *Labeled Transition Systems* (LTSs). Solving the problems of test derivation for LTSs therefore solves the problem for LOTOS behaviors as well. In fact, testing a LOTOS specification is done by testing the LTS representation of the specification behavior. An LTS can be considered as a special type of finite state machine [HU79].

Definition Labeled Transition System

As defined in [Bri88], a Labeled Transition System (LTS) is a 4-tuple (S, A, T, s_0) , where :

1. S is a (countable) non empty set of states;
2. A is a (countable) set of visible actions a ;
3. $T = \{-b \rightarrow \subseteq S \times S \mid b \in A \cup \{i\}\}$ is a set of transitions, where i is the invisible action.
4. s_0 is the initial state;

Let s denote a string of actions, $s = a_{a_1} \dots a_n$, and let i^k be a string of k internal actions:

$B \xrightarrow{s} B'$ is defined as follows: $\exists B_1, \dots, B_n \mid B \xrightarrow{a_1} B_1 \dots B_{n-1} \xrightarrow{a_n} B_n = B'$

$B \xRightarrow{s} B'$ is defined as follows: $\exists i^{k_0} a_1 i^{k_1} a_2 \dots a_n i^{k_n} \mid B \xrightarrow{i^{k_0} a_1 i^{k_1} a_2 \dots a_n i^{k_n}} B'$

$P \xRightarrow{\sigma} P'$ is defined as follows: $\exists P' \mid P \xRightarrow{\sigma} P'$

The set of traces of B , $\text{Tr}(B) = \{\sigma \mid B \xRightarrow{\sigma} \}$

Given a LOTOS specification of a system and its corresponding LTS, the state s_0 is the root of the LTS; it represents the behavior of the system when no action has been performed. An element of the set S expresses a state in the LTS, which is a behavior in the LOTOS

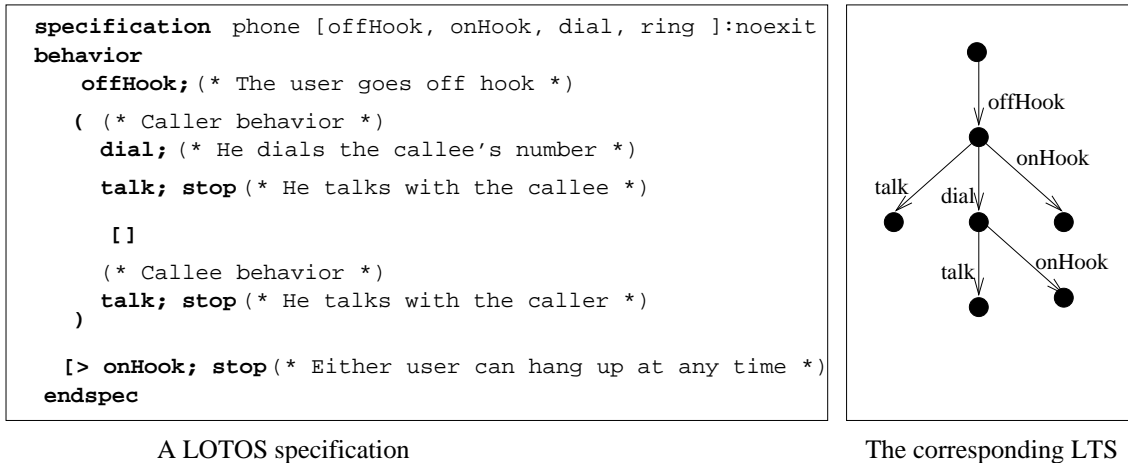


Figure 3.1: A LOTOS specification and its corresponding LTS

specification. A transition from a state of the LTS to another represents the execution of a LOTOS action transforming the behavior of the system. Actions can be external (visible) or internal. A trace of the system is a sequence of external actions that the system can perform.

Figure 3.1 shows a simple LOTOS specification and its corresponding LTS. In the specification, `offHook` can be executed at first. No other actions are possible. This is represented in the corresponding LTS by an action labeled `offHook` in the transition leaving the root of the LTS and going to an other state. The LOTOS behavior between parentheses in the specification can be disabled by the action `onHook`. The possible traces specified within the brackets are `talk`, or `dial` then `talk`. The alternatives between these LOTOS actions are expressed in the LTS by different transitions leading to different states.

When testing a LOTOS specification using a tool, the tool converts the specification into an LTS. Verification and Validation is performed on this structure.

3.4.1 Verification of LOTOS Specifications

Interactive simulation is a verification technique used often to verify a LOTOS specification. It is mainly used in the early stages of the design process. The interactive simulation is performed by applying the *step-by-step execution* of the LTS corresponding to the specification. This technique is considered as white-box testing (definition in section 3.2) since all possible paths in each process of the LOTOS specification are tested.

LOLA was used in this research to perform step-by-step execution of our LOTOS specification using the `Step` option. A set of possible LOTOS actions is proposed after each execution of an action. One action is chosen to go to a next state. The interactive simulation stops when no further actions can be performed.

3.4.2 Validation of LOTOS Specifications

The validation of a specification is performed against initial requirements. This may be regarded as a black-box testing (definition in section 3.2) since we are concerned only with the observable behavior of the system.

Using the LOLA tool, we can perform the validation of a LOTOS specification by applying the `testexpand` command on the specification. Inputs are the LOTOS specification and LOTOS tests. The LOTOS tests are usually generated manually from the requirements (a contribution of this thesis is the automation of this step).

We distinguish two types of tests:

- Tests that check against expected behaviors of the system according to the requirements. They are called *acceptance tests*,
- Tests that express faulty behaviors of the system. They are called *rejection tests* [CaT95].

A LOTOS test (also called LOTOS scenario) is composed of a LOTOS behavior (most commonly a sequence of LOTOS actions) followed by a special gate (for instance **SUCCESS** for acceptance tests and **REJECT** for rejection tests). The LOTOS test and the current behavior are composed in parallel, synchronizing on all the gates but the special gate.

The `testExpand` option is executed in LOLA as follows:

```
lola> testExpand special_gate TestProcess
```

LOLA analyses whether the execution of the test on the specification reaches the special gate or not. Three types of results can be obtained:

- **Must Pass**: all possible executions are successful, they reach the special gate.
- **May Pass**: some executions are successful and some unsuccessful.
- **Reject**: all executions failed to reach the special gate.

In case the acceptance tests aren't **Must Pass** or the rejection tests aren't **Reject**, the specification is reviewed in order to correct its behavior since it has either unexpected deadlocks or unwanted behaviors.

Feature Interaction Detection

The validation of telecommunication systems also checks for possible *Feature Interactions*. The feature interaction problem represents the case where a feature isn't working properly according to its intent because of some unexpected interactions with other features in the system.

An example of feature interaction involving *Originating Call Screening* (OCS) and *Call Forward Always* (CFA) features (definitions in section 4.4) is as follows: user A has OCS to user C. User B has CFA to C. When A calls B, the call is forwarded to C because of CFA,

but A should not talk to C because of OCS. Should C's device ring because the call is meant for B or should the call be denied because A isn't allowed to talk to C?

The choice on how the system should behave is decided by the designers.

A feature interaction scenario involves the presence of two or more features in the system and expresses a possible behavior of the system.

The application of our work to feature interaction detection is discussed briefly in section 4.8

Structural Coverage

Another way to validate a specification is *structural coverage* [AL00]. It consists in inserting counters in different parts of the specification and measuring the structural coverage of a LOTOS specification against validation test suites. It is very useful since it detects incomplete test suites and unreachable parts of the specification.

Model Checking

Model checking is also a form of validation. Desirable properties for the system expressed in terms of temporal logic formulas can be checked on the LOTOS specification using the LMC (Lotos Model Checker) tool [Gri92].

Note that structural coverage and model checking techniques were not used in this research, so they are not discussed further.

3.5 Some Existing Test Generation Tools from Specifications

Since we built specifications only in LOTOS, our interest in this section is focused on test generation tools from LOTOS.

The TTCN language (definition in 2.6) has been defined for writing abstract test suites and for performing conformance testing on existing implementations. It is now supported by many test generation tools. Test generation tools from formal specifications are often used to generate TTCN test suites for the purpose of testing the conformance of an implementation against a specification.

3.5.1 Structure of a Test Suite

Conformance testing involves applying test suites. Test suites are used to describe control and observation of the implementation under test and assign verdicts to test outcomes.

The structure of a test suite is hierarchical. It comprises a number of test cases and is associated with a unique standardized test purpose.

A test case is a sequence of actions describing all the interactions occurring between an *Implementation Under Test* (IUT) (i.e. an implementation which is being assessed on its

correctness by testing) and a tester which wants to verify that an implementation conforms with the specification according to a test purpose. In an industrial context, test cases are often described using TTCN. Some transitions are decorated with verdicts that could have one of these values: (PASS), PASS, FAIL, INCONCLUSIVE (definitions in section 2.6).

The test cases may be grouped into test groups, which in turn may be grouped into larger test groups. The test cases may also be decomposed into test steps and the test steps into test events (or test sequences).

The area of test suite generation, especially concerning techniques based on formal specifications, is an area of active research and development.

3.5.2 Test Suite Generation with TGV

TGV (Test Generation with Verification technology) [FJJ96] is a tool dedicated to the automatic generation of conformance tests based on formal techniques such as SDL and LOTOS. It is part of *CADP* (CAESAR/ALDEBARAN Development Package) [FGM92], a toolset that offers a wide range of functionalities, from interactive simulation (using the tool *Caesar*) to the most recent formal verification techniques (using *TGV*) for formal languages such as LOTOS.

As seen in Figure 3.2, an LTS is generated from a LOTOS specification using Caesar within the CADP toolset. LOTOS test purposes (randomly generated from the LOTOS specification or manually generated from the user's requirements) are also converted into LTSs within the CADP toolset. *TGV* takes as inputs the LTSs obtained from the LOTOS specification and the formalized test purpose, and generates a more complex LTS decorated with verdicts. The obtained test format is called *Aldebaran* format. An Aldebaran to TTCN converter, *Aut2ttn*, is used to obtain TTCN tests.

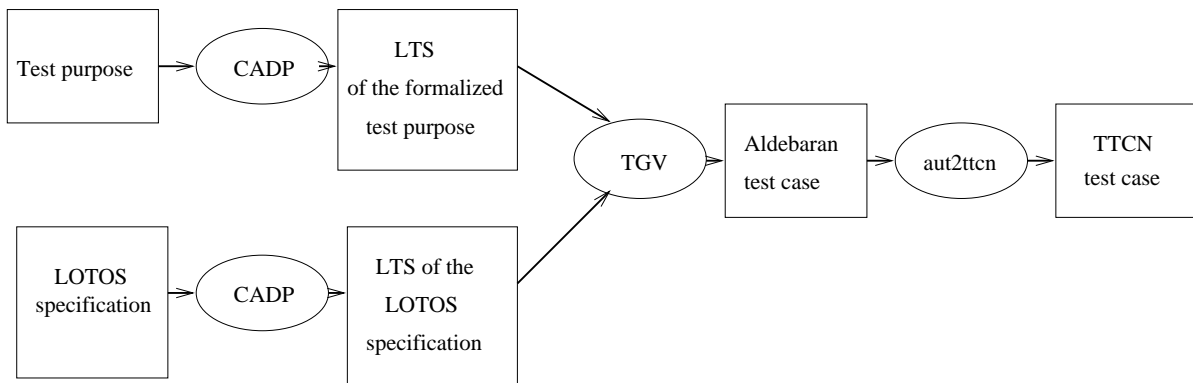


Figure 3.2: Test generation using TGV

TGV was used in the case study presented in section 6.4. Some problems encountered while using this tool are discussed in section 6.5.

3.5.3 Test Generation and Execution with TorX

TorX [BFV99] is a tool dedicated to the test generation and execution from formal specifications. It allows *on-the-fly* testing for LOTOS and PROMELA, and batch test derivation for SDL. On-the-fly test generation techniques perform the generation of tests cases from the composition of a specification and a test purpose without complete generation of the specification state space but only of the part that behaves like the test purpose. In addition to test generation, TorX performs test execution on implementations.

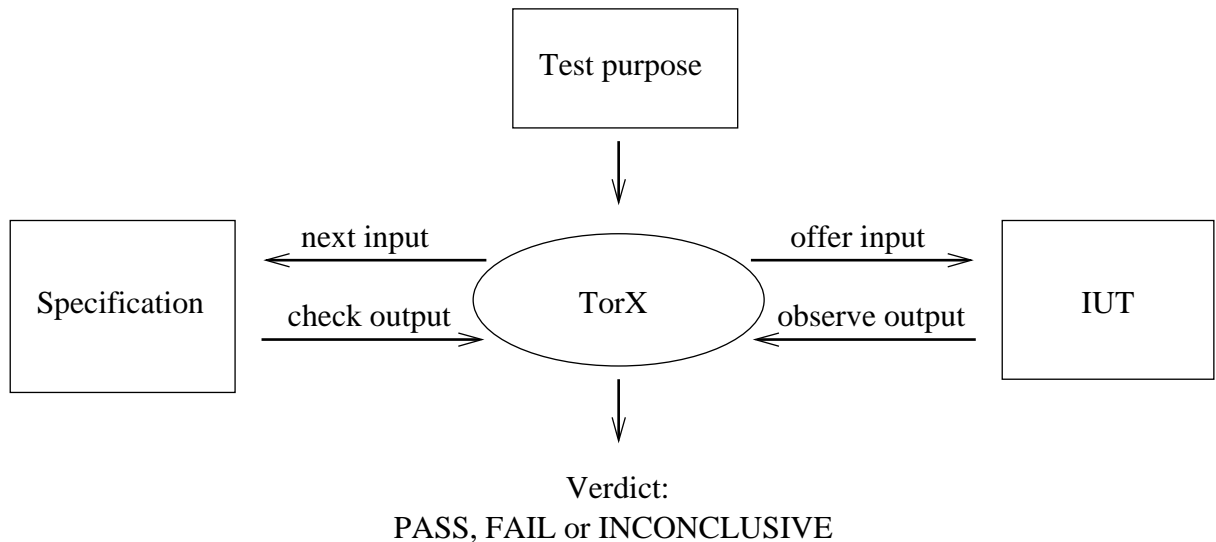


Figure 3.3: On-the-fly testing with TorX

In case of test generation from a LOTOS specification (Figure 3.3), test purposes are manually generated from requirements or automatically generated from the specification. TorX applies the actions of a test purpose to the specification and the implementation and observes the corresponding outputs. A verdict is generated from the comparison between the specification and the implementation outputs.

3.6 Conclusion

The first part of the chapter discusses a number of commonly used telecommunications software testing techniques and phases. While testing software, a combination of techniques and phases is usually necessary to develop a good set of test cases against which the software can be evaluated.

The second part of the chapter introduces two testing tools for LOTOS models: TGV and TorX. The use of TGV is proposed in the development methodology presented in chapter 6 of the thesis.

Chapter 4

LOTOS Specification of Telephony Features for a New PBX

This chapter describes in detail the steps followed in the construction of a LOTOS model from a UCM model of a telephony system.

4.1 Introduction

In this chapter, we describe our work in the Fast Spec-to-Test project. After presenting the project, we will review the different phases of the development process that we achieved such as the analysis of the requirements (in UCM form), the LOTOS specification of the telephony system described in the UCMs, and the verification and validation of the specification using manually generated scenarios from the UCMs.

4.2 Presentation of the Project

The Fast Spec-to-Test project was a collaborative project between Mitel Corporation and the University of Ottawa. It was intended to build a formal model of a new generation of a Private Branch eXchange (PBX) using a development methodology based on formal methods such as: LOTOS, SDL, MSC and TTCN.

The principal aims of this project were to:

- Demonstrate the rapid and inexpensive formal modeling phase using Use Case Maps to specify the requirements and using LOTOS or SDL (or both) to specify the formal model of the system.

- Rapidly develop a set of test cases which will be used for conformance testing purposes at the implementation stage. The set of test cases assures the full coverage of the UCM paths.

Three teams were involved in the Fast Spec-to-Test project. The Mitel team provided the set of requirements written in UCM form. Two teams from the University of Ottawa were in charge of the formal modeling stage (or *specification stage*). The LOTOS team was working on the early specification stage using LOTOS, and the SDL team on the late specification stage using SDL.

The LOTOS team had to provide to the SDL team:

- Possible design errors found in the UCMs.
- Some non-existent design details that had to be created for the LOTOS specification.
- A set of MSCs that attempted to cover all the LOTOS specification. They were used by the SDL team as an input to the SDL specification.

The work done by the LOTOS team is part of this thesis; therefore, it will be detailed later on in this chapter.

4.3 General Overview of the Development Methodology

A number of different specification languages and techniques are available today. They are partially complementary and can be used in combination. The justification of the use of several specification techniques is given in [AAL99]. The development methodology followed in the Fast Spec-to-Test project is essentially based on the use of:

- Use Case Maps for the description of the telephony features at the requirements stage,
- LOTOS and SDL for features specification, verification and validation at the formal modeling stage.

Several tools are used to link different phases of the formal modeling stage and to transform different notations into others.

The starting point of this methodology are the UCMs as a semiformal description of the features. Some plain English documentation (informal description of the features, roles of the system components, etc.) can also be part of the requirements.

- (1) From the UCMs, a LOTOS specification is derived. The translation from UCMs to LOTOS corresponds to the transformation and formalization of an abstract, semiformal model into a less abstract, formal and executable one. First, the UCMs are analyzed. Design errors can already be corrected at this stage. Second, a LOTOS specification is derived following some UCM to LOTOS mapping rules presented in section 4.6.3.

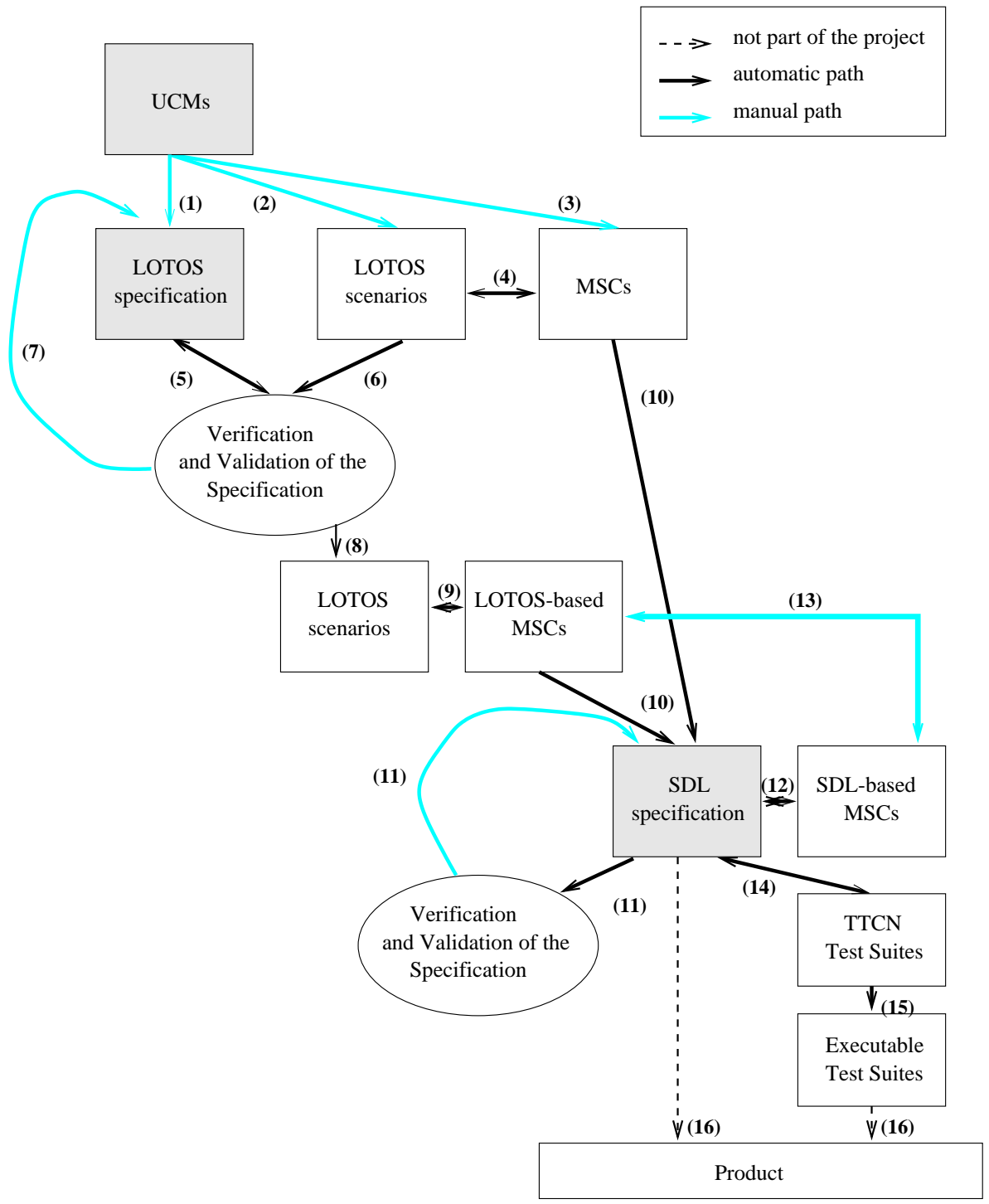


Figure 4.1: The software development methodology approach

In this experience, the LOTOS specification that we obtained from the UCMs was hand-prepared. However, some work has been done by Amyot to formalise the transformation of a UCM map into a LOTOS specification [Amy94]. In addition, the automation of the transformation of UCMs into a skeleton of LOTOS specification is currently being investigated within the LOTOS research group of the University of Ottawa. This automation is not part of our work.

- (2) LOTOS scenarios are manually generated from the UCMs in such a way as to cover the UCM. LOTOS scenarios are useful for the validation of the specification against the requirements and for the verification of conformance of the implementation against the specification.

The method **Ucm2LotosTests** designed and implemented in this thesis (details in chapter 5) automates this step of LOTOS scenario generation from UCMs.

- (3) MSCs are manually generated from UCMs. They are used at different stages of the development process.
- (4) Using the `Msc2lotos` converter and its conventions, MSCs are automatically converted into LOTOS traces.
- (5) Verification of the detailed behavior of the LOTOS specification is performed using step-by-step execution of the specification with the LOLA tool.
- (6) LOTOS scenarios are executed on the specification for validation purposes. If the execution is not successful, then the specification does not behave correctly and has to be modified.
- (7) Corrections are made to the specification in case of bad design.
- (8) LOTOS traces are generated from the specification. They are added to the set of LOTOS scenarios already generated in (2). This new set of tests is used to validate the SDL specification and to test the implementation.
- (9) Using the `Lotos2msc` converter and its conventions, LOTOS scenarios are automatically converted into MSCs. We call these MSCs *LOTOS-based MSCs*.
- (10) An SDL specification is built from the MSCs.
- (11) Verification and Validation of the SDL specification is carried out using the Tau toolset.
- (12) A set of MSCs that covers the complete SDL specification is generated with the Tau toolset. We call these MSCs *SDL-based MSCs*.
- (13) Cross-validation is performed between the LOTOS-based MSCs and the SDL-based MSCs. The differences found are discussed and possible changes on the SDL specification are made.

- (14) TTCN test suites are generated from the SDL specification using the Autolink tool integrated in the Tau toolset.
- (15) TTCN test suites are transformed into executable ones.
- (16) Implementation code is generated and conformance testing of the product is performed by executing the tests generated in (15) on the product.

One of the most important features of this methodology is that it uses more than one formal method to model the design (LOTOS, SDL and MSC). Therefore, using the cross-validation strategy (13), the specification obtained is most likely to have the expected behavior. The results of the cross-validation on the Fast Spec-to-Test project are presented in section 4.9.2.

4.4 Presentation of the Telephony Features

A *feature* is a collection of services packaged together that can be commercialized. 7 telephony features are part of the project. They are listed below:

- Basic Call (BC): It represents the basic call connection between two users interfaced by their access devices. An access device is most commonly a phone. The BC feature includes the connection request by a caller and the answer of the callee.
- Call Forward On Busy (CFO): User A has CFO to user B means that when A is busy, the incoming calls are forwarded to B.
- Call Forward Always (CFA): User A has CFA to user B means that each time A has an incoming call, this call is forwarded to B.
- Outgoing Call Screening (OCS): User A has OCS for user B means that its outgoing calls to B are blocked.
- Call Hold (CH): It enables a user to place the current call on hold, and/or retrieve the held calls.
- Call Transfer (CT): It allows a user to place a call on hold, then to dial and consult with a third party, then to transfer the second party to the third party.
- Call Pickup (CP): It allows a user within a group of users in a same local network to answer a call connection request made to the group of users from his own access device. If more than one party in the group attempts to pick up the call, the call will be completed to the first party seizing the call, and other parties will receive the reorder tone.

4.5 Presentation of the Requirements

The requirements of the system to specify in LOTOS describe the behavior of the 7 features presented in section 4.4. They were described in UCM form. The set of UCMs contains:

- 28 static stubs,
- 29 dynamic stubs,
- 49 plug-ins.

The terms *originating* and *terminating* will be used later on in this chapter. An originating component (or party) is a component (or a party) from the caller side. Terminating ones are from the callee (or called party) end.

The Features

The *BC map* (Basic Call map), which deals with the basic call connection request, is common to the UCMs of all the features. Either they include it or they are part of it. The BC map has several plug-ins and different plug-ins are executed depending on the features as follows:

- The BC feature is defined in the BC map. Its behavior is represented in the default plug-ins of the stubs composing the BC map.
- The CT feature is defined by a map containing the BC stub (which points to the BC map).
- The CP feature is also defined by a map containing the BC stub.
- The CH feature is represented by a map with, as a precondition, the BC map.
- The CFA and CFO features are defined by the BC map. But, at a certain point, a plug-in that redirects the call to a third party is chosen against a default one used for the BC.
- The OCS feature is defined by the BC map with a choice of plug-ins dealing with the failure of the BC between the two parties.

All the features either contain the BC feature or are part of it. The BC feature was the first one to be modeled in LOTOS. The specification of the other features was built on top of the BC model.

The Architecture

The architecture of the system, as presented in the requirements, is defined by the following entities:

- Device Element Block (DEB): it represents the physical end point of a call. It could be an actual telephone, a computer, or some other device. It is the only entity of the system that interacts with the user (or the customer).
- Communicating Entity Block (CEB): it represents a user's profile in the system. It has information about restrictions and privileges of the user.
- Logical Element Block (LEB): it represents the logical endpoint of a call in the switch. Typically, this is described as a role in an organization, e.g. the Director of sales.
- Call Object (CO): It is an intermediary component between LEBs that participate in a call. It is dynamically created when an originating LEB wants to communicate with a terminating LEB during a call connection request.

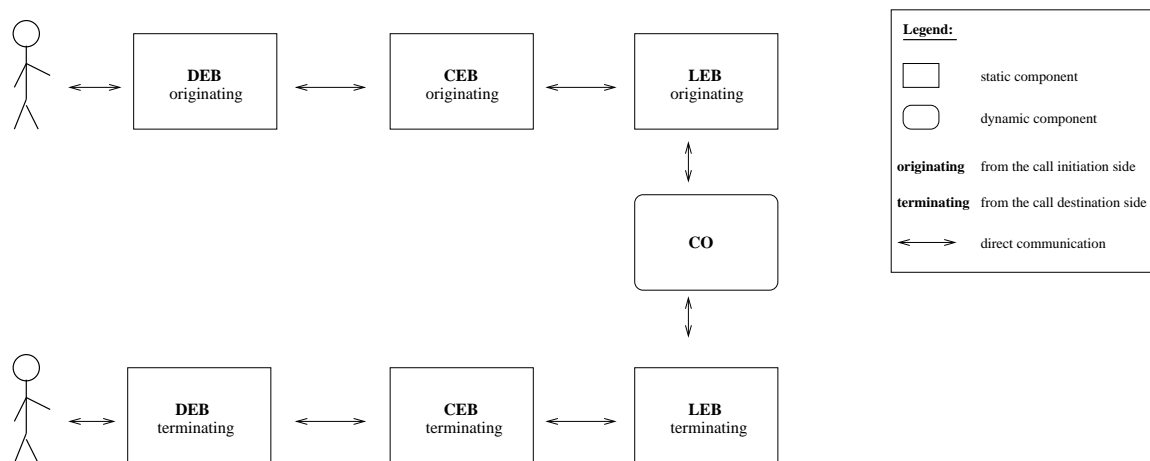


Figure 4.2: System architecture

As specified in the UCMs, a call between two ends involves seven components as shown in Figure 4.2. A call is initiated by an originating user through a DEB (this DEB will be the originating DEB for this call). The call connection request to a second party (or terminating party) is propagated to the originating LEB. A CO is dynamically created to connect the originating LEB to the terminating LEB. The call connection request is propagated to the terminating DEB. If the terminating end accepts the call, then the two parties establish a connection through the 7 entities involved in the call. In connections like the three-way calling, a third party is involved in the connection so another DEB, CEB and LEB are involved in the call.

Note that DEB, CEB and LEB are now called respectively DA (Device Agent), PA (Personal Agents) and FA (Functional Agents). However in this thesis we use the terminology that was prevalent at the time we did the bulk of our work.

4.6 UCM to LOTOS Transformation

The UCM to LOTOS transformation was performed following multiple steps:

1. Analysis of the UCMs by looking at the decomposition of the maps into stubs and at the components involved in each map.
2. Representation in a graph of the decomposition of the UCM maps and the stubs into stubs/plugin-ins.
3. Establishment of mapping rules between UCM and LOTOS elements.
4. Specification of a LOTOS model for a simplified behavior of the BC feature.
5. Specification of a LOTOS model for the 7 features described in section 4.5.

The following sections are going into more details about the LOTOS specification construction from the simplified BC specification to the specification of the 7 features.

4.6.1 Analysis of the Requirements

After analyzing the requirements, some syntax errors (i.e. unlabeled UCMs) and inconsistencies in the UCMs were found. The UCM designers had to fix them before the corresponding LOTOS specification could be started.

The UCMs were very complex, and deep in terms of sub-map encapsulation (up to 5 levels). Moreover, many stubs were repeatedly used in different maps. The graph representation presented in the next section allowed us to have a better idea of the UCMs composition into stubs.

4.6.2 Graph Representation of the Features

In order to have a picture of the hierarchy and the sequential order between the stubs of the UCMs, we built a graph as follows:

- In the graph, a feature and a static stub are represented by rectangles, a dynamic stub by a diamond and a plug-in by an ellipse.
- Features are represented on the top of the graph,
- In the UCM representation of a feature, the latter is composed by a sequence of stubs; this list is presented at the level below the one of the feature in the graph. The sequential order of the stubs is shown as a number on the label on each arrow linking the feature to each stub composing it.
- If a stub S is static, the level below it in the graph represents the sequence of stubs involved in S . The sequential order of the stubs is represented by the incremental labels i on the arrows linking S to each stub.

- If a stub **S** is dynamic, the level below it in the graph represents the alternative plug-ins for **S**. If the arrow leading to the stub **S** is **i**, then the arrows going from **S** to each alternative plug-in is **i.j**, where **j** is different for each plug-in. This way, from the representation of a stub **S** in a graph, we can know which stubs/plug-ins are composed in sequence and which are alternatives in **S**.

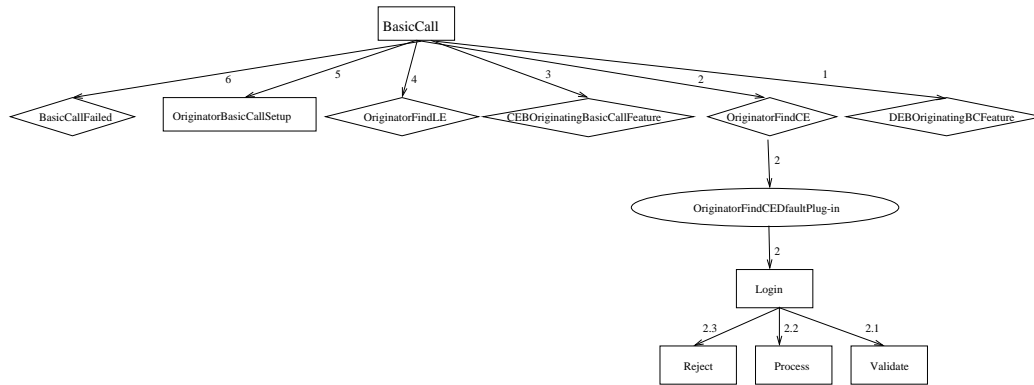


Figure 4.3: Hierarchical relationship between stubs of the BC map

Figure 4.3 represents a part of the graphical representation of the BC feature. We understand that the BC map contains 6 stubs. One of them is static: **OriginatorBasicCallSetup** (in fact, it is rectangular). The others are dynamic (they are diamonds). The dynamic stub **OriginatorFindCE** contains one plug-in **OriginatorFindCEDefaultPlugIn** (it is elliptic). The Figure shows that this plug-in has one static stub **Login** that has 3 static stubs: **Validate** followed by **Process** and then **Reject**. The sequential order of the stubs is represented by the arrow labels 2.1, 2.2, 2.3.

The graph of Figure 4.3 is automatically generated using the tool Graphviz [Att99]. This tool generates graphs from text files. A part of the Graphviz input file corresponding to the graph of Figure 4.3 is shown below:

```

digraph BasicCall {
name="Basic Call Feature Graph" node [style=empty];

BasicCall -> DEBOriginatingBCFeature
        -> DEBOriginatingBCFeatureDefaultPlugIn [style=bold color=red label=1];

BasicCall -> OriginatorFindCE
        -> OriginatorFindCEDefaultPlugIn
                -> Login [style=bold color=red label=2];
                Login
                        -> Validate [style=bold color=red label=2.1];
                Login
                        -> Process [style=bold color=red label=2.2];
                Login
                        -> Reject [style=bold color=red label=2.3];
BasicCall -> CEBOriginatingBCFeature
  
```

```

-> CEBOrganizingBCFeatureDefaultPlugIn [style=bold color=red label=3];

BasicCall -> ...
}

```

The graph representation of all the stubs/plugin composing the UCMs helped us to build a LOTOS specification by making sure that it followed the architectural behavior in the UCMs.

4.6.3 UCM to LOTOS Mapping

The transformation from the semi-formal notation UCMs to the formal language LOTOS is not straightforward. Some *mapping* rules that transform UCM into LOTOS elements were set up for this purpose. The mapping presented below is partially based on previous work by Amyot [Amy94].

- **start point** It is most generally a LOTOS action having the start point's label. It could also be a sequence of actions, a guarded behavior, or nothing if the start point's label is empty.
- **end point** It is most generally a LOTOS exit action carrying a value having the end point's label. It could also be an action or sequence of actions, or nothing if the end point's label is empty.
- **responsibility** A LOTOS action having the responsibility's label or sequence of actions.
- **or-fork** LOTOS choice operator $[]$ preceding each branch that represents a path on the right side of the or-fork.
- **and-fork** Parallel composition operator preceding each branch that represents a path on the right side of the and-fork.
- **or-join, and-join** Enable operator (\gg).
- **components C1, C2** Processes C1 and C2. If there is a direct UCM path from C1 to C2 in the map then processes C1 and C2 communicate through the gate C1_to_C2.
- **static stub, plug-in** Process having the stub's label. Responsibilities in the stub must be in the process gate list.
- **dynamic stub** Process having the stub's label and composed of processes representing each plug-in of this dynamic stub.
- **alternative between plug-ins** $[]$
- **timed waiting place** $[]$ between sequences of actions representing each alternative UCM path from the timed waiting place.

- **access to database** The database is represented by a LOTOS process. Each LOTOS action in this process corresponds to an action performed on the database of the UCM.

Figure 4.4 shows an example of a UCM and a derived LOTOS specification using the mapping presented above. The **Login** plug-in and the **Validate**, **Process** and **Reject** stubs are represented in LOTOS by processes of the same name. The causal relationship among the 3 stubs is represented by the enable operator (\gg). The or-fork is represented by the choice operator ($[]$). The query `updateDEdatabase` to the database `associatedCEdatabase` is represented by the action `Datachannel !updateDEDatabase` and the query `storeCE` by the action `Datachannel !storeCE`. The end points `registered` and `rejected` are respectively represented by the actions `exit(registered)` and `exit(rejected)`. We have not shown for simplicity the components `DEB` and `CEB`. These would have to be defined as processes, containing the actions above. In section 4.7.2 we will show additional details of this specification.

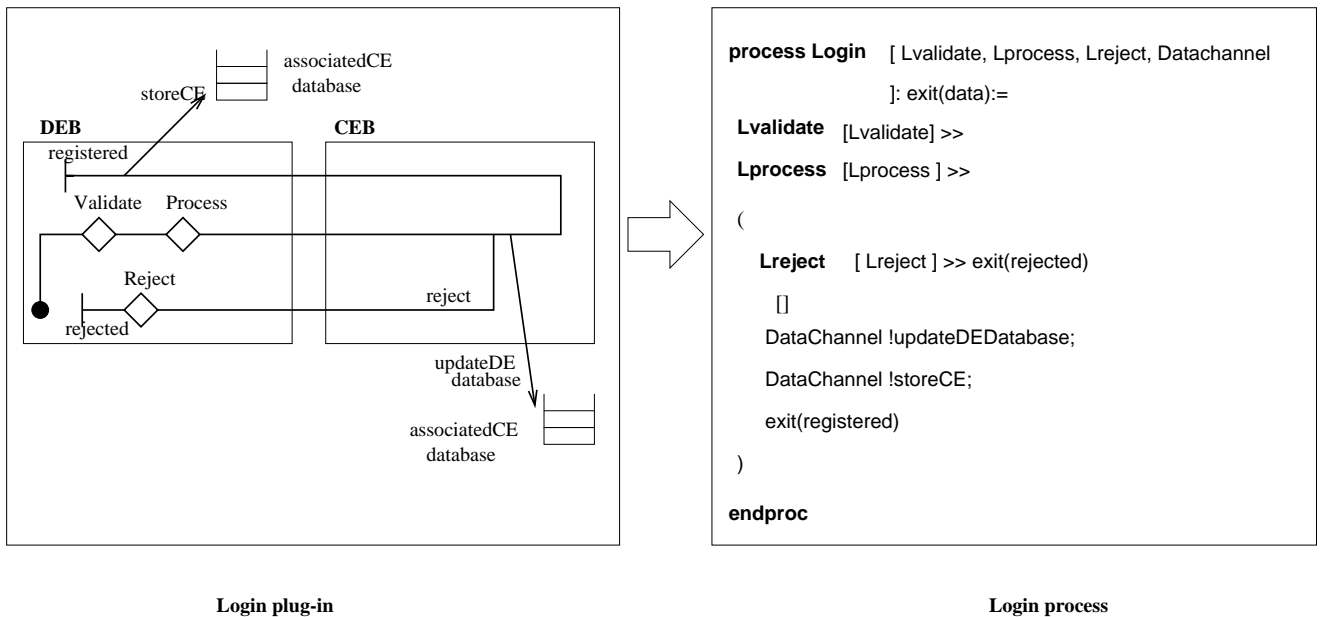


Figure 4.4: Example of UCM to LOTOS transformation using the mapping rules

Using the UCM to LOTOS mapping rules presented above, as Figure 4.4 shows, we were able to build (manually) the **Login** process corresponding to the UCMs of the Login plug-in.

It should be noted that recent work has provided an algorithm, which has been implemented, to translate UCMs into LOTOS skeletons. However, this did not exist at the time we did our research.

4.6.4 Specification of a Simplified Basic Call

We have modeled the simplified BC using four related UCMs:

- SimplifiedBasicCall: UCM representing the originating party that attempts to initiate a call (Figure 4.6(b)). In order to get this simple UCM, we *flattened* all the stubs of the BC map (Figure 4.6(a)); this way, we were able to represent in one map all the design details that are hidden by the stubs of the BC map.

The process of *flattening stubs* is the process of replacing the stubs of a map by their internal behavior. Figure 4.5 shows examples of how static and dynamic stubs are flattened, but it doesn't cover all the possible behaviors of stubs.

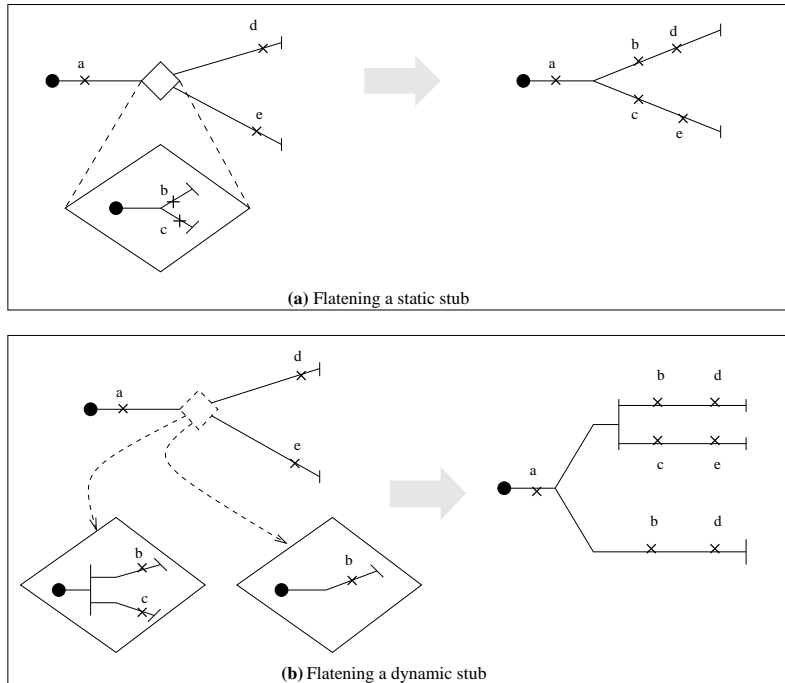


Figure 4.5: Process of flattening stubs

- Answer: This UCM is a post condition of the SimplifiedBasicCall UCM. It starts when a phone rings on the terminating side. Its scenario is as follows: If the terminating party answers, then the phone stops ringing, the connection timer is released, and the connection is established.
- HangUpOrig: It represents the termination of a call from the originating party.
- HangUpTerm: It represents the termination of a call from the terminating party.

After building the simplified BC UCM, the transformation into a LOTOS model was performed. We had to think about several points, especially how to represent the responsibilities, the timed waiting places, the entities DEB, CEB, LEB and CO and the communication between them in the LOTOS specifications.

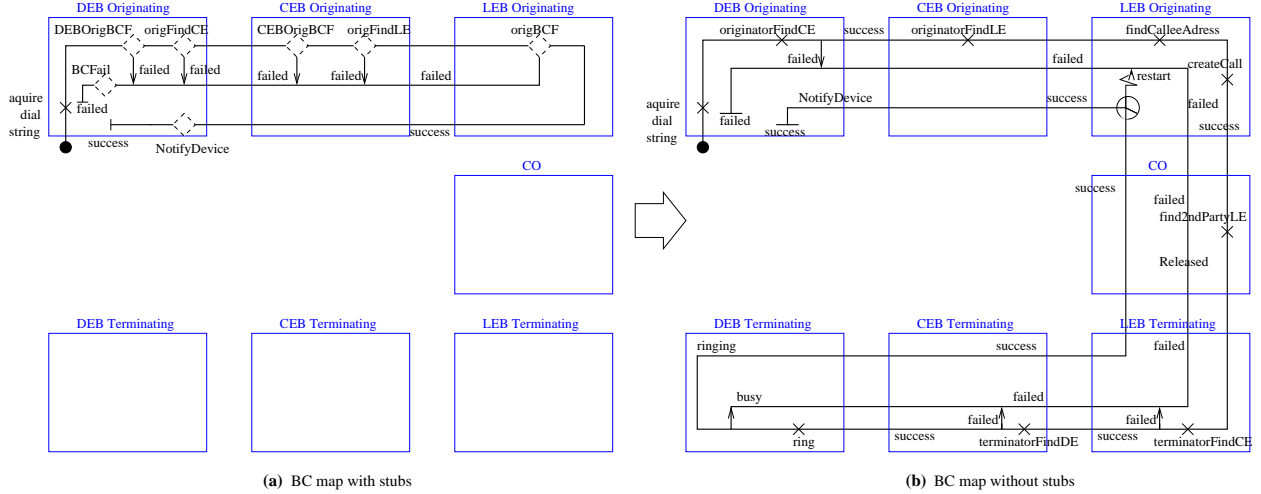


Figure 4.6: BC map flattened into a simplified BC map without stubs

Representation of the Entities and their Intercommunication

The DEBs, CEBs and LEBs are represented in the LOTOS model as processes. A process DEB is identified by its name (of sort `DebID`). The processes CEB and LEB are identified by both their names (of sort `CebID` and `LebID`) and their instances (of sort `Instance`). In fact, CEBs and LEBs could be involved in more than one call. In this case, different instances of the same process are created to manage each call.

Whenever a LEB originating wants to communicate with LEB terminating, a Call Object is created to be able to receive messages from the LEB origin and forward it to the LEB destination. This is due to the fact that a LEB does not know the location of the other LEBs. So the CO plays the interface between LEBs. The process CO is identified by an instance (of sort `Instance`).

In our LOTOS specification, two entities communicate through a LOTOS gate if there is a path in the UCMs that goes directly from one entity to another. From Figure 4.6, the DEB communicates only with the CEB, the CEB communicates with both the DEB and the LEB, the LEB with the CEB and the CO, and the CO with the LEBs. This communication between different components is represented in LOTOS by a synchronization through gates between processes DEB, CEB, LEB, and CO.

We decided to make these processes communicate through unidirectional gates as follows:

- a DEB communicates with CEBs through gates `DE_to_CE` and `CE_to_DE`,
- a CEB communicates with DEBs through `DE_to_CE` and `CE_to_DE` and with LEBs through `CE_to_LE` and `LE_to_CE`,
- a LEB communicates with CEBs through `LE_to_CE` and `CE_to_LE` and with a CO through `LE_to_CO` and `CO_to_LE`.

The choice of having two unidirectional gates between two processes instead of one bi-directional gate was made because of two reasons: first, in order for LOTOS traces to conform to Lotos2msc’s conventions; in fact, the tool needs to know, from the two entities involved in a message exchange, which is the sender and which is the receiver (more detail about Lotos2msc’s conventions are described in section 2.4).

Representation of the Timed Waiting Place in LOTOS

Since we cannot represent time in LOTOS, the part of the UCM map containing a timed waiting place is represented in LOTOS by a process containing a choice between two sequences of actions. One sequence of actions represents the system behavior when the timeout expires and the other represents the system behavior when the timeout does not expire.

The UCM map of Figure 4.6 contains a timed waiting place. The corresponding LOTOS process is as follows:

```

process WaitForAnswerFromCaller[ CO_to_LE, Timeout
                                ](leb:LebID, lebInst:Instance,
                                   ceb:CebID, cebInst:Instance,
                                   co:Instance): exit(Data):=
  CO_to_LE !co !leb !lebInst !answerInd;
  exit(success)
  []
  Timeout;
  exit(restart)
endproc

```

The sequence of actions `CO_to_LE !co !leb !lebInst !answerInd; exit(success)` corresponds to the UCM path describing the case where the timeout does not expire (success path in Figure 4.6). The sequence of actions `Timeout; exit(restart)` corresponds to the UCM path describing the case where the timeout expires (restart path in Figure 4.6).

Dealing with the Difference of Abstraction Level between UCMs and LOTOS

UCMs are abstract and are not meant to be very detailed. For this purpose, they are suitable for requirement specification. On the other hand, LOTOS specifications can be very detailed; they represent the interactions among system components in terms of message passing, rendezvous and synchronization. One of the reasons for this difference is that LOTOS specifications are executable, UCMs are not. When transforming UCMs into LOTOS, messages and other design details are added into the specification. This step was done by our team in full interaction with the UCMs designers in order to represent faithfully the system’s expected behavior.

One simple example of addition of new information in the LOTOS specification is the transformation of the responsibility `acquireDialString` in the BC map (see Figure 4.6) by a LOTOS compositions with the following visible actions from the environment:

```

USER_to_DE !userA !debA0 !offHook;
DE_to_USER !debA0 !userA !dialTone;
USER_to_DE !userA !debA0 !dial !2002;

```


4.6.5 Specification of the Other Features

The LOTOS model obtained from the simplified BC UCM (explained in section 4.6.4) was extended to the specification of the 6 other features (discussed in section 4.4). The LOTOS specification built for the Basic Call feature was enhanced by the addition of:

- New LOTOS processes, each one corresponds to a stub in the UCMs.
- New message types. In fact, new messages have to be added in order to express all message exchanges in the system with the 7 features.
- New users, involved in multi-user scenarios.

Some further details about the construction of the specification are presented in the next section.

4.7 LOTOS Specification of the Telecommunication System

In this section, we will describe the LOTOS model built from the UCMs of the 7 features. We will describe its ADT part, its top-level specification, the process `DEB` that describes the behavior of the entity `DEB` and the process `Database` that describes the actions performed to or from the database.

4.7.1 Abstract Data Types

12 types were created in the specification. The values defined in the data types are used as experiments in the LOTOS actions. In our LOTOS specification, these values are sent with the messages going through the system entities. They transport information about the sender and the receiver's identity, and about the user and the call's status.

Data Values of type `Data` express the status of the user or the call. For instance, the `busy` value of a message received from a user expresses a user in the `busy` state. The `connectReq` value of a message sent from a `CO` to a `LEB` expresses a connection call request.

Instance Values of type `Instance` express different instances of `CEBs`, `LEBs` and `COs`. A first instance of a process is used to manage the first call made in the system. Incrementally new instances of processes are created for each new call to manage.

UserID It was decided to have 4 users in the system which was found to be sufficient to represent all scenarios that interested us. We defined the `UserId` data type with four possible values: `userA`, `userB`, `userC` and `userD`.

DialString Each user has a phone number of type DialString. The four possible values are 2001, 2002, 2003 and 2004.

DebID is the DEB type. Each value of type DebID represents a user's access device. A value `nodeb` is added to represent a non-existent DEB for a user.

CebID is the CEB type. Each value of type CebID represents a user's CEB. A value `noceb` is added to represent an inexistent CEB for a user.

LebID is the LEB type. Each value of type LebID represents a user's LEB. A value `noleb` is added to represent an inexistent LEB for a user.

DataComm It represents the type of messages sent to and received from the database.

FeatureID is the feature type. The values of FeatureID are: CFO, CFA, OCS, CH, CT, CP.

FArg It represents a variable or a set of variables of type DialString. This data type is used to check what are the enabled features for a user.

FeatureSet It represents a couple of types (FeatureID, FArg) used to enable a feature for a user. A possible value is (CFA, (2001, 2002)). It means that the feature CFA is set up from the phone 2001 to the phone 2002.

Database This data type is used to express the access to the database.

4.7.2 Behavior of the Specification

The LOTOS specification is described by a top-level behavior followed by process definitions. The top-level behavior of the specification was built based on the system architecture in Figure 4.2.

Two more gates, `DE_to_CO` and `CO_to_DE`, were added to the specification, in fact, in some UCMs, a direct path links the CO to the DEB.

One process Database represents all the databases; one gate `DataChannel` is a shared gate between all the entities. Every access to the database is performed through this gate.

The corresponding LOTOS behavior is as follows:

```
behavior
  hide
  (* Agreed-upon interfaces *)
    CO_to_LE,          (* from Call Object to LEB *)
    LE_to_CO,          (* from LEB to Call Object *)
    LE_to_CE,          (* from LEB to CEB *)
    CE_to_LE,          (* from CEB to LEB *)
```

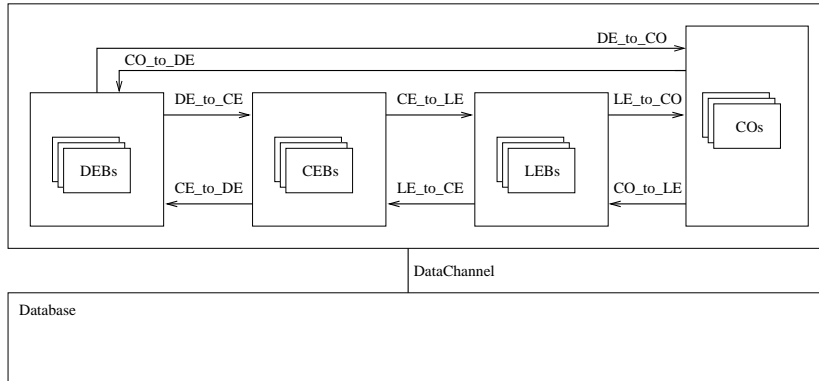


Figure 4.7: Graphical representation of the top-level specification

```

CE_to_DE,          (* from CEB to DEB *)
DE_to_CE,          (* from DEB to CEB *)
DE_to_CO,          (* from DEB to Call Object *)
CO_to_DE,          (* from Call Object to DEB *)
(* other interfaces, to access the database *)
DataChannel,      (* DEB, CEB, LEB components and Database *)
(* special interface, for dynamic creation of call objects *)
CreateCall        (* between LEB and Call Object *)
in
(
  (
    (
      (* Instantiates four concurrent DEB entities *)
      DEB[ DE_to_USER, USER_to_DE, DE_to_CE, CE_to_DE, DE_to_CO, CO_to_DE,
          HoldingTimeOut, DataChannel ](deba0, userA, 2001)
      |||
      DEB[ DE_to_USER, USER_to_DE, DE_to_CE, CE_to_DE, DE_to_CO, CO_to_DE,
          HoldingTimeOut, DataChannel ](debB0, userB, 2002)
      |||
      DEB[ DE_to_USER, USER_to_DE, DE_to_CE, CE_to_DE, DE_to_CO, CO_to_DE,
          HoldingTimeOut, DataChannel ](debC0, userC, 2003)
      |||
      DEB[ DE_to_USER, USER_to_DE, DE_to_CE, CE_to_DE, DE_to_CO, CO_to_DE,
          HoldingTimeOut, DataChannel ](debD0, userD, 2004)
    )
    |[ DE_to_CE, CE_to_DE ]|
  )
  (
    (* Instantiates four concurrent CEB entities *)
    CEB[ CE_to_DE, DE_to_CE, CE_to_LE, LE_to_CE, DataChannel ](cebA, 0 of Instance)
    |||
    CEB[ CE_to_DE, DE_to_CE, CE_to_LE, LE_to_CE, DataChannel ](cebB, 0 of Instance)
    |||
    CEB[ CE_to_DE, DE_to_CE, CE_to_LE, LE_to_CE, DataChannel ](cebC, 0 of Instance)
    |||
    CEB[ CE_to_DE, DE_to_CE, CE_to_LE, LE_to_CE, DataChannel ](cebD, 0 of Instance)
  )
)

```

```

    |[ LE_to_CE, CE_to_LE ]|
    (
        (* Instantiates two concurrent LEB entities *)
        LEB[ LE_to_CO, CO_to_LE, LE_to_CE, CE_to_LE, CreateCall,
            DataChannel, RingingTimeOut ](lebA, 0 of Instance)
        |||
        LEB[ LE_to_CO, CO_to_LE, LE_to_CE, CE_to_LE, CreateCall,
            DataChannel, RingingTimeOut ](lebB, 0 of Instance)
    )
)

|[ CreateCall, LE_to_CO, CO_to_LE, DE_to_CO, CO_to_DE ]|

(* Creates Call Objects dynamically *)
CallObjectCreator[ CreateCall, LE_to_CO, CO_to_LE, CO_to_DE, DE_to_CO, DataChannel
    ](0 of Instance)
)

|[ DataChannel ]|

(* Process which holds all databases. *)
DatabaseInit[ Init, DataChannel ]
)

```

4.7.3 Process DEB

The DEB, as mentioned before, represents the access device. It represents the interface between the telephony network and the user. All the user's actions (from the call origination to the call termination) are inputs of the DEB.

The DEB is either a DEB originating, if it receives a call request coming from a user, or a DEB terminating, if it receives a call request coming from a DEB.

```

process DEB[ DE_to_USER,      (* communication between the DEBs and the user *)
            USER_to_DE,
            DE_to_CE,        (* communication between the DEBs and the CEBs *)
            CE_to_DE,
            DE_to_CO,        (* communication between the DEBs and the COs *)
            CO_to_DE,
            HoldingTimeOut,
            DataChannel      (* communication between the DEBs and the database *)
    ] (deb:DebID, user:UserID, lds:DialString): exit:=

hide NewCEB, SetBusy, HandleOnHook, IncomingCall in (
    (* Process managing possible incoming calls if busy *)
    AlertDevice[ CO_to_DE, DE_to_CO, CE_to_DE, DE_to_CE, DE_to_USER, HandleOnHook,
                SetBusy, IncomingCall] (user, deb, 0 of Instance, success of Data)
    |[ HandleOnHook, SetBusy, IncomingCall ]|
    (
        (* Process containing DEB basics (simple calls management) *)
        BasicDEB [ DE_to_USER, USER_to_DE, DE_to_CE, CE_to_DE, DE_to_CO, CO_to_DE, NewCEB,
                  SetBusy, IncomingCall, HandleOnHook, HoldingTimeOut, DataChannel]
    )
)

```

```

                (deb, user, lds)
            |||
            DEBRedirectOccured[ NewCEB, CE_to_DE, DE_to_USER
                ] (user, deb, noceb of CebID, 0 of Instance)
        ) ) endproc

```

4.7.4 Process Database

The UCMs involve different databases. Our LOTOS specification represents the access to these databases as a single process with multiple synchronization points. The process Database is executed whenever an access to the database is made. What follows is a part of the process Database.

```

process Database[ Init, DataChannel ]( db:Database ):exit:=
    (* new Initialisation of the database while test running *)
    DatabaseInit[ Init, DataChannel ]
    [] (* getCE, gives the corresponding CEB for a DEB *)
    (* cebA is debA0's correspondant CEB *)
    DataChannel !getCE !debA0 !cebA; Database[ Init, DataChannel ]( db )
    []
    (* cebB is debB0's corresponding CEB *)
    DataChannel !getCE !debB0 !cebB; Database[ Init, DataChannel ]( db )
    []
    (* cebC is debC0's corresponding CEB *)
    DataChannel !getCE !debC0 !cebC; Database[ Init, DataChannel ]( db )
    []
    (* cebD is debD0's corresponding CEB *)
    DataChannel !getCE !debD0 !cebD; Database[ Init, DataChannel ]( db )
    []
    (* No corresponding CEB for any other DEB *)
    DataChannel !getCE ?deb:DebID !failure [ (deb <> debA0) and (deb <> debB0) and
        (deb <> debC0) and (deb <> debD0) ];
    Database[ Init, DataChannel ]( db )
    []
    ...
    []
    (* Database Query: returns "true" iff UserID is subscribed to FeatureID *)
    DataChannel ?u:UserID ?f:FeatureID !true [isFeatureSubscribed(u, f, db)];
    Database[ Init, DataChannel ]( db )
    []
    DataChannel ?u:UserID ?f:FeatureID !false [not(isFeatureSubscribed(u, f, db))];
    Database[ Init, DataChannel ]( db )
    []
    exit
endproc (* Database *)

```

4.8 Verification of the LOTOS Specification

The LOLA tool (described in section 3.4) was used for the verification and validation of the specification. First, we performed step-by-step execution of the specification in order to verify if the sequences of internal actions and message exchanges between the components correspond to the expected behavior of the system. Second, we performed validation against the user's requirements. The observable behavior of the LOTOS model is validated against the initial requirements. A set of test scenarios is generated for this purpose. In this section we develop the generation of different kinds of LOTOS scenarios to test the specification.

In this work, LOTOS test scenarios were manually generated from the requirements. They intended to cover all possible behaviors in the system. The scenario-based testing phase was performed using functional test scenarios (which describe only interactions between the user and the system). The scenarios were generated by sets, using the following decomposition:

- **Basic System Properties:** They focus on testing the basic properties that the system must fulfill.
- **Individual Features Properties:** They consist on verifying the expected behavior of the system when there is only one feature activated.
- **Feature Interaction Scenarios:** They focus on detecting possible incorrect behavior of the system when more than one feature is activated (a formal definition of feature interaction is presented in section 3.4).

4.8.1 Test Scenarios

Basic System Properties

We generated 9 scenarios that test the basic properties of the system. One of them is shown below. It is shown in LOTOS form below and in MSC form in Figure 4.8. It represents a user A trying to call a user B, then getting connected with him. The call ends with A hanging up first. The scenario ends with B getting the dial tone and hanging up.

```
process Scenario1[ USER_to_DE, DE_to_USER, Init, scenario1 ]: noexit:=
  (* initialisation of the database using default values *)
  Init !emptyDB;
  USER_to_DE !userA !debA0 !offHook;          (* A goes offHook *)
  DE_to_USER !debA0 !userA !dialTone;         (* A gets dialTone *)
  USER_to_DE !userA !debA0 !dial !2002;      (* A dials B *)
  DE_to_USER !debA0 !userA !callInProgress;   (* A gets callInProgress *)
  DE_to_USER !debB0 !userB !ringingOn;        (* B gets ringingOn *)
  DE_to_USER !debA0 !userA !ringBackTone;     (* A gets ringBackTone *)
  USER_to_DE !userB !debB0 !offHook;         (* B goes offHook *)
  DE_to_USER !debB0 !userB !ringingOff;       (* B gets ringingOff *)
  DE_to_USER !debA0 !userA !toneOff;         (* A gets toneOff *)
  DE_to_USER !debA0 !userA !voiceOn !2002;
```

```

DE_to_USER !debB0 !userB !voiceOn !2001; (* parties talk *)
USER_to_DE !userA !debA0 !onHook; (* A goes onHook *)
DE_to_USER !debA0 !userA !voiceOff !2002;
DE_to_USER !debB0 !userB !voiceOff !2001;
DE_to_USER !debB0 !userB !dialTone; (* B gets dialTone *)
USER_to_DE !userB !debB0 !onHook; (* B goes onHook *)
DE_to_USER !debB0 !userB !toneOff; (* B gets toneOff *)
scenario1; stop (* success *)
endproc

```

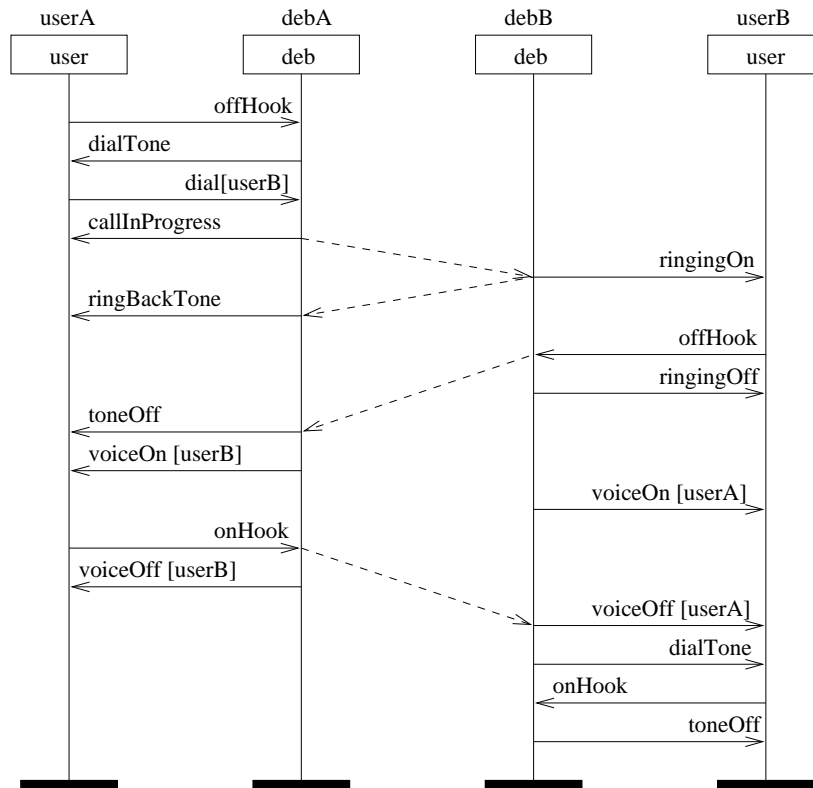


Figure 4.8: Message Sequence Chart of Scenario1

Individual Features Properties

We generated 14 scenarios to test the features individually: 2 for OCS, 3 for CFA, 1 for CFB, 4 for CH, 2 for CR, 2 for CT and 1 for CP.

Scenario2 of Figure 4.9 verifies the correctness of the CH feature. It describes a user A trying to call a user B, then A obtains communication with B. After that, A puts B on hold and calls a user C then talks to him. C hangs up. A gets then the dial tone, he switches afterwards back to B. After the talk AB hangs up, A gets the dial tone then hangs up.

Feature Interaction Scenarios

In our project, the feature interaction problem (defined in section 3.4.2) was solved at the requirements stage: priorities between features were established. When a user has 2 or more features activated, the scenario of the call will follow the behavior of the highest priority activated feature. LOTOS scenarios were built to verify that this mechanism works correctly.

It was decided that the OCS feature has a higher priority than the CFA feature. **Scenario3** of Figure 4.10 verifies that when OCS and CFA are involved, the system follows OCS behavior. This scenario involves 3 users: A, B and C. User A has OCS for C. User B has CFA to C. When A tries to call B, he gets forwarded to C due to CFA then gets the fast busy tone because he is not allowed to call C due to OCS to C.

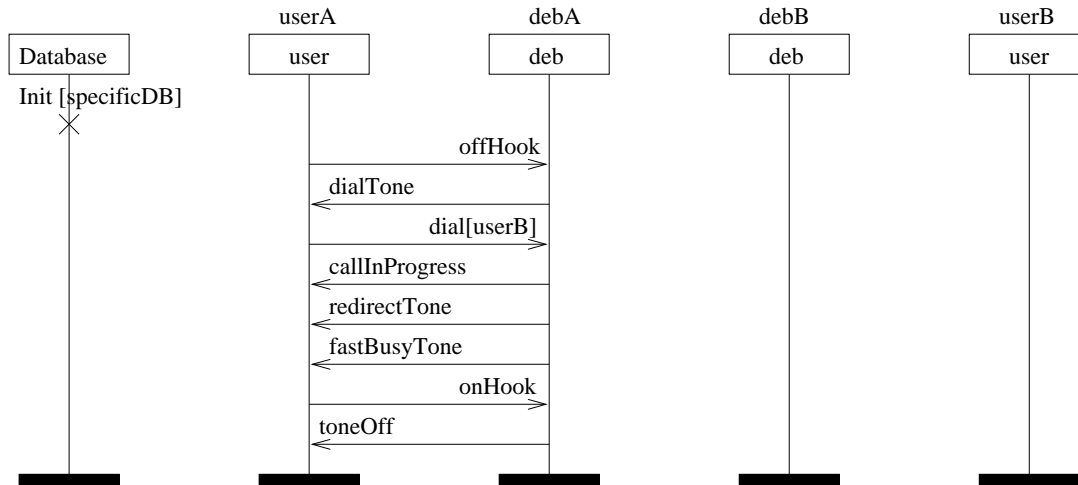


Figure 4.10: Message Sequence Chart of Scenario3

4.9 Analysis of the Results

4.9.1 Testing Results

A total of 33 scenarios were generated to test the specification. Some unexpected deadlocks and unwanted behaviors were corrected. Then all the tests were executed again on the specification. No other errors were found.

4.9.2 Cross-Validation

The cross-validation step (Figure 4.1) was performed after both the LOTOS team and the SDL team had built their specifications and had generated sets of scenarios that verified the specifications, with the intention of covering all possible behaviors of the specified system. It is intended to make sure that the SDL-based scenarios are complete and to add scenarios if this is not the case. After this step, a set of TTCN test cases was to be generated from

the complete SDL-based scenarios and the SDL specification to test the implementation. This step consists of comparing the LOTOS-based and the SDL-based scenarios. In order to be able to compare these scenarios, the two teams agreed to present them on the same format, the MSC format. The SDL-based MSCs were automatically generated from the SDL model using the Tau toolset. The LOTOS-based MSCs were automatically translated from LOTOS scenarios using the Lotos2msc converter. After comparing the two sets of scenarios, two major kinds of differences were discovered:

- Missing messages: MSCs for identical scenarios produced by the two teams were different. The SDL team represented the part of the scenario when two parties are talking by a `talk` state in the MSC. The LOTOS team represented it as a period starting with the reception by the users of messages `voiceOn` and ending with the reception of messages `voiceOff`. It was concluded that the scenarios are equivalent and that the difference of representation of the talk state does not make any scenario wrong. No changes were made to the specifications nor to the scenarios.
- Missing Scenarios: The set of LOTOS-based scenarios had 3 less scenarios than the set of SDL-based ones to test the CT feature, 1 more scenario to test the BC feature, and 1 more scenario to test the CH feature. These differences were due to some lack of clarity in the requirements. It was decided to add to the SDL specification the missing CH scenario.

4.10 Conclusion

The use of two formal methods (LOTOS and SDL) for the formal modeling stage was very productive. In fact, the cross-validation allowed to correct some design errors or misunderstandings of the requirements and to come up with an SDL model that, as far as we could tell, represents the expected system behavior. The automatic TTCN test suite generation from SDL allowed to generate rapidly a set of TTCN test suites that was used to validate the implementation by testing its conformance to the specification.

The use of only one formal method for the formal modeling stage can also be considered. In fact, chapter 6 proposes a development methodology based on UCMs at the requirements stage and LOTOS at formal modeling stage. Executable tests suites can be automatically generated using **Ucm2LotosTests**, a method developed in this thesis (it is explained in chapter 5), in combination with the TGV tool for the TTCN test suites generation. The exclusive use of SDL for the formal modelling stage is an other option that allows the rapid generation of TTCN tests [PUW00].

Chapter 5

Ucm2LotosTests: Automatic LOTOS Test Generation From Use Case Maps

*This chapter describes the design of **Ucm2LotosTests**, a new functionality added in *UCMNav* tool for the automatic LOTOS scenarios generation from UCMs.*

5.1 Motivation

UCMs are used to describe requirements and high-level designs with graphical scenarios. Such scenarios can serve as a basis for validation of systems against their requirements by ensuring that detailed designs and implementations conform to the original UCM description.

As seen in the development methodology presented in chapter 4, a LOTOS specification is validated by executing LOTOS scenarios. The set of LOTOS scenarios is manually generated from the UCMs. The same LOTOS scenarios can be used for conformance testing purposes at the implementation stage. Our goal is to automate the LOTOS scenarios generation from UCMs. Thus, the validation of the LOTOS specification can be performed faster, and the set of automatically generated scenarios is more complete than in a manual generation where the tester can miss some scenarios.

In our context, the traditional question “How much testing is enough?” will be answered by: “When all the UCM paths are covered!”. By *coverage* of the UCM paths we mean generation of tests that verify these paths.

Since UCMs are supported by the *UCMNav* tool (Section 2.2), the automatic test generation from UCMs can be carried out by using their internal representation in *UCMNav*.

Generally, a LOTOS specification built from UCMs is less abstract and represents a more realistic behavior of the system to be designed than the one in the UCMs. In fact, it is impossible to think of a fully automatic transformation from UCMs to LOTOS scenarios without the participation of the user to fill in the missing information in the UCMs. The automatic generation method that we implemented allows the interaction with the user during this process.

5.2 Desired Behavior of the Ucm2LotosTests Functionality

UCMNav is the tool that supports the UCM notation. It creates, edits and modifies UCMs. Our goal was to add a new feature to it called **Ucm2LotosTests** that generates a set of LOTOS scenarios from a UCM map. The Ucm2LotosTests functionality appears in the option list of a start point in a UCM map. The execution of this functionality generates text files; each file corresponds to a LOTOS scenario generated from the UCM triggered by the selected start point.

Ucm2LotosTests generates also scenarios corresponding to two or more start points. In this case, it generates LOTOS scenarios corresponding to the UCM paths triggered by all the selected start points.

5.3 Internal Representation of the UCMs in UCMNav

In UCMNav, the data structures used to store the UCM map are based on a variation of a standard graph grammar called a hypergraph. Basically, a hypergraph is a graph described by a collection of nodes connected together by hyperedges.

The internal representation of a UCM map is a list of connected hyperedges. An example of UCM is shown in Figure 5.1. Its corresponding internal representation is shown in Figure 5.2.

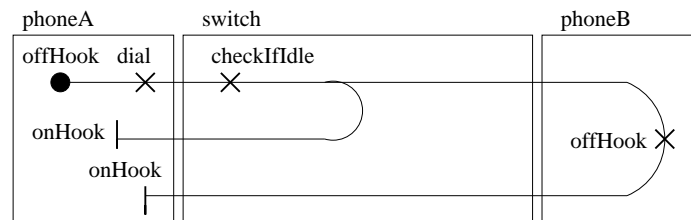


Figure 5.1: Example of UCM

The implementation of UCMNav was performed using Object-Oriented design. Each UCM element (definition in section 2.2) is implemented as a class with methods that correspond to possible actions on the element. These classes are sub-classes of the class **Hyperedge**. The classes being used to implement Ucm2LotosTests are: **Start** (for the

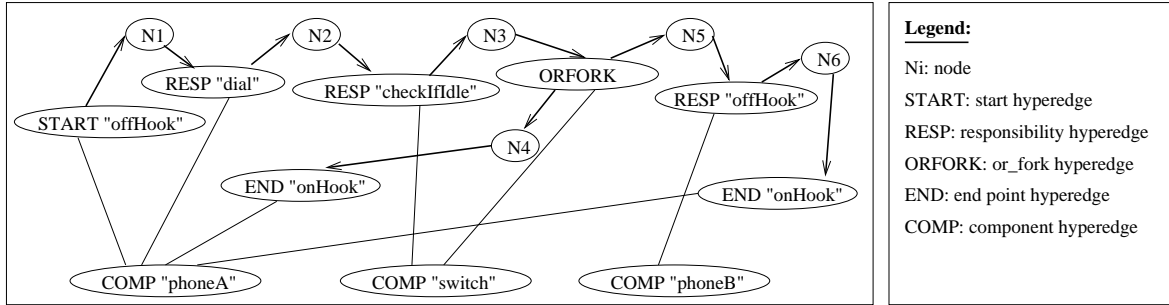


Figure 5.2: UCMNav internal representation of the UCM of Figure 5.1

implementation of start point), **Result** (for end point), **Synchronization** (for and_fork and and_join), **Stub** (for static and dynamic stub), **Responsibility**, **OrFork**, **OrJoin** and **Loop**.

5.4 Design of the Ucm2LotosTests Functionality

The design of Ucm2LotosTests is composed of two steps:

- *UCM Route Generation*: This step deals with the generation of a set of scenarios from a UCM. The generated scenarios are called *UCM routes*. Each of them is a sequence of UCM elements that exist in the UCM. Routes are automatically generated depending on the visited UCM elements (details in section 5.4.1). The set of generated routes should cover all the paths of the UCM.
- *UCM Routes to LOTOS Scenarios generation*: This step is performed after the route generation is carried out. It transforms each generated route into a LOTOS scenario. It is done in interaction with the user: For each UCM element in the generated route, the UCM to LOTOS mapping is requested (details in section 5.4.2).

5.4.1 UCM Route Generation

UCM Route, UCM Segment

In this section we introduce the notion of *UCM route* (or *route*) and *UCM segment* (or *segment*). A route is a UCM that is limited to start points, end points, conditions in or_forks, responsibilities, and_forks, and and_joins. A route does not contain or_forks, or_joins, or stubs. Our algorithm generates sets of routes from UCMs, by eliminating or_forks, or_joins and stubs. This is defined in detail in the following algorithm.

A segment in a UCM is any part of a UCM route. It is generally composed of sequences of UCM elements that exist in the UCM route. Contrary to routes, segments are not obliged to have start and end points.

The UCM of Figure 5.3 (a) has two UCM routes (Figure 5.3 (b)). Each of them is composed of many possible UCM segments. One possible segment is the responsibility a followed by the responsibility b.

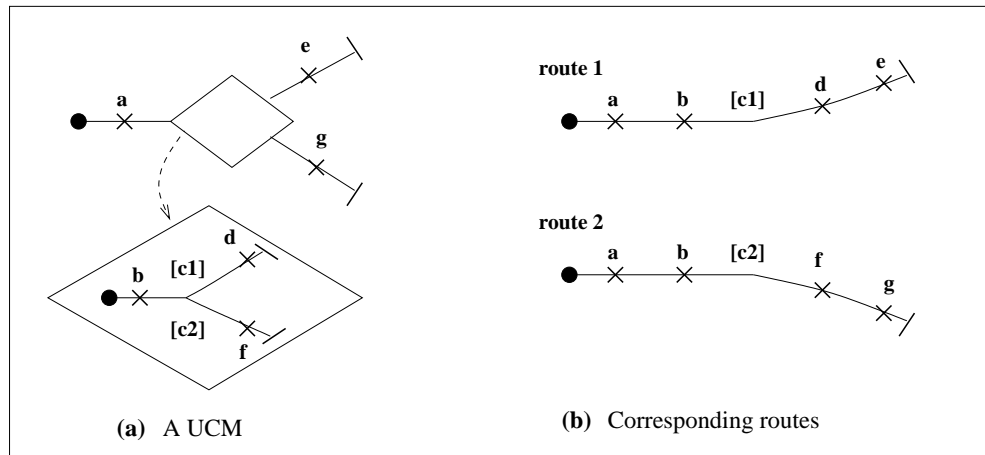


Figure 5.3: Routes of a UCM

UCM Route Generation

Given a UCM, the route generator visits every UCM element and generates its corresponding UCM segment (following the algorithm detailed below). After visiting all the elements, a set of UCM routes is obtained from the composition of the generated UCM segments.

During the process, the route generator can have multiple paths to visit individually. In this case, a stack is used to store the entry of each path that the route generator will visit after visiting the current one.

The route generator proceeds as follows when visiting the UCM elements. Figures 5.4 to 5.14 present the UCMs as compositions of UCM segments (labeled a to f) without specifying the composition of these segments in order to treat general UCM behaviors:

- **Start point** The start point of a UCM triggers the route generator. This latter includes the start point into the route being generated, and then goes to the next UCM element in the current path.

If two or more start points trigger the route generator, this latter processes the route generation for one of the start points and puts the others in a stack. After finishing the route generation of the path following the current start point, it does the same processing for the remaining start points in the stack.

- **Responsibility** When a responsibility is visited, the route generator adds it to the route that is being generated, then goes to the next UCM element in the current path.
- **End point** When the route generator reaches an end point, it checks if this end point is included in a stub. If it is the case, it follows what comes after the stub. If not, it checks if the current path is part of a parallel branch (started by an and_fork). In this case, it goes back to the beginning of the parallel branch, to follow the remaining parallel paths. If neither of the two cases is true, then the end of the UCM is reached. In this case, the UCM route being generated is now complete.

- **Or_fork** (Figure 5.4) When an `or_fork` is visited, the route generator follows each path on the right side of the `or_fork` after adding its corresponding `or_fork` condition to the route being generated. Then, it generates different routes for each followed path. Each route is composed of the path on the left side of the `or_fork` followed by one of the paths on the right side of this `or_fork`. Thus, the number of generated routes corresponds to the number of paths on the right side of the `or_fork`.

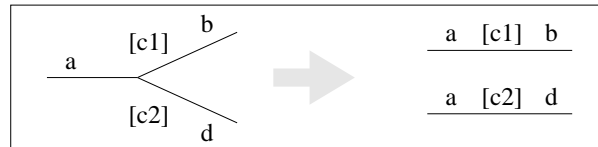


Figure 5.4: Route generation with `or_fork`

- **Or_join** (Figure 5.5) When an `or_join` is visited, the route generator follows the path on the right side of the `or_join`. When it reaches the end of it, it generates a route corresponding to the path visited on the left side followed by the one on the right side of the `or_join`.

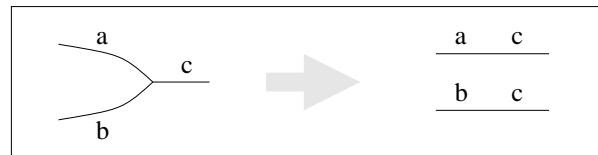


Figure 5.5: Route generation with `or_join`

The route generator as defined for the `or_fork` and `or_join` will generate routes as shown in Figure 5.6 when a UCM contains an `or_join` after an `or_fork`.

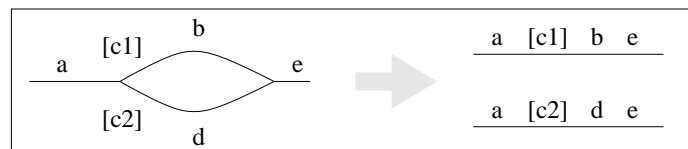


Figure 5.6: Route generation with `or_fork` followed by `or_join`

- **And_fork** (Figure 5.7) When an `and_fork` is visited, the route generator follows all the paths on the right side of it, one by one, then integrates all of them in the route to generate. This means that the generated route has the same behavior as the one of the UCM.

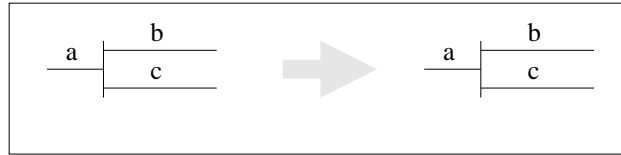


Figure 5.7: Route generation with `and_fork`

- **And_join** (Figure 5.8) When a `and_join` is visited, the route generator checks if all the paths on the left side of the `and_join` were already visited. If it is not the case, then it visits the remaining paths, else it groups all the generated routes into one and adds the right side of the `and_join` to it. The obtained route is identical to the UCM.

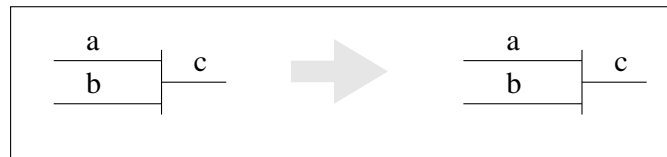


Figure 5.8: Route generation with `and_join`

The route generator as defined for the `and_fork` and `and_join` will generate routes as shown in Figure 5.9 when a UCM contains an `and_join` after an `and_fork`.

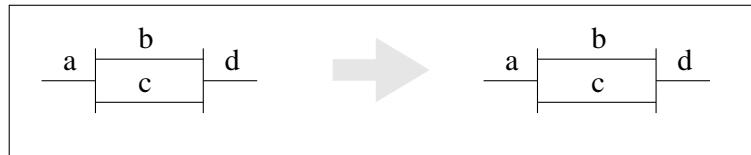


Figure 5.9: Route generation with `and_fork` followed by `and_join`

- **Static stub** (Figure 5.10) The route generator visits the start point of the stub and then proceeds to add the subsequent elements it visits to the route it is generating. When an end point of the stub is reached, the route generator accesses the path continuation in the upper level and continues ahead with the route generation.
- **Dynamic stub** (Figure 5.11) A different route is generated for each plug-in of the dynamic stub. The route generation of each plug-in processes identically as the one for a static stub. There will be as many sets of routes as there are plug-ins in this dynamic stub.
- **Loop** (Figure 5.12) A convention for the coverage of the loop by tests was adopted. We chose to cover the loop at most once. Therefore, two possible routes can be generated: one route that does not go through the loop and one route that goes through it once.

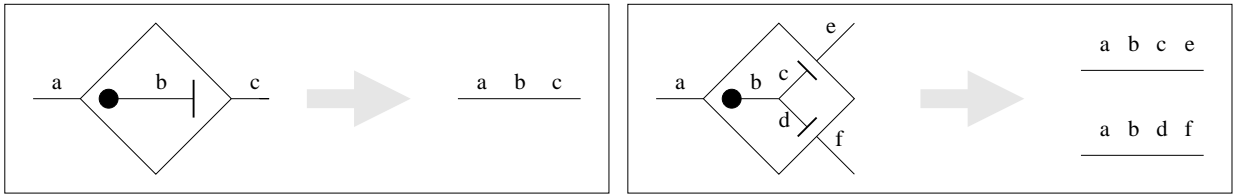


Figure 5.10: Route generation for the static stub

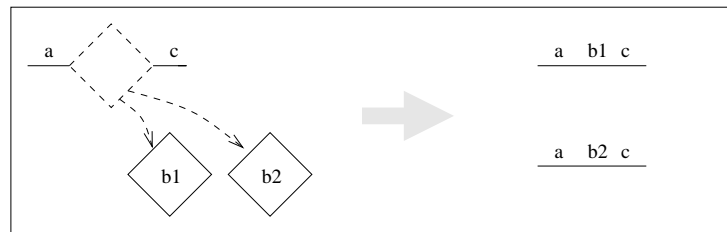


Figure 5.11: Route generation for the dynamic stub



Figure 5.12: Route generation for the loop

- **Waiting place** (Figure 5.13) When a waiting place is visited, the route generator goes to the next UCM element in the current path.



Figure 5.13: Route generation for the waiting place

- **Timed Waiting Place** (Figure 5.14) When a timer is visited, two routes are generated. One of them contains the continuation path and the other contains the timeout path.

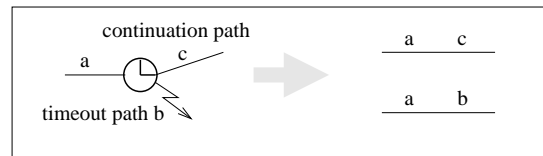


Figure 5.14: Route generation for the timed waiting place

- When a visited UCM element is not one of the UCM elements mentioned above, the route generator goes directly to the next UCM element in the current path.

Summary of the Route Generator

The following pseudocode provides a summary of the major steps involved in the UCM route generation. Each class corresponding to a UCM elements mentioned above has the method `GenerateLOTOS(hyperedge)` that performs its route generation. When a visited UCM element is any of those mentioned above, the method `GenerateLOTOS(hyperedge)` of the class `Hyperedge` is executed.

```

Start::GenerateLOTOS(hyperedge)
// implements the route generation for a start point
  addInRoute(hyperedge)      // start point is added to the route being generated
  next(hyperedge)           // next UCM element is visited
endMethod

Result::GenerateLOTOS(hyperedge)
// implements the route generation for the end point
  addInRoute(hyperedge)      // end point is added to the route being generated
  if (existsNext(hyperedge)) // end point is not the last element in a path
    next(hyperedge)         // access to the UCM element following it
  elseif inStub(hyperedge)  // end point is in a sub-map
    nextInUpperLevel(hyperedge) // access to the path linked to the end point

```

```

    else return(route) // end point is the last element in a path, a route is generated
  endIf
endMethod

Responsibility::GenerateLOTOS(hyperedge)
// implements the route generation for the responsibility
  addInRoute(hyperedge) // responsibility is added to the route
  next(hyperedge) // next UCM element is visited
endMethod

OrJoin::GenerateLOTOS(hyperedge)
// implements the route generation for the or_join
  next(hyperedge) // next UCM element is visited
endMethod

OrFork::GenerateLOTOS(hyperedge)
// implements the route generation for the or_fork
  if (nonVisited(hyperedge)) // first pass on the or_fork
    addInRoute(hyperedge->condition) // one condition of the or_fork is added to the route
    pushInStack(hyperedge, conditionList) // the other conditions are stored
    next(hyperedge->condition) // next UCM element of the current or_fork branch is visited
  else // or_fork is already visited
    element <- popConditionFromStack() // one condition is retrieved from stack
    addInRoute(element)
    next(element)
  endIf
endMethod

Synchronization::GenerateLOTOS(hyperedge)
// implements the route generation for the and_fork, and_join
  if (nonVisited(hyperedge)) // current or_join first time visited
    tempList <- addOrJoin() // add or_join in a temporary list tempList
    pushInStack(hyperedge, conditionList) // all possible paths from or_join stored
    next(hyperedge->condition) // first possible path from or_join is visited
  elseif (nonEmptyStack())
    element <- popConditionFromStack() // access to a remaining non visited path from or_join
    tempList <- addInRoute(element) // add path to tempList
    next(element)
  else // all paths from or_join are visited
    tempList <- addAndJoin() // add and_join to tempList
    addInRoute(tempList) // add tempList to route
  endIf
endMethod

Stub::GenerateLOTOS(hyperedge)
// implements the route generation for the static and dynamic stub
  if (nonVisited(hyperedge))
    addInRoute(hyperedge->plugIn->entry) // each start point in current plug-in is added in route
    pushInStack(hyperedge, plugInList) // store all plug-ins in stack
    next(hyperedge->plugIn->entry) // access to next start point in plug-in
  else
    currentPlugIn <- popPlugInFromStack() // retrieve next plug-in
    addInRoute(currentPlugIn->entry) // access to each start point of plug-in
  endIf
endMethod

```

```

        next(element)                // next UCM element is visited
    endIf
endMethod

Hyperedge::GenerateLOTOS(hyperedge)
// implements the route generation for the hyperedge
    next(hyperedge)    // next UCM element is visited
endMethod

```

Coverage of UCMs by the Generated Routes

The way the UCM route generator is designed, all possible paths and elements of a UCM are visited. Thus, the obtained UCM routes cover all the paths of the UCM.

5.4.2 UCM Routes to LOTOS Scenarios Transformation

After the UCM route generation is carried out, a LOTOS scenario is generated for each UCM route. We define two ways to generate LOTOS scenarios:

- Default LOTOS scenarios generation: The LOTOS scenario generation for the UCM routes is performed without user intervention. This way is not intended to be used frequently. The generated LOTOS scenarios give an idea of the routes that cover a UCM, but they do not constitute acceptable traces of LOTOS specifications since LOTOS design details are not present in the scenarios.
- LOTOS scenarios generation with user intervention: During the generation process, inputs from the user are requested. This way is the one to be chosen by the LOTOS specification designers to get LOTOS scenarios that constitute acceptable traces of LOTOS specifications.

Default LOTOS Scenarios Generation

A UCM route is a composition of UCM elements, and a LOTOS scenario is a composition of LOTOS elements. In order to generate the correct LOTOS behavior from a route, we set up some mapping rules to transform, in a deterministic way, each UCM route into a LOTOS scenario that expresses the same behavior of the system. These mapping rules define the corresponding LOTOS element to each UCM element. A UCM route is only composed by start points, end points, conditions in or_forks, responsibilities, and_forks and and_joins. In this case, by defining the appropriate LOTOS mapping of each of these UCM elements, we are able to generate a LOTOS scenario that corresponds to the entire UCM route.

It was decided that the mapping rules are as follows: A UCM element that is a start point, an end point, a conditions or a responsibility is mapped into a LOTOS action having its label. The other possible UCM elements in a route are mapped into LOTOS actions and operators as shown in Figure 5.15.

Figure 5.16 shows an example of automatic transformation of a UCM route into a LOTOS scenario.





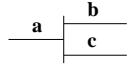

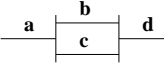
UCM route composition	Corresponding LOTOS statement
responsibility 	LOTOS action r1
start point 	LOTOS action s1
end point 	LOTOS action e1
condition 	LOTOS action c1
and_fork 	LOTOS behavior a; (b; exit c; exit)
and_join 	LOTOS behavior (b; exit c; exit) >> d; exit
or_join followed by and_join 	LOTOS behavior a; (b; exit c; exit) >> d; exit

Figure 5.15: Mapping UCM to LOTOS for the default LOTOS scenarios generation

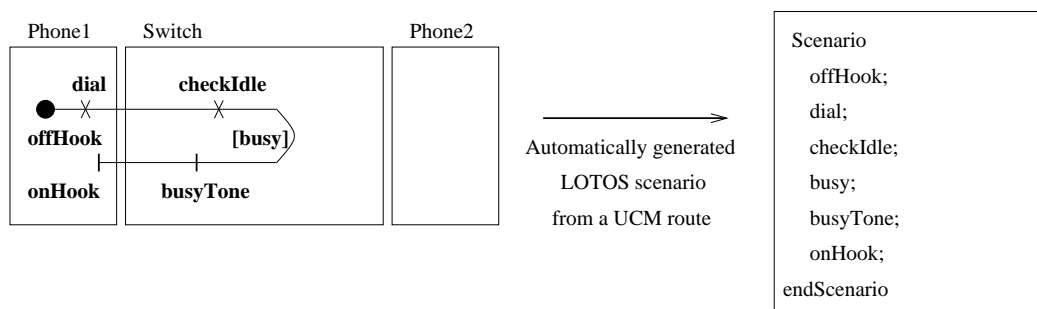


Figure 5.16: Example of UCM route to LOTOS scenario generation using the default LOTOS scenarios generation

LOTOS Scenarios Generation with User's Intervention

UCMs are an abstract view of the system while the LOTOS specification is closer to the real system behavior in terms of actions performed and message exchanges between components. For these reasons, in order to be able to generate, from UCMs, LOTOS scenarios that have the details of the LOTOS specification, the user's participation is requested to fill in the lack of information between the UCMs and the already built LOTOS specification.

The user who participates in the test generation is generally the designer of the LOTOS specification. The same mapping used to build the LOTOS specification from the UCMs is used here to generate LOTOS scenarios from the same UCMs.

It is requested of the user to provide mapping for responsibilities, start points, end points and conditions of the UCMs. The other possible UCM elements in a route (and_fork and and_join) are automatically mapped into LOTOS elements using the mapping rules presented in Figure 5.15. We obtain the mapping rules of Figure 5.17.

The mapping provided by the user is a LOTOS behavior. This LOTOS behavior is either an action, or a sequence of actions, or a choice or parallel composition between actions or sequences of actions. It is not composed of processes. It is up to the users to define the corresponding mappings depending on the way they designed the LOTOS specifications from the given UCMs.





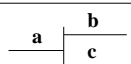
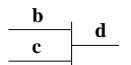
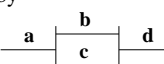
UCM route composition	Corresponding LOTOS behavior
responsibility 	$M(r1)$
start point 	$M(s1)$
end point 	$M(e1)$
condition 	$M(c1)$
and_fork 	$M(a); (M(b); \text{exit} \parallel M(c); \text{exit})$
and_join 	$(M(b); \text{exit} \parallel M(c); \text{exit}) \gg M(d); \text{exit}$
or_join followed by and_join 	$M(a); (M(b); \text{exit} \parallel M(c); \text{exit}) \gg M(d); \text{exit}$

Figure 5.17: UCM to LOTOS elements transformation using a mapping M entered by the user

Figure 5.19 shows an example of automatic transformation of a UCM route into a LOTOS scenario using the mapping M presented in Figure 5.18.

If the user wants to generate scenarios representing only the external behavior of the system, no mapping is provided to each UCM element representing an internal action in the

system. For instance, the `checkIdle` responsibility and the `[busy]` conditions of Figure 5.19 are not translated into LOTOS elements (see Figure 5.18) since they are internal actions to the `Switch` component.

Name	Element	Component	UCM to LOTOS mapping
<code>offHook</code>	start point	Phone1	<code>user_to_phone !user1 !phone1 !offHook;</code> <code>phone_to_user !phone1 !user1 !dialTone;</code>
<code>dial</code>	resp	Phone1	<code>user_to_phone !user1 !phone1 !dial !user2;</code>
<code>checkIdle</code>	resp	Switch	
<code>[busy]</code>	condition	Switch	
<code>busyTone</code>	end point	Switch	<code>phone_to_user !phone1 !user1 !busyTone;</code>
<code>onHook</code>	end point	Phone1	<code>user_to_phone !user1 !phone1 !onHook;</code>

Figure 5.18: Mapping M entered by the user and used for the generation of the LOTOS scenario of Figure 5.19

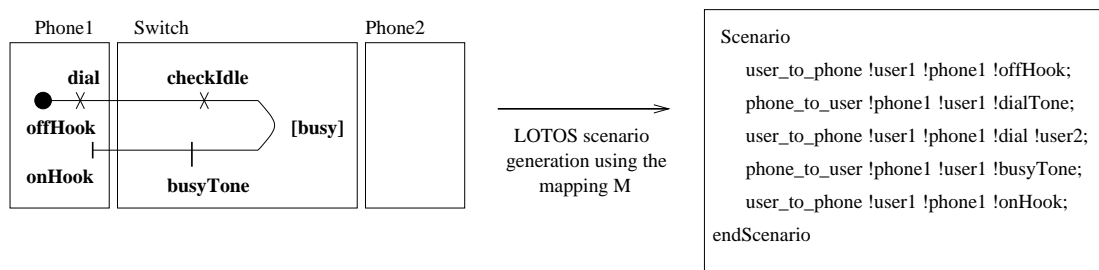


Figure 5.19: Example of UCM route to LOTOS scenario generation using the mapping M of Figure 5.18

5.5 Example of Test Generation with Ucm2LotosTests

The `Ucm2LotosTests` execution on the UCM map of Figure 2.1 generated 3 possible scenarios (see Figure 5.21): `Scenario1` represents the case where the called party is busy, `Scenario2` represents the case where the caller hangs up first after talking to the called party and `Scenario3` represents the case where the called party hangs up first. The corresponding LOTOS scenarios were generated using the mapping of Figure 5.20.

5.6 Usefulness of Ucm2LotosTests

The generated LOTOS scenarios with `Ucm2LotosTests` are useful for validation purposes, because of the fact that they are automatically derived from the UCMs, so they describe expected behaviors of the system. They can be considered as acceptance tests (definition

Name	Element	Component	UCM to LOTOS mapping
offHook	start point	Phone1	user_to_phone !user1 !phone1 !offHook; phone_to_user !phone1 !user1 !dialTone;
dial	resp	Phone1	user_to_phone !user1 !phone1 !dial !user2;
checkIdle	resp	Switch	
[busy]	condition	Switch	
busyTone	end point	Switch	phone_to_user !phone1 !user1 !busyTone;
onHook	end point	Phone1	user_to_phone !user1 !phone1 !onHook;
[idle]	condition	Switch	phone_to_user !phone2 !user2 !ring; phone_to_user !phone1 !user1 !ringBack;
offHook	start point	Phone2	user_to_phone !user2 !phone2 !offHook;
onHook	start point	Phone1	user_to_phone !user1 !phone1 !onHook; phone_to_user !phone2 !user2 !discTone;
onHook	end point	Phone2	user_to_phone !user2 !phone2 !onHook;
onHook	start point	Phone2	user_to_phone !user2 !phone2 !onHook; phone_to_user !phone1 !user1 !discTone;
onHook	end point	Phone1	user_to_phone !user1 !phone1 !onHook;

Figure 5.20: Possible mapping of the UCM elements of Figure 2.1

in 3.4.2) of the system. In this case, when they are executed on a LOTOS specification that is built from the same UCMs, if they do not pass, then it can be concluded that the specification does not behave as expected and has to be corrected.

These scenarios can be used for several purposes. First, they can be used in order to test whether the specification exhibits the behaviors that are specified in the UCM requirements. Second, they can be used for white-box, grey-box or black-box testing of implementations. Scenarios like the ones in Figure 5.21 are used for black-box testing since they include only message exchanges between the environment and the system. In fact, all the LOTOS actions of the obtained scenarios are messages sent from a user to a phone (using the `user_to_phone` gate) or from a phone to a user (using the `phone_to_user` gate).

If we want to generate scenarios for white-box testing, we have to provide the internal actions of the system in the mapping. For instance, if we want to perform white-box testing on the system described in Figure 5.21, we can provide the mapping of Figure 5.22. The LOTOS actions in bold are the ones that represent the internal actions, such as the message exchanges between the switch and the phones (represented by the LOTOS actions `phone_to_switch` and `switch_to_phone`). However, white-box and grey-box testing are not discussed further in this thesis.

5.7 Automatic Test Generation Issues

The scenarios generated with our method are supposed to represent the expected behavior of the system. Therefore, when they are composed in parallel with the LOTOS specification

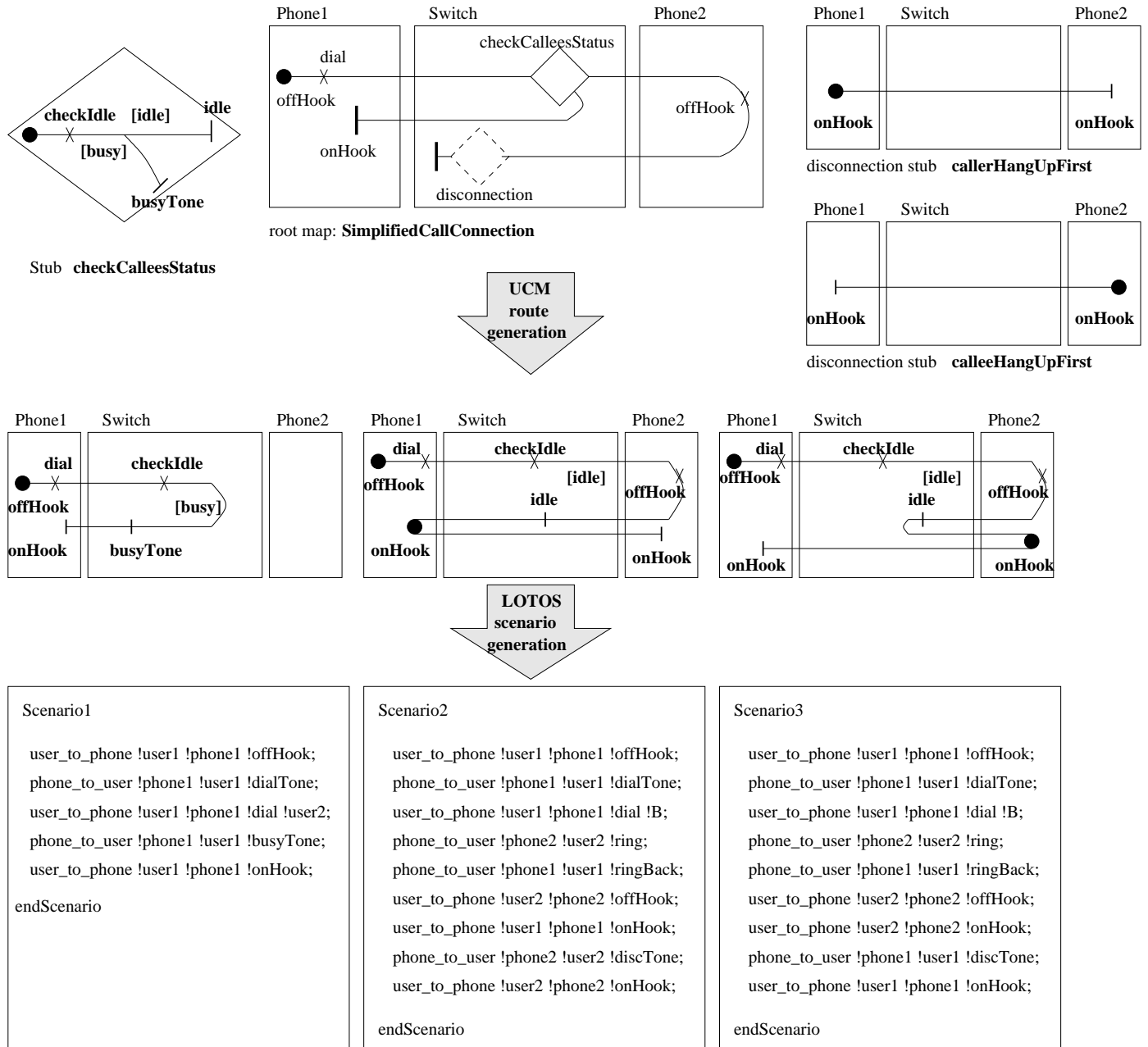


Figure 5.21: Automatic generation of 3 possible scenarios in LOTOS from a UCM

Name	Element	Component	UCM to LOTOS mapping
offHook	start point	Phone1	user_to_phone !user1 !phone1 !offHook; phone_to_user !phone1 !user1 !dialTone;
dial	resp	Phone1	user_to_phone !user1 !phone1 !dial !user2;
checkIdle	resp	Switch	switch_to_phone !phone2 !checkIdle;
[busy]	condition	Switch	phone_to_switch !phone2 !busy;
busyTone	end point	Switch	switch_to_phone !phone1 !busy; phone_to_user !phone1 !user1 !busyTone;
onHook	end point	Phone1	user_to_phone !user1 !phone1 !onHook;
[idle]	condition	Switch	switch_to_phone !phone2 !incomingCall; phone_to_user !phone2 !user2 !ring; phone_to_user !phone1 !user1 !ringBack;
offHook	start point	Phone2	user_to_phone !user2 !phone2 !offHook;
onHook	start point	Phone1	user_to_phone !user1 !phone1 !onHook; phone_to_switch !phone1 !disc; switch_to_phone !phone2 !disc; phone_to_user !phone2 !user2 !discTone;
onHook	end point	Phone2	user_to_phone !user2 !phone2 !onHook;
onHook	start point	Phone2	user_to_phone !user2 !phone2 !onHook; phone_to_switch !phone2 !disc; switch_to_phone !phone1 !disc; phone_to_user !phone1 !user1 !discTone;
onHook	end point	Phone1	user_to_phone !user1 !phone1 !onHook;

Figure 5.22: Possible mapping of the UCM elements of Figure 5.21 for white-box testing

during validation, if there are deadlocks, then the specification is checked for unwanted behaviors but not the scenarios.

In some cases, `Ucm2LotosTests` generates scenarios that do not express a valid behavior of the system, and therefore are rejected by the specification (definition in 3.4.2).

Such cases may occur when the UCMs behavior depends on data values. Since the UCMs do not include a data model, behaviors depending on data values may not be properly generated in the test generation process. The example in figure 5.23 presents the case where four scenarios are automatically generated with `Ucm2LotosTests` from a UCM, but only two of them represent the expected behavior of the system. The other two are rejected by the specification.

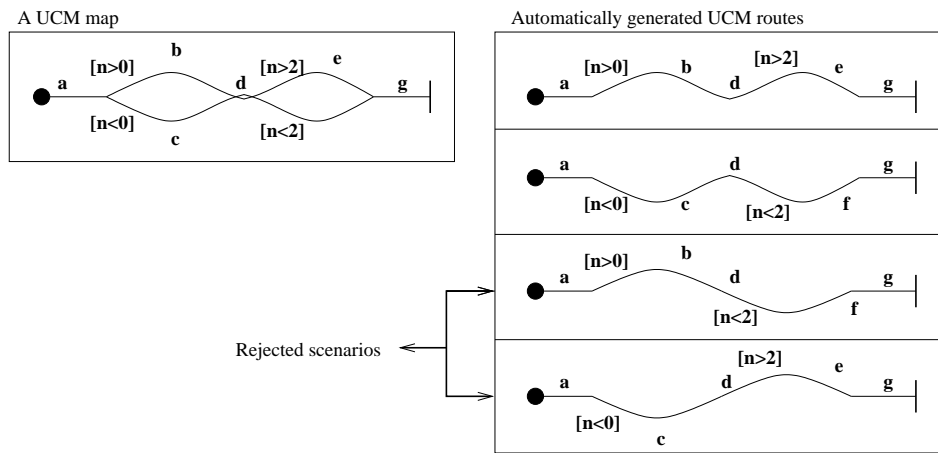


Figure 5.23: Automatic generation of rejected scenarios

This problem is being addressed by the UCM community, with the inclusion of a data model in UCMs.

For the time being, one way to address this problem is the manual analysis of the generated scenarios in order to know which are the acceptance scenarios and which are the rejection ones. The two sets of scenarios can be used for validation purposes. If a specification conforms to its requirements, the first set of scenarios is expected to be successfully executed with the specification and the second set is expected to be rejected. If this is not the case then the specification has to be corrected and validated again.

Another issue is how to test the case where a stub calls itself in the UCM. This presents implementation problems (e.g. fixing the number of times that recursion is executed) and so we assume that such recursion is not present in our UCMs.

5.8 Conclusion

In this chapter, we presented the design of the new functionality `Ucm2LotosTests` implemented for the automatic generation of LOTOS scenarios from UCMs. The generated sce-

narios assure the full coverage of the UCMs, and they can be used for white-box, grey-box and black-box testing.

In the next chapter, we propose a development methodology for the design of telecommunication systems that integrates this functionality for fast validation of specifications and automatic TTCN test suite generation from requirements.

Chapter 6

Proposition of a Development Methodology based on Fast Test Generation

This chapter proposes a new development methodology based on UCMs for the requirements stage and on LOTOS for the formal modeling stage and it proposes an automatic TTCN test suite generation from UCMs.

6.1 Introduction

In chapter 4, we introduced a software development methodology based on UCMs for the requirements, and on LOTOS and SDL for the formal modeling stage. LOTOS was first used at the early specification stage to specify a system from UCMs, and to detect design errors. Then, SDL was used for the late specification stage. Finally, TTCN test suites were automatically generated from the SDL specification using the tool Tau.

In this chapter, we propose a new development methodology. It proposes new ways of testing the system being designed by using the Ucm2LotosTests functionality designed in this thesis and the TGV tool applied on LOTOS. Advantages and limitations of this development methodology are presented later on in this chapter.

6.2 Development Methodology based on UCMs and LOTOS

Figure 6.1 presents the essential phases of the development methodology:

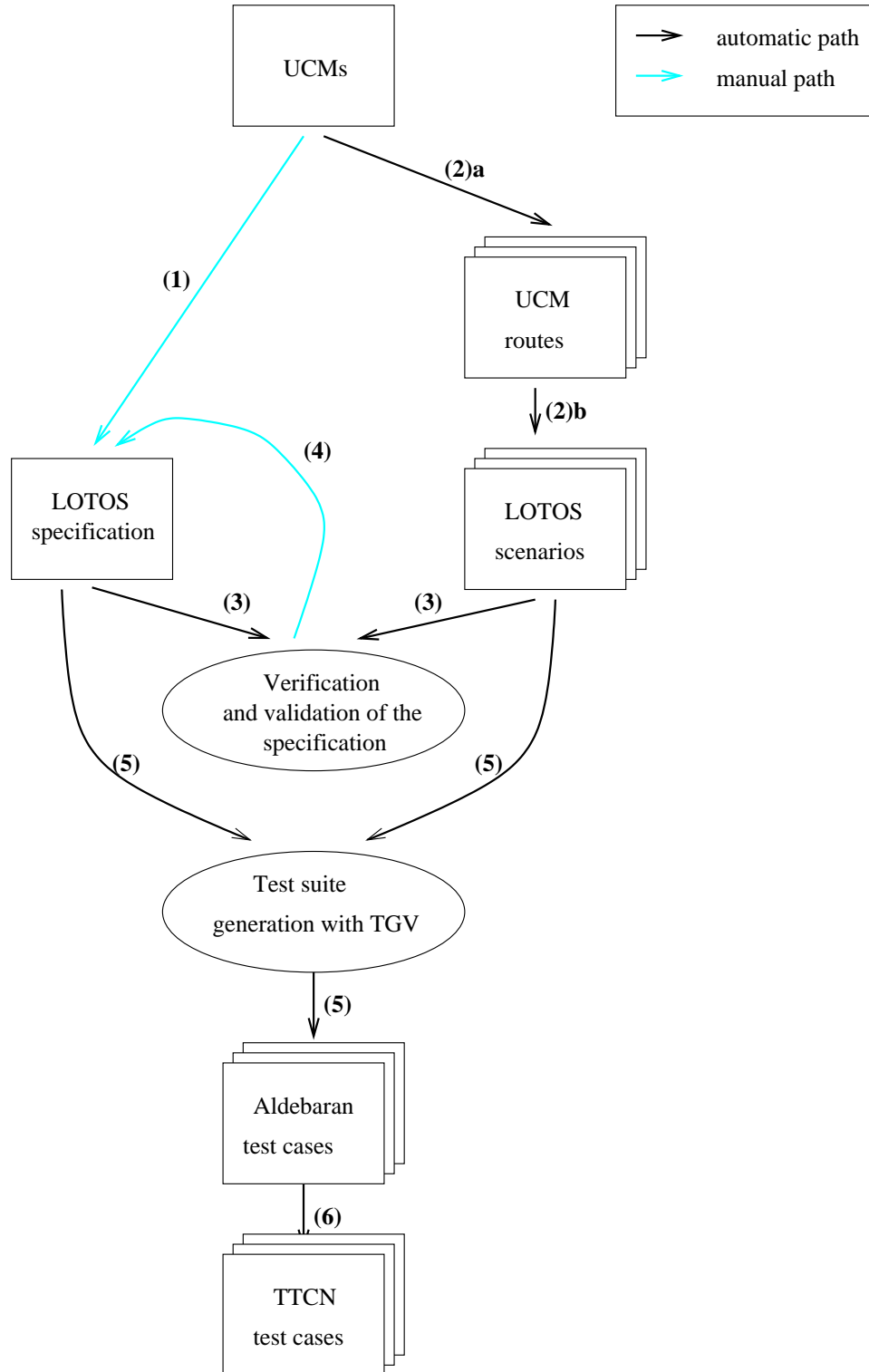


Figure 6.1: New software development methodology approach based on fast test generation

- (1) The requirements of a system designed in UCMs are transformed into a LOTOS behavior. This is done by following a general mapping rule presented in section 4.6.3. The UCM elements are mapped into LOTOS statements. The correspondence between each UCM element and each LOTOS statement is called: mapping M .
- (2) `Ucm2LotosTests` is applied to the UCMs to generate the set of LOTOS scenarios that cover the UCMs. First ((2)a), the UCM routes are generated. Second ((2)b), they are translated into LOTOS scenarios using the mapping M that has been chosen to build the LOTOS specification from UCMs in (1) (details in chapter 5).
- (3) The verification and validation of the LOTOS specification is performed using the tool Caesar (which is part of the CADP toolset introduced in section 3.5.2). Simulation of the specification and validation against the generated scenarios is performed (details on verification and validation of LOTOS specifications are presented in section 3.4).
- (4) Corrections may be made to the specification if any of the executions of the LOTOS scenarios on it are unsuccessful.
- (5) Aldebaran test cases are generated from the LOTOS specification and the LOTOS scenarios using the tool TGV introduced in section 3.5.2.
- (6) TTCN test cases are obtained from the Aldebaran test cases using the tool `Aut2ttn`.

It is important to note that our methodology is based on the assumption that the UCM themselves are correct. This may not be the case in practice, and similar methods to the ones considered here could be used to validate the UCMs [Amy01], however, this is outside of the scope of this thesis.

The main differences between our methodology and the one presented in chapter 4 is:

- The first methodology uses LOTOS and SDL at the formal modeling stage, but the second uses only LOTOS.
- In the first methodology, TTCN test suite generation is performed from SDL using the tool Tau, but in the second, it is obtained from LOTOS using the tool TGV.

6.3 Advantages of the Methodology

The development methodology presented above has several advantages. In the next section, we will demonstrate these advantages by means of a simple case study. These advantages are:

- Use of UCMs, which is a semi formal and scenario-oriented notation. It has been found to be very useful to express requirements. Requirements written in UCM are precise, though semi-formal, and acceptable to developers due to the ease of learning the UCM notation.

- Use of LOTOS which is well suited for the formal specification of telecommunication systems.
- Fast test scenario generation using Ucm2LotosTests method which allows to generate LOTOS scenarios that cover all the UCMs.
- Automatic generation of abstract test suites from requirements expressed by UCMs, using Ucm2LotosTests and TGV.

6.4 Case Study

6.4.1 Introduction

In this section we propose an example of specification and verification of a system using the presented methodology. We will present:

- The UCM requirements of the system to specify,
- The LOTOS specification manually built from the requirements,
- One of the automatically generated LOTOS scenarios from the UCMs using Ucm2LotosTests,
- Validation of the LOTOS specification with the tool Caesar, using the generated black-box scenarios,
- Automatic generation of TTCN test suites using TGV.

This case study was built in order to demonstrate the usefulness and benefits of our development methodology and to experiment with the tools it proposes to use (Ucm2LotosTests, Caesar and TGV).

We were interested in applying the design methodology to the telephony system presented in chapter 4. This was not possible because of two problems:

- The LOTOS specification built from the telephony system requirements was not supported by the tool Caesar because of its very large number of states.
- The LOTOS test generation from the UCMs using Ucm2LotosTests was not successful. This was due to the fact that some stubs of the UCMs were recursive (i.e. they call themselves). Ucm2LotosTests is unable to generate tests in the presence of this type of recursion..

Therefore, the decision was taken to work with a smaller set of UCMs. The UCMs of the Basic Call (BC) feature were extracted from the complete set of requirements of the telephony system presented in chapter 4. From these UCMs, a specification and test scenarios were generated. The steps followed for carrying out our process in this new system with one feature are detailed in the next sections.

6.4.2 Requirements

The UCMs of the example that we discuss here are the UCMs of the basic call feature of the PBX design discussed in chapter 4. The root map of the basic call feature is shown in Figure 6.2.

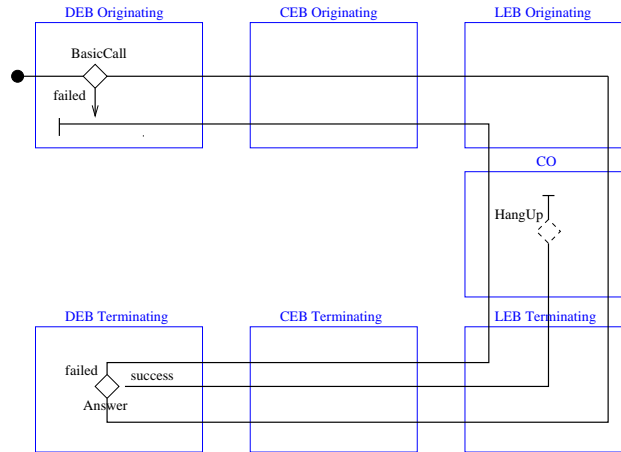


Figure 6.2: Root map of the Basic Call map

6.4.3 LOTOS Specification

A LOTOS specification was manually and quickly built from the UCMs of the BC feature. The mapping rules presented in section 4.6.3 were followed. The top level of the specification is shown below.

```

behavior
(
  (
    (
      DEB [DE_to_USER, USER_to_DE, DE_to_CE, CE_to_DE](origDEB of DebID)
      |||
      DEB [DE_to_USER, USER_to_DE, DE_to_CE, CE_to_DE](destDEB of DebID)
    )
    |[ DE_to_CE, CE_to_DE ]|
  )
  (
    CEB [CE_to_DE, DE_to_CE, CE_to_LE, LE_to_CE](origCEB of CebID)
    |||
    CEB [CE_to_DE, DE_to_CE, CE_to_LE, LE_to_CE](destCEB of CebID)
  )
  |[ LE_to_CE, CE_to_LE ]|
  (
    LEB [LE_to_CO, CO_to_LE, LE_to_CE, CE_to_LE](origLEB of LebID)
    |||
    LEB [LE_to_CO, CO_to_LE, LE_to_CE, CE_to_LE](destLEB of LebID)
  )
)
  
```

```

)
|[ CreateCall, LE_to_CO, CO_to_LE ]|
CallObjectCreator[ CreateCall, LE_to_CO, CO_to_LE](0 of Instance)
)

|[ getCE, FindLE, Find2ndPartyLE, FindCE, FindDE]|

Database[ getCE, FindLE, Find2ndPartyLE, FindCE, FindDE]
)

```

The top-level behavior of the specification is the same as the top level specification built for the system with all the features included, since the physical and logical components are the same. However, since only the Basic Call feature is specified in this LOTOS model, only the relevant processes are specified.

6.4.4 LOTOS Test Generation

From the UCMs (root map is shown in Figure 6.2), Ucm2LotosTests automatically generated 10 tests. The mapping provided during the test generation is the one used to built the LOTOS specification.

A scenario representing a successful connection between two users is shown in figure 6.3 in MSC form. It shows user A dialing, user B ringing and user A receiving ring back. B then goes off hook. A goes on hook, causing dial tone to be received by B which then goes on hook.

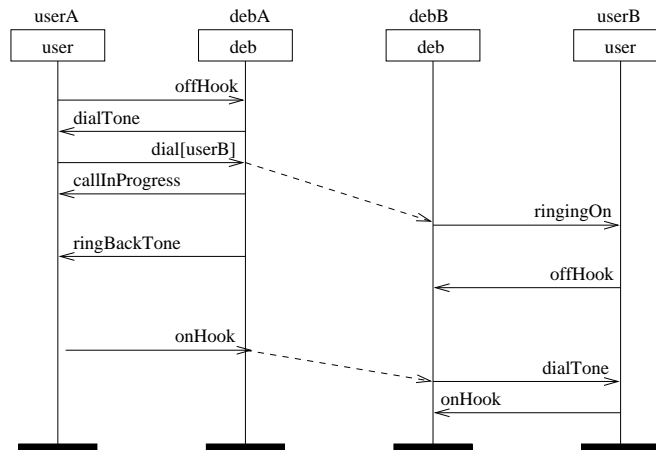


Figure 6.3: Scenario corresponding to a successful connection ending with caller hanging up first

6.4.5 Testing the LOTOS specification

Testing the LOTOS specification for BC was performed using the tool Caesar. Verification of the specification was performed using the step-by-step execution utility of the tool; And

validation of the specification was performed by executing the generated scenarios on the specification.

The 10 tests were executed on the specification. All executions were successful, although we cannot state that the whole LOTOS specification was covered by the tests. Thus, we cannot make sure that all behaviors of the specification are correct. However, we can make sure that the specification behaves as expected in the requirements since all the behaviors present in the requirements are verified in the specification by executing the 10 generated scenarios on it.

6.4.6 TTCN Test Generation

As explained in section 3.5.2, the tool TGV generates Aldebaran test cases from a LOTOS specification and a LOTOS scenario, and the tool Aut2ttn converts Aldebaran test cases into TTCN test cases. We used the tool TGV to obtain Aldebaran test cases from our LOTOS specification and the 10 generated LOTOS scenarios. Then, when we tried passing the Aldebaran test cases through the converter Aut2ttn, we realized that the Aldebaran format expected by the converter is slightly different from the format of the automatically generated tests by TGV. We had to modify the Aldebaran files by modifying information of the headers of the files and the format of the test steps that compose the test cases. From the manually modified Aldebaran files, we were able to obtain TTCN test cases using Aut2ttn.

We believe that, at the time we are writing this thesis, there is a new version of Aut2ttn that takes as input Aldebaran tests that have the same format as the ones automatically generated by TGV.

The following is the TTCN test corresponding to the LOTOS scenario represented in Figure 6.3 in MSC form. As seen in this example, there are no FAIL verdicts and other alternatives of the system behavior are absent. This is due to the fact that TGV doesn't generate the behavior of the system where the scenario fails. This TTCN format includes the field *Constraints Ref* that identifies with a unique name each action performed in the system. For instance the action *offHook* is identified by the name *scenario1_001*.

```

+-----+
|                                     |
|                               Test Case Dynamic Behaviour                               |
|-----+
| Test Case Name      : scenario1                                             |
| Group               : EndToEndCalls/                                       |
| Purpose             : User1 successfully calls User2 and User1 hangs up first |
| Default             :                                                         |
| Comments            :                                                         |
+-----+-----+-----+-----+-----+-----+
| Nr |Label| Behaviour Description      | Constraints Ref      | Verdict|Comments|
+-----+-----+-----+-----+-----+-----+
| 1 |    | user1 !offHook            | scenario1_001       |        |        |
| 2 |    | user1 ?dialTone          | scenario1_002       |        |        |
| 3 |    | user1 !dial              | scenario1_003       |        |        |
| 4 |    | user1 ?callInProgress    | scenario1_004       |        |        |
| 5 |    | user2 ?ringingOn         | scenario1_005       |        |        |
| 6 |    | user1 ?ringBackTone      | scenario1_006       |        |        |

```


could not handle some of the LOTOS constructs that were used in our specifications. This problem would not exist if Caesar had been used from the very beginning of this work.

Limitations of TGV

The TGV tool does not directly generate TTCN test suites from LOTOS; rather, it generates tests in Aldebaran format. From Aldebaran tests, TTCN test suites are generated using the Aut2ttn converter. Details on how TGV works are given in section 3.5.2. In many cases, the generated tests in Aldebaran format have to be manually modified in order for the Aut2ttn converter to be able to transform them into TTCN tests.

6.6 Conclusion

In our case study, a set of LOTOS tests were automatically generated to quickly validate the specification. We believe that this methodology can be further developed and eventually applied to real industrial systems, leading to partial automation of test case generation for such systems.

Chapter 7

Conclusion

This thesis is in the context of formal modeling and test generation of telecommunication systems using UCMs and LOTOS. The essential steps followed in this work are the description of system requirements using the semi-formal notation UCMs, and the formal modeling of the system using the language LOTOS.

7.1 Contributions of the Thesis

The thesis provides a number of contributions. These were announced in section 1.2. Some additional concluding remarks follow.

UCM to LOTOS Formal Modeling (Chapter 4)

This research is in the framework of an explanatory project towards a new development methodology for telecommunication systems which is part of a joint project between the University of Ottawa and Mitel Corporation. The proposed methodology is based on UCMs for designing the requirements of a system, and on LOTOS and SDL for specifying the system at the formal modeling stage. For testing purposes, this methodology proposes the use of the Tau toolset in order to generate TTCN test suites from SDL. This methodology proposes the cross-validation step, usually performed at the end of the formal modeling phase of the development process. This step takes advantage of the use of two formal languages to specify the system and verify that they behave the same way. Some design errors and inconsistencies can be found using this technique.

A first contribution of our work was to perform the formal modelling stage in LOTOS as mentioned in the presented methodology. For this purpose,

- We defined mapping rules transforming each UCM element into a specific LOTOS operator or statement according to their meanings,
- We built a LOTOS model using the mapping rules that we defined. The obtained LOTOS specification using this mapping has the desired behavior of the system specified in UCMs.

- We manually generated black-box tests from the UCMs and showed how these tests can be used to validate the specification.

Automatic LOTOS Test Generation from UCMs (Chapter 5)

One of the motivations of this thesis was the fast test generation from models specified in LOTOS and UCMs. We proposed and implemented a new functionality in the tool UCMNav called Ucm2LotosTests. This new functionality generates a set of LOTOS scenarios from UCMs. The execution of Ucm2LotosTests is performed with interaction with the user who will provide the correspondence between UCM and LOTOS elements. The generated LOTOS scenarios cover all the paths in the UCMs. They can be used as a way to validate the LOTOS specification built from the same UCMs, and as an input of the tool TGV that is used by testers to generate TTCN test suites.

New Development Methodology Proposing a Fast Test Generation (Chapter 6)

The thesis describes a second development methodology for specifying telecommunication systems. This methodology proposes the use of UCMs at requirements stage, and the use of LOTOS at the formal modeling stage. In addition, it proposes new techniques to conduct validation and testing when using LOTOS. It proposes, first, the use of Ucm2LotosTests, a new functionality of UCMNav tool implemented in the thesis, which generates automatically a set of LOTOS scenarios from UCMs. This set is complete in the sense that it covers all the paths in the UCM. Second, the methodology proposes the use of the tools TGV and Aut2ttn to generate automatically TTCN test suites from LOTOS scenarios. These test suites are used for conformance testing purposes at the implementation stage. As explained in the case study in section 6.4.6, we had problems using the tool Aut2ttn since the version we had available didn't accept the format of the automatically generated Aldebaran tests from TGV.

We applied this development methodology to a case study. It allows the automatic generation of tests that will cover the UCMs; Such tests verify the expected behavior of the system in the LOTOS specification. Finally, they are transformed into TTCN tests and are ready to be executed on implementations. However, this methodology currently encounters some problems due to shortcomings in the tools (section 6.5).

7.2 Future Work

Further work can be done to improve the quality of design, tests and to permit the reuse of the already specified systems. Several suggestions would be:

Improving Ucm2LotosTests

Ucm2LotosTests is not able to generate tests from UCMs presenting recursive stubs. This inconvenience can be corrected by checking if the visited stubs have already been covered. In this case, the user can choose to generate or not tests that cover the stubs again.

Furthermore, it was decided in the design of the `Ucm2LotosTests` that, in case a UCM path contains a loop, only two tests would be generated: one test covering the path that does not include the loop, and one that covers one pass in the loop. An improvement to the test generation of `Ucm2LotosTests` in case of presence of loops would be to allow the generation of tests that cover the loop more than once. The user could decide on the number of times a loop would be covered by tests.

Automatic Generation of LOTOS specifications from UCMs

The implementation of the `Ucm2LotosTests` functionality allows the automatic LOTOS test generation from UCMs. However, the specification of a LOTOS model from UCMs is still a manual step. The automation of this step is an open research subject. The UCM to LOTOS mapping rules presented in the thesis for the production of our LOTOS specification can be automated but we cannot insure that the newly obtained LOTOS specification would reflect the behavior of the system to specify. The main issues of this automation deal with the difference of abstraction between a system designed with UCMs and specified with LOTOS.

Regression Testing with `Ucm2LotosTests`

During the development process of software, the requirements could change. This implies that a specification in progress has to be retested against new requirements. By applying regression testing techniques, we can avoid retesting the untouched parts of a system after the modifications. This technique can be applied to `Ucm2LotosTests`. This latter could be applied to a UCM, then to its modification and generate different sets of tests:

- a set of tests that cover the untouched UCM paths,
- a set of tests that cover the modified UCM paths,
- a set of tests that cover the new UCM paths.

All of these can be considered to be regression tests.

Improvements on Caesar, TGV and `Aut2ttn`

It was mentioned in the thesis that the use of TGV for TTCN test generation from LOTOS is performed necessitate the compliance of the LOTOS specification with Caesar requirements. This is inconvenient when the specification is built with a different tool. One way to avoid this inconvenience is to modify the TGV tool so that it could be applied to any kind of LOTOS specifications. Shortcomings were also identified in the `Aut2ttn` tool, however these appear to belong to the programming, rather than to the conceptual domain.

The problems we have dealt with in this thesis have many aspects and implications, both of theoretical and practical significance. We believe that a positive contribution was given in the directions that we were able to pursue.

Appendix A

Case Study Details

This appendix gives details of the case study presented in section 6.4. This case study was used to illustrate the development methodology presented in chapter 6. The case study presents the specification and validation of the basic call feature designed in the project described in chapter 4.

A.1 Presentation of the Requirements

The requirements corresponding to the design of the basic call feature were presented in UCM form. They are described by 9 maps.

Figure A.1 is the root map of the UCMs of the system. It is a sequence of three stubs, the BC stub (Figure A.2), the Answer stub (Figure A.3) and the HangUp stub. The two possible plugins are shown in Figure A.4 and A.5.

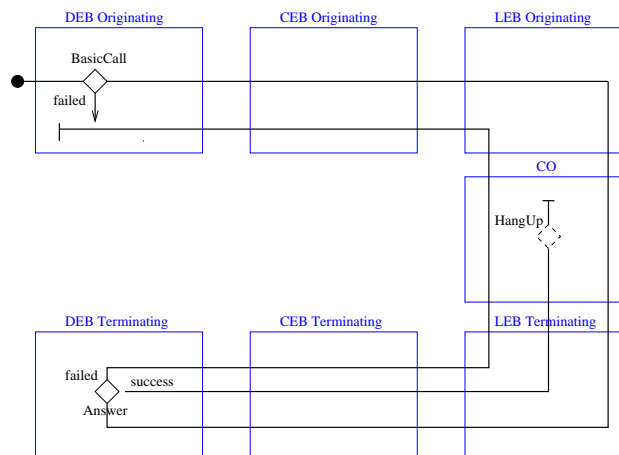


Figure A.1: Root map of the Basic Call map

Figure A.2 describes the Basic Call connection request initiated in the DEB originating.

Figure A.3 describes the system behavior when the callee answers the call connection request through the DEB terminating.

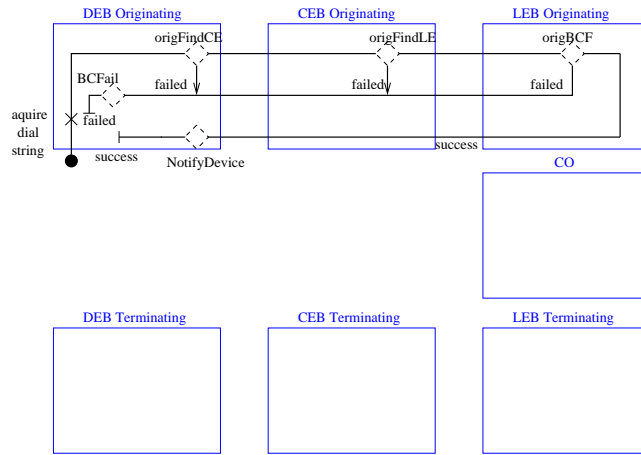


Figure A.2: Root map of the Basic Call map

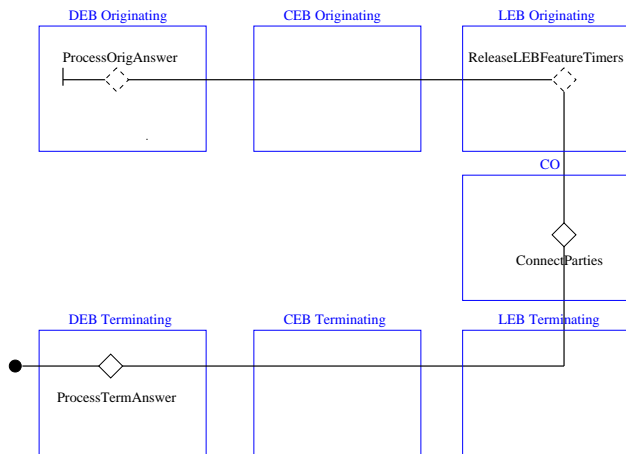


Figure A.3: Answer submap

Figure A.4 describes the termination of the call from the caller end (through the DEB originating).

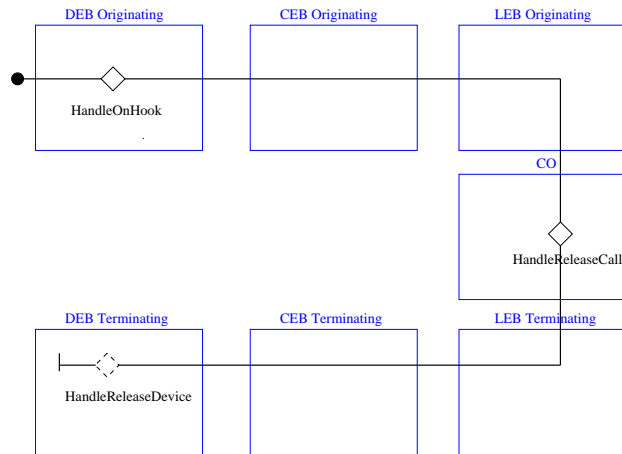


Figure A.4: Hangup originating submap

Figure A.5 describes the termination of the call from the callee end (through the DEB terminating).

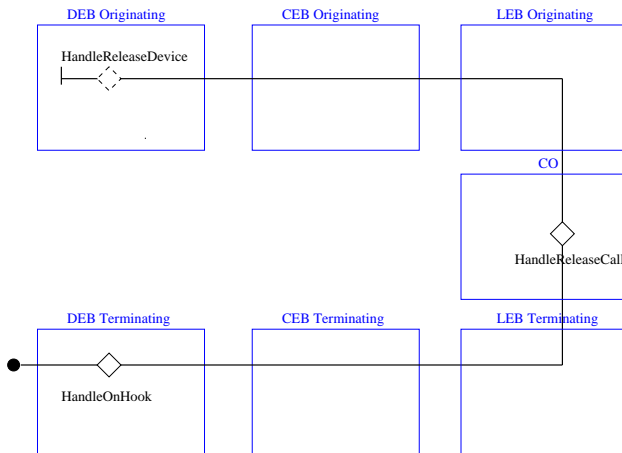


Figure A.5: Hangup terminating submap

A.2 LOTOS Specification

A LOTOS specification describing the Basic Call connection was built from the UCMs of the system (parts are shown in section A.1) following the UCM to LOTOS mapping rules presented in section 4.6.3. The start points, the responsibilities, the conditions and the end points of the UCMs were mapped into LOTOS actions as presented in Figure A.6.

Name	Element	Component	UCM to LOTOS mapping
offHook	start point	Phone1	user_to_phone !user1 !phone1 !offHook; phone_to_user !phone1 !user1 !dialTone;
dial	resp	Phone1	user_to_phone !user1 !phone1 !dial !user2;
checkIdle	resp	Switch	switch_to_phone !phone2 !checkIdle;
[busy]	condition	Switch	phone_to_switch !phone2 !busy;
busyTone	end point	Switch	switch_to_phone !phone1 !busy; phone_to_user !phone1 !user1 !busyTone;
onHook	end point	Phone1	user_to_phone !user1 !phone1 !onHook;
[idle]	condition	Switch	switch_to_phone !phone2 !incomingCall; phone_to_user !phone2 !user2 !ring; phone_to_user !phone1 !user1 !ringBack;
offHook	start point	Phone2	user_to_phone !user2 !phone2 !offHook;
onHook	start point	Phone1	user_to_phone !user1 !phone1 !onHook; phone_to_switch !phone1 !disc; switch_to_phone !phone2 !disc; phone_to_user !phone2 !user2 !discTone;
onHook	end point	Phone2	user_to_phone !user2 !phone2 !onHook;
onHook	start point	Phone2	user_to_phone !user2 !phone2 !onHook; phone_to_switch !phone2 !disc; switch_to_phone !phone1 !disc; phone_to_user !phone1 !user1 !discTone;
onHook	end point	Phone1	user_to_phone !user1 !phone1 !onHook;

Figure A.6: UCM to LOTOS mapping

The important parts of the specification are as follows:

1. Lines 22 through 66 define the data type `Data` used to represent the possible values of the messages exchanged between entities of the system.
2. Lines 68 through 77 define the data type `User` used to represent different users in the system.
3. Lines 79 through 91 define the data type `DEB` used to represent different DEBs in the system.
4. Lines 93 through 105 define the data type `CEB` used to represent different CEBs in the system.
5. Lines 108 through 120 define the data type `LEB` used to represent different LEBs in the system.
6. Lines 122 through 132 define the data type `DialString` used to represent different strings to enter when dialing.
7. Lines 134 through 186 describe the behavior of the specification, which consists of the process `Database` synchronizing with a behavior containing DEBs, CEBs and LEBs and a `CallObjectCreator` synchronizing on their shared gates.
8. Lines 188 through 260 define the process which specifies the Database behavior.
9. Lines 262 through 382 define the process which specifies the DEB behavior.
10. Lines 384 through 440 define the process which specifies the CEB behavior.
11. Lines 442 through 497 define the process which specifies the LEB behavior.
12. Lines 500 through 569 define the process which specifies the CO behavior.
13. Lines 571 through 658 define the process describing the internal behavior of the stubs included in the maps of `BasicCall` and `Answer` stubs.

```
01      (* ***** *)
02      (* Specification of the Basic Call Feature *)
03      (* ***** *)
04      specification BasicCall[ CO_to_LE, LE_to_CO,
05                               LE_to_CE, CE_to_LE,
06                               CE_to_DE, DE_to_CE,
07                               DE_to_USER, USER_to_DE,
08                               (* Responsibilities *)
09                               AcquireDialedString,          validate,
10                               processs,                    reject,
11                               updateDEDatabase,           storeCE,
12                               Ring,                        getCE,
13                               FindLE,                      FindDE,
14                               FindCalleeAddress,          CreateCall,
```

```

15             FindCE,                               Timeout,
16             Find2ndPartyLE,                       ProcessTerminatorAnswer,
17             ProcessOriginatorAnswer,             ConnectParties,
18             HandleReleaseDevice,                 HandleOnHook,
19             HandleReleaseCall
20         ]: exit
21
22 library
23     Boolean, NaturalNumber
24 endlib
25
26 (* Type Data, used to represent the possible values of the
27 * messages exchanged between entities of the system. *)
28 type Data is Boolean
29     sorts Data
30         opns  offHook (*! constructor *),          (* offHook from the environment *)
31              onHook (*! constructor *),           (* onHook from the environment *)
32              dialToneOn (*! constructor *),       (* dial tone on *)
33              dial      (*! constructor *),       (* dial string indication *)
34              digitReceived (*! constructor *),   (* ack from the deb of the digit collection *)
35              ring (*! constructor *),            (* ring signal, for the caller *)
36              ringBackOn (*! constructor *),      (* caller receive the ring back *)
37              ringBackOff (*! constructor *),     (* caller receives answer *)
38              ringingOn (*! constructor *),      (* ringing signal *)
39              busyToneOn (*! constructor *),      (* busy signal, for the caller *)
40              busyInd (*! constructor *),         (* busy signal, for the system *)
41              connectReq (*! constructor *),     (* connect Request *)
42              connectInd (*! constructor *),     (* connect Indication *)
43              connectResp (*! constructor *),    (* connect response *)
44              connectConf (*! constructor *),    (* connect confirmation *)
45              answerReq (*! constructor *),      (* answer notification *)
46              answerInd (*! constructor *),     (* answer indication *)
47              disconnectReq (*! constructor *),  (* disconnection Request *)
48              disconnectInd (*! constructor *),  (* disconnection Indication *)
49              registered (*! constructor *),     (* registered in the CE DB *)
50              rejected (*! constructor *),      (* rejected from the CE DB *)
51              result (*! constructor *),        (* success or failure *)
52              success (*! constructor *),       (* success notification *)
53              failure (*! constructor *),       (* failure notification *)
54              restart (*! constructor *)        (* restart notification *)
55         : -> Data
56         _ == _ : Data, Data -> Bool
57         _ <> _ : Data, Data -> Bool
58
59     eqns forall x,y: Data
60 ofsort Bool
61         x <> y = not(x == y)
62     endtype
63     (* instances of CEBs, LEBs and COs *)
64     type Instance is NaturalNumber renamedby
65         sortnames Instance for Nat
66     endtype
67

```

```

68     type UserID is Boolean
69         sorts UserID
70         opns caller (! constructor *),
71             callee (! constructor *) :-> UserID
72             _ == _ : UserID, UserID -> Bool
73             _ <> _ : UserID, UserID -> Bool
74
75         eqns forall x,y: UserID ofsort Bool
76             x <> y = not(x == y)
77     endtype
78
79     (* DEB: Device Entity Block : phone1, phone2, ... *)
80     type DebID is Boolean
81         sorts DebID
82         opns phone1 (! constructor *), (* originating DEB *)
83             phone2 (! constructor *), (* terminating DEB *)
84             nodeb (! constructor *) (* no deb found *)
85             :-> DebID
86             _ == _ : DebID, DebID -> Bool
87             _ <> _ : DebID, DebID -> Bool
88
89         eqns forall x,y: DebID ofsort Bool
90             x <> y = not(x == y)
91     endtype
92
93     (* CEB: Communicating Entity Block : terminating, originating, ... *)
94     type CebID is Boolean
95         sorts CebID
96         opns terminating (! constructor *), (* originating CEB *)
97             originating (! constructor *), (* terminating CEB *)
98             noceb (! constructor *) (* no ceb found *)
99             :-> CebID
100            _ == _ : CebID, CebID -> Bool
101            _ <> _ : CebID, CebID -> Bool
102
103     eqns forall x,y: CebID ofsort Bool
104         x <> y = not(x == y)
105     endtype
106
107
108     (* LEB: Location Entity Block *)
109     type LebID is Boolean
110         sorts LebID
111         opns ottawa (! constructor *), (* originating LEB *)
112             montreal (! constructor *), (* terminating LEB *)
113             noleb (! constructor *) (* no leb found *)
114             :-> LebID
115             _ == _ : LebID, LebID -> Bool
116             _ <> _ : LebID, LebID -> Bool
117
118         eqns forall x,y: LebID ofsort Bool
119             x <> y = not(x == y)
120     endtype

```

```

121
122 (* Possible dial strings *)
123 type DialString is Boolean
124     sorts DialString
125     opns termNum (*! constructor *),
126     origNum (*! constructor *) :-> DialString
127     _ == _ : DialString, DialString -> Bool
128     _ <> _ : DialString, DialString -> Bool
129
130     eqns forall x,y: DialString ofsort Bool
131         x <> y = not(x == y)
132     endtype
133
134     behavior
135     (
136     (
137     ( (* DEB entities *)
138     DEB[ DE_to_USER, USER_to_DE, DE_to_CE, CE_to_DE,
139     validate, processs, reject, updatedEDatabase, storeCE,
140     Ring, getCE, ProcessOriginatorAnswer, HandleOnHook,
141     HandleReleaseDevice, ProcessTerminatorAnswer
142     ] (phone1 of DebID)
143
144     |||
145
146     DEB[ DE_to_USER, USER_to_DE, DE_to_CE, CE_to_DE,
147     validate, processs, reject, updatedEDatabase, storeCE,
148     Ring, getCE, ProcessOriginatorAnswer, HandleOnHook
149     HandleReleaseDevice, ProcessTerminatorAnswer
150     ](phone2 of DebID)
151     )
152     )
153     |[ DE_to_CE, CE_to_DE ]|
154
155     ( (* CEB entities *)
156     CEB[ CE_to_DE, DE_to_CE, CE_to_LE, LE_to_CE,
157     FindLE, FindDE ](terminating of CebID, 0 of instance)
158     |||
159
160     CEB[ CE_to_DE, DE_to_CE, CE_to_LE, LE_to_CE,
161     FindLE, FindDE ](originating of CebID, 0 of instance)
162     )
163     |[ LE_to_CE, CE_to_LE ]|
164
165     ( (* LEB entities *)
166     LEB[ LE_to_CO, CO_to_LE, LE_to_CE, CE_to_LE, FindCalleeAddress,
167     CreateCall, FindCE, Timeout ](montreal of LebID, 0 of instance)
168     |||
169
170     LEB[ LE_to_CO, CO_to_LE, LE_to_CE, CE_to_LE, FindCalleeAddress,
171     CreateCall, FindCE, Timeout ](ottawa of LebID, 0 of instance)
172     )
173

```



```

174
175         [[ CreateCall, LE_to_CO, CO_to_LE ]|
176
177         CallObjectCreator[ CreateCall, LE_to_CO, CO_to_LE, Find2ndPartyLE,
178                           ConnectParties, HandleReleaseCall](0 of Instance)
179
180     )
181
182     [[ getCE, FindLE, Find2ndPartyLE, FindCE, FindDE, updateDEDatabase, storeCE]|
183
184     Database[ getCE, FindLE, Find2ndPartyLE, FindCE, FindDE, updateDEDatabase, storeCE]
185     )
186     where
187
188     process Database[getCE, FindLE, Find2ndPartyLE, FindCE, FindDE, UpdatedDEDatabase, StoreCE]
189     :exit:=
190     (
191     (
192         (* a DEB wants to Find its CEB, login code could be added *)
193         (
194         getCE !phone1 !originating; exit
195         []
196         getCE !phone2 !terminating; exit
197         []
198         getCE ?deb: DebID !failure [ (deb <> phone2) and (deb <> phone1)]; exit
199         )
200
201         []
202
203         (* a DEB wants to update the DE database by adding the new CE *)
204
205         UpdateDEDatabase ?deb:DebID ?ceb:CebID; exit
206
207         []
208
209         (* a DEB wants to store the new CE *)
210
211         StoreCE ?deb:DebID ?ceb:CebID; exit
212
213         []
214
215         (* a CEB wants to Find its LEB *)
216         (
217         FindLE !originating !ottawa; exit
218         []
219         FindLE !terminating !montreal; exit
220         []
221         FindLE ?ceb:CebID !failure [ (ceb <> originating) and (ceb <> terminating)]; exit
222         )
223
224         []
225
226         (* a Call Object wants to Find the terminating LEB *)
227         (

```

```

228     Find2ndPartyLE !termNum !montreal; exit
229     []
230     Find2ndPartyLE !origNum !ottawa; exit
231     []
232     Find2ndPartyLE ?ds:DialString !failure [ (ds <> termNum) and (ds <> origNum)]; exit
233 )
234
235     []
236
237     (* a LEB wants to Find a corresponding CEB *)
238     (
239     FindCE !termNum !terminating; exit
240     []
241     FindCE !origNum !originating; exit
242     []
243     FindCE ?ds:DialString !failure [ (ds <> termNum) and (ds <> origNum) ]; exit
244     )
245
246     []
247
248
249     (* a CEB wants to Find its DEB, policies can be added *)
250     (
251     FindDE !originating !phone1; exit
252     []
253     FindDE !terminating !phone2; exit
254     []
255     FindCE ?ceb:CebID !failure [ (ceb <> originating) and ( ceb <> terminating) ]; exit
256     )
257
258     (* And we replicate the process to be able to continue *)
259 ) >> Database[ getCE, FindLE, Find2ndPartyLE, FindCE, FindDE, UpdatedDEDatabase, StoreCE]
260 endproc
261
262 process DEB[DE_to_USER, USER_to_DE, DE_to_CE, CE_to_DE,
263     AcquireDialedString, Ring, getCE, ProcessOriginatorAnswer,
264     ProcessTerminatorAnswer, HandleReleaseDevice
265     HandleOnHook](deb:DebID): exit:=
266     (
267     (* env chooses to use this DEB and logs in *)
268     USER_to_DE !caller !deb !offHook;
269     DE_to_USER !deb !caller !dialToneOn;
270     (
271     (* management of the incoming calls *)
272     CE_to_DE ?ceb:CebID ?cebInst:Instance !deb !connectInd;(* busy tone *)
273     DE_to_CE !deb !ceb !cebInst !busyToneOn; exit
274     []
275     HandleReleaseDevice; exit (* onHook *)
276     []
277     HandleOnHook; exit
278     )
279
280     |[HandleReleaseDevice, HandleOnHook]|

```

```

281
282     (* management of call *)
283     USER_to_DE !caller !deb !dial ?ds: DialString;
284     DE_to_USER !deb !caller !digitReceived;
285     (
286         OriginatorFindCE[getCE](deb, result) >>
287         accept ceb: CebID, result:Data in
288         (
289             [result == success] ->
290             (
291                 DE_to_CE !deb !ceb ?cebInst:Instance !connectReq !ds;
292                 (
293                     (
294                         CE_to_DE !ceb !cebInst !deb !failure;
295                         DE_to_USER !deb !caller !dialToneOn; exit
296                     )
297                     []
298                     (
299                         CE_to_DE !ceb !cebInst !deb !connectConf;
300                         DE_to_USER !deb !caller !ringBackOn;
301                         CE_to_DE !ceb !cebInst !deb !answerInd;
302                         ProcessOriginatorAnswer;
303                         DE_to_USER !deb !caller !ringBackOff;
304                         (* parties are talking together *)
305                         (
306                             (
307                                 CE_to_DE !ceb !cebInst !deb !disconnectInd;
308                                 HandleReleaseDevice;
309                                 DE_to_USER !deb !caller !dialToneOn;
310                                 USER_to_DE !caller !deb !onHook; exit
311                             )
312                             []
313                             (
314                                 USER_to_DE !caller !deb !onHook;
315                                 HandleOnHook;
316                                 DE_to_CE !deb !ceb !cebInst !disconnectReq; exit
317                             )
318                         )
319                     )
320                 )
321             )
322             []
323             [result == failure] -> DE_to_USER !deb !caller !dialToneOn; exit
324         )
325     )
326 )
327 )
328 []
329 (* Incoming Call from a CEB *)
330 CE_to_DE ?ceb:CebID ?cebInst:Instance !deb !connectInd;
331 (
332     Ring;
333     (

```

```

334 DE_to_USER !deb !callee !ringingOn;
335 DE_to_CE !deb !ceb !cebInst !connectResp;      (* ringing *)
336 USER_to_DE !callee !deb !offHook;            (* env answers *)
337 ProcessTerminatorAnswer;
338 DE_to_CE !deb !ceb !cebInst !answerReq;
339 (
340     (
341         USER_to_DE !callee !deb !onHook;        (* onHook, end of the call *)
342         HandleOnHook;
343         DE_to_CE !deb !ceb !cebInst !disconnectReq; exit
344     )
345     []
346     (
347         CE_to_DE !ceb !cebInst !deb !disconnectInd; (* onHook, end of the call *)
348         HandleReleaseDevice;
349         DE_to_USER !deb !caller !dialToneOn;
350         USER_to_DE !caller !deb !onHook; exit
351     )
352 )
353 )
354 )
355 )
356 (* Recall of the process to be able to be used again *)
357 >> DEB[ DE_to_USER, USER_to_DE, DE_to_CE, CE_to_DE, AcquireDialedString,
358     Ring,
359     getCE, ProcessOriginatorAnswer, ProcessTerminatorAnswer,
360     HandleReleaseDevice, HandleOnHook](deb)
361
362 endproc
363
364 process CEB[ (* comms with DEB *) CE_to_DE, DE_to_CE,
365             (* comms with LEB *) DE_to_CE, CE_to_LE, LE_to_CE,
366             (* Responsibilities*) FindLE,FindDE](ceb:CebID, cebInst:Instance): exit:=
367 (* originating CEB receives a request from an originating DEB *)
368 DE_to_CE ?deb:DebID !ceb !cebInst !connectReq ?ds:DialString;
369 (
370     (
371         OriginatorFindLE[ FindLE] (ceb) >>
372         accept leb: LebID, result: Data in
373         (
374             [result == success] ->                (* Find LE succeeded *)
375             CE_to_LE!ceb!cebInst!leb?lebInst:Instance!connectReq!ds;(* contact LE and ds *)
376             LE_to_CE !leb !lebInst !ceb !cebInst ?result:Data;(* receive result notification *)
377             CE_to_DE !ceb !cebInst !deb !result;    (* and pass it back to the DEB *)
378             (
379                 [ result == connectConf ] ->
380                 (
381                     LE_to_CE !leb !lebInst !ceb !cebInst !answerInd;
382                     CE_to_DE !ceb !cebInst !deb !answerInd;
383                     (
384                         (
385                             DE_to_CE !deb !ceb !cebInst !disconnectReq;
386                             CE_to_LE !ceb !cebInst !leb !lebInst !disconnectReq; exit

```

```

387         )
388     []
389     (
390         LE_to_CE !leb !lebInst !ceb !cebInst !disconnectInd;
391         CE_to_DE !ceb !cebInst !deb !disconnectInd; exit
392     )
393 )
394 )
395 )
396 []
397 [result == failure] -> (* Find LE failed *)
398     CE_to_DE !ceb !cebInst !deb !failure; exit (* pass it back to the DEB *)
399 )
400 )
401 )
402
403 []
404
405 (* terminating CEB receives a request from a terminating LEB *)
406 LE_to_CE ?leb:LebID ?lebInst:Instance !ceb !cebInst !connectInd;
407 (
408     TerminatorFindDE[ FindDE ](ceb) >>
409     accept deb: DebID, result: Data in
410     (
411         [ result == success ] -> (* FindDE succeeded *)
412         (
413             CE_to_DE !ceb !cebInst !deb !connectInd; (* Contact the DEB *)
414             DE_to_CE !deb !ceb !cebInst ?result:Data; (* receive result notification *)
415             CE_to_LE !ceb !cebInst !leb !lebInst !result;(* and pass it back to the LEB *)
416             (
417                 [ result == connectResp ] ->
418                 (
419                     DE_to_CE !deb !ceb !cebInst !answerReq;
420                     CE_to_LE !ceb !cebInst !leb !lebInst !answerReq;
421                     (
422                         (
423                             LE_to_CE !leb !lebInst !ceb !cebInst !disconnectInd;
424                             CE_to_DE !ceb !cebInst !deb !disconnectInd; exit
425                         )
426                         []
427                         (
428                             DE_to_CE !deb !ceb !cebInst !disconnectReq;
429                             CE_to_LE !ceb !cebInst !leb !lebInst !disconnectReq; exit
430                         )
431                     )
432                 )
433             )
434         )
435     []
436     [ result == failure ] -> (* Find DE failed *)
437         CE_to_LE !ceb !cebInst !leb !lebInst !failure; exit
438 )
439 )

```

```

440 endproc
441
442 process LEBs[(*comms with Call Object*) LE_to_CO, CO_to_LE,
443             (*comms with CEB*) LE_to_CE, CE_to_LE,
444             (*Responsibilities*) FindCalleeAddress, CreateCall, FindCE, Timeout]
445             (leb:LebID, lebInst:Instance): exit:=
446 (* LEB receives a request from a CEB *)
447 CE_to_LE ?ceb:CebID ?cebInst:Instance !leb !lebInst !connectReq ?ds:DialString; (
448 (
449 (* Entering STATIC stub OriginatorBasicCallSetup *)
450 OriginatorBasicCallSetup[ LE_to_CO, CO_to_LE, LE_to_CE, CE_to_LE,
451 FindCalleeAddress, CreateCall, FindCE, Timeout ](leb, lebInst, ceb, cebInst, ds)
452 >> accept result:Data, co:Instance in (
453 LE_to_CE !leb !lebInst !ceb !cebInst !result; ((* pass result to the CEB *))
454 (
455 [ result == answerInd ] -> (
456 CE_to_LE !ceb !cebInst !leb !lebInst !disconnectReq;(* disconnectReq from CEB *)
457 LE_to_CO !leb !lebInst !co !disconnectReq; exit
458 []
459 CO_to_LE !co !leb !lebInst !disconnectInd;(* disconnectInd from CallObj *)
460 LE_to_CE !leb !lebInst !ceb !cebInst !disconnectInd; exit
461 )
462 )
463 )
464 []
465 ( [ result <> answerInd ] -> exit )
466 )
467 )
468 )
469 )
470
471 []
472
473 (* LEB receives a request from a Call Object *)
474 CO_to_LE ?co:Instance !leb !lebInst !connectInd ?ds:DialString; (
475 (
476 LEBTerminatingBasicCallSetup[ LE_to_CO, CO_to_LE, LE_to_CE, CE_to_LE, FindCE
477 ](leb, lebInst, co, ds) >> accept result:Data, ceb:CebID, cebInst:Instance in (
478 LE_to_CO !leb !lebInst !co !result; (
479 (
480 [ result == connectResp ] -> (
481 CE_to_LE !ceb !cebInst !leb !lebInst !answerReq;
482 LE_to_CO !leb !lebInst !co !answerReq; (
483 CE_to_LE !ceb !cebInst !leb !lebInst !disconnectReq;(* disconnectReq from CEB *)
484 LE_to_CO !leb !lebInst !co !disconnectReq; exit
485 []
486 CO_to_LE !co !leb !lebInst !disconnectInd; (* disconnectInd from CallObj *)
487 LE_to_CE !leb !lebInst !ceb !cebInst !disconnectInd; exit
488 )
489 )
490 )
491 []
492 ( [ result <> connectResp ] -> exit )
493 )

```

```

494 )
495 )
496 )
497 endproc
498
499 process CallObject[ LE_to_CO, CO_to_LE, Find2ndPartyLE,
500                     ConnectParties, HandleReleaseCall
501                     ](co:Instance): exit:=
502 (* we receive a request from a LEB *)
503 LE_to_CO ?leb0:LebID ?leb0Inst:Instance !co !connectReq ?ds:DialString; (
504 Find2ndPartyLE[ Find2ndPartyLE ](ds) >> accept result:Data, lebT:LebID in (
505 [ result == failure ] -> (
506     CO_to_LE !co !leb0 !leb0Inst !failure; exit
507 )
508 []
509 [ result == success ] -> (
510     OriginatorBasicCallTerminate[ LE_to_CO, CO_to_LE,
511                                     ConnectParties, HandleReleaseCall
512                                     ](co, leb0, leb0Inst, lebT, ds)
513     >> accept result:Data, lebTInst:Instance in (
514         [ result == connectResp ] -> (
515             CO_to_LE !co !leb0 !leb0Inst !connectConf;
516             LE_to_CO !lebT !lebTInst !co !answerReq;
517             ConnectParties;
518             CO_to_LE !co !leb0 !leb0Inst !answerInd; (
519                 LE_to_CO !lebT !lebTInst !co !disconnectReq;
520                 CO_to_LE !co !leb0 !leb0Inst !disconnectInd;
521                 HandleReleaseCall; exit
522             []
523             LE_to_CO !leb0 !leb0Inst !co !disconnectReq;
524             CO_to_LE !co !lebT !lebTInst !disconnectInd;
525             HandleReleaseCall; exit
526         )
527     )
528     []
529     [ result <> connectResp ] -> exit
530 )
531 )
532 )
533 )
534 endproc
535
536 process CallObjectCreator[ CreateCall, LE_to_CO, CO_to_LE, Find2ndPartyLE,
537                             ConnectParties, HandleReleaseCall](inst:Instance):exit:=
538     CreateCall !inst; (
539     CallObject[LE_to_CO,CO_to_LE,Find2ndPartyLE,ConnectParties,HandleReleaseCall](inst)
540     |||
541     (
542     let inst:Instance = succ(inst) in (
543         CreateCall !inst;
544         CallObject[LE_to_CO,CO_to_LE,Find2ndPartyLE,ConnectParties,HandleReleaseCall](inst)
545         |||
546         (

```

```

547     let inst:Instance = succ(inst) in ( CreateCall !inst;
548         CreateCall !inst; CallObject[ LE_to_CO,CO_to_LE,Find2ndPartyLE,
549             ConnectParties,HandleReleaseCall ](inst)
550     |||
551     (
552         let inst:Instance = succ(inst) in (
553             CreateCall !inst; CallObject[ LE_to_CO,CO_to_LE,Find2ndPartyLE,
554                 ConnectParties,HandleReleaseCall](inst)
555         |||
556         (
557             let inst:Instance = succ(inst) in (
558                 CreateCall !inst; CallObject[ LE_to_CO,CO_to_LE,Find2ndPartyLE,
559                     ConnectParties,HandleReleaseCall](inst)
560             )
561         )
562     )
563 )
564 )
565 )
566 )
567 )
568 )
569 endproc
570
571 (* ----- DEB plug-ins -----*)
572 process OriginatorFindCE[ getCE ] (deb: DebID, result: Data): exit(CebID, Data):=
573     (* registered == true in the ucm, CE found in other words *)
574     getCE !deb ?ceb: CebID; exit(ceb, success)
575     []
576     (* registered == false *)
577     getCE !deb !failure; exit(noceb, failure)
578 endproc
579
580 (* ----- CEB originating plug-ins -----*)
581 process OriginatorFindLE[ FindLE ](ceb:CebID): exit(LebID, Data):=
582     FindLE!ceb?leb:LebID;exit(leb,success) [] FindLE?ceb:CebID!failure;exit(noleb,failure)
583 endproc
584
585 (* ----- LEB terminating plug-ins -----*)
586 process TerminatorFindDE[ FindDE ] (ceb: CebID): exit(DebID, Data):=
587     FindDE !ceb ?deb:DebID; (* FindDE succeeded *) exit(deb, success)
588     []
589     FindDE !ceb !failure; (* Find DE failed *) exit(nodeb, failure)
590 endproc
591
592 (* ----- LEB plug-ins -----*)
593 process LEBTerminatingBasicCallSetup[ LE_to_CO, CO_to_LE, LE_to_CE, CE_to_LE, FindCE
594     ](leb:LebID,lebInst:Instance,co:Instance,ds:DialString)
595     : exit(Data, CebID, Instance):=
596
597     TerminatorFindCE[ FindCE ](ds) >> accept result:Data, ceb:CebID in (
598     [ result == failure ] -> exit(failure, noceb, 0 of Instance)
599     []

```



```

600     [ result == success ] ->
601     (
602         LE_to_CE !leb !lebInst !ceb ?cebInst:Instance !connectInd;
603         CE_to_LE !ceb !cebInst !leb !lebInst ?result:Data; (
604             [ result == connectResp ] -> exit(connectResp, ceb, cebInst)
605             []
606             [ result == failure ] -> exit(failure, ceb, cebInst)
607             []
608             [ result == busyInd ] -> exit(busyInd, ceb, cebInst)
609         )
610     )
611 )
612 endproc
613
614 process TerminatorFindCE[ FindCE](ds: dialString): exit(Data, CebID):=
615     FindCE !ds ?ceb:CebID; exit(success, ceb) [] FindCE !ds !failure; exit(failure, noceb)
616 endproc
617
618 process OriginatorBasicCallSetup[ (* comms with Call Object *) LE_to_CO, CO_to_LE,
619                                     (* comms with CEB *) LE_to_CE, CE_to_LE,
620                                     (* Responsibilities *) FindCalleeAddress, CreateCall,
621
622                                     FindCE, Timeout
623                                     ](leb:LebID, lebInst:Instance,
624                                     ceb:CebID, cebInst:Instance, ds:DialString)
625     : exit(Data, Instance):=
626     FindCalleeAddress; (* take care of resp FindCalleeAddress *)
627     CreateCall ?co:Instance; LE_to_CO !leb !lebInst !co !connectReq !ds; (
628     CO_to_LE !co !leb !lebInst !connectConf; (* connection succeeded *)
629     LE_to_CE !leb !lebInst !ceb !cebInst !connectConf;
630     WaitForAnswer[ CO_to_LE, LE_to_CE, Timeout ](leb, lebInst, ceb, cebInst, co)
631     >> accept result :Data in (
632         [ result == answerInd ] -> (
633             exit(answerInd, co)
634         )
635     )
636 )
637 []
638 CO_to_LE !co !leb !lebInst !failure; (* connection failed *) exit(failure, co)
639 )
640 endproc
641
642 process WaitForAnswer[ CO_to_LE, LE_to_CE, Timeout
643                                     ](leb:LebID, lebInst:Instance, ceb:CebID, cebInst:Instance,
644                                     co:Instance) : exit(Data):=
645     CO_to_LE !co !leb !lebInst !answerInd; exit(answerInd) [] Timeout; exit(failure)
646 endproc
647
648 (* Call Object plug-ins *)
649 process Find2ndPartyLE[ Find2ndPartyLE ](ds:DialString): exit(Data, LebID):=
650     Find2ndPartyLE!ds ?lebT:LebID; exit(success,lebT) [] Find2ndPartyLE!ds !failure;
651     exit(failure,noleb)
652 endproc

```

```

653
654 process OriginatorBasicCallTerminate[LE_to_CO,CO_to_LE,ConnectParties,HandleReleaseCall
655                                     ](co:Instance, leb0:LebID, leb0Inst:Instance,
656                                     lebT:LebID, ds:DialString): exit(Data, Instance):=
657   CO_to_LE !co !lebT ?lebTInst:Instance !connectInd !ds;
658   LE_to_CO !lebT !lebTInst !co ?result:Data; exit(result, lebTInst)
659 endproc

```

A.3 LOTOS scenarios generated with Ucm2LotosTests

From the requirements presented in section A.1, the Ucm2LotosTests functionality was used to generate LOTOS scenarios following the mapping of Figure A.6. The obtained LOTOS scenarios are shown below.

Scenario1 describes a call connection request from a user who doesn't have a CEB.

```

process Scenario1[ USER_to_DE, DE_to_USER, scenario1 ]: noexit:=
  USER_to_DE !caller !phone1 !offhook;
  DE_to_USER !phone1 !caller !dialtone;
  USER_to_DE !caller !phone1 !dial !termNum;
  DE_to_USER !phone1 !caller !digitreceived;
  getCE !phone1 !noCEB;
  DE_to_USER !phone1 !caller !busyTone;
  USER_to_DE !caller !phone1 !onHook;
  scenario1; stop
endproc

```

Scenario2 describes a call connection request from a user who doesn't have a LEB.

```

process Scenario2[ USER_to_DE, DE_to_USER, scenario2 ]: noexit:=
  USER_to_DE !caller !phone1 !offhook;
  DE_to_USER !phone1 !caller !dialtone;
  USER_to_DE !caller !phone1 !dial !termNum;
  DE_to_USER !phone1 !caller !digitreceived;
  getCE !phone1 !originating;
  DE_to_CE !phone1 !originating !0 of instance !connectreq !termnum;
  findLE !originating !noleb;
  CE_to_DE !originating !0 of instance !phone1 !failure;
  DE_to_USER !phone1 !caller !busyTone;
  USER_to_DE !caller !phone1 !onHook;
  scenario2; stop
endproc

```

Scenario3 describes a call connection request to a user who doesn't have a LEB.

```

process Scenario3[ USER_to_DE, DE_to_USER, scenario3 ]: noexit:=
  USER_to_DE !caller !phone1 !offhook;
  DE_to_USER !phone1 !caller !dialtone;
  USER_to_DE !caller !phone1 !dial !termNum;
  DE_to_USER !phone1 !caller !digitreceived;
  getCE !phone1 !originating;
  DE_to_CE !phone1 !originating !0 of instance !connectreq !termnum;
  findLE !originating !ottawa;

```

```

CE_to_LE !originating !0 of instance !ottawa !0 of instance !connectreq !termnum;
findCalleeAddress;
createCall !0 of instance;
LE_to_CO !ottawa !0 of instance !0 of instance !connectreq !termnum;
find2ndPartyLE !termnum !noleb;
DE_to_USER !phone1 !caller !busyTone;
USER_to_DE !caller !phone1 !onHook;
scenario3; stop
endproc

```

Scenario4 describes a call connection request to a user who doesn't have a CEB.

```

process Scenario4[ USER_to_DE, DE_to_USER, scenario4 ]: noexit:=
  USER_to_DE !caller !phone1 !offhook;
  DE_to_USER !phone1 !caller !dialtone;
  USER_to_DE !caller !phone1 !dial !termNum;
  DE_to_USER !phone1 !caller !digitreceived;
  getCE !phone1 !originating;
  DE_to_CE !phone1 !originating !0 of instance !connectreq !termnum;
  findLE !originating !ottawa;
  CE_to_LE !originating !0 of instance !ottawa !0 of instance !connectreq !termnum;
  findCalleeAddress;
  createCall !0 of instance;
  LE_to_CO !ottawa !0 of instance !0 of instance !connectreq !termnum;
  find2ndPartyLE !termnum !montreal;
  CO_to_LE !0 of instance !montreal !0 of instance !connectind !termnum;
  findCE !termnum !noceb;
  DE_to_USER !phone1 !caller !busyTone;
  USER_to_DE !caller !phone1 !onHook;
  scenario4; stop
endproc

```

Scenario5 describes a call connection request to a user who doesn't have a DEB.

```

process Scenario5[ USER_to_DE, DE_to_USER, scenario5 ]: noexit:=
  USER_to_DE !caller !phone1 !offhook;
  DE_to_USER !phone1 !caller !dialtone;
  USER_to_DE !caller !phone1 !dial !termNum;
  DE_to_USER !phone1 !caller !digitreceived;
  getCE !phone1 !originating;
  DE_to_CE !phone1 !originating !0 of instance !connectreq !termnum;
  findLE !originating !ottawa;
  CE_to_LE !originating !0 of instance !ottawa !0 of instance !connectreq !termnum;
  findCalleeAddress;
  createCall !0 of instance;
  LE_to_CO !ottawa !0 of instance !0 of instance !connectreq !termnum;
  find2ndPartyLE !termnum !montreal;
  CO_to_LE !0 of instance !montreal !0 of instance !connectind !termnum;
  findCE !termnum !terminating;
  LE_to_CE !montreal !0 of instance !terminating !0 of instance !connectind;
  findDE !terminating !nodeb;
  DE_to_USER !phone1 !caller !busyTone;
  USER_to_DE !caller !phone1 !onHook;
  scenario5; stop
endproc

```

Scenario6 describes a call connection request to a user who is busy.

```
process Scenario6[ USER_to_DE, DE_to_USER, scenario6 ]: noexit:=
  USER_to_DE !caller !phone1 !offhook;
  DE_to_USER !phone1 !caller !dialtone;
  USER_to_DE !caller !phone1 !dial !termNum;
  DE_to_USER !phone1 !caller !digitreceived;
  getCE !phone1 !originating;
  DE_to_CE !phone1 !originating !0 of instance !connectreq !termnum;
  findLE !originating !ottawa;
  CE_to_LE !originating !0 of instance !ottawa !0 of instance !connectreq !termnum;
  findCalleeAddress;
  createCall !0 of instance;
  LE_to_CO !ottawa !0 of instance !0 of instance !connectreq !termnum;
  find2ndPartyLE !termnum !montreal;
  CO_to_LE !0 of instance !montreal !0 of instance !connectind !termnum;
  findCE !termnum !terminating;
  LE_to_CE !montreal !0 of instance !terminating !0 of instance !connectind;
  findDE !terminating !phone2;
  CE_to_DE !terminating !0 of instance !phone2 !connectind;
  DE_to_CE !phone2 !terminating !0 of instance !busytoneon;
  CE_to_LE !terminating !0 of instance !montreal !0 of instance !busytoneon;
  le_to_co !montreal !0 of instance !0 of instance !busytoneon;
  CO_to_LE !0 of instance !ottawa !0 of instance !busytoneon;
  LE_to_CE !ottawa !0 of instance !originating !0 of instance !busytoneon;
  CE_to_DE !originating !0 of instance !phone1 !busytoneon;
  DE_to_USER !phone1 !caller !busyTone;
  USER_to_DE !caller !phone1 !onHook;
  scenario6; stop
endproc
```

Scenario7 describes a call connection request to a user who is not responding.

```
process Scenario7[ USER_to_DE, DE_to_USER, scenario7 ]: noexit:=
  USER_to_DE !caller !phone1 !offhook;
  DE_to_USER !phone1 !caller !dialtone;
  USER_to_DE !caller !phone1 !dial !termNum;
  DE_to_USER !phone1 !caller !digitreceived;
  getCE !phone1 !originating;
  DE_to_CE !phone1 !originating !0 of instance !connectreq !termnum;
  findLE !originating !ottawa;
  CE_to_LE !originating !0 of instance !ottawa !0 of instance !connectreq !termnum;
  findCalleeAddress;
  createCall !0 of instance;
  LE_to_CO !ottawa !0 of instance !0 of instance !connectreq !termnum;
  find2ndPartyLE !termnum !montreal;
  CO_to_LE !0 of instance !montreal !0 of instance !connectind !termnum;
  findCE !termnum !terminating;
  LE_to_CE !montreal !0 of instance !terminating !0 of instance !connectind;
  findDE !terminating !phone2;
  CE_to_DE !terminating !0 of instance !phone2 !connectind;
  ring;
  DE_to_USER !phone2 !callee !ringingon;
  DE_to_CE !phone2 !terminating !0 of instance !connectresp;
```

```

CE_to_LE !terminating !0 of instance !montreal !0 of instance !connectresp;
LE_to_CO !montreal !0 of instance !0 of instance !connectresp;
CO_to_LE !0 of instance !ottawa !0 of instance !connectconf;
LE_to_CE !ottawa !0 of instance !originating !0 of instance !connectconf;
CE_to_DE !originating !0 of instance !phone1 !connectconf;
DE_to_USER !phone1 !caller !ringbackon;
timeout;
DE_to_USER !phone1 !caller !busyTone;
USER_to_DE !caller !phone1 !onhook;
scenario7; stop
endproc

```

Scenario8 describes a call between two users. The caller hangs up first.

```

process Scenario8[ USER_to_DE, DE_to_USER, scenario8 ]: noexit:=
  USER_to_DE !caller !phone1 !offhook;
  DE_to_USER !phone1 !caller !dialtone;
  USER_to_DE !caller !phone1 !dial !termNum;
  DE_to_USER !phone1 !caller !digitreceived;
  getCE !phone1 !originating;
  DE_to_CE !phone1 !originating !0 of instance !connectreq !termnum;
  findLE !originating !ottawa;
  CE_to_LE !originating !0 of instance !ottawa !0 of instance !connectreq !termnum;
  findCalleeAddress;
  createCall !0 of instance;
  LE_to_CO !ottawa !0 of instance !0 of instance !connectreq !termnum;
  find2ndPartyLE !termnum !montreal;
  CO_to_LE !0 of instance !montreal !0 of instance !connectind !termnum;
  findCE !termnum !terminating;
  LE_to_CE !montreal !0 of instance !terminating !0 of instance !connectind;
  findDE !terminating !phone2;
  CE_to_DE !terminating !0 of instance !phone2 !connectind;
  ring;
  DE_to_USER !phone2 !callee !ringingon;
  DE_to_CE !phone2 !terminating !0 of instance !connectresp;
  CE_to_LE !terminating !0 of instance !montreal !0 of instance !connectresp;
  LE_to_CO !montreal !0 of instance !0 of instance !connectresp;
  CO_to_LE !0 of instance !ottawa !0 of instance !connectconf;
  LE_to_CE !ottawa !0 of instance !originating !0 of instance !connectconf;
  CE_to_DE !originating !0 of instance !phone1 !connectconf;
  DE_to_USER !phone1 !caller !ringbackon;
  USER_to_DE !callee !phone2 !offhook;
  processterminatoranswer;
  DE_to_CE !phone2 !terminating !0 of instance !answerreq;
  CE_to_LE !terminating !0 of instance !montreal !0 of instance !answerreq;
  LE_to_CO !montreal !0 of instance !0 of instance !answerreq;
  connectparties;
  CO_to_LE !0 of instance !ottawa !0 of instance !answerind;
  LE_to_CE !ottawa !0 of instance !originating !0 of instance !answerind;
  CE_to_DE !originating !0 of instance !phone1 !answerind;
  USER_to_DE !caller !phone1 !onhook;
  handleonhook;
  DE_to_CE !phone1 !originating !0 of instance !disconnectreq;
  CE_to_LE !originating !0 of instance !ottawa !0 of instance !disconnectreq;

```

```

LE_to_CO !ottawa !0 of instance !0 of instance !disconnectreq;
CO_to_LE !0 of instance !montreal !0 of instance !disconnectind;
handlereleasecall;
LE_to_CE !0 of instance !terminating !0 of instance !disconnectind;
CE_to_DE !terminating !0 of instance !phone2 !disconnectind;
de_to_user !phone2 !callee !dialTone;
user_to_de !callee !phone2 !onhook;
scenario8; stop
endproc

```

Scenario9 describes a call between two users. The callee hangs up first.

```

process Scenario9[ USER_to_DE, DE_to_USER, scenario9 ]: noexit:=
  USER_to_DE !caller !phone1 !offhook;
  DE_to_USER !phone1 !caller !dialtone;
  USER_to_DE !caller !phone1 !dial !termNum;
  DE_to_USER !phone1 !caller !digitreceived;
  getCE !phone1 !originating;
  DE_to_CE !phone1 !originating !0 of instance !connectreq !termnum;
  findLE !originating !ottawa;
  CE_to_LE !originating !0 of instance !ottawa !0 of instance !connectreq !termnum;
  findCalleeAddress;
  createCall !0 of instance;
  LE_to_CO !ottawa !0 of instance !0 of instance !connectreq !termnum;
  find2ndPartyLE !termnum !montreal;
  CO_to_LE !0 of instance !montreal !0 of instance !connectind !termnum;
  findCE !termnum !terminating;
  LE_to_CE !montreal !0 of instance !terminating !0 of instance !connectind;
  findDE !terminating !phone2;
  CE_to_DE !terminating !0 of instance !phone2 !connectind;
  ring;
  DE_to_USER !phone2 !callee !ringingon;
  DE_to_CE !phone2 !terminating !0 of instance !connectresp;
  CE_to_LE !terminating !0 of instance !montreal !0 of instance !connectresp;
  LE_to_CO !montreal !0 of instance !0 of instance !connectresp;
  CO_to_LE !0 of instance !ottawa !0 of instance !connectconf;
  LE_to_CE !ottawa !0 of instance !originating !0 of instance !connectconf;
  CE_to_DE !originating !0 of instance !phone1 !connectconf;
  DE_to_USER !phone1 !caller !ringbackon;
  USER_to_DE !callee !phone2 !offhook;
  processterminatoranswer;
  DE_to_CE !phone2 !terminating !0 of instance !answerreq;
  CE_to_LE !terminating !0 of instance !montreal !0 of instance !answerreq;
  LE_to_CO !montreal !0 of instance !0 of instance !answerreq;
  connectparties;
  CO_to_LE !0 of instance !ottawa !0 of instance !answerind;
  LE_to_CE !ottawa !0 of instance !originating !0 of instance !answerind;
  CE_to_DE !originating !0 of instance !phone1 !answerind;
  USER_to_DE !callee !phone2 !onhook;
  handleonhook;
  DE_to_CE !phone2 !terminating !0 of instance !disconnectreq;
  CE_to_LE !terminating !0 of instance !montreal !0 of instance !disconnectreq;
  LE_to_CO !montreal !0 of instance !0 of instance !disconnectreq;
  CO_to_LE !0 of instance !ottawa !0 of instance !disconnectind;

```

```

    handlereleasecall;
    le_to_ce !montreal !0 of instance !terminating !0 of instance !disconnectind;
    ce_to_de !terminating !0 of instance !phone1 !disconnectind;
    de_to_user !phone1 !caller !dialTone;
    user_to_de !caller !phone1 !onhook;
    scenario9; stop
endproc

```

A.4 TTCN test case example generated with TGV

From the LOTOS scenarios automatically generated using Ucm2LotosTests, we generated TTCN test suites using TGV and aut2ttn. The TTCN test case generated from Scenario 1 is shown below.

Test Case Dynamic Behaviour					
Test Case Name		: scenario1			
Group		:			
Purpose		:			
Default		:			
Comments		:			
Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		user1 !offHook	scenario1_001		
2		user1 ?dialTone	scenario1_002		
3		user1 !dial	scenario1_003		
4		user1 ?digitReceived	scenario1_004		
5		getCE !noceb	scenario1_005		
6		user1 ?busyTone	scenario1_006		
7		user1 !onHook	scenario1_007	PASS	
8		user1 !onHook	scenario1_007	PASS	
9		user1 !onHook	scenario1_007	PASS	
10		user1 !onHook	scenario1_007	PASS	
11		user1 !onHook	scenario1_007	PASS	

Bibliography

- [AAL99] D. Amyot, R. Andrade, L. Logrippo, J. Sincennes, and Z. Yi. *Formal Methods for Mobility Standards*. IEEE 1999 Emerging Technology Symposium on Wireless Communications & Systems, Dallas (TX), USA, April 1999. Editor: Traci King, Samsung Telecommunications America. Publisher: Steve Bootman, Hitachi Telecom.
- [ACG00] 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*. Proceedings of the Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems, Glasgow, May 2000.
- [AHL98] D. Amyot, N. Hart, L. Logrippo, and P. Forhan. *Formal Specification and Validation using a Scenario-Based Approach: The GPRS Group-Call Example*. Object-Time Workshop on Research in OO Real-Time Modeling, Ottawa, Canada, January 1998.
- [ALB99] D. Amyot, L. Logrippo, R.J.A. Buhr, and T. Gray. *Use Case Maps for the Capture and Validation of Distributed Systems Requirements*. Fourth International Symposium on Requirements Engineering, Limerick, Ireland, June 1999.
- [AL00] D. Amyot, L. Logrippo. *Structural Coverage for LOTOS* IFIP TC6/WG6.1 13th International Conference on Testing of Communicating Systems (TestCom 2000), Ottawa, August 2000.
- [AmL00] D. Amyot, L. Logrippo. *Use Case Maps and LOTOS for the Prototyping and Validation of a Mobile Group Call System* Computer Communications 23(8), 2000.
- [Amy94] D. Amyot. *Formalization of Timethreads Using LOTOS*. M.Sc. Thesis, University of Ottawa, 1994.
- [Amy01] D. Amyot. *Specification and Validation of Telecommunications Systems with Use Case Maps and LOTOS*. PhD. Thesis, University of Ottawa, 2001.
- [And00] R. Andrade. *Applying Use Case Maps and Formal Methods to the Development of Wireless Mobile ATM Networks*. The Fifth NASA Langley Formal Methods Workshop, Williamsburg, Virginia, USA, June 2000.

- [Att99] AT&T, 1999. *Graphviz - Graph Drawing Software*.
<http://www.research.att.com/sw/tools/graphviz/>
- [BB87] T. Bolognesi, E. Brinksma. *Introduction to the ISO Specification Language LOTOS*. Computer Networks and ISDN systems 14, pages 25-59, 1987.
- [BFV99] A. Belifante, J. Feenstra, R. G. de Vries, J. Tretmans. *Formal Test Automation: A Simple Experiment*. IFIP TC6 12th International Workshop on Testing of Communicating Systems, Budapest, 1999.
- [Boe81] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.
- [Bri88] E. Brinksma. *A Theory for the Derivation of Tests*. In P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 235-247. Elsevier Science Publishers B. V., 1989.
- [Bur96] R. J. A. Buhr and R. S. Casselman. *Use Case Maps for Object-Oriented Systems*. Prentice-Hall, 1996.
- [Bur98] R. J. A. Buhr. *Use Case Maps as Architectural Entities for Complex Systems*. IEEE Transactions on Software Engineering, Special Issues on Scenario Management. Vol. 24, No. 12, pages 1131-1155, 1998.
- [CaT95] R. H. Carver and K. C. Tai. *Test Sequence Generation from Formal Specifications of Distributed Programs*. 15th International Conference on Distributed Computing Systems, pages 360-367, May 1995.
- [EM85] B. Ehrig, B. Mahr. *Fundamentals of Algebraic Specifications*. Springer-Verlag, 1985.
- [Fac95] M. Faci. *Detecting Feature Interactions in Telecommunications Systems Designs*. PhD. Thesis, University of Ottawa, 1999.
- [FGM92] J-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodríguez, and J. Sifakis. *A Toolbox for the Verification of LOTOS Programs*. Proceedings of the 14th International Conference on Software Engineering ICSE'92 (Melbourne, Australia), pages 246-259, 1992.
- [FJJ96] J-C. Fernandez, C. Jard, T. Jeron and C. Viho. *Using On-the-fly Verification Techniques for the Generation of Test Suites* CAV'96, Conference on Computer Aided Verification, New Jersey, 1996.
- [FL94] M. Faci, L. Logrippo. In: L.G. Bouma and H. Velthuisen. *Specifying Features and Analyzing their Interactions in a LOTOS Environment*. (eds.) Feature Interactions in Telecommunications Systems. IOS Press, 1994 (Proc. of the 2nd International Workshop on Feature Interactions in Telecommunications Systems, Amsterdam) pages 136-151.

- [FLS90] M. Faci, L. Logrippo, B. Stepien. *Formal Specification of Telephone Systems in LOTOS*. In E. Brinksma, G. Scollo, and C. A. Vissers, editors, Protocol Specification, Testing and Validation, IX, pages 25-34, Elsevier Science Publishers B. V., 1990.
- [FLS91] M. Faci, L. Logrippo, B. Stepien. *Formal Specification of Telephone Systems in LOTOS: The Constraint-Oriented Approach*. Computer Networks and ISDN Systems 21 (1991) 53-67.
- [Gri92] B. Ghribi. *A Model Checker for LOTOS* M.Sc. Thesis, University of Ottawa, 1992.
- [GWW00] J. Grabowski, A. Wiles, C. Willcock, D. Hogrefe. *On the design of the new testing language TTCN-3*. IFIP TC6/WG6.1 13th International Conference on Testing of Communicating Systems (TestCom 2000), Ottawa, August 2000.
- [HU79] J. E. Hopcroft, J. D. Ullman. *Introduction to Automata Theory, Language and Computation*. Addison-Wesley. 1979.
- [Hum90] W. S. Humphrey. *Managing Software Process*. Addison-Wesley. 1990.
- [ISO89] ISO 8807 International Standard: Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Organization for Standardization, Geneva, 1989.
- [ISO92] ISO/IEC. Open Systems Interconnection - Conformance Testing Methodology and Framework - part 3 - *The Tree and Tabular Combined Notation (TTCN)*. ISO/IEC IS 9646-3, Geneva, 1992.
- [ITU96-1] ITU-T recommendation Z. 100. *Specification and Description Language (SDL)*. ITU, Geneva, 1996.
- [ITU96-2] ITU-T recommendation Z. 120. *Message Sequence Chart (MSC)*. ITU, Geneva, 1996.
- [Kam96] J. Kamoun. *Formal Specification and Feature Interaction Detection in the Intelligent Network*. M.Sc. Thesis, University of Ottawa, 1996.
- [KVZ99] H. Kahlouche, C. Viho, M. Zendri. *Hardware testing using a communication protocol conformance testing tool*. Proceedings of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Amsterdam, the Netherlands, March 1999.
- [KW91] J. Kroon, A. Wiles. *A Tutorial on TTCN*. Proceedings of the 11th International IFIP WG 6.1 Symposium on Protocol, specification, Testing and Verification, 1991.
- [LFH92] L. Logrippo, M. Faci, M. Haj-Hussein. *An Introduction to LOTOS: Learning by Examples*. Computer Networks & ISDN Systems, Vol.23, No. 5, pages 325-342, 1992.

- [MaP01] N. Mansourov, R. L. Probert. *Improving time-to-market using SDL tools and techniques*. Computer Networks, 35, pages 667-691, 2001.
- [McC76] T. McCabe. *A Software Complexity Measure*. IEEE Trans. Software Engineering, vol. 2, December 1976.
- [Mig98] 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/>
- [Mye79] G. J. Myers. *The art of software testing*. John Wiley & Sons. 1979.
- [Pre97] R.S. Pressman. *Software Engineering, A Practitioner's Approach*. McGraw Hill. 1997.
- [PrW99] R. L. Probert, A. W. Williams. *Fast Functional Test Generation using an SDL model*. Proceedings of the 12th annual International Workshop on the Testing of Communicating Systems (IWTCS '99), Budapest Hungary, September 1999, pp. 299-315.
- [PUW00] R. L. Probert, H. Ural, A. W. Williams. *Rapid generation of functional tests using MSCs, SDL and TTCN*. Computer Communications, 24, pages 374-393, 2001.
- [QPF88] J. Quemada, S. Pavon, A. Fernandez. *Transforming LOTOS specifications with LOLA: The Parametrized Expansion*. In: K. J. Turner (Ed), Formal Description Techniques, I, IFIP/North Holland, pages 45-54, 1988.
- [SL95] B. Stepien and L. Logrippo. *Feature interaction detection by using backward reasoning with LOTOS*. S.T. Vuong and S.T. Chanson. Protocol Specification, Testing and Verification XIV. Chapman & Hall, pages 71-86, 1995.
- [Ste00] B. Stepien. *Lotos2Msc Converter - User's Manual*. University of Ottawa LOTOS group, January 2000.
- [StL95] B. Stepien and L. Logrippo. *Representing and Verifying Intentions in Telephony Features Using Abstract Data Types*. IFW'95, Kyoto, Japan.
- [Toc89] A. J. Tocher. *LOTOS and the Formal Specification of Communication Standards: An Example*. Formal Methods: Theory and Practice, Chapter 2, edited by P. N. Scharbach, BP Research, 1989.
- [Tuo96] R. Tuok. *Modeling and Derivation of Scenarios for a Mobile Telephony System in LOTOS*. M.Sc. Thesis, University of Ottawa, 1996.
- [Tur98] K. J. Turner. *Validating Architectural Feature Description using LOTOS*. Fifth International Workshop on Feature Interactions in Telecommunications Software systems, IOS Press.

- [VSS91] C. A. Vissers, G. Scollo, M. v. Sinderen, and E. Brinskma. *Specification Styles in Distributed Systems Design and Verification*. Theoretical Computer Science, 89: 179-206, 1991.