THE DEPLOYMENT OF FEATURES IN DFC/ECLIPSE

By

Yuxiao Wang

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

in Information and Systems Science

Department of Systems and Computer Engineering

Carleton University

Ottawa, Canada, K1S 5B6

December 2002

©2002, Yuxiao Wang

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-79822-4

Canada

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis

THE DEPLOYMENT OF FEATURES IN DFC/ECLIPSE

submitted by

Yuxiao Wang

in partial fulfillment of the requirements for
the degree of Master of Science in Information and Systems Science

Thesis Supervisor

Chair, Department of Systems and Computer Engineering

Carleton University

December 2002

ABSTRACT

Distributed Feature Composition (DFC) is a new architecture for the description of telecommunication features. It was designed for modularizing features, analyzing feature interactions, and separating services and transmission layers. ECLIPSE implements DFC on an IP platform. It provides a framework for developing and deploying telecommunication features rapidly. However, currently all features in the ECLIPSE network run on the router inside the network. This feature deployment scheme wastes the ample processing power and the large storage capacity of intelligent end systems. It may also cause the security problem. This thesis will explore several feature deployment schemes in the ECLIPSE network and propose a flexible feature scheme which will allow users to select a best location for his features. A modified ECLIPSE architecture is proposed with the introduction of a line interface controller. In the new architecture, according to the properties of individual features, some features can be deployed inside the network, some features can be deployed in end systems, and some features are better to be deployed in both places, with the network part of the feature as the backup to the end system part for reliability reasons (split features). End systems can dynamically load features from the Web server, or they can download features from third parties. The modified architecture is also compatible with the original version.

Based on the modified ECLIPSE architecture, a split Voice Mail (VM) feature is designed and implemented. The VM feature will take callers' messages when the end user is offline, online but busy, or online but does not answer the call. It will also allow the end user to retrieve saved messages and record customized greeting messages.

ACKNOWLEDGEMENTS

I would like to gratefully acknowledge the enthusiastic supervision of Professor Bernard Pagurek, who has been a constant source of support and inspiration during the research of this thesis. I would also like to give my thanks to all my colleagues in the Network Management Lab for their help and support.

I would like to thank AT&T for providing this project and financial aid.

Finally, I am forever indebted to my parents and my husband, Xiaofei, for their understanding, patience and encouragement when it was most required. I am also grateful for my son, Danny, for his endless love and support during the course of my studies.

ACRONYMS

3WC	Three Way Calling
CFB	Call Forwarding on Busy
CW	Call Waiting
DFC	Distributed Feature Composition
DTMF	Dual-Tone Multi-Frequency
EBI	Emergency Break-In
ECLIPSE	Extended Communications Layered on IP - Synthesis Environment
FB	Feature Box
FSM	Finite State Machine
GUI	Graphical User Interface
ICS	Incoming Call Screening
IETF	Internet Engineering Task Force
IP	Internet Protocol
ITU-T	International Telecommunication Union - Telecommunication Standardization Sector
LI	Line Interface
LIC	Line Interface Controller
MB	Media Box
MGCP	Media Gateway Control Protocol
PSTN	Public Switched Telephone Network
RI	Resource Interface

RMI	Remote Method Invocation
SIP	Session Initiation Protocol
SPT	Signal Path Termination
TI	Trunk Interface
UML	Unified Modeling Language
VM	Voice Mail

TABLE OF CONTENTS

ABSTRACT.....	I
ACKNOWLEDGEMENTS.....	II
ACRONYMS.....	III
TABLE OF CONTENTS.....	V
LIST OF FIGURES.....	IX
CHAPTER 1 INTRODUCTION	1
1.1 BACKGROUND.....	1
1.2 MOTIVATION AND OBJECTIVES	3
1.3 THESIS CONTRIBUTION	4
1.4 THESIS ORGANIZATION	5
CHAPTER 2 DISTRIBUTED FEATURE COMPOSITION	6
2.1 OVERVIEW	6
2.2 DFC COMPONENTS.....	7
2.3 CUSTOMER CALL CONSTRUCTION.....	10
2.4 ROUTING.....	12
2.5 MEDIA.....	14
2.6 SIGNALS/MESSAGES.....	15
2.6.1 MESSAGES IN THE CALL PROTOCOL	15
2.6.2 MESSAGES IN THE MEDIA CHANNEL PROTOCOL	16
2.6.3 STATUS MESSAGES	17
CHAPTER 3 THE ECLIPSE PROJECT	19
3.1 OVERVIEW	19

3.2	BOXES AND PORTS	21
3.3	ADMINISTRATION.....	22
3.4	ROUTING.....	24
3.5	DATA AND PROVISIONING.....	26
3.6	MEDIA IMPLEMENTATION	27
3.6.1	GENERAL MEDIA REPRESENTATIONS IN DFC	27
3.6.2	MBOXES	29
3.6.3	MBOX MODELS	31
3.6.4	MBOX MODEL COMMANDS.....	32
3.6.5	SIGNAL PATH TERMINATIONS	33
3.7	BOX PROGRAMMING	34
3.8	A BASIC USAGE SET-UP AND TEARDOWN.....	40
3.9	LIMITATIONS OF THE ECLIPSE PROJECT.....	43
 CHAPTER 4 IMPROVEMENTS AND EXTENSIONS OF THE ECLIPSE		
	ARCHITECTURE.....	46
4.1	INTRODUCTION.....	46
4.2	ROUTER EXTENSIONS.....	47
4.3	LINE INTERFACE CONTROLLER	51
4.4	MODIFIED ECLIPSE ARCHITECTURE	54
4.5	ROUTING ALGORITHM EXTENSIONS.....	58
4.5.1	MODIFICATIONS TO THE ROUTERNode CLASS	59
4.5.2	MODIFICATIONS TO THE ROUTINGALGORITHM CLASS	65
4.5.3	AN EXAMPLE FOR THE MODIFIED ROUTING ALGORITHM.....	72
4.6	COMPARISON WITH THE ORIGINAL ARCHITECTURE AND OTHER MODIFIED ARCHITECTURE.....	80

4.6.1	COMPARISON WITH THE ORIGINAL ARCHITECTURE	80
4.6.2	COMPARISON WITH THE OTHER MODIFIED ARCHITECTURE	84
4.7	ERROR FEATURE BOXES	85
4.8	THE TIMEOUT FUNCTION IN FINITE STATE MACHINES	86
CHAPTER 5 DESIGNING THE ARCHITECTURE OF THE VOICE MAIL		
	FEATURE	88
5.1	OVERVIEW	88
5.2	THE ARCHITECTURE OF THE VOICE MAIL FEATURE	90
5.2.1	RECEIVEVMBOX AND RECEIVEVMBOXCONTROLLERVERSION	92
5.2.2	RETRIEVEVMBOX	94
5.2.3	RECORDGREETINGVMBOX	95
5.2.4	THE VOICE MAIL SERVER AND THE LOCAL VOICE MAIL DATABASE	96
5.3	THE BEHAVIORS OF THE VOICE MAIL FEATURE	98
5.3.1	LEAVING MESSAGES	98
5.3.1.1	LEAVING MESSAGES ON OFFLINE	98
5.3.1.2	LEAVING MESSAGES ON BUSY	103
5.3.1.3	LEAVING MESSAGES ON NO ANSWER	106
5.3.2	RETRIEVING MESSAGES	108
5.3.3	RECORDING GREETING MESSAGES	110
CHAPTER 6 IMPLEMENTATION OF THE VOICE MAIL FEATURE AND		
	TESTING	111
6.1	IMPLEMENTATION OVERVIEW	111
6.2	THE FSMs OF THE VM FEATURE	112
6.2.1	THE FSMs OF THE RECEIVEVM FEATURE BOX	112
6.2.2	THE FSMs OF THE RETRIEVEVM AND RECORDGREETING FEATURE BOXES	128

6.2.3 THE FSMs OF THE PLAYER/RECORDER BOX.....	130
6.3 COMPARISON WITH THE OTHER VOICE MAIL SYSTEM ON ECLIPSE.....	137
6.4 TESTING.....	138
6.4.1 TEST CASE ONE – THE VM FEATURE.....	139
6.4.2 TEST CASE TWO – THE VM AND CW FEATURE.....	142
CHAPTER 7 CONCLUSIONS AND FUTURE WORK	144
7.1 CONCLUSIONS.....	144
7.2 FUTURE WORK.....	146
REFERENCES.....	148
APPENDIX A BUILT-IN FSMs RELATING TO THE VM FEATURE	152
APPENDIX B A SAMPLE OF THE CODE OF A FSM.....	163
APPENDIX C TRACE FILES FOR THE TWO TEST CASES.....	165

LIST OF FIGURES

FIGURE 2.1 COMPONENTS OF THE DFC ARCHITECTURE.....	7
FIGURE 2.2 A LINEAR USAGE	11
FIGURE 2.3 A NONLINEAR USAGE	12
FIGURE 3.1 THE STRUCTURE OF THE ECLIPSE NETWORK.....	20
FIGURE 3.2 MESSAGE EXCHANGING BETWEEN PEER PORTS.....	22
FIGURE 3.3 DCHANNELS AND DLINKS IN A DFC USAGE	28
FIGURE 3.4 MODELS IN THE MBOXES.....	31
FIGURE 3.5 A MESSAGE TRANSITION.....	36
FIGURE 3.6 AN INTERNAL TRANSITION.....	37
FIGURE 3.7 NESTED FINITE STATE MACHINES.....	37
FIGURE 3.8 A NESTED HISTORY STATE	39
FIGURE 3.9 ALTERNATIVE STATE DIAGRAMS	40
FIGURE 3.10 THE SEQUENCE DIAGRAM FOR A BASIC USAGE SET-UP AND TEARDOWN	41
FIGURE 4.1 NEW ROUTER CLASS HIERARCHY	49
FIGURE 4.2 THE MODIFIED ECLIPSE ARCHITECTURE.....	55
FIGURE 4.3 A LINEAR USAGE RUNNING ON THE MODIFIED ARCHITECTURE	58
FIGURE 4.4 PSEUDOCODE FOR THE METHODS RELATED TO THE ROUTING ALGORITHM IN THE ROUTERNode CLASS.....	60
FIGURE 4.5 PSEUDOCODE FOR THE METHODS RELATED TO THE ROUTING ALGORITHM IN THE ROUTINGALGORITHM CLASS.....	66
FIGURE 4.6 AN EXAMPLE OF THE USAGE SET-UP IN THE MODIFIED ARCHITECTURE	72
FIGURE 4.7 ROUTING ERROR OCCURS WHEN THE CONTROLLER IS OFFLINE	78

FIGURE 4.8 ROUTING ERROR OCCURS WHEN THE LI BOX IN THE CONTROLLER IS CLOSED..	79
FIGURE 4.9 ROUTING ERROR OCCURS WHEN THE LI BOX IS OFFLINE.....	80
FIGURE 4.10 THE SEQUENCE DIAGRAM FOR SETTING UP AND TEARING DOWN A USAGE CONTAINING ONE FEATURE BOX IN THE ORIGINAL ECLIPSE ARCHITECTURE..	81
FIGURE 4.11 THE SEQUENCE DIAGRAM FOR SETTING UP A USAGE CONTAINING ONE FEATURE BOX IN THE MODIFIED ECLIPSE ARCHITECTURE	83
FIGURE 4.12 THE SEQUENCE DIAGRAM FOR THE ROUTING ERROR BOX	86
FIGURE 5.1 THE SNAPSHOTS OF THREE USAGES INVOLVING THE THREE FEATURE BOXES...	91
FIGURE 5.2 RECEIVEVMBox.....	93
FIGURE 5.3 THE SIGNALING LAYER AND THE MEDIA LAYER RELATED TO THE RECEIVEVMBox.....	94
FIGURE 5.4 RETRIEVEVMBox.....	95
FIGURE 5.5 RECORDGREETINGVMBox.....	96
FIGURE 5.6 THE SEQUENCE DIAGRAM FOR LEAVING MESSAGE ON OFFLINE.....	100
FIGURE 5.7 THE SEQUENCE DIAGRAM FOR LEAVING MESSAGE ON BUSY	104
FIGURE 5.8 THE SEQUENCE DIAGRAM FOR LEAVING MESSAGE ON NO ANSWER	106
FIGURE 5.9 AN ALTERNATIVE CASE FOR LEAVING MESSAGE ON NO ANSWER	107
FIGURE 5.10 THE SEQUENCE DIAGRAM FOR RETRIEVING MESSAGES	109
FIGURE 6.1 STATECHART FOR RECEIVEVMBoxFSM.....	113
FIGURE 6.2 STATECHART FOR INITIALTRANSPARENT2LINKSFSMV2.....	119
FIGURE 6.3 STATECHART FOR CALLRESOURCEFSM	121
FIGURE 6.4 STATECHART FOR SIGNALCALLFSM	122
FIGURE 6.5 STATECHART FOR MEDIACALLNoACCEPTFSM	123
FIGURE 6.6 STATECHART FOR CONNECTRESOURCEFSM.....	124
FIGURE 6.7 STATECHART FOR RECEIVEVMBoxFSMCONTROLLERVERSION	126

FIGURE 6.8 STATECHART FOR RETRIEVEVMBoxFSM.....	129
FIGURE 6.9 STATECHART FOR PLAYERBoxFSM	132
FIGURE 6.10 STATECHART FOR PLAYERLineFSM.....	133
FIGURE 6.11 STATECHART FOR PLAYERBUSYFSM.....	136
FIGURE 6.12 SNAPSHOT OF STARTING A CONNECTION BETWEEN TWO CUSTOMERS.....	140
FIGURE 6.13 SNAPSHOT OF A SECOND CALL ATTEMPT	140
FIGURE 6.14 SNAPSHOT OF PLAYING A GREETING MESSAGE.....	141
FIGURE 6.15 SNAPSHOT OF LEAVING A MESSAGE	141
FIGURE 6.16 ONE USAGE DURING TEST CASE TWO	143
FIGURE AA.1 STATECHART FOR TRANSPARENTFSM	152
FIGURE AA.2 STATECHART FOR OPEN2LINKSFSM.....	153
FIGURE AA.3 STATECHART FOR INITIALTRANSPARENT2LINKSFSM.....	154
FIGURE AA.4 STATECHART FOR TRANSPARENT2LINKSFSM.....	156
FIGURE AA.5 STATECHART FOR OPEN2LINKSONPORTSFSM	157
FIGURE AA.6 STATECHART FOR MEDIACALLFSM.....	159
FIGURE AA.7 STATECHART FOR OPENEXTCHANFSM	159
FIGURE AA.8 STATECHART FOR ANSWERFSM.....	161
FIGURE AA.9 STATECHART FOR MEDIAANSWERFSM	162

CHAPTER 1 INTRODUCTION

1.1 BACKGROUND

Internet telephony provides voice or multimedia communication services that are transported via the Internet in real-time [1]. It has experienced rapid evolution over recent years, although it still needs to overcome some technical challenges, such as no guarantee of bandwidth, packet loss, higher delay and jitter which affect the quality of voice [2]. Comparing to the public switched telephone network (PSTN), Internet telephony has some significant benefits, such as lower cost, integration with other Internet services and creation of new services which are impossible using the PSTN [3]. However, some problems which have hindered the development and innovation of the PSTN are still existing in Internet telephony and the most well-known of them is the feature interaction problem [4, 5].

In telephony networks, a feature or service refers to “an optional or incremental unit of functionality” [6,7]. In the context of this thesis, the term “feature” and “service” are used interchangeably since they are almost synonymous. Feature interactions refer to situations in which several features or services interact with each other, and the desired operation of some of the features will be affected. The feature interaction problem has been a major problem in the PSTN. It made brittle systems and large switches, difficult to maintain. It will be much worse in Internet telephony since multimedia services provided

by the Internet telephony will introduce more new feature interactions [7, 8].

Feature interactions are inevitable. On the other hand, while many feature interactions are bad and undesirable, many are good and desirable. Therefore the architectures for Internet telephony must provide an approach to analyze and manage feature interactions. Distributed Feature Composition (DFC) was proposed under these circumstances [9, 10]. DFC is a virtual architecture that describes telecommunication services in a modular way. It provides a framework for managing and analyzing feature interactions. In DFC features are independent modules and do not depend on each other. A customer call is processed by a dynamic assembly of feature components and featureless internal calls. The order in which features occur in a customer call is governed by precedence relations. Feature interactions can be analyzed and managed by applying different precedence relations and the desired precedence relation can be found and preserved. The current implementation of DFC, the Extended Communications Layered on IP - Synthesis Environment (ECLIPSE) network, is a virtual telecommunications network based on IP. It was developed by AT&T Research Labs and used to determine whether DFC can be used as the service architecture for next-generation networks [11].

Prior to the proposition of DFC, there already existed two main sets of standards for Internet telephony and they are still dominant nowadays. One is the Session Initiation Protocol (SIP) proposed by the Internet Engineering Task Force (IETF) [12]. The other is the H.323 protocol suite developed by the International Telecommunication Union - Telecommunication Standardization Sector (ITU-T) [13]. These protocols provide the

functionality for Internet telephony signaling, e.g. call set-up, management and teardown, and support advanced services [14,15,16]. Unfortunately, neither of them addresses the feature interaction problem [17].

1.2 MOTIVATION AND OBJECTIVES

In the long term, Internet telephony is motivated by the prospect of creating many services which are not possible in traditional telecommunications networks [3, 18]. With more features or services being added, the service placement (feature deployment) becomes a crucial issue because endpoints in Internet telephony can take a larger role in providing services than in the PSTN [19]. Where should services reside in Internet telephony systems? There are three possible ways: one way is to place all services in the network, just like in the PSTN; another way is to place all services in the endpoints; and the third way is to place services both in the network and in the endpoints. The motivation of this thesis is to examine the three possible ways of service placement for one feature in the DFC architecture and find a best fit for DFC.

The main objectives of this thesis are to investigate various schemes of feature deployment in the DFC architecture, to propose a flexible feature deployment for DFC which will allow users to select a best location for his features, to modify the current ECLIPSE architecture to allow features to run on endpoints, and to design and develop a new feature, the Voice Mail (VM) feature, to demonstrate and verify the new architecture.

1.3 THESIS CONTRIBUTION

This thesis proposes a new architecture which is modified from the original ECLIPSE architecture. In the new architecture, according to the properties of individual features, some features can be placed inside the network, some features can be placed in end systems, and some features are better to be placed in both places, with the network part of the feature as the backup to the end system part for reliability reasons. The last kind of features are called split features. The new architecture is compatible with the original architecture, in which all features are deployed inside the ECLIPSE network. The thesis makes a number of contributions as follows:

- Extended the original ECLIPSE router to allow two or more routers to share the same set of media boxes. By doing this, multiple routers can be run simultaneously in the ECLIPSE network. Whereas, in the original ECLIPSE network, only one router can be run due to the limitation of media implementation.
- Introduced a new component, the line interface controller, into the original ECLIPSE architecture, which will allow features to run on end systems. It also allows end users to download service blocks (feature boxes) from a third party without the need to ask the permission of the network.
- Modified the Routing Algorithm to accommodate the line interface controller.
- Explored different possibilities of feature deployment on the new architecture and proposed a flexible feature deployment for ECLIPSE.
- Designed and implemented a Voice Mail feature on the modified ECLIPSE network.

Its main purpose is to demonstrate and justify the new architecture. The core part of

the VM feature is designed as a split feature: one part is running in the end system and the other part is running inside the network. Both parts work together, providing a graceful solution for handling callers' messages. In addition, the VM feature can also allow users to retrieve saved messages and record greeting messages.

1.4 THESIS ORGANIZATION

The rest of the thesis is organized as follows. Chapter 2 gives a general introduction to the DFC architecture. Chapter 3 gives a detailed review of the ECLIPSE project. This is followed by an analysis of the limitations of the current ECLIPSE project. In Chapter 4, a new component, the line interface controller, which allows features to run on end systems, is introduced first. Next the modified ECLIPSE architecture is presented, which provides the flexibility to deploy features inside the network, in end systems, or in both places. Since the routing algorithm is needed to be extended correspondingly to accommodate the new component, the modified routing algorithm is described using pseudocode in the following section. In the last sections of Chapter 4, some other extensions to overcome the limitations of the original ECLIPSE project are described. The detailed design, implementation and testing of the VM feature, based on the new architecture, are described in Chapter 5 and 6. Finally, Chapter 7 summarizes the thesis with conclusions and provides suggestions for related future works.

CHAPTER 2 DISTRIBUTED FEATURE COMPOSITION

2.1 OVERVIEW

Distributed Feature Composition (DFC) is a virtual architecture that describes telecommunication services in a modular and analyzable way [9, 10, 20]. “It was designed for feature modularity, structured feature composition, analysis of feature interactions, and separation of services and transmission layers” [21, 22, 23]. The original version of DFC [9] that only provides voice services has been extended to accommodate mobile and multimedia services. Lots of features have been described informally and analyzed within the DFC architecture, and it shows that all services can be fit into it.

Since DFC is a component-based software architecture in which features are modules, it has more advantages over other architectures [24]. First, features can be added or modified with relative ease due to modularity. Second, since a structure is imposed on feature composition, it can be used to analyze potential feature interactions. Third, features can be protected from each other due to the independent property of feature modules.

This chapter will briefly describe the DFC architecture.

2.2 DFC COMPONENTS

The components of the DFC architecture are shown in Figure 2.1 [20]. Single rectangles represent boxes, including interface boxes and feature (*F*) boxes. There are three types of interface boxes: line interface (*LI*) boxes, trunk interface (*TI*) boxes, and resource interface (*RI*) boxes. The dots on the boxes represent ports. All global data are represented by double rectangles.

Figure 2.1 Components of the DFC architecture

Boxes are independent modules and communicate with each other only through internal DFC calls. They place and receive internal calls through their ports and the virtual network carries these internal calls. Among the boxes, interface boxes are persistent and addressable, while feature boxes are non-addressable and created only

when they are needed. Feature boxes will be destroyed after each call except bound feature boxes (see below).

Line interface boxes are connected to telecommunication devices, such as traditional telephones or PCs, through external lines. Trunk interface boxes are connected to other networks, such as PSTNs, through external trunks. Resource interface boxes are connected to media resources through external lines. In DFC media processing, such as recording, playing, mixing, monitoring, and media conversion requires both control and status events. A device that performs control-intensive media processing is referred to as a (media) resource, such as an announcement player or a message recorder [25].

Feature boxes implement feature logic. A feature box can have any number of ports, according to the requirements of its function. It has full control over all internal calls it receives or places and will operate without external assistance.

When a feature box does not need to function, it will behave transparently, which means it will not do any processing over the signals and media streams it receives. Those streams will pass transparently through the box as if the box does not exist. When the feature box is required to function, it can behave differently, e.g. absorbing signals, generating new signals, re-routing internal calls, processing media streams, etc.

Feature boxes can be fit into two categories: free feature boxes and bound feature boxes. A free feature box is anonymous, and interchangeable with others of the same type. A bound feature box is a unique and persistent module which is bound to an interface box.

The router is the core of the port-to-port virtual network. It dynamically assembles feature boxes by routing internal calls from one box to another, and finally routes internal calls to the destination addresses.

In order to accomplish its responsibilities, the router needs data on configurations, feature subscriptions, feature precedences and dialling patterns. All of the above are global data. The configuration data records the set of boxes existing in the DFC system and the addresses of the interface boxes. The subscription data records compulsory features assumed to be subscribed to by all customers, such as the Emergency Break-In feature (EBI), and optional features subscribed to by customers, such as the Voice Mail (VM) feature and the Call Forwarding on Busy (CFB) feature. The feature precedence data governs the order of features when the router assembles feature boxes. Features are related explicitly to each other only through these precedence relations. The dialling patterns describe the correct and effective dialling numbers in the system.

There is another kind of global data called operational data, which is used by feature boxes. For example, the 800 number feature box will retrieve the real routable phone numbers corresponding to the virtual 800 phone numbers from its operational data. Access to operational data is partitioned by features, customers, or both. This means that operational data can only be shared by features or customers of the same type, which guarantees feature modularity.

2.3 CUSTOMER CALL CONSTRUCTION

In the DFC context a customer call refers to a call between two customers. It is responded by a usage, which is a dynamic assembly of boxes and internal calls [20].

An internal call is a featureless connection from a DFC port on one box to a DFC port on another box. Each internal call has a set-up phase and a teardown phase. Between these two phases there always exists a two-way signaling channel and any number of media channels, each carrying a different medium [23]. During the set-up phase, the initiating port sends a setup signal to the router, and the router chooses a box and forwards the signal to it. The receiving box finds an idle port and sends a signal back to the initiating port to acknowledge the call. If there is no idle port, the call establishment fails. If any participant port sends a teardown signal to the other part to terminate the call, the call enters the teardown phase. The system will do some cleanup work.

Figure 2.2 shows a linear usage and Figure 2.3 shows a nonlinear usage. The internal call is shown as an arrow from the initiating port to the receiving port.

In Figure 2.2 (a) all feature boxes are free feature boxes. Customer b who owns a line interface box (LI_b) subscribes to a Call Forwarding on Busy (CFB) feature and the Voice Mail (VM) feature on its target zone. Both features deal with busy treatments. Upon receipt of a setup signal, the CFB box sends the router a setup signal, which is directed to customer b , and then monitors the signals coming back to it. If it receives a busy signal, it will tear down the current outgoing internal call and places a new outgoing internal call which is directed to the forwarding address c . The remaining chain of boxes is replaced

by a new chain of feature boxes which are subscribed to by customer c , as shown in the Figure 2.2 (b).

Figure 2.2 A linear usage

If customer c is busy talking with someone else and there is no idle port on its line interface box (LI_c), LI_c will send back a busy signal upstream. Suppose customer c does not subscribe to any busy treatment features, the busy signal will arrive at the VM box. The VM box will ask customer a to leave a message. Thus, from this example, we can see that, by carefully arrange the order of the VM feature and the CFB feature, we can provide a good solution to the busy treatment.

In Figure 2.3 there is a bound feature box representing the Call Waiting (CW) feature. The CW box is bound to the line interface box of customer c . At first, customer c made a successful call to customer a . The call passes transparently through CW just because c subscribes to Call Waiting. Later, when customer b attempts to call customer c , an internal call is routed to the CW box. The CW box first sends a signal back to LI_b to

notify it that a customer call has reached the target address, and then alerts LI_c with a call waiting notification. The CW box then monitors the voice channel from LI_c for the flash signal. Every time when it receives a flash signal, it will switch the voice channel between the one connecting c with a and the one connecting c with b .

Figure 2.3 A nonlinear usage

2.4 ROUTING

The main job of the DFC router is to assemble boxes and internal calls dynamically and construct a usage. It communicates with interface boxes using the setup signal. The setup signal consists of five important fields: *source*, *dialled*, *target*, *command*, and *route* [20]. There are three kinds of commands: *new*, *continue* and *update*. The route is a sequence of routing pairs. Each routing pair consists of two components: *box type* and

zone. There are three kinds of zones: *source*, *dialled*, and *target*.

In assembling feature boxes, the router uses the precedence rules to govern the order in which features can occur in a route. The precedence relation is a partial order, i.e., all boxes in the source zone precede all boxes in the dialled zone, and all boxes in the dialled zone precede all boxes in the target zone. Feature boxes are assigned a priority value. Feature boxes with higher priority are placed closer to the source of their triggering signal. For example in Figure 2.2, both *VM* and *CFB* are busy treatment features. A busy signal is emitted by the target line interface. The *CFB* box is placed near the target line interface box, so it will absorb and respond to the busy signal earlier than the *VM* box.

I shall use the example in Figure 2.2 to explain the function of the router. In Figure 2.2 (a), *LI_a* initiates a call by sending a setup signal to the router. The setup signal has a *source* field containing the address of *LI_a*, a *dialled* string containing the dialled string, and a *command* field containing *new*. The *target* field and the *route* field are empty. When the router received the setup signal, it fills the *target* field with the address of *LI_b* exacted from the dialled string. Since the command is *new*, the router will compute a new route and fill the *route* field with it. Customer *a* subscribes to one feature in its source zone, so the first routing pair is (*F1*, *source*). The dialled string matches the triggering pattern of the feature *F2* in the dialled zone, so the next routing pair is (*F2*, *dialled*). Customer *b* subscribes to two features *VM* and *CFB* in the target zone, so the last two routing pairs are (*VM*, *target*) and (*CFB*, *target*).

After the router finishes computing the route, it strips the first routing pair off the route, which is *F1*, and routes the internal call to a free copy of *F1*. As *F1* does not need

to re-direct the route, it simply copies the setup signal, changes the command to *continue* and sends the modified setup signal to the router through an outgoing port. The *continue* command instructs the router not to recompute the route, so the router simply strips another routing pair. With the chain unfolded, each box will do the same thing as *F1*. Finally, the route is empty, so the router routes the last internal call to *LI_b*.

Because customer *b* is talking to somebody else and there is no idle port on *LI_b*, *LI_b* sends a busy signal back to the *CFB* box. Upon receiving the busy signal, the *CFB* box springs into action. It tears down the current outgoing call and makes another outgoing call by sending a setup signal to the router. This time the target field contains the forwarding address *c* and the command field contains *update(target)*. The *update(target)* command instructs the router to remove the remaining target-zone route for customer *b* and replaces it with a newly computed target-zone route for customer *c*. Because customer *c* subscribes to the features *F3* and *F4*, the new usage will be like Figure 2.2 (b).

2.5 MEDIA

A DFC system can offer any number of media. Each medium is a distinct and universal form of communication [22]. The commonly used media in telecommunications industry are voice, video, text, images and audio.

The DFC architecture fully separates the signaling layer from the media layer. The

media layer can be implemented in any manner, using any protocols and representations provided that the implementation satisfies the specifications of the DFC architecture.

In DFC a call has a two-way signaling channel and any number (including none) of two-way media channels, each of which carrying a distinct medium. Media channels during a call must be opened and closed explicitly by signals from the signaling channel, because without it there would be no way to tell a call which medium it is going to use.

2.6 SIGNALS/MESSAGES

In DFC internal calls are provided by port-to-port communications. DFC ports use signals, or messages, to communicate with each other [22]. In this thesis, messages have the same meaning with signals and they can be used interchangeably. There are several sets of signals/messages defined in the DFC architecture. I shall briefly explain them in the subsections below.

2.6.1 MESSAGES IN THE CALL PROTOCOL

The primary purpose of the call protocol is for DFC boxes and routers to set up and tear down internal calls.

The set of messages in the call protocol is: $\{Setup, Upack, Upnack, Other, Teardown, Downack\}$. The message *Setup* is used to request the setup of an internal call. It is sent by

a port of one box to another box via the router. The message *Upack* indicates that a *Setup* message has been received by a box, a port has been allocated, and an internal call has been established. The message *Upnack* indicates that a *Setup* message has been received by a box, but no idle port can be allocated, and the requested internal call has been refused. The message *Teardown* is used to tear down the internal call. The message *Downack* is used to acknowledge a *Teardown* message. The message *Other* represents additional message types that can be sent as a part of a call.

One import thing needed to point out is that feature boxes should set up internal calls in a piecewise fashion. Otherwise, the feature boxes cannot do any functional signaling until the call has reached the target address, and this will make some features difficult or impossible to program. Feature boxes should also tear down internal calls in a piecewise fashion. In this way, ports are free to be reused as soon as possible.

2.6.2 MESSAGES IN THE MEDIA CHANNEL PROTOCOL

Media channel protocol is used for the signaling layer of DFC to open and close media channels explicitly. It is nested inside the call protocol. During a call, a media channel can be opened and closed any number of times, and either port of a call can initiate the opening operation.

The set of messages in the media channel protocol is: {*Open*, *OAck*, *ONack*, *Close*, *CloseAck*}. The message *Open* is used to request a new media channel. The message *OAck* is used to acknowledge a request for a media channel. The message *ONack* is used

to refuse a request for a media channel. The message *Close* is used to close a media channel. The message *CloseAck* is used to acknowledge the closing of a media channel.

Unlike a chain of internal calls, a chain of related media channels should be opened end-to-end. Because media streams are interpreted by endpoints, two endpoints of a call will only be able to communicate with each other until all media channels between them have been opened. But a chain of related media channels should be closed piecewise since there is no point to close it end-to-end.

2.6.3 STATUS MESSAGES

The status messages related to a call as a whole are: *{Unknown, Avail, Unavail}*. These status messages are used to provide end-to-end communications. They are usually generated by the interface boxes that receive communication requests, and are usually propagated back to the interface boxes that initiate the communications. The message *Unknown* indicates an invalid target address. The message *Avail* indicates that an interface box has been reached successfully. The message *Unavail* indicates that an interface box could not be reached.

The status messages related to a media channel are: *{Ready, Wait, Accept, Reject}*. The message *Ready* indicates that the interface box receives an acknowledgement (*OAck*) message after it sends an *Open* message. This means the media channel between two interface boxes is fully available. The message *Wait* indicates that the generating interface box is attempting to enable communication on the media

channel, usually by alerting a person to come to the device. The message *Accept* indicates that the end user is willing to communicate through the requested media channel. The message *Reject* indicates that the end user is unwilling or unable to communicate through the requested media channel.

CHAPTER 3 THE ECLIPSE PROJECT

3.1 OVERVIEW

The Extended Communications Layered on IP - Synthesis Environment (ECLIPSE) project was developed by AT&T Research Labs [11] and the main purpose of the project is to determine whether DFC can be used as the service architecture for future telecommunication networks [10, 20]. DFC is the foundation of ECLIPSE and ECLIPSE implements DFC on an IP platform.

In DFC signaling and media transmission are separated. In order to use bandwidth efficiently, media should be transmitted through the shortest path. Signaling is less expensive and must be reliable, so it is implemented in a different way. Therefore, the whole ECLIPSE system is divided into two major subsystems. One is called the signaling subsystem, which provides administration, routing, signaling, control, and global data. Java RMI [26] is used to implement this subsystem. The other is called the media subsystem, which provides media processing and transmission. The media paths are quite different from the signaling paths to which they correspond.

Figure 3.1 shows the structure of the ECLIPSE network. It also shows a DFC usage running in the signaling layer on top of the media layer. All the nodes should be IP hosts.

Figure 3.1 The structure of the ECLIPSE network

ECLIPSE routers are completely different from IP routers. They are just logical routers and use IP routers to transport packets in the network.

In the current ECLIPSE network, users can communicate through voice or text. The ECLIPSE network connects to its end users through line or trunk interfaces. The interfaces are built in two ways. One way is to build new software and hardware devices to use DFC signaling directly. The other way is to implement some new software to

interface between DFC signaling and the protocols used by the existing end devices, such as ITU-T's H.323, IETF's SIP, or IETF/ITU's Megaco/H.248 [27].

3.2 BOXES AND PORTS

In ECLIPSE, DFC components such as boxes and ports are supported by means of Java classes and interface hierarchies. Most DFC signaling protocols are encapsulated in these classes, ensuring that attempted misuse will result in an exception.

In ECLIPSE, there are six important kinds of ports: switch ports, box ports, caller ports, callee ports, dual ports, and external ports.

A switch port is created for each router node. A switch port has no fixed peer port and no state. Normally, it receives *Setup* messages from a box's caller port, and redirects the *Setup* messages to a box's box port.

A box port is responsible for receiving messages from the router. It is automatically created whenever a box is created. The peer of a box port is the switch port of the router.

Caller ports initiate DFC internal calls to their peer boxes. Callee ports receive DFC internal calls from their peer boxes. Dual ports can behave like either a caller port or a callee port from their initial state depending on how they are initialized. These three kinds of ports are generally called linkable ports. Namely, they have a defined link state with their peers, whereas other ports don't have.

External ports are used to exchange messages between a DFC box and the world outside an ECLIPSE network.

Each interface box has a certain number of normal DFC ports, which are called internal ports, and it must have at least one external port. Line, trunk, or resource joins the interface box at the external port. Feature boxes can only have internal ports.

Figure 3.2 shows how boxes establish connections between each other by exchanging DFC messages between peer ports. This figure only shows the initial phase of call set-ups, and the complete call lifecycle, along with how and what messages needed to be transmitted, will be presented in Section 3.8.

Figure 3.2 Message exchanging between peer ports

3.3 ADMINISTRATION

For each ECLIPSE administrative domain, there are a *Provisioning Manager*, a

Registration Manager, a *Central Data Server*, and a *Monitor Manager*. There can be any number of routers (In principle it is true. But the current version of ECLIPSE only allows one router to be running due to the limitation of media implementation. See details in Section 3.9).

The *Provisioning Manager* handles provisioning requests, such as add/delete a router, add/delete an address, add/delete a feature type, and add/delete a customer, etc. It will also update the *Central Data Server* according to the requests.

The *Central Data Server* serves as a database of provisioned data and is accessed by the *Provisioning Manager*, the *Registration Manager* and the routers.

The *Registration Manager* receives notifications when routers and remote interface boxes are up or down and informs the changes of their status to the interested observers. For dumb end devices, like traditional telephones, their LI boxes should reside in the routers. But for intelligent end devices, like PCs, their LI boxes should reside in end devices, and are referred to as remote interface boxes. Up means the router or the remote interface box is running and down means that it has stopped functioning.

When a router comes up, it will inform the *Registration Manager* of its node ID, switch port and some other data. The node ID is a unique identification for each router. The router then registers as an observer of remote interface box registration events. If a remote interface box associated with this router changes its status, the router will be informed. Afterwards, the router can route *Setup* messages to that box if necessary.

When a remote interface box comes up, it first registers as an observer of the router

associated with its address. Once the router is up, the remote interface box will receive a router's up status message, which includes a reference to the router's switch port. So the box can send *Setup* messages to the router afterwards. The box will then inform the *Registration Manager* of its up status, address and box port.

The *Monitor Manager* collects events generated by all node monitors and then propagates them to monitor clients, such as the *VizManager* which will demonstrate the visual presentation of the events.

In the ECLIPSE implementation the *Provisioning Manager* is running as a RMI server, and the *Registration Manager* and the *Central Data Server* co-exist with it. Other class instances contact with them by looking up the object associated with the name "*ProvisioningManager*", "*RegistrationManager*", or "*CentralDataServer*" in a specified rmiregistry (typically they use the same rmiregistry). They can also call the *connect()* method which returns a first-in-first-out queue into which messages can be placed for the *Provisioning Manager* or the *Registration Manager*.

3.4 ROUTING

Each ECLIPSE router runs independently and supports only a subset of all customers and requires data relating to this subset. While pure DFC is implicitly centralized and has no concept of location, the ECLIPSE implementation is explicitly distributed. All ECLIPSE routers work together, forming a distributed implementation of the DFC router.

ECLIPSE routers are responsible for computing and updating logical routes and applying a feature box by routing an internal call to it. A lazy evaluation of zones is used when computing logical routes [10]. For example, consider a situation in which a source customer whose data resides at one router node calls a target customer whose data resides at a different router node. The router at the source node only computes the source zone of feature box types from the local subscription data and leaves the target zone to be null. *Setup* messages travels through those feature boxes and dialled zone feature boxes and by the time the target zone is reached, the signaling path of the usage has extended all the way to the router node at the target node. This router node will compute the remaining target zone of feature box types from its own local subscription data.

In ECLIPSE each router is associated with a box factory which is responsible for instantiating new feature boxes as needed. The code of feature boxes is placed at a Web server. When the feature box type is first needed, it is dynamically loaded into the router using Java Classloader and stays there afterwards. The factory then generates an instance of the feature box type from the loaded code using Java Reflection.

When generating feature boxes, the factory treats free boxes and bound boxes differently. For free feature boxes, a new instance is created and then passed the relevant *Setup* message. For bound feature boxes, the factory first checks if there has already existed an instantiated bound box. If so, the relevant *Setup* message is sent to that box. If not, a new box is created and the *Setup* message is passed to it.

3.5 DATA AND PROVISIONING

DFC global data are organized as relations and are implemented as Java Hashtables.

In ECLIPSE the *Provisioning Manager* distributes data only to the routers that require it. The remaining data are currently handled by the *Central Data Server*.

Some of the routing data can be partitioned by interface address. For example, the subscription data is implemented as a hashtable and the keys are interface addresses, so each router can operate with only the subscription data for the interface addresses belonging to it. On the other hand, the configuration data and the feature precedence rules must be logically centralized.

Feature operational data are stored in the *Central Data Server* and are loaded to the router when necessary. It can be partitioned by features, customers, or both. Since feature boxes have access only to the appropriate data partitions, the system can prevent privacy violations.

To the ECLIPSE network, provisioning means configuring router nodes and managing feature operational data. There are three kinds of provisioning commands: *administrative commands* can only be used by a network administrator, e.g., add/remove a router node, *programming commands* can only be used by authorized feature programmers, e.g., add/remove a feature box type, and *customer commands* can be used by anyone with system access, e.g., subscribe/unsubscribe a feature.

3.6 MEDIA IMPLEMENTATION

In ECLIPSE a general-purpose media layer was designed and implemented [23]. Since the signaling layer is fully distributable, that is, a box program of a feature box should be able to run anywhere in the ECLIPSE network, the media layer needs to be equally distributable. Thus in ECLIPSE the media aspects of a feature box should be able to be implemented anywhere in the network and could be separated from the box program that controls them.

A media switch was developed to switch, replicate, mute, and sum media streams. A media switch implements all internal media processing capabilities of a box, including interface boxes and feature boxes. Please note that in the ECLIPSE network the only place to produce, interpret, split, merge, translate, or bridge multiple media is in an interface box or a device hiding behind an interface box.

The media implementation of ECLIPSE can support several kinds of media, such as voice, video, text, image, and audio. However, new media types can be added to the network without limitation. For each medium, the implementation is separate and independent.

3.6.1 GENERAL MEDIA REPRESENTATIONS IN DFC

In DFC an internal call always has a caller port and a callee port. Between these two ports there can be zero or any number of bi-directional media channels. Each media

channel has an identifier and two channel terminations, one at each port of the internal call. A channel termination is identified by a pair of identifier and port.

Channels can be internal or external. External channels connect media streams between an external port and a device, a resource, or another network. Channels between an interface box and a feature box, or between two feature boxes are called internal channels.

Note: In this usage, first c calls a, then b calls c.

Figure 3.3 Dchannels and Dlinks in a DFC usage

The media processing state inside a box is represented by a set of links. Each link is a unidirectional media connection between two channel terminations of the same box.

Henceforth components in the signaling layer of the DFC architecture will be

referred to as Dboxes, Dports, Dchannels, Dlinks and Dcalls in order to distinguish from components in the media layer.

Figure 3.3 shows the Dchannels and Dlinks in a DFC usage. Solid lines represent internal calls. Dashed lines represent Dchannels, labelled with a channel identifier (an integer). Dashed arrows within a box represent Dlinks.

3.6.2 MBOXES

In the media layer of the ECLIPSE network, there is a software component called an Mbox which consists of a media switch, a controller and a model. The ports on the media switch are called Mports, and the media connections between two Mports on different media switches are called Mcalls. In Figure 3.3, m and n are Mports on the Mbox MB_1 , and o , p and q are Mports on the Mbox MB_2 . The media connection between n and p is an Mcall. The implementation of external Dchannels (channel 1 in Figure 3.3) requires Mcalls too.

Each Dbox is assigned an Mbox of the required medium at the time of its creation and the assignment does not change during the lifetime of the Dbox. The corresponding Mbox implements all medium processing of the Dbox. The dot-dashed lines in Figure 3.3 show the associations between the Dboxes and the assigned Mboxes. For example, the *CW* feature box subscribed to by customer c is assigned the Mbox MB_2 when it is created.

There are no requirements about the configurations between Dboxes and Mboxes,

either in number or location. Since the implementation of Mboxes is very complicated, fewer Mboxes are better than more Mboxes, and it is better to give a Dbox an Mbox in its own network node. But that is not a limitation, and the implementers are given all degrees of freedom to implement what they think is best.

The model is a representation of the current media processing states of all the Dboxes to which this Mbox is assigned.

The controller in the Mbox creates Dchannels and Dlinks in response to the commands from Dboxes and the representations of those Dchannels and Dlinks are maintained in the model. If there is a Dchannel connecting two Dboxes with two different Mboxes, then the Dchannel should appear in the models of both Mboxes and the controllers of the two Mboxes cooperate in creating and destroying the corresponding Mcall. For example, in Figure 3.3 the Dchannel between the channel terminations $(h, 7)$ and $(e, 7)$ is corresponded by an Mcall between the Mports n and p .

It is the responsibility of the controller to control the media switch so that the media output at an Mport should be the sum of the inputs to some set of the Mports of the same Mbox. As shown in Figure 3.3, all Mports are labeled with the correct output sets. For example, currently customer c is talking with customer a , so the output of q is from o . On the other hand, since customer b is being held, the output of p is null.

3.6.3 MBOX MODELS

The model in an Mbox is represented by a dynamic graph. Since the media processing states of the Dboxes to which this Mbox is assigned are affected by the set-up and teardown of each call, the model will be updated dynamically. Figure 3.4 shows the models inside the two Mboxes of Figure 3.3. Nodes represent Dchannel terminations. Each node on the border is allocated an Mport and the interior nodes are not. Plain lines represent Dchannels and arrow lines represent Dlinks.

Figure 3.4 Models in the Mboxes

There are two kinds of internal Dchannels. One is the Dchannel connecting two Dboxes, both of which have the same Mbox, for example, the Dchannel between $(b, 3)$ and $(c, 3)$. The Dchannel appears only in the model of one Mbox, and its Dchannel terminations will not lie on the border. The other one is the Dchannel connecting two Dboxes which have different Mboxes, for example, the Dchannel between $(e, 7)$ and $(h,$

7). The Dchannels appears in the models of both Mboxes, and its Dchannel terminations lie on the border. An Mport is allocated to each border node.

External Dchannels are created at the provisioning time and are put into the Mbox by some provisioning operations.

3.6.4 MBOX MODEL COMMANDS

There are six commands that a Dbox can send to its Mbox and notify the Mbox to create corresponding representations of Dchannels and Dlinks in its model. A Dbox sends a command to its Mbox and must wait for a reply to that command before sending another one. The commands are as follows:

- `OpenLink(db:Dbox, from:Dchannel_termination, to:Dchannel_termination)`: add a Dlink between two channel terminations of a Dbox. Note that the two Dports of the Dchannel terminations must be distinct.
- `CloseLink(db:Dbox, from:Dchannel_termination, to:Dchannel_termination)`: remove a Dlink between two channel terminations of a Dbox.
- `Open2Links(db:Dbox, from:Dchannel_termination, to:Dchannel_termination)`: add two Dlinks in different directions between two Dchannel terminations.
- `Close2Link(db:Dbox, from:Dchannel_termination, to:Dchannel_termination)`: remove two Dlinks in different directions between two Dchannel terminations.
- `OpenChan(db:Dbox, mb1:Mbox, ct1:Dchannel_termination, mb2:Mbox, ct2:Dchannel_termination)`: add a Dchannel after the Dbox receives an *open* message from another Dbox in the signaling layer. Note that the two Dchannel terminations

must have distinct Dports and the same Dchannel identifier.

- `CloseChan(db:Dbox, ct:Dchannel_termination)`: remove a Dchannel after the Dbox receives a *close* message in the signaling layer.

ECLIPSE specifies two rules about which media commands Dboxes should issue and when they should issue them [23]. These rules help maintain synchronization between the signaling and media layers. They are as follows:

- “A Dbox that receives an *Open* signal must issue an *OpenChan* command, if and only if it is going to respond to the *Open* with an *OAck*. Furthermore, it must receive an *ack* from its Mbox for the *OpenChan* before responding with the *OAck*.”
- “A Dbox that receives a *Close* signal must issue a *CloseChan* command. Furthermore, it must receive an *ack* from its Mbox for the *CloseChan* before responding with the *CloseAck*.”

3.6.5 SIGNAL PATH TERMINATIONS

In ECLIPSE users do not have direct access to channel terminations, instead a signal path termination (*SPT*) is used to represent an endpoint of a signaling path between peer channel ports on adjacent boxes. A signal path termination is associated with a channel port. After a signal path termination is created for a port, a user can output messages to the signal path termination just as if it were a port itself. Users can use the signal path termination to control the media channel associated with that signal path termination. There is a one-to-one mapping relationship between channel terminations and signal path

terminations. In general, a signal path termination is just an endpoint through which signaling messages about a specific media channel can be received and sent.

3.7 BOX PROGRAMMING

In ECLIPSE the feature logic is defined using ECLIPSE Statecharts [28]. It allows a programmer to easily represent a feature as an assembly of finite state machines (FSMs). The ECLIPSE Statecharts language was developed to simplify the design, implementation and analysis of feature boxes. It was inspired by the Unified Modeling Language (UML) state diagrams [29, 30, 31, 32], and customized for DFC feature box developing. Like UML state diagrams, the ECLIPSE Statecharts language is a graphical language based on finite state machines and is suitable for describing the complicated behaviour of a system. It supports concepts such as history states and nested FSMs etc., and also supports some specific concepts of DFC, such as ports, messages, and transition priorities. But it does not support some UML concepts, such as concurrent FSMs, inter-FSM synchronization and timeout transitions.

In the ECLIPSE system, ECLIPSE Statecharts are implemented as a set of Java classes, i.e. state classes, transition classes and an interpreter class. Usually, when implementing feature boxes, box programmers use a translator, which was developed by AT&T, to translate the ECLIPSE Statecharts language into Java code. But since we did not have access to this translator, the Java code of the feature boxes implemented in this thesis was written directly from the finite state machines (See details in Chapter 6).

Since feature boxes communicate with each other only via their ports, the statecharts of the box define how the box reacts to messages it receives on its ports and what actions the box can perform in response to those messages. Actions may include sending some messages out from its ports. Message exchange between feature boxes is asynchronous. Usually the arrival of a message will cause a transition from one state to another, provided that the required guard condition evaluates to be true. Alternatively, the message can have a null effect, in which case the FSM remains in the same state.

In ECLIPSE Statecharts, the syntax for a transition label has three parts, each of which is optional: *event[guard]/action*. Events are messages input to a port. Actions usually are messages output to a port's peer port, but they can be arbitrary java code. Guards are arbitrary Boolean expressions.

ECLIPSE uses the following notations to represent events and actions:

p1?m1: message *m1* input from port *p1*

p2!m2: message *m2* output to port *p2*

Transitions have the following form:

p1?m1[guard] / p2!m2; action

which means in a state, if *m1* is received from *p1* and *guard* evaluates to be true, the transition can fire to permit *m2* to be output to *p2*, and *action* is executed. The finite state machine then enters a new state.

ECLIPSE defines two kinds of transitions: internal (eventless) transitions and message transitions. After firing a message transition, the FSM interpreter attempts to fire

as many internal transitions as possible. When there are no more internal transitions can be fired, the interpreter will wait for a message to arrive. When a message arrives, the interpreter will attempt to fire the message transition. If no message transitions and internal transitions can be fired, the interpreter will return.

Figure 3.5 A message transition

Figure 3.5 is an example of a message transition. Suppose currently the FSM is in *state1*. When *event* occurs, the *guard* condition is tested. If the condition is true, the transition to *state2* is allowed. A sequence of actions is then executed: first the exit action *action2*, then the transition action *action3*, at last the entry action to *state2* (*action4*). The FSM enters into *state2*. It should be noted that *action3* is shown as an action on the state transition from *state1* to *state2*. This is because it is executed only on that particular transition into *state2* and not on the other transitions. Also note that in each state, every combination of events and conditions only allows the FSM to enter just one state, which means that multiple targets are not permitted.

Figure 3.6 shows an example of an internal transition. Suppose currently the FSM is in *state1*. When *guard* is true, the transition to *state2* is allowed. Before entering *state2*, the transition action *action1* is executed.

Figure 3.6 An internal transition

Figure 3.7 gives an example of nested FSMs. Normally a nested state machine is recreated each time its parent state is entered. *FSM0* is the super state machine of two nested state machines: *FSM1* and *FSM2*.

Figure 3.7 Nested finite state machines

The initial state of *FSM0* is *s1*. On entry into the *s1* state, the instantaneous entry

action *a1* is executed. A nested state machine named *FSM1* is then created. *FSM1* immediately enters its initial state *s11*. On entry into the *s11* state, the entry action *a3* is executed. At any time from now on, if the *e1* event takes place and the *g1* condition is true, *FSM0* will transition from the *s1* state to the *s2* state. A sequence of actions will be executed: *a2* and *a5* if *FSM1* is in the *s11* state, or *a4*, *a2* and *a5* if *FSM1* is in the *s12* state. When *FSM0* enters into the *s2* state, it will create another nested state machine named *FSM2*.

If *FSM1* is in the *s11* state, the *e2* event occurs and *g2* is true, *FSM1* will transition into the *s12* state. From now on, if the *e3* event takes place, *FSM0* will transition from the *s1* state into the *s2* state, and *FSM2* will be created just like above. The exit actions are executed as following: first the exit action *a4* of the nested state machine is executed, and then the exit action *a2* of the super state machine is executed.

After *FSM2* is created, it will enter the initial state *s21*. When the guard condition *g4* becomes true, it will transition into the *s22* state. At this state, when the event *e4* occurs, *FSM2* is ended and *FSM0* will transition from *s2* to *s3*. Since no internal transitions and message transitions are defined for the *s3* state, *FSM0* will return.

ECLIPSE has also defined a nested history state. When the parent state is designated as a history state, the nested state machine maintains its state between invocations. For example, in Figure 3.8, if the super FSM enters *s2* for the first time via the transition *t1*, a new instance of *FSM2* is created and its initial state *s21* will be entered. If the *s2* state is exited and re-entered via the transition *t2*, the previous state of *FSM2* will be entered. If re-entering via the transition *t3*, then *s21* will be entered.

Figure 3.8 A nested history state

Feature logics are usually very complicated and it is normal for a feature box to include several nested FSMs inside a state. Therefore if we put all states and transitions in a single state diagram, it would make the diagram very cluttered and difficult to read. In this thesis I will introduce a notation to simplify statecharts and provide the readers with a better visual effect.

Figure 3.9 depicts the same statecharts as Figure 3.7, but the one big state diagram in Figure 3.7 has been decomposed into three small diagrams. In the state diagram of *FSM0* (Figure 3.9 (a)), the nested state machines *FSM1* and *FSM2* have not been extended. This would save more space as more states are added to *FSM0*. The state diagrams of *FSM1* and *FSM2* are drawn separately (Figure 3.9 (b) and (c)). Some labels of transitions have been changed a little. For example, the meaning of the transition $e3[@s12]$ is: if the nested state machine *FSM1* is in the *s12* state and when the event *e3* takes place, *FSM0* will transition from the *s1* state to the *s2* state. This notation will be used very often in Chapter 6.

Figure 3.9 Alternative state diagrams

3.8 A BASIC USAGE SET-UP AND TEARDOWN

In ECLIPSE a basic usage set-up and teardown refers to a usage between two end users, neither of which subscribes to any features. It is the simplest usage. But from it we can see how ECLIPSE defines the interface behaviours. Figure 3.10 uses the UML

sequence diagram to show a basic usage set-up and teardown.

Figure 3.10 The sequence diagram for a basic usage set-up and teardown

Note that since LI boxes are persistent from the time they are created and media links inside the boxes have already been opened during the provisioning time, they will not

need to send *OpenLink* commands to their Mboxes during call set-ups. They just need to send the *OpenChan* command to open a media channel between the two LI boxes.

The usage starts when *user a* picks up the phone off the hook and dials the number of *user b* (or enters the address of *user b* as the target address of the call in the graphical user interface). The line interface box of *user a* (LI_a) generates a new *Setup* message and sends the message to the router through its caller port. The router will compute a new route and put it into the route field of the *Setup* message. Since both *user a* and *user b* do not subscribe to any features, the route field is empty. Instructed by the empty route, the router forwards the *Setup* message to the line interface box of *user b* (LI_b). Upon receiving the *Setup* message, LI_b first allocates a callee port and sends an *Upack* message to its peer port (LI_a 's caller port) through this callee port. Next LI_b sends an *Avail* message to LI_a , indicating that the usage has successfully reached the target interface box.

Once LI_a receives the *Avail* message, it will send an *Open(voice)* message to LI_b , notifying LI_b to open a voice channel. LI_b will then send an *OpenChan* command to its Mbox. The Mbox responds to LI_b with an *OpenChanAck* message if the voice channel between two boxes is established successfully. Upon receipt of *OpenChanAck*, LI_b will send an *OAck* response back to LI_a .

LI_a sends a *Ready* message to LI_b , indicating that it has received an *OAck*. On receiving the *Ready* message, LI_b starts alerting the device of *user b* and sends a *Wait* message to LI_a . LI_a will then play a ringback tone to *user a*.

If *user b* answers his phone, LI_b will stop alerting and send LI_a an *Accept* message.

Upon receiving the *Accept* message, LI_a will stop the ringback tone. The two users can start to talk.

If the two users do not want to talk any more, either one of them can terminate the call. For example, if *user a* puts his phone back on the hook, his LI box will generate a *Teardown* message and send it to its peer box. LI_b will respond to it with a *Downack* message. LI_b will then send its Mbox a *CloseChan* command. After the Mbox closes the voice channel between the two boxes, it will send LI_b a *CloseChanAck* message. The Usage has been torn down successfully. From now on, the two interface boxes are capable of receiving another internal call.

3.9 LIMITATIONS OF THE ECLIPSE PROJECT

In the ECLIPSE network all features are deployed on routers inside the network. Providing services in the network definitely has its own benefits, such as taking advantages of the ample computational power and storage capacity of the network nodes, upgrading services easily, and continuing to offer services while endpoints are not connected or accessible, etc. But it inevitably wastes the processing power of the end systems since in Internet telephony lots of end systems have much more intelligence than traditional telephone end devices. Providing services in end devices also has the advantage that on-the-spot communication with users is much easier, while network

services can only communicate with each other via protocol messages and media contents. Furthermore, because people would like to protect their privacy and are reluctant to store their information in the network node, moving the services to the end devices is much preferable.

In many cases, features should be deployed in either end devices or the network. Because even in Internet telephony networks, there will still exist a large number of “dumb” devices, such as traditional black telephones, basic versions of most services have to be available via network nodes.

However, since end devices in Internet telephony are usually not permanently network-connected and are likely to be powered off every so often, it would be necessary for some features to be designed as a split feature: one part is running on the end device performing the normal functions, and the other part is running inside the network as a backup, performing functions when the end device is offline.

Since not just one feature is to be deployed in end devices, more features will be moved out of the network to the end devices. If the creation and assembling of these features are still done by the router inside the network, the *Setup* messages between the router and the features will travel through the network. Obviously transporting messages through the network is very expensive in terms of performance. Thus it is necessary to extend the architecture of ECLIPSE and allow a feature controller to run on the end device to manage its own features. Actually the feature controller is just a lightweight router since it only needs to manage a small set of features. It will run the LI box of the end device and route internal call from one feature box to another according to the

precedence rule. The difference between the feature controller and the ordinary router is that the feature controller only accesses the data related to its LI box and not others.

Another problem with the ECLIPSE implementation is related to the Mboxes. Every router is associated with a group of its own Mboxes, each of them dealing with a particular type of media. All boxes (interface boxes and feature boxes) belonging to one router share the same group of Mboxes. The media channel between two boxes on two routers should be implemented as a Mcall. But in the ECLIPSE project full media implementation has not been achieved yet. Mcalls are not available. Therefore it is impossible to test a case in which two LI boxes on two different routers communicate with each other via a type of media. As mentioned above, a feature controller is lightweight router and a call is definitely going to be set up between two LI boxes on two feature controllers. It is necessary to think of a way to make this possible. Chapter 4 will discuss this subject and propose an alternative way to solve this problem

A third limitation is associated with the ECLIPSE Statecharts. It does not support timeout functions. But in reality timeout is an important design strategy which is used everywhere. It should be explicitly implemented in the ECLIPSE Statecharts.

In ECLIPSE some system feature boxes such as error boxes are preloaded in the system. Some of them did not perform the necessary functions which are defined in the DFC specification. They need to be redesigned too.

CHAPTER 4 IMPROVEMENTS AND EXTENSIONS OF THE ECLIPSE ARCHITECTURE

4.1 INTRODUCTION

In this chapter, I will give a more detailed description about how to extend the original ECLIPSE architecture to get rid of the limitations which I mentioned in the last chapter. First the router is extended to allow two routers to share the same set of media boxes. Next a new component, the line interface controller (also called as the feature controller), is introduced into the architecture, which will allow features to run on the end devices. Routing algorithm is then modified to accommodate the new component.

The modified architecture is more flexible than the original architecture. In the new architecture, according to the properties of individual features, some features can be deployed inside the network, some features can be deployed in end systems, and some features are better to be deployed in both places, with the network part of the features as the backup to the end system part for reliability reasons. End users have the options to subscribe to features which are going to run in the network and/or end devices. The new architecture is also compatible with the original architecture, in which all features are deployed only inside the DFC network.

4.2 ROUTER EXTENSIONS

In the original ECLIPSE implementation, each router is associated with a set of its own media boxes, each of them dealing with a particular type of media. The media boxes are stored in a media box table in the router and are initialized at the time of the router's creation. Because in this thesis we are mainly interested in intelligent end devices, their LI boxes run on end devices. LI boxes residing on the end devices are called as remote LI boxes. Trunk interface boxes are not to be considered either, since we do not allow end devices to be the gateway connecting to other networks.

In ECLIPSE when a remote LI box comes up, it first registers with the *Registration Manager* as an observer of router registration events associated with its box address. It also registers the existence of the box itself, and provides its box port and monitor as message parameters. If the box address has been provisioned to a router and the router is up, then the *Registration Manager* will send a router status message to the box. The status message contains the references to the router's switch port and media box table. The references will be passed to the box, and from now on, all media processing of the box will be done by the media boxes in the table. Therefore all LI boxes belonging to the same router share the same set of media boxes.

Feature boxes running on the same router share the same set of media boxes too. Since in the original ECLIPSE all feature boxes are residing in routers, they are assigned the group of media boxes associated with their router when they are created by the box factory of the router.

Two LI boxes or feature boxes belonging to two routers are assigned different Mboxes, thus the media channel between them should be implemented as an Mcall, as mentioned in Section 3.6.2. But for some reasons the Mcall has not been implemented in ECLIPSE yet or it is malfunctioned. Thus although these two boxes can communicate via their signaling channel, they cannot communicate via their media channel.

To solve this problem, two or more routers can be allowed to share one set of media boxes. In this way, the media channel between two LI boxes or feature boxes on two routers is just a normal internal channel and does not require an Mcall. In ECLIPSE there is no required relationship of any kind between the configurations of boxes (LI boxes and feature boxes) and their media boxes, either in location or number. It is usually better to assign a box a media box in the box's own network node. But there are many reasons for exceptions to these heuristics. For example, in the next section, I would like to introduce a lightweight router which will be running on end devices. When the implementation of media boxes is too complicated and it is unreasonable to run them on the end devices, the lightweight router will not create its own media boxes, and will be given the reference of the media boxes associated with a primary router (normal ECLIPSE router). There is no need to run a separate set of media boxes on end devices. All the media processing related to the lightweight router will be done in the media boxes of the primary router.

In order to implement this, I would like to introduce a secondary router which will not create its own media boxes and only use the media boxes of a primary router.

In the original ECLIPSE implementation, there is a class named *RouterNode*, which performs the normal functions of an ECLIPSE router. In order to introduce the secondary

router, I split the *RouterNode* class into two classes: a super class called *RouterNode* and a subclass called *PrimaryRouterNode*. The *PrimaryRouterNode* class represents normal ECLIPSE routers and replaces the original *RouterNode* class. Wherever the original *RouterNode* is need, the *PrimaryRouterNode* will be used instead.

The new *RouterNode* class extracts most functions from the original *RouterNode* class, such as operating on the received *Setup* messages and assembling proper feature boxes, handling various kinds of messages besides the *Setup* message, and running interface boxes when they resides in this router node (for “dumb” end devices), etc. It does not create any media boxes and leaves the register function to its subclasses too. Each *PrimaryRouterNode* creates a set of media boxes at the time of its creation. The *PrimaryRouterNode* then informs the *Registration Manager* of its node ID, switch port and some other data. It also registers with the *Registration Manger* as an observer of remote interface box registration events. When a remote interface box associated with this router changes its status (up/down), the router will be informed.

Figure 4.1 New router class hierarchy

A new *SecondaryRouterNode* class is introduced to represent a secondary router. It extends the super class *RouterNode*. The new router class hierarchy is shown in Figure 4.1. The *RouterNode* class is specialized to be a *PrimaryRouterNode*, a

SecondaryRouterNode, or a *LIController*. *LIController* is a lightweight router which will be discussed more in details in the next section.

The difference between the *SecondaryRouterNode* and *PrimaryRouterNode* is that the *SecondaryRouterNode* does not create any media boxes at the time of its creation. The node ID (name) of the primary router is provided to the secondary router through a properties file. When a secondary router is up, it first registers with the *Registration Manager* as an observer of router registration events associated with a primary router. It also registers the existence of itself too, and provides its node ID, switch port and some other data as message parameters. When the primary router comes up, the *Registration Manager* will send a router status message to this secondary router. The message contains a reference to the primary router's media box table. Using this information the values in the media box table of the secondary router are initialized. From now on all media processing related to the secondary router will be done in the media boxes of the primary router. In this way two boxes on two different routers are assigned the same set of media boxes. The media channels between the two boxes are normal internal channels, which do not need the assistance of an Mcall, so they can communicate freely via these media channels.

The implementation of the primary router and secondary router is proved successful, and has been tested and used by another member of our group, Mr. Ruiguo Li, in his research [33].

4.3 LINE INTERFACE CONTROLLER

In the original ECLIPSE network all features are deployed on routers inside the network. But this feature deployment scheme wastes the strong processing and storage power of end devices. It may cause performance problem, especially when the feature number becomes very large [33]. It may also cause security problem, as feature operational data is to be stored in the network. For users who do not want their private information to be stored inside the network, it is a big issue. Therefore moving features to end devices is a better solution.

There are two methods to deploy features in end devices. First, an application program can be run on the end device which will create instances of those features subscribed to by the user. All these features are running on the end device, but are managed by the router inside the network. Second, a local router is to be run on the end device and manages the features subscribed to by that user.

There are some problems with the first approach. For example, when assembling feature boxes during a usage, *Setup* messages have to travel through the network from the router to the feature boxes and back forth. It is very expensive and will cause performance problem. Another problem with this approach is that features need to access feature operational data remotely since feature operational data is stored in the router. That is not efficient. The third problem is that it is difficult to upgrade the system. In ECLIPSE only routers are allowed to load the features from the Web server. Therefore, when new versions of features have been put into the Web server, there is no way for the application program running on the end device to load the new versions and upgrade

the corresponding features.

The second approach solves the above problems. A router can be run on the end device. This router is not necessary to be a normal ECLIPSE router. In fact it should be a lightweight router since it only needs to run the LI box of its user and manage those features subscribed to by the user. It does not care about other users.

This approach has many advantages over the first one. For example, the lightweight router can dynamically load feature types when needed, and features that used to be run inside the network can be run on end devices without change. The most important of all is that it makes the ECLIPSE network more distributed while keeping the modularity property of the DFC features.

The only problem with this feature deployment scheme is the reliability problem. Because end devices, such as PCs, may not connect to the network permanently and are likely to be powered off every so often, the features residing in the end devices will not function when the end users are offline. For example, suppose the Voice Mail feature is running on the end device. If the end user is offline, the Voice Mail feature cannot function and callers are not able to leave messages.

To solve this problem, some features, like the Voice Mail feature, should be designed as a split feature. One part runs on the end devices and performs the normal functions. The other part runs inside the network as a backup. When the end user is online, the network part of the feature behaves transparently and the end device part functions. If the end user is offline, the network part of the feature will spring up into function. As to some

other features, like the Call Waiting feature, there is no need to design it as a split feature, since it is absolutely no meaning for it to run if the end user is offline. This kind of features should be run on end devices.

The lightweight router is also called a line interface controller (LIC) and is implemented as an *LIController* class. This class extends the super class *RouterNode* (see Figure 4.1). Since the common functions have been defined in the *RouterNode* class, the *LIController* class is only responsible for registering with the *Registration Manager* and running its own LI box and related RI boxes.

Like a secondary router, the LIC is designed to not create a set of media boxes of its own. It uses the media boxes associated with a primary router. The node ID (name) of the primary router is provided to the LIC in a properties file.

When an LIC comes up, it first registers with the *Registration Manager* as an observer of router registration events associated with a primary router. It also registers the existence of itself too, and provides its node ID, switch port and some other data as message parameters. When the primary router comes up, the *Registration Manager* will send a router status message back to the LIC. The message contains a reference to the primary router's media box table. Using this information the values in the media box table of the LIC are initialized. Therefore the boxes on the LIC and the primary router share one set of media boxes. From now on all media processing related to the LIC will be done in the media boxes of the primary router.

Next the LIC will create an instance of an LI box and related RI boxes, such as

player interface boxes and recorder interface boxes. The boxes are initialized to use the specified box properties defined in a properties file. The RI boxes are private and can only be accessed by the local LI box. They provide some basic auxiliary functions to the LI box.

Introducing the LIC into the ECLIPSE architecture makes it possible for end users to download service blocks (feature boxes) from a third party without the need to ask the permission of the network. End users would like to choose service blocks from third parties because these services may be good in performance, or simply because they are free.

Usually the code for all feature boxes provided by the ECLIPSE network is placed at a Web server. When the feature box type is first needed, it is dynamically loaded into the router and stays there afterwards. In our approach of the LIC, end users can download feature boxes from a third party and store them in its local drive. When the LIC needs to load feature boxes, it will first check its local drive and try to load them from there. If it cannot find any code locally, it will load them from the specified remote Web server, just like normal ECLIPSE routers. LI and RI boxes can be loaded in this way too.

4.4 MODIFIED ECLIPSE ARCHITECTURE

As compared to Figure 3.1, Figure 4.2 shows the modified ECLIPSE architecture with a DFC usage running on it. The new architecture only adds one component, the line interface controller, to the original version. The modified architecture has several

characteristics:

Figure 4.2 The modified ECLIPSE architecture

- LI and RI boxes are allowed to run as a standalone program on end devices, without

being put in an LIC. In this case, feature boxes have to run on routers. This makes the modified architecture compatible with the original version.

- LICs are able to run in the source zone, the target zone, or both. In other words, wherever an LI box is used, an LIC can be used too. It makes the modified architecture more general and symmetric.
- When routing internal calls, the router will automatically detect if there is an LIC running on the end device. If there is one, control will be given to the LIC and let it construct the remaining usage. If there is none, the router itself will assemble the feature boxes for the end user.
- LICs are allowed to subscribe to features. In the new architecture, features running inside the network should be subscribed to by the end user through his LIC, while features running on the end device should be subscribed to by the end user through his LI box. DFC specifies that customers use addresses to subscribe to features. So, if the end user wants to use his LIC to subscribe to features, the LIC must have an address. But in DFC, only interface boxes have addresses, and routers do not have addresses. An LIC does not obtain an address automatically from the DFC specification since it is a lightweight router. Therefore in our design, each LIC will be given a unique address associated with the address of the LI box. For example, “ericController” is the name for the LIC which hosts the LI box “eric”. The system keeps the associations in its database.

I'll use the usage in Figure 4.2 to explain the above characteristics. In Figure 4.2, there is a line interface controller, called LIC_b , running on the end device of user b . $F2$ is a split feature. The network part of $F2$, $F2_N$, is running on user b 's router, while the end

device part of $F2$, $F2_E$, is running on LIC_b . The LIC_b subscribes to $F2_N$ in LIC_b 's target zone, and LI_b subscribes to $F2_E$ in LI_b 's target zone.

When the router of user b receives the *Setup* message which is originally emitted by LI_a , it will first attempt to find if there is a line interface controller running for user b . In our case, there is. Next it will compute the remaining route according to the subscription list of LIC_b . LIC_b subscribes to the feature $F2_N$, so only one pair ($F2_N$, target) appears in the route. The router then routes the internal call to $F2_N$. $F2_N$ has no initial need to control the routing, so it simply copies the *Setup* message and changes the command to *continue* and send the modified *Setup* message to the router. This time when the router receives the *Setup* message, it cannot find a feature box to route the internal call to since the route is empty. So it will transfer the control to LIC_b . When LIC_b receives the *Setup* message, it will compute the route according to the subscription data of LI_b . Since LI_b only subscribes to $F2_E$, LIC_b will route the internal call to the $F2_E$ box. Finally, in the last internal call, the route is empty so LIC_b routes to LI_b . The routing procedure will be explained more in details in Section 4.5.3, using some examples.

If both source and target LI addresses have a controller running on their end devices, the feature boxes will be executed in such an order: feature boxes subscribed to by the source LI address, feature boxes subscribed to by the source controller, dialed zone feature boxes, feature boxes subscribed to by the target controller, and finally feature boxes subscribed to by the target LI address. Hereafter the source/target address refers to the address of an LI box, as compared to the controller address referring to the address of

an LIC. Figure 4.3 shows the order of feature execution in a linear usage with the controllers running on both source and target end devices. Note that since the LICs are a kind of router, they will not be shown in a usage.

Figure 4.3 A linear usage running on the modified architecture

4.5 ROUTING ALGORITHM EXTENSIONS

The key to assembly of necessary usage configurations is the ECLIPSE routing algorithm. It operates on the *Setup* message that initiates each internal call. Each time the router receives a *Setup* message, it will call the routing algorithm and attempt to find the next applicable box (LI, RI or feature boxes) and send the modified *Setup* message to it. In this way, a usage is established piece by piece and finally reaches the callee interface box. Since in the new architecture the router needs to automatically detect the existence of an LIC and compute the route according to whether it encounters an LIC or an LI box, the routing algorithm needs to be extended and modified.

There are two classes related to the routing algorithm: the *RouterNode* Class and the *RoutingAlgorithm* class. They both need to be modified. In this section I'll use java-like

pseudocode to explain the modified routing algorithm. Two sections of pseudocode will be presented. The first section of pseudocode describes how the *RouterNode* class handles *Setup* messages, that is, how the router nodes determine which component they are going to send the *Setup* messages to: a feature box, an interface box, a line interface controller, or another router? The second section of pseudocode is related to the pure routing algorithm, that is, how the *RoutingAlgorithm* class computes a route and finds the next feature box to route the internal call. Both sections of code cooperate with each other and help the router construct the right usage.

In the next two subsections I will describe the two sections of pseudocode in general. Since the algorithm is very complex, I will use an example to illustrate how it works in the last subsection.

4.5.1 MODIFICATIONS TO THE ROUTERNODE CLASS

Figure 4.4 (Part 1 and Part 2) in the next two pages shows the first section of pseudocode. The 52 lines of pseudocode correspond to 800 lines of java source code, and describe all parts in the *RouterNode* class which are related to the routing algorithm. Code included in the dashed square is the modified part added by me.

```

1    private RouterData routerData;
2    private RoutingAlgorithm routingAlgorithm;

3    private void handleMessage(Message m) {
4        ... ..
5        if (m is an instance of the Setup message) {
6            Port w = setup(m);
7            output the Setup message to the port w via my switch port;

8            if (an error occurs while outputting the Setup message) {
9                create a routing error box;
10               output the Setup message to the box port of the routing error box
11               via my switch port;
12           }
13       }
14       ... ..
15   }

16   private Port setup(SetupMessage s) {
17       RoutingRequest nextFeatureBox = routingAlgorithm.setup(s, routerData);

18       case 1 (nextFeatureBox is not null) :
19           create the free feature box, or locate the bound feature box;
20           return the box port of the feature box;

21       case 2 (nextFeatureBox is null) :
22           get the box port of the target line interface box from the router's
23           local address-port table;

24       sub case 1 (no error occurs while getting the box port) :
25           return the box port of the line interface box;

26       sub case 2 (an error occurs while getting the box port) :
27           if (the target address has a controller) {
28               get the controller's switch port from the central data server;
29           }
30       }
31   }

```

Figure 4.4 (Part 1) Pseudocode for the methods related to the routing algorithm in the *RouterNode* class

```

25         if (the switch port is null) {
26             create a routing error box;
27             return the box port of the routing error box;
28         } else if (myself is not the controller) {
29             set the command field of the Setup message to UPDATE;
30             set the modifier field of the Setup message to TARGET;
31             return the switch port of the controller;
32         } else {
33             create a routing error box;
34             return the box port of the routing error box;
35         }
36     } else {
37         create a routing error box;
38         return the box port of the routing error box;
39     }

```

```

40     case 3 (a ChangeRouter error occurs) :
41         get the new router's switch port from the central data server
           according to the address parameter in the ChangeRouter error message;

42         if (the switch port is null) {
43             create a dialing error box;
44             return the box port of the dialing error box;
45         } else {
46             set the command field of the Setup message to CONTINUE;
47             return the switch port of the new router;
48         }

49     other cases:
50         create a routing error box;
51         return the box port of the routing error box;
52 }

```

Figure 4.4 (Part 2) Pseudocode for the methods related to the routing algorithm in the *RouterNode* class

As mentioned in Section 4.2, the *RouterNode* class defines the general attributes and operations for all router nodes. It is specialized to be a primary router node, a secondary router node or a line interface controller (lightweight router). There are many attributes defined in the *RouterNode* class, including *routerData* and *routingAlgorithm* (line 1 and

2). The *routerData* attribute stores the local router data, and the *routingAlgorithm* refers to the pure routing algorithm which will be discussed in Section 4.5.2. Among the common operations defined in the *RouterNode*, there are two methods related to the routing algorithm: the *handleMessage(Message m)* method and the *setup(SetupMessage s)* method.

The *handleMessage(Message m)* method is used by the *RouterNode* class to handle various kinds of messages, including the *Setup* message. When the router node receives a *Setup* message (line 4), it calls the *setup(SetupMessage s)* method of itself to get a proper port value (line 5), which may refer to the box port of a feature box or an interface box, or the switch port of a line interface controller or another router. Once the router node gets the port value, it will output the *Setup* Message to that port via the switch port of itself (line 6). If an error occurs while sending the *Setup* message (line 7), it will create a routing error box and output the *Setup* message to the box port of the routing error box via its switch port. The routing error box will be discussed more in Section 4.7.

The *setup(SetupMessage s)* method is used by the router node to find the next applicable component to pass the *Setup* message. Depending on various circumstances, the component could be a feature box, an interface box, a line interface controller, or another router.

At the beginning of the *setup(SetupMessage s)* method, the router node calls the *setup(SetupMessage s, RouterData routerData)* method of the *RoutingAlgorithm* class to get the information about the next feature box to be applied (line 14). The information is wrapped in an instance of the *RoutingRequest* class. If the information about the next

feature box can be found (line 15), the router node will create a new free feature box or locate the original bound feature box. The box port of the feature box is then returned.

If the information about the next feature box cannot be found (line 18), it means that the router node runs out of feature boxes residing on it. So it is time for the router node to route the internal call to an LI box or an LIC.

First the router node looks up its address-port table to check if there is a box port recorded for the target LI address (line 19), because whenever a remote LI box comes up, the router will always record the box port in its address-port table. If the box port can be found and no error occurs while getting the box port from the table (line 20), it means that the target LI box belongs to this router. So the box port of the LI box is returned

If during the procedure of looking up the box port of the target LI box, an error occurs (line 22), that is because the router cannot find the target address in its local address-port table. One possible reason is that the target LI box is currently down. But there may be another possibility: although the target address does not belong to this router, it may be included in an LIC and the LIC might belong to this router. So it is necessary to check whether there is an LIC running on the target end device. The code from line 23 to line 39 is added to accomplish this task.

First the router node will ask the *Central Data Server* to check if there is a LIC registered for the target address (line 23). If yes, it will get the value of the switch port from the *Central Data Server* (line 24). If an LIC exists but the value of the switch port is null (line 25), that is because the controller has not come up for the first time. In this case,

a routing error box is created and its box port is returned. If an LIC exists and the value of the switch port is not null, that means the controller is running on the end device at present. But it needs to make a further checkup to see if the current router node is the LIC itself, because an LIC may be a router node too. If not (line 28), it is time to transfer the control to the controller now. A sequence of operations is to be performed: the command field of the *Setup* message is set to be *UPDATE*, and the modifier field of the *Setup* message is set to be *TARGET*, which means the target route needs to be updated. Finally the switch port of the LIC is returned. If the switch port of the LIC is not null, the current router node is the controller itself, and the target address is still not able to be found out (line 32), that is probably because the LI box is closed intentionally.

If no LIC is registered for the target address (line 36) and the LI box cannot be found, the LI box must be down currently. So a routing error box is created and the box port of the routing error box is returned.

The code from line 40 to 48 deals with the case whenever it is necessary to change to another router node. When the router node calls the *setup(SetupMessage s, RouterData routerData)* method of the *RoutingAlgorithm* to ask for the information about the next feature box, the method may return a *ChangeRouter* error message. This error message tells the route node it is time to change to another router, because the source address or the target address included in the *Setup* message belongs to another router. Sometimes it needs to change to the controller's router. The current router will attempt to get the switch port of the new router from the *Central Data Server* (line 41). If the switch port is null (line 42), in this case an unknown address is encountered. Thus a dialing error box is

created and the box port of the dialing error box is returned. If the switch port is not null, the command field of the *Setup* message is changed to *CONTINUE* and the switch port of the new router is returned. The new router will continue to handle the *Setup* message.

The code from line 49 to 51 deals with the general errors which may occur during a call to the *setup(SetupMessage s, RouterData routerData)* method.

4.5.2 MODIFICATIONS TO THE ROUTINGALGORITHM CLASS

Figure 4.5 (Part 1 and Part 2) shows the second section of pseudocode. The 65 lines of pseudocode correspond to about 450 lines of java source code, and describe the methods in the *RoutingAlgorithm* class which perform the pure routing algorithm. Code included in the dashed squares is modified by me to extend the original routing algorithm. In the pseudocode below, the source/target address is the value in the source/target field of the *Setup* message. It refers to the address of the LI box of a source/target user.

As mentioned before, the *RoutingAlgorithm* class is responsible for performing the pure routing algorithm, that is, computing a route and looking for the next feature box to route the internal call. The *setup(SetupMessage s, RouterData routerData)* method of the *RoutingAlgorithm* performs this task. It operates on the *Setup* message and returns a *RoutingRequest* including the information about the next feature box to be applied or a *ChangeRouter* error to notify the caller of this method to change to another router.

```

1      Private Boolean changeToRouterOfTargetControllerFlag = false;

2      public RoutingRequest setup(SetupMessage s, RouterData routerData) {
3          fill the target field of the Setup message using the dialed string;

          // Step 1: compute the route.
4          check the command field of the Setup Message;
5          case 1 (command = NEW) :
6              compute the source route according to the source address;
7              call the computeTargetRoute method to compute the target route;

8          case 2 (command = UPDATE) :
9              check the modifier field of the Setup message;
10             sub case 1 (modifier = SOURCE) :
11                 recompute the source route according to the source address;
12             sub case 2 (modifier = TARGET) :
13                 call the computeTargetRoute method to recompute the target route;

14         case 3 (command = CONTINUE) :
15             sub case 1 (the source route is null) :
16                 compute the source route according to the source address;
17             sub case 2 (the target route is null) :
18                 if (the target address does NOT have a controller) {
19                     compute the target route according to the target address;
20                 } else {
21                     compute the target route according to the controller's address;
22                 }

          // Step 2: check the route and return the proper value to the caller of
          this method.
23         sub step 1: check the source route;
24         case 1 (the source route is null) :
25             return a ChangeRouter error with the source address as the parameter;
26         case 2 (the source route is not empty) :
27             get the first feature box off the source route;
28             return a RoutingRequest with info about the feature box;
29         case 3 (the source route is empty) :
30             if (the source address has a controller AND
31                 it is the first time to check if the source address has a controller) {
32                 sub case 1 (this router is the source address's controller and the
33                     source and target addresses do not belong to same controller) :
34                     return a ChangeRouter error with the controller's address as
35                     the parameter;

```

Figure 4.5 (Part 1) Pseudocode for the methods related to the routing algorithm in the *RoutingAlgorithm* class

```

33      sub case 2 (the controller belongs to this router) :
34          compute the source route according to the controller's address;
35          if (the source route is not empty) {
36              get the first feature box off the source route;
37              return a RoutingRequest with info about the feature box;
38          }
39      }

40  sub step 2: check the target route;
41  case 1 (the target route is null) :
42      sub case 1 (changeToRouterOfTargetControllerFlag is true) :
43          return a ChangeRouter error with the target controller's address as the
            parameter;
44      sub case 2 (changeToRouterOfTargetControllerFlag is false) :
45          return a ChangeRouter error with the target address as the parameter;
46  case 2 (the target route is not empty) :
47      get the first feature box off the target route;
48      return a RoutingRequest with info about the feature box;
49  case 3 (the target route is empty) :
50      end of step 2, doing nothing;

51  return null;
52  }

53  private computeTargetRoute(SetupMessage s, RouterData routerData) {
54      check the target address;
55      case 1 (the target address does NOT have a controller) :
56          compute the target route according to the target address;
57      case 2 (the target address has a controller AND
            the controller belongs to this router) :
58          compute the target route according to the controller's address;
59      case 3 (the target address has a controller AND
            the controller does NOT belong to this router) :
60          if (this router is the target address's controller) {
61              compute the target route according to the target address;
62          } else {
63              changeToRouterOfTargetControllerFlag = true;
64          }
65  }

```

Figure 4.5 (Part 2) Pseudocode for the methods related to the routing algorithm in the *RoutingAlgorithm* class

The actions performed by the *setup(SetupMessage s, RouterData routerData)* method are divided into two steps. In step one, it computes the route according to the value in the command field of the *Setup* message. In step two, it checks the route and returns a *RoutingRequest* or *ChangeRouter* error accordingly. Note that we do not include code about how to compute a dialing route since the original routing algorithm does not implement it and the focus of our research is not aimed at that.

At the beginning of step one, the command field of the *Setup* message is checked (line 4). Three cases would be possible: *NEW*, *UPDATE*, and *CONTINUE*.

In the first case, the command value is *NEW* (line 5). The *NEW* command tells the router to compute a new route. For the source route, it is very straightforward. The router just computes the source route according to the address of the source LI box. It does not need to consider whether the source user has a controller, since the feature boxes subscribed to by the source LI box will always be executed first. But for the target user, it needs to check whether it has a controller or not. If yes, the feature boxes subscribed to by the target controller will be executed first. If not, the feature boxes subscribed to by the target LI box will be executed. The *computeTargetRoute* method is called to compute the target route accordingly.

The functionality of the *computeTargetRoute* method is included in line 53 to 65. If the target user does not have a controller running on its end device (line 55), it will just compute the target route according to the address of the target LI box. If the target user does have a controller and the controller belongs to this router (line 57), which means that the router has the subscription data of the controller, it will compute the target route

according to the controller's address. If the target user has a controller but the controller does not belong to this router (line 59), which means that the router cannot find any the subscription data about the controller, two possibilities may occur. One is that this router itself is the controller (line 60). In this case the controller is responsible for routing the internal calls, so it is time to compute the target route subscribed to by the address of the target LI box. The other possibility is that this router is a normal router and not a controller (line 62). In this case, the *changToRouterOfTargetControllerFlag* is set to be true, which will make this program know where to change the router in the next step.

In the second case, the command value is *UPDATE* (line 8). The *UPDATE* command causes the router to remove the remnants from the current route and replace them with newly computed values. Instructed by the value contained in the modifier field of the *Setup* message, the router may update the source, dialed, or target route accordingly. The updating procedure is the same with the first case.

In the third case, the command value is *CONTINUE* (line 14). Usually when the command value is *CONTINUE*, the router is not going to recompute anything in the route. If the source or target route is null, then this router must have been changed from another router just now and the route field has not been put into any values. So the source route or target route is computed accordingly.

After step one, the source, dialed, or target route should have been computed. Now it is time to find a feature box to route the internal call to. If a feature box can be found, the information about it will be returned to the caller of this method. If not, an error message will be returned.

The source route is checked first (line 23). If the source route is null (line 24), which means no subscription data about the address of the source LI box has been found in this router, it is time to change to another router. Thus a *ChangeRouter* error message with the source address as the parameter is returned. Upon receipt the *ChangeRouter* error message, the caller of this method (*RouterNode*) will extract the address from the *ChangeRouter* message and use this address to decide which router will be changed to. If the source route is not empty (line 26), the first feature box is stripped off the source route and a *RoutingRequest* with the information about this feature box is returned. If the source route is empty (line 29), the situation will get a little complicated since it may need to check whether or not the source user has a controller running on its end device.

If the feature boxes subscribed to by the address of the source LI box have just been exhausted, it is necessary to check whether the source user has a controller or not. If not, operating on the source route is ended and it is time to check the target route (or dialed route if it was implemented). If yes (line 30), it is the right time to compute the source route subscribed to by the controller. Two sub cases will be possible.

Sub case one is when this router itself is the source LI box's controller and the source and target addresses do not belong to the same controller. In this case, the empty source route means that the features subscribed to by the source LI box have just been exhausted, so it is time to change to the controller's router and compute the source route subscribed to by the controller. Thus a *ChangeRouter* error message with the controller's address as the parameter is returned. Upon receiving this message, the caller of this method will change to the router to which the controller belongs. At the new router, the

features subscribed to by the controller will be found out. When it is the first time to get to that new router, the source route is still empty. So sub case two applies to that situation (line 33). The new router will compute the source route according to the controller's address. If it is not empty, the first feature box will be striped off the source route and then a *RoutingRequest* with the information about that feature box is returned. Note that sub case two is only applied for once, and the following operations on the source route will be performed by the normal procedure (line 23 to 29). If the source and target addresses belong to the same controller, everything will be done inside this controller, there is no need to change to the controller's router.

If finally the source route is empty again, it is time to consider the target route (should be dialed route, but it is omitted here due to the reason mentioned above). The target route is checked (line 40). Just like the source route, three cases are possible. If the target route is null (line 41), the subscription data about the target LI box cannot be found on this router. If *changToRouterOfTargetControllerFlag* is true (line 42), it is time to change to the router of the target controller. If it is false, it is time to change to the router of the target LI box. A *ChangeRouter* error message with a proper address will be returned. If the target route is not empty, the first feature box will be striped off the target route and a *RoutingRequest* with the information about the feature box is returned. If the target route is empty, all fields of route have been handled and it is time to go to an LI box or an LI controller. Thus a null value is returned, the caller of this method, the *RouterNode* class, will decide which component it will direct to (case 2 in pseudocode 1).

4.5.3 AN EXAMPLE FOR THE MODIFIED ROUTING ALGORITHM

The routing algorithm described in the last two sections is very complicated and high level. In this section, I will use an example to demonstrate how the routing algorithm works. In the descriptions below, I use P1 to represent pseudocode in Figure 4.4, P2 to represent pseudocode in Figure 4.5, L1 to represent line 1, and so forth. For example, P1:L2 will refer to line 2 of pseudocode in Figure 4.4.

Figure 4.6 An example of the usage set-up in the modified architecture

Figure 4.6 shows how a usage is constructed in the modified architecture. In this example the caller, customer *a*, is going to call customer *b*. Both the caller and callee

have LICs, LIC_a and LIC_b , running on their end devices. The two controllers belong to two different routers: *Router I* and *Router II*. Customer a subscribes to a feature $F1$ in the source zone using his LI address, and subscribes to a feature $F2$ in the source zone using his LIC address. Customer b subscribes to a feature $F4$ in the target zone using his LI address, and subscribes to a feature $F3$ in the source zone using his LIC address. All feature boxes in this example are of free feature box type.

The call set-up begins when the line interface box LI_a of customer a emits a *Setup* message (represented by 1 in Figure 4.6) to its controller. The *Setup* message has a source field containing a , a dialed field containing the dialed string b , and a command field containing *NEW*. The values in the target and route fields are null. Upon receiving the *Setup* message (P1:L4), the controller will call the setup method of its own to determine to which port it is going to route the internal call (P1:L5).

At the beginning of the setup method, the controller will call its routing algorithm to determine which feature box is going to be applied next (P1:L14). The routing algorithm will first extract the address b from the dialed string, and put it into the target field (P2:L3).

Next the routing algorithm, instructed by *NEW* (P2:L5), computes a new route and puts it into the route field. Customer a subscribes to a feature $F1$ in the source zone using its LI address, so the source route will include $F1$ (P2:L6). The routing algorithm will then call its own *computeTargetRoute* method to compute the target route (P2:L7). Since the target address b has a controller but the controller does not belong to this router, the *changToRouterOfTargetControllerFlag* variable is set to be true (P2:L63).

Now the routing algorithm of the controller has finished computing the route, and needs to find a box to route the internal call. The source route is examined first (P2:L23). Since the source route is not empty (P2:L26), the first element *FI* is stripped off the source route, and a *RoutingRequest* including the information about *FI* is returned to the controller.

Upon receiving the *RoutingRequest*, the controller will check its content. Since the *RoutingRequest* contains *FI* and is not null (P1:L15), the controller creates a fresh copy of *FI* and then sends the modified *Setup* message to *FI* (P1:L6, also 2 in Figure 4.4).

Upon receiving the message, *FI* sends an acknowledgement message *Upack* to *LI_a* (3 in Figure 4.4). The connection between *FI* and *LI_a* is established. *FI* then sets the command value of the *Setup* message to *CONTINUE* and sends it to the controller (4 in Figure 4.4).

Instructed by *CONTINUE*, the routing algorithm of the controller will do nothing about the route of the *Setup* message. (Actually, the target route is null, so the routing algorithm indeed has computed the target route. Since the target controller belongs to another router, the target route is still to be null.) The source route is empty (P2:L29), so the routing algorithm needs to check whether *LIC_a* has subscribed to any features. Now since the router itself is the controller, sub case 1 applies (P2:L31), a *ChangeRouter* error message with the controller's address as the parameter is returned to the controller by the routing algorithm. The controller will get the new router's switch port from the *Central Data Server* according to the controller's address (P1:L41). Since *LIC_a* belongs to *Router*

I, the switch port of *Router I* is found, and the *Setup* message is sent to it (P1:L6, 5 in Figure 4.4).

Upon receiving the *Setup* message, the routing algorithm of *Router I* finds that the source route is empty (P2:L29) and the controller belongs to this router (P2:L33). So it computes the source route according to the controller's address. Since LIC_a subscribes to a feature *F2*, the new source route will include *F2*. *F2* is then stripped off the route (P2:L36) and a *RoutingRequest* is returned. A new instance of *F2* is created by *Router I* and the *Setup* message is sent to it (6 in Figure 4.4). *F2* will send an *Unpack* message to *F1* (7 in Figure 4.4) and the connection between *F1* and *F2* is established.

F2 will in turn send the *Setup* message (8 in Figure 4.4) to *Router I* with the command value still being *CONTINUE*. This time when *Router I* receives the *Setup* message, it will check the source route first. The source route is empty and the features subscribed to by the controller have already been exhausted, it is time to go to the next step to check the target route.

The target route is null now and *changeToRouterOfTargetControllerFlag* is true, so a *ChangeRouter* error message with the target controller's address as the parameter is returned to *Router I* by its routing algorithm (P2:L43). *Router I* will attempt to find the switch port of the router to which LIC_b belongs (P1:L41-48). So the switch port of *Router II* is found out and the *Setup* message is sent to it (9 in Figure 4.4).

When *Router II* receives the *Setup* message, instructed by the *CONTINUE* command and the null target route (P2:L17), its routing algorithm will compute the target route

according to the address of the controller. Since LIC_b subscribes to the $F3$ feature, the target route will include $F3$. The information about $F3$ is returned to *Router II* by its routing algorithm. $F3$ is created and the *Setup* message is sent to it (10 in Figure 4.4). $F3$ will send an *Upack* message back to $F2$ (11 in Figure 4.4) and the connection between $F2$ and $F3$ is established. $F3$ has no need to redirect the call too, so it passes the *Setup* message to *Router II* (12 in Figure 4.4).

This time when *Router II* receives the *Setup* message from $F3$, both the source route and the target route are empty. Thus a *RoutingRequest* with a null value is returned to *Router II* by the routing algorithm (P2:L51). When *Router II* receives this null *RoutingRequest*, it will first attempt to get the box port of the target address b (P1:L19). Since it will not find the target address b in its local address-port table, an error will occur (P1:L22). So it needs to check if there is a controller running for the target address b . In our case, there is. So *Router II* will transfer the control to LIC_b .

Router II will get LIC_b 's switch port from the *Central Data Server* (P1:L24). Since the switch port will not be null (the controller is running) and *Router II* is certainly not a controller (P1:L28), the command field of the *Setup* message is set to *UPDATE* and the modifier field is set to *TARGET* (P1:L29-30). *Router II* sends the *Setup* message to LIC_b (13 in Figure 4.4).

When LIC_b receives the *Setup* message, instructed by the *UPDATE* command and the *TARGET* modifier, it will recompute the target route (P2:L12). The routing algorithm will compute the target route subscribed to by the target address b (P2:L61). Since

customer b subscribes to the feature $F4$ in its target zone using address b , the new target route only includes $F4$. LIC_b routes the internal call to $F4$ as normal (14 in Figure 4.4). $F4$ sends an *Upack* message and a *Setup* message accordingly (15 and 16 in Figure 4.4). This time when LIC_b receives the *Setup* message, the source route and target route are both empty. So a *RoutingRequest* with a null value is returned to LIC_b by its routing algorithm.

Because the *RoutingRequest* is null (P1:L18), LIC_b will attempt to get the box port of the LI box for the target address b from its local address-port table. Since the box port of LI_b can be found successfully (P1:L20), the *Setup* message will be sent to the box port of LI_b (17 in Figure 4.4). LI_b will send an *Upack* message to $F4$ (18 in Figure 4.4). The connection between $F4$ and LI_b is established. Since all connections between LI_a and LI_b are set up successfully, the media channel is opened (19 in Figure 4.4). Customer a and customer b can talk freely.

The above example demonstrates just one routing scenario and it is the most complicated one. The routing algorithm also accommodates the simplest situation, that is, neither the caller nor the callee has a controller running on its end device. This is the general case handled by the original routing algorithm. In this case only interface boxes reside in end devices, and all feature boxes reside in the routers.

The routing algorithm can also handle various error situations. Figure 4.7 shows what happens when an LIC cannot be found because the target user is offline. In this case, the new ECLIPSE routing algorithm routes to a routing error box. This is a kind of

system feature boxes, like a free feature box, and is preloaded in the system. It is the usual source of *Unknown* messages (will be explained in details in the next section).

After the controller has been up for some time, now the user gets offline. Although the controller is not in use, the switch port is still being recorded in the *Central Data Server*. So the router of the controller can find the value of switch port, but an error will take place in an attempt to output the *Setup* message to it since the switch port is not in use currently (P1:L7). Thus a routing error box is created by the router and the *Setup* message is sent to the box port of the routing error box.

Figure 4.7 Routing error occurs when the controller is offline

If the user's controller has never come up, a routing error box is also to be created when somebody calls the user now (P1:L25-27). The figure for this case is the same with

Figure 4.7.

There is another possibility: although the controller is running normally, the LI box is closed by the user for some reason, sometimes accidentally. If somebody calls the user now, the box port of the LI box cannot be found on the local address-port table in the controller. An error occurs and a routing error box is created (P1:L32-35), as shown in Figure 4.8.

Figure 4.8 Routing error occurs when the LI box in the controller is closed

Another possibility is that when the target address does not have a controller running in its end device, but only a LI box. In this case, when the end user is offline, the value of the box port of the LI box cannot be found in the router. Thus a routing error feature box is created too (P1:L36-39), as shown in Figure 4.9.

If for some reasons an error occurs when computing the route, a routing error box is created to handle this general case (P1:L49-51).

Figure 4.9 Routing error occurs when the LI box is offline

4.6 COMPARISON WITH THE ORIGINAL ARCHITECTURE AND OTHER MODIFIED ARCHITECTURE

4.6.1 COMPARISON WITH THE ORIGINAL ARCHITECTURE

The introduction of the line interface controller into the ECLIPSE architecture allows features to be executed on end devices and assembled and destroyed locally. Compared to the original architecture, in which all features are deployed on the routers inside the network, this feature deployment scheme reduces the workload of the network node and

saves the network's storage space. It also eliminates the security risk caused by storing personal information inside the network. Since signaling messages usually cost network bandwidth and introduce signal transmission delay, the following paragraphs compare the modified architecture with the original one in terms of the numbers of signaling messages.

Figure 4.10 The sequence diagram for setting up and tearing down a usage containing one feature box in the original ECLIPSE architecture

Figure 4.10 shows the sequence diagram for setting up and tearing down a usage in the original ECLIPSE architecture. In this case, only customer *a* subscribes to a feature box. After the feature box receives an *Open* message, it needs to send an *Openchan* command and an *Open2Links* command to its assigned Mbox. The Mbox will respond with an *OpenChanAck* message and an *Open2LinksAck* message. If the callee picks up the phone, its LI box will issue an *Accept* message. Once the LI box of the caller receives the *Accept* message, the caller and callee can talk with each other.

In the teardown phase, the boxes send the *Teardown* messages to their linkable peer ports, and issue the *Closechan* commands and the *Close2Links* commands to their Mbox, without involving the router. Only at the end of this phase, the feature box needs to send a *Destroyed* message to the router to destroy itself.

Totally thirty-five signaling messages are needed to set up and teardown a usage containing one feature box. Since in the original ECLIPSE architecture, feature boxes reside in the router, some messages, such as *Setup 2* in Figure 4.10, are transmitted locally. Considering the general case, in which LI boxes reside in end devices, feature boxes reside in the routers, and Mboxes reside in the routers, twenty-four messages of the thirty-five messages need to be transported over the network (marked by * in Figure 4.10).

Figure 4.11 shows the sequence diagram for setting up a usage in the modified ECLIPSE architecture. In this case, both customer *a* and customer *b* have LICs running on their end devices, and customer *a* subscribes to a feature box which will be executed on its end device. Comparing this diagram with the one in Figure 4.10, you will find that

only two extra *Setup* messages are added in the call set-up phase. The message sequences after *Unpack 8* are almost the same with Part 2 in Figure 4.10 since these messages do not involve the router and LICs, except that at the end of the teardown phase the feature box needs to send a *Destroyed* message to its controller to destroy itself.

Figure 4.11 The sequence diagram for setting up a usage containing one feature box in the modified ECLIPSE architecture

As we can see in the diagram, thirty-seven messages are needed to set up and teardown a call in the modified architecture. Considering the general case for the new architecture, in which LI and feature boxes reside in the controllers of end devices and Mboxes and routers reside in the network node, twenty-three messages are transported over the network (marked by # in Figure 4.10 and 4.11). Therefore, although the number of the total messages increases, the number of the messages transported over the network does not increase. Since transporting messages over the network costs network bandwidth and increases the transmission delay, reducing this kind of messages is more desirable.

As we check these twenty-three messages further, we found that many messages are transmitted between the feature box and its assigned Mbox over the network. That is

because currently feature boxes in end devices need to access Mboxes residing in the router remotely, due to the reasons mentioned in Section 3.9 and 4.2. In the future if LICs are allowed to create its own Mboxes in its local host, these messages will not need to be transmitted over the network. The number of signaling messages transported over the network will be reduced dramatically. But obviously other issues regarding the deployment of Mboxes and communications between two Mboxes will be considered too.

4.6.2 COMPARISON WITH THE OTHER MODIFIED ARCHITECTURE

In the parallel research to the DFC/ECLIPSE architecture, one of our group members, Mr. Ruiguo Li, has proposed a FDDP (Features Deployed in Dual Places - network and end device) version of the original DFC, which also aims to move feature boxes to end devices [33]. In the FDDP version of DFC, the original DFC architecture is modified with the addition of a local feature manager, a proxy of the user's line interface box and a Blind Call Transfer feature. This architecture allows features to be deployed either inside the network, at the end devices or in both places, depending on the characteristics of the features.

The local feature manager is much like our line interface controller. (Actually it was modified from the *SecondaryRouterNode* class provided by me.) But our line interface controller is more specialized. First, it creates the interface boxes of its end user and only manages those feature boxes subscribed to by this end user. Second, it not only can dynamically load features from the ECLIPSE network, but also is able to load features

from a third party. That makes our architecture more open and flexible.

In the FDDP version of DFC, the proxy of the user's line interface box is used to allow the user to subscribe to features running inside the network, using the proxy's address. In our architecture, since the line interface controller is assigned an address, there is no need to run a proxy on the router. Besides, since our routing algorithm can automatically detect the existence of a line interface controller and will transfer the control to the controller when necessary, there is no need to run a Blind Call Transfer feature on the router too. This will reduce the workload of the router when more and more end devices have line interface controllers running on them.

Another critical problem with his approach is that he does not address how to handle the case in which the local feature manager appears in the source zone of a call. Our architecture is more general since the line interface controller can run anywhere in a call, either in a source zone or a target zone (see our test case two in Section 6.4.2).

4.7 ERROR FEATURE BOXES

In the original ECLIPSE network, error feature boxes are system features and are preloaded in the system. There are three kinds of error feature boxes: routing error box, dialing error box, and trunk error box. But they were not implemented. Only a general error box was defined and its responsibility was to reject *Setup* messages, that is, sending back an *Upnack* message as soon as it received a *Setup* message. That does not comply with the DFC specifications [22]. For example, DFC defines that a routing error box is

the usual source of *Unknown* messages. After the routing error box receives a *Setup* message, it should first send back an *Upack* message, and then an *Unknown* message immediately, to tell its peer that an unknown target address is met.

In this thesis, a routing error box class, called *RoutingErrorBox*, was designed to meet the above requirements. It extends the super class *ErrorBox*, which is modified from the original version and does not send any messages. It is the responsibility of its subclass to send proper messages as needed. Figure 4.12 shows how the routing error box reacts to the *Setup* message.

Figure 4.12 The sequence diagram for the routing error box

4.8 THE TIMEOUT FUNCTION IN FINITE STATE MACHINES

In reality timeout is an import design strategy which is used everywhere, but one of the limitations of the ECLIPSE Statecharts is that they do not support the timeout

function and they do not support concurrent finite state machines either. For example, the finite state machine cannot busy wait for the timeout to be expired while monitoring the box's ports to receive any messages. Normally during a state, it will wait for the timeout to be finished, and then begins to receive messages from the box's ports. Before the timeout is expired, it cannot receive any messages. Since a timeout function does not involve receiving any message from outside, transitions related to timeout functions are classified as an internal transition. The ECLIPSE Statecharts define that the FSM interpreter will attempt to fire as many internal transitions as possible at first. Where there is no more internal transitions can be fired, the interpreter will wait for a message to arrive. It is due to this reason that the timeout function cannot be executed while waiting for new messages to arrive.

To solve this problem, in my design, the finite state machines of a box will send the *SetTimer* and *CancelTimer* messages to the box's box port. Since the box port is a separate thread, it will start the timer when it receives a *SetTimer* message and cancel the timer when it receives a *CancelTimer* message. After the timer is run out, the box port thread will send a *Timeout* message to the finite state machine that sends the *SetTimer* message. In this way, the timeout function is just like a general message transition, because the finite state machines will always monitor the box port and all other ports for message arriving.

CHAPTER 5 DESIGNING THE ARCHITECTURE OF THE VOICE MAIL FEATURE

5.1 OVERVIEW

The modified ECLIPSE architecture allows features to be deployed inside the network, in the end device, or in both places, according to the requirements for the features. The Voice Mail (VM) feature is used as an example to explore these feature deployments. The design of the VM feature is very challenging because it is not clear where this feature should be placed, and also because it interacts with other features [34].

In Internet telephony, end devices may be dumb devices, such as traditional telephones, or intelligent devices, such as PCs. Dumb devices are always connected to the network, while intelligent devices can be connected to the network all the time, or they may be powered off every so often. Therefore when we design the VM feature, we must give considerations to all the above situations. Besides, subscribers must be given the options to choose in which way they want to use their VM features.

In DFC, end users are connected to the network through line interfaces. For traditional telephones, since their line interfaces reside inside the network, the VM feature will also be placed inside the network.

Things get a little complicated with PC users. If the PC is always connected to the

network, the user can choose to place the VM feature in his end device. There are some advantages to this deployment: retrieving stored messages may be completed in the end device without the participation of the network, and the subscriber does not need to pay a monthly fee to the service provider. But at present the PCs usually are not permanently network-connected, so there is also a big disadvantage in this deployment scheme: during the period when the PC user is offline, the VM feature in the end device will not function.

Therefore in this thesis, we present a split VM feature. One part is running on the router and the other part is running on the end device. When the subscriber is offline, the part of the VM feature running on the router will do the job on behalf of the subscriber: playing greeting messages to callers, taking callers' messages and storing the messages in the network. If the subscriber is online, the part of the VM feature running on the router will behave transparently. The *Setup* message travels through it and the end device part of the VM feature, and signals the subscriber that a call is coming. If the subscriber is busy or does not answer the call for a period of time, the end device part of the VM feature will spring into function. Messages will be stored in the end device.

If the messages are stored inside the network, the VM feature and the player in the network can play back the messages to the subscriber. On the other hand, the subscriber can ask the VM feature and the player in the end device to play the messages stored both in the network and in the end device. For the latter case, messages stored in both places are merged together and are sorted according to the time stamp of each message. The subscriber can also check the messages stored in the end device without getting online. In addition, the subscriber can delete the messages if he wishes.

5.2 THE ARCHITECTURE OF THE VOICE MAIL FEATURE

In the ECLIPSE architecture, the VM feature can be implemented by three kinds of feature boxes (FBs): *ReceiveVMBox*, *RetrieveVMBox*, and *RecordGreetingVMBox*. The *ReceiveVMBox* plays back a greeting message to the caller and takes the caller's message if its subscriber is busy or does not answer the call. The *RetrieveVMBox* plays back stored messages to a caller. The *RecordGreetingVMBox* records a user's greeting message. Ms. Dongyang Zhang first proposed a VM architecture in her thesis [35], and our VM feature is based on her design.

The *ReceiveVMBox* has a network version: *ReceiveVMBoxControllerVersion*. The *ReceiveVMBox* and the *ReceiveVMBoxControllerVersion* form the split *ReceiveVM* feature. The *ReceiveVMBox* is the end device part of the *ReceiveVM* feature, while the *ReceiveVMBoxControllerVersion* is the network part. The *ReceiveVMBox* will run on the end device and the *ReceiveVMBoxControllerVersion* will run on the router. In the situation that a user does not have a controller and all of his features run on the router, the user will only need to subscribe to the *ReceiveVMBox* and this box will run on the router. Since this case is very simple, in this thesis we would focus our description on the split *ReceiveVM* feature.

The split *ReceiveVM* feature covers all cases to store callers' messages when the subscriber is offline, online but busy talking to someone else, or online but does not answer the call. When the subscriber is offline, neither the controller nor the feature boxes on the controller will be running. Thus the *ReceiveVMBoxControllerVersion* on the router will spring into function and is responsible for taking callers' messages. When the

subscriber is online, the controller and the feature boxes on the end device will be running. So the *ReceiveVMBoxControllerVersion* will behave transparently, and the *ReceiveVMBox* on the end device will be responsible for taking callers' messages. The two feature boxes work together and provide a graceful solution of taking callers' messages.

(a) A usage for taking callers' messages
(*ReceiveVMControllerVersion* residing in the router, *ReceiveVM* residing in the end device)

(b) A usage for retrieving stored messages

(c) A usage for recording greeting messages

Figure 5.1 The snapshots of three usages involving the three feature boxes

The *ReceiveVMBox/ReceiveVMBoxControllerVersion*, *RetrieveVMBox*, and *Record-*

GreetingVMBox are all free feature boxes residing in target zones. The *ReceiveVMBox* is a feature box in the target zone of the subscriber's LI address. The *ReceiveVMBoxControllerVersion* is a feature box in the target zone of the subscriber's LIC. The *RetrieveVMBox* is in the target zone of a player box, which is a kind of RI boxes and connects to a voice mail server or a local voice mail database. The *RecordGreetingVMBox* is in the target zone of a recorder box, which is also a kind of RI boxes and connects to the voice mail server or the local voice mail database too. Figure 5.1 shows the snapshots of three usages involving the three feature boxes.

5.2.1 RECEIVEVMBOX AND RECEIVEVMBOXCONTROLLERVERSION

The *ReceiveVMBox*, shown in Figure 5.2, and the *ReceiveVMBoxControllerVersion* both employ 5 ports: a box port (labeled '*box*') and four dual ports (labeled '*participant*', '*subscriber*', '*player*', '*recorder*'). Actually the four dual ports can be replaced as either caller ports or callee ports. The reason to choose dual ports is due to the convenience of implementation. Since some built-in finite state machines developed by AT&T were used during the implementation of the VM feature and ports in these FSMs are all dual ports, in my design I had to use dual ports too.

For the *ReceiveVMBox*, after port *box* receives a *Setup* message from the router, port *participant* will send back an *Upack* message to its upstream feature box or LI box. Port *subscriber* will initiate an outgoing internal call and relay the *Setup* message to the LI box of its subscriber. If the subscriber is busy or does not answer the call for a certain

period of time, the *ReceiveVMBox* will terminate the current outgoing internal call and port *player* will place a second outgoing internal call to a player box. The player box is the resource interface of a message player. It will play a greeting message to the caller. After the greeting message is finished, the *ReceiveVMBox* will terminate the second outgoing internal call and port *recorder* will place a third outgoing internal call to a recorder box. The recorder box is the resource interface of a message recorder. It will record the caller's message and save it. Note that the player box and the recorder box are connected to a local voice mail database, since in the split *ReceiveVM* feature, the *ReceiveVMBox* is running on the end device.

Figure 5.2 *ReceiveVMBox*

The *ReceiveVMBox* must establish a media channel to play a greeting message and record a voice message. Figure 5.3 shows the signaling layer and the media layer relating to the *ReceiveVMBox*. User *c*'s LI box, the *ReceiveVMBoxControllerVersion*, and the *ReceiveVMBox* are connected to the *MB2* media box. The player box and the recorder

box which are called by the *ReceiveVMBox* are also connected to *MB2*. The interface boxes and feature boxes will send commands to their media boxes for opening and closing channels and links.

Figure 5.3 The signaling layer and the media layer related to the *ReceiveVMBox*

The *ReceiveVMBoxControllerVersion* is just like the *ReceiveVMBox* except that it will call a player box and a recorder box which are connected to the voice mail server running inside the network.

5.2.2 RETRIEVEVMBOX

The *RetrieveVMBox*, as shown in Figure 5.4, employs three ports: a box port (labeled

'box') and two dual ports (labeled '*participant*' and '*subscriber*').

Figure 5.4 RetrieveVMBox

When port *box* receives a *Setup* message from the router, port *participant* will send back an *Upack* message to its upstream feature box or LI box. Port *subscriber* will initiate an outgoing internal call and relay the *Setup* message to its subscriber, a player box. The player box will play the stored messages to the caller. Like the *ReceiveVMBox*, the *RetrieveVMBox* also must establish a media channel to play back stored messages. It will send commands to its assigned media box for media controls.

5.2.3 RECORDGREETINGVMBox

The *RecordGreetingVMBox* is used to record customized greeting messages. The *RecordGreetingVMBox*, as shown in Figure 5.5, has three ports: a box port (labeled '*box*') and two dual ports (labeled '*participant*' and '*subscriber*').

The functionality of the *RecordGreetingVMBox* is almost the same as the *RetrieveVMBox* except that it will call the resource interface of a message recorder, namely the recorder box. The recorder box will record the new greeting message of a user. Like the *ReceiveVMBox* and the *RetrieveVMBox*, the *RecordGreetingVMBox* also

must send its assigned media box for media controls since it must establish a media channel to record greeting messages.

Figure 5.5 RecordGreetingVMBox

5.2.4 THE VOICE MAIL SERVER AND THE LOCAL VOICE MAIL DATABASE

The voice mail system usually stores a large amount of messages. Although messages can be stored inside an ECLIPSE router, they will use more router storage. Storing messages in a separate storage server separates message processing from the normal functions of a router. Therefore, in our design, all messages needed to be stored inside the network will be stored in a separate voice mail server.

A certain number of player interface boxes and recorder interface boxes are connected to the VM server. They can subscribe to the *RetrieveVMBox* or *Record-GreetingVMBox*. When callers call the player boxes or recorder boxes, these boxes will play the stored message or record the new greeting messages in the VM server. The player boxes and recorder boxes can be called by the *ReceiveVMBoxControllerVersion* too, and will play the greeting message to the caller and save the caller's message in the VM server.

Messages can also be stored in a special kind of storage server, the local voice mail database running on the end device. A certain number of player boxes and recorder boxes are connected to each local VM database. The local player boxes and recorder boxes will be connected to the remote network VM server too (using Java RMI). These boxes work in this way: for example, when the user calls the local player box, the box will connect to both the network VM server and the local VM database. Messages stored in both places are merged together by the local player box and are sorted according to the time stamp of each message. The player box then plays the sorted messages to the caller one by one.

It is possible for users to retrieve their messages stored in end devices even if they are not connected to the ECLIPSE system. That is, messages may be retrieved without the participation of the network. However, messages have to be stored in the VM server that is permanently connected to the ECLIPSE system when the users are offline.

The VM server in the network can be implemented as a Java RMI server, just like the *Central Data Server*. Any kind of database server can be chosen as the local VM database server. Several tables are implemented to meet the requirements of the VM system. One is the saved message table which stores messages left by callers for all users. Another one is the greeting message table which stores greeting messages of all users. Some auxiliary tables may be needed, such as a password table, etc.

For each message left by the caller, it should be stored with its recording time, its recording data and the “from” address.

5.3 THE BEHAVIORS OF THE VOICE MAIL FEATURE

In this section three aspects regarding the behaviors of the VM feature are discussed: leaving messages, retrieving message, and recording greeting messages. The UML sequence diagrams are used to show signaling relating to the corresponding VM boxes.

5.3.1 LEAVING MESSAGES

Let's consider a case when user *a* attempts to make a customer call to user *b*, who has subscribed to a split *ReceiveVM* feature. User *b* uses its LI address to subscribe to the *ReceiveVMBox*, which is supposed to be running on user *b*'s end device. It also uses its LIC address to subscriber to the *ReceiveVMBoxControllerVersion*, which is supposed to be running on the router. When user *b* is offline, busy and does not answer the call for a period of time, the split *ReceiveVM* feature will play the greeting message of user *b* and take the caller's messages. In the subsections bellow, I will discuss the three scenarios in details.

5.3.1.1 LEAVING MESSAGES ON OFFLINE

In this scenario, since user *b* is offline, the LIC of user *b* will not function, and all feature boxes which should run on the end device will not be executed too. Thus only those feature boxes which should run on the router will be executed. In our case, since

user *b* subscribes to the network part of the *ReceiveVM* feature, the *ReceiveVMBoxControllerVersion*, only this feature box will be executed.

In Figure 5.6 a sequence diagram shows signaling relating to the *ReceiveVMBoxControllerVersion* that occurs in this scenario. In the sequence diagram, a “*port?message*” notation means that the message is received by this port and a “*port!message*” notation means that the message is sent by this port. These two notations are used only for the *ReceiveVMBoxControllerVersion* to set up and tear down internal calls.

Note that in this sequence diagram and the sequence diagrams below, we do not show *Setup* signals sent to and from the router and the LIC. Omitting them is just for simplicity and makes the diagrams more concise. The *Setup* messages are only shown from one box to another. Actually they have been sent from one box to the router/controller, and after being modified, they are sent to another box by the router/controller. Signals sent to and from the Mbox are not shown either. Between the feature box receives an *Open* message and sends back an *OAck* message, it usually asks the Mbox to open media channels and links. After the channels and links have been opened successfully, an *OAck* message is sent out. They are normal operations and will be executed by every feature, so the signals relating to these operations are not shown in these diagrams. Readers can refer to Figure 3.10 and 4.10 for more details.

In the sequence diagram of Figure 5.6, *par* and *sub* are abbreviated for port *participant* and *subscriber*.

Figure 5.6 The sequence diagram for leaving message on offline

The whole procedure will be divided into three steps. Step one begins when the *ReceiveVMBoxControllerVersion* receives a *Setup* message, generated by user *a*'s call attempt, from its box port. Its port *par* will then send an *Upack* message to the LI box of user *a*. And it will place an outgoing internal call (*Setup 1.2a*), which is directed to user *b*'s LI box, through its port *sub*. Port *sub* monitors the messages coming back to it on its original outgoing call. Since user *b* is offline, the router cannot find the LIC of user *b* and it will generate a routing error box. The routing error box sends an *Unknown* message following an *Upack* message (*Upack 1.4a*) to port *sub* of the *ReceiveVMBoxControllerVersion*, indicating that the subscriber is offline. Afterwards, the routing error box attempts to tear down its connection to the *ReceiveVMBoxControllerVersion*, as it knows that it is no longer useful. Step one is ended now.

During the beginning of Step two, the *ReceiveVMBoxControllerVersion*'s port *player* places a second outgoing internal call (*Setup 2.1a*) to a player box to play back a greeting message. If port *player* receives an *Avail* message following an *Upack* message, it means that the player box is available. Thus port *par* will pass the *Avail* message to *LI_a*. Upon receiving the *Avail* message, *LI_a* will send out an *Open(voice)* message to the *ReceiveVMBoxControllerVersion*. The *ReceiveVMBoxControllerVersion* will attempt to open the voice channel between itself and *LI_a* and the voice links inside itself (This part has not been shown in this figure. Please refer to Figure 4.10). The *Open(voice)* message is also passed to the player box. The player box will attempt to open the voice channel between itself and the *ReceiveVMBoxControllerVersion*. If this is successful, an *OAck* message is sent to the *ReceiveVMBoxControllerVersion*'s port *player*. And if the channel

and links have been successfully opened by the *ReceiveVMBoxControllerVersion* too, an *OAck* message is sent to LI_a through the *ReceiveVMBoxControllerVersion*'s port *par*. When LI_a receives the *OAck* message, it means that a media(voice) path is established between user a 's LI box and the player box.

The subsequent status messages relating to the voice channel are sent through the signal path terminations of the *ReceiveVMBoxControllerVersion*. As we mentioned before (Section 3.6.5), a signal path termination is like a wrapper class for the media channel termination. For example, *parSPT* is a signal path termination of port *par*, and messages relating to a specific media channel (in our case, voice) are sent and received only through this termination. In this way, signal path terminations separate signaling messages from one type of media to another.

Upon receiving the *OAck* message, LI_a will send a *Ready* message and indicates that it is ready to communicate. The *ReceiveVMBoxControllerVersion* will pass this *Ready* message to the player box. The player box will send a *Wait* message, indicating that it is preparing the device. After the device is ready, the player box will send an *Accept* message to the *ReceiveVMBoxControllerVersion*. The *ReceiveVMBoxControllerVersion* will pass this *Accept* message to LI_a . The *ReceiveVMBoxControllerVersion* will also send a *UserAddress* message to tell the player box the address of the subscriber. The player box will use this address to load the subscriber's greeting message and play it to the caller. Once the greeting message has been played, the player box will send a *Beep* message to the *ReceiveVMBoxControllerVersion*. The second internal call will be terminated (*Teardown 2.17a*) by the *ReceiveVMBoxControllerVersion*.

The last step is Step three. It begins with the *ReceiveVMBoxControllerVersion*'s port recorder placing a third outgoing internal call (*Setup 3.1a*) in order to record the caller's message. If the recorder box accepts the call, the *ReceiveVMBoxControllerVersion* will notify the recorder box of the subscriber's address and the current caller's address (*UserAddress 3.9a*, *CallerAddress 3.10a*), which are going to be stored with the caller's message. Afterwards, the caller can leave a message for the subscriber. Once the caller hangs up his phone, the *ReceiveVMBoxControllerVersion* will teardown the third outgoing internal call (*Teardown 3.12a*).

5.3.1.2 LEAVING MESSAGES ON BUSY

This scenario and the next scenario, leaving messages on no answer, will only be applied when user *b* is online. In this scenario, since user *b* is online, the line interface controller of user *b* will be running and all feature boxes supposed to be running on the end device will be executed too. All features subscribed to by user *b* using its controller's address will also be executed on the router. In our case, since user *b* subscribes to both the network part of the *ReceiveVM* feature, the *ReceiveVMBoxControllerVersion*, and the end device part of the *ReceiveVM* feature, the *ReceiveVMBox*, these feature boxes will be executed. In Figure 5.7 a sequence diagram shows signaling relating to both feature boxes that occur in this scenario.

Figure 5.7 The sequence diagram for leaving message on busy

Since the *ReceiveVMBox* will take callers' messages and the *ReceiveVMBoxControllerVersion* will behave transparently, the behavior of the *ReceiveVMBoxControllerVersion* is much simpler. It receives messages from one port and sends out the messages through another port without modifying it. Therefore I didn't specify its port names during message transmission. The "port?message" and "port!message" notations are used only for the *ReceiveVMBox* to set up and tear down internal calls.

In this scenario, if user *a* makes a customer call, the router will route the *Setup* message to the *ReceiveVMBoxControllerVersion*. The *ReceiveVMBoxControllerVersion* sends back an *Upack* message (*Upack 1.3b*), indicating that the internal call is successful, and passes the *Setup* message (*Setup 1.2b*) to the next feature box, the *ReceiveVMBox*. The *ReceiveVMBox* will place its first outgoing internal call (*Setup 1.4b*) directed to the subscriber, user *b*, as before.

If user *b* is busy, all the ports of user *b*'s LI box are occupied with other calls, so his LI box will send an *Upnack* message (*Upnack 1.6b*) to the *ReceiveVMBox*'s port *sub*, indicating that the final internal call set-up is failed.

Afterwards, the *ReceiceVMBox* places the second outgoing internal call (*Setup 2.1b*) to the player box for playing back the greeting message, and then place the third outgoing internal call (*Setup 3.1b*) to the recorder box for recording the caller's message.

5.3.1.3 LEAVING MESSAGES ON NO ANSWER

Figure 5.8 The sequence diagram for leaving message on no answer

This scenario, as shown in Figure 5.8, is applied when user *b* is online but does not pick up the phone within *X* seconds. When the *ReceiveVMBox* receives the *Wait* message (*Wait 1.19c*), which indicates that the LI box of user *b* is alerting the user to answer the call, it starts the timer by sending a *SetTimer* message to the box port of itself. If *X* seconds pass, the box port will send itself a *Timeout* message. Upon receiving the message, the *ReceiveVMBox* will teardown its first outgoing internal call and begin to call the player box.

Figure 5.9 An alternative case for leaving message on no answer

Figure 5.9 shows an alternative case for this scenario. If at any time within these *X*

seconds, user *b* picks up the phone, the usage will go through the *ReceiveVMBox* transparently and the timer will be cancelled.

There is also another possibility that although the LIC of user *b* is running, his LI box has been closed intentionally for some reason. In this case, when user *a* calls user *b*, user *b*'s controller cannot find its LI box. Thus a routing error box is created by the controller and an *Unknown* message is sent out by the routing error box. Upon receiving the message, the *ReceiveVMBox* running on the end device will spring into function and take the caller's message. The sequence diagram of this case is almost the same as that in Figure 5.6, except that the *ReceiveVMBoxControllerVersion* is replaced by the *ReceiveVMBox*.

5.3.2 RETRIEVING MESSAGES

The sequence diagram for retrieving messages is shown in Figure 5.10. To retrieve messages stored in the VM server in the network or in the end devices, the user has to call the player boxes attached to them. The player boxes should subscribe to the *RetrieveVMBox* in their target zone.

It is the responsibility of the player boxes to connect to the VM server, local VM databases, or both. The network player box will only be attached to the VM server, while the local player box will be connected to both the VM server and the local database. If the user calls the network player box, only messages stored in the VM server will be played. If the user calls the local player box, the player box will play back the messages stored in

both places. In addition, the user can directly access the local VM database to retrieve the stored messages without being connected to the network.

Figure 5.10 The sequence diagram for retrieving messages

After the media channel between the caller and the player box have been established (*Accept 16e*), the *RetrieveVMBox* will send the player box a *RetrieveMessages* message, notifying the box of the caller's address. Once the player box receives the *RetrieveMessages* message, it will validate the caller's name and password. The caller presses the numbers on the keyboard to enter the password, and these numbers are sent to the player box through the *DTMF* (Dual-Tone Multi-Frequency) messages. If the authentication is successful, the player box will send the information about the saved messages to the

caller, such as, “You have 5 new messages, 10 messages in total. Press 1 to listen to new messages; Press 2 to listen to all messages; Press * to exit.” If the user selects 1 or 2, the messages will be played back one by one. During listening to each message, the user will be given the options to keep the message or delete it. For example, after one message is played, the user will hear a message: “Press 3 to keep the message; Press 4 to delete the messages; Press * to exit.” If the caller decides to abandon the retrieving operation, he can press the star key or just hang up the phone to tear down this call.

5.3.3 RECORDING GREETING MESSAGES

When a user subscribes to a VM feature, he will usually record a greeting message. He may modify the greeting message at a later time. In the ECLIPSE network, greeting messages can be stored in the VM server, the local VM database, or both.

To record a greeting message, the user has to call the recorder box attached to the VM server or the local VM database. The recorder box should subscribe to the *RecordGreetingVMBox* in its target zone. If the user calls the network recorder box, the greeting message will be stored in the VM server. If the user calls the local recorder box, the greeting message will be stored in the local database.

The behavior of the *RecordGreetingVMBox* is almost the same as the *RetrieveVMBox*, except that the *RecordGreetingVMBox* sends out the *RecordingPrompt* message instead of the *RetrieveMessages* message. Note that it is also possible for users to record greeting messages in the local end device without being connected to the network.

CHAPTER 6 IMPLEMENTATION OF THE VOICE MAIL FEATURE AND TESTING

6.1 IMPLEMENTATION OVERVIEW

In ECLIPSE features are implemented as an assembly of finite state machines. Usually a feature box has a main FSM, which defines its general operations. As the function of the feature box becomes more complicated, the states of its main FSM may have nested child state machines. These child state machines themselves may also have nested state machines, and so on.

As mentioned before, AT&T has developed a translator which can translate the ECLIPSE Statecharts language into Java code easily. But we do not have access to this translator. So, we have to implement Java code directly from FSMs. The challenge of this FSM to Java code mapping is that we have to strictly follow the specifications of the ECLIPSE project. For example, each class of finite state machines must extend the super class *FSM*, and the standard methods, such as “*addState*”, “*addInternalTransition*”, “*addMessageTransition*”, should be used to add states and transitions for each individual finite state machine. Appendix B gives an example of such a class of finite state machine.

ECLIPSE has defined some built-in fragments of FSMs which can be utilized by feature box programmers. Since some operations are common to most feature boxes,

reusing the existing classes may save a lot of effort for programmers. The implementation of the VM feature in this thesis has made use of the existing FSMs as much as possible. Descriptions about those FSMs can be found in Appendix A.

6.2 THE FSMs OF THE VM FEATURE

6.2.1 THE FSMs OF THE RECEIVEVM FEATURE BOX

The *ReceiveVM* feature is composed of two feature boxes: *ReceiveVMBox* and *ReceiveVMBoxControllerVersion*. Figure 6.1 shows the main FSM of the *ReceiveVMBox*: *ReceiveVMBoxFSM*. Figure 6.7 shows the main FSM of the *ReceiveVMBoxControllerVersion*: *ReceiveVMBoxFSMControllerVersion*.

ReceiveVMBoxFSM

Twelve states are shown on the statechart in Figure 6.1, and ten of them are superstates. Each superstate has one or several nested FSMs, which are further decomposed into their own statecharts, as shown on the following figures respectively. The event sequence numbers are shown on this diagram and the diagrams of its nested FSMs, corresponding to the message transmissions previously described in the sequence diagrams in Figure 5.7-5.8. Note that only part of the related sequence numbers on those sequence diagrams are shown in this FSM, and the rest of them are buried in the nested machines and can be found there.

Figure 6.1 Statechart for ReceiveVMBoxFSM

The initial state of the *ReceiveVMBoxFSM* is the *START* state. After entering the initial state, the *ReceiveVMBoxFSM* immediately creates a nested FSM, the *InitialTransparent2LinksFSMV2*, which attempts to set up a single media connection between two dual ports, port *par* and *sub*. Depending on the result of the attempt, the *InitialTransparent2LinksFSMV2* may enter three possible states: *LINK_OPENED*, *CALLER_UNLINKED*, and *CALLEE_UNKNOWN*.

When the *InitialTransparent2LinksFSMV2* enters the *LINK_OPENED* state, it means that a media channel has been opened successfully between port *par* and *sub* (port *sub* receives an *OAck* message). The *InitialTransparent2LinksFSMV2*'s entry into the *LINK_OPENED* state makes the super FSM, the *ReceiveVMBoxFSM*, transition from the initial *START* state to the *CALLEE_OPENED* state.

In the *CALLEE_OPENED* state, the *ReceiveVMBoxFSM* keeps on waiting for the arrivals of subsequent media status messages. If the signal path termination of port *par*, *parSPT*, receives a *Ready* status message, the *ReceiveVMBoxFSM* transitions from the *CALLEE_OPENED* state to the *WAITING_CALLEE* state. During this transition, it will pass the *Ready* message to the LI box of its subscriber through the signal path termination of port *sub*, *subSPT*.

In the *WAITING_CALLEE* state, the *ReceiveVMBoxFSM* is waiting for the response from the LI box of its subscriber. If the LI box begins to alert the subscriber, it will generate a *Wait* message. Upon receiving the *Wait* message, the *ReceiveVMBoxFSM* will pass it to the caller and at the mean time, enable a timer. If the subscriber picks up the phone within the timeout period, his LI box will generate an *Accept* message and send it

to the *ReceiveVMBox*. The arrival of the *Accept* message causes the *ReceiveVMBoxFSM* to transition to the *CALLER_CALLEE_CONNECTED* state. Before entering this state, the *Accept* message is passed to the caller and the timer is canceled.

A nested *Transparent2LinksFSM* (see Appendix A) is created on entry into the *CALLER_CALLEE_CONNECTED* state. In this state, the caller and callee can communicate freely through the media channel, and the *Transparent2LinksFSM* will allow the caller or callee to open another media channel if he wishes. If the caller or callee tears down the customer call, a *Teardown* event occurs and a transition is made to the *END* state.

After entering the *END* state, the *ReceiveVMBoxFSM* creates four nested *DualTeardownFSMs* to tear down the four ports: *sub*, *par*, *player* and *recorder*. The *DualTeardownFSM* is developed by AT&T Research Labs and reused here to tear down dual ports.

If in the *WAITING_CALLEE* state a *Timeout* message arrives, that means the subscriber does not pick up the phone within the timeout period. The *Timeout* event causes a transition to the *CALL_PLAYER* state. During this transition, two actions are performed: port *sub* tears down its outgoing call to the subscriber, and port *player* sends a *Setup* message to the player box.

On entry into the *CALL_PLAYER* state, the *ReceiveVMBoxFSM* creates a nested *CallResourceFSM*, which attempts to set up the media channel to a resource interface. Depending the result of the attempt, the *CallResourceFSM* may enter three possible

states: *CONNECTED*, *REJECTED*, and *DISCONNECTED*.

The transition “*playerSPT?Accept[@MEDIA_CALL.OPENED] / parSPT!Accept -> MEDIA_CALL.CONNECTED*” needs a little explanation. It means that when the *CallResourceFSM* is in the *OPENED* substate of the *MEDIA_CALL* superstate, the arrival of the *Accept* message on *playerSPT* will make the *CallResourceFSM* transition to the *CONNECTED* substate of the *MEDIA_CALL* superstate. During this transition, the *Accept* message is send out through *parSPT*. This transition has to be defined outside the *CallResourceFSM*, because it cannot handle propagating message between two ports.

When the *CallResourceFSM* enters the *CONNECTED* state, it means that a media channel to the resource interface is opened successfully. The super *ReceiveVMBoxFSM* will make a transition to the *OPEN2LINKS_TO_PLAYER* state. When the *CallResourceFSM* enters the *REJECTED* or *DISCONNECTED* state, it means that the attempt to open a media channel to the resource interface fails. Thus a transition is made to the *END* state.

After entering the *OPEN2LINKS_TO_PLAYER* state, the *ReceiveVMBoxFSM* creates a nested *Open2LinksOnPortsFSM* (see Appendix A), which is responsible for opening the media links between two ports, port *par* and *player*. When the nested *Open2LinksOnPortsFSM* enters the *LINKED_OPENED* state, it means that the links between two ports *par* and *player* have been opened. Thus the media connection between the caller interface box and the player interface box is set up successfully. The super *ReceiveVMBoxFSM* will then transition to the *PLAYING_GREETING* state, in which a greeting message is played back. Before entering this state, a *UserAddress* message to

sent out through the signal path termination of port *player* to notify the player box of the subscriber's address. The player box will use this address to retrieve the greeting message.

On entry into the *PLAY_GREETING* state, a nested *Transparent2LinksFSM* is created, which will allow the caller and the player box to open another media channel besides the current one.

If the player box finishes playing the greeting message, it will send a *Beep* message to the *ReceiveVMBox*. The arrival of the *Beep* message causes the *ReceiveVMBoxFSM* to transition from the *PLAY_GREETING* state to the *CALL_RECORDER* state. Before entering this state, two actions are performed: port *player* tears down its outgoing call to the player box, and port *recorder* sends a *Setup* message to the recorder box.

Just like in the *CALL_PLAYER* state, the *ReceiveVMBoxFSM* attempts to open a media channel to the recorder box in the *CALL_RECORDER* state. In this state, the arrival of the *Accept* message on the signal path termination of port *recorder*, *recorderSPT*, will make the *CallResourceFSM* transition from the *OPENED* substate of the *MEDIA_CALL* superstate to the *CONNECTED* substate of the *MEDIA_CALL* superstate. There is no need to pass this *Accept* message to the caller through *parSPT*, since the caller has already received an *Accept* message during the *CALL_PLAYER* state.

If a media channel to the recorder box is opened successfully, the super *ReceiveVMBoxFSM* will then transition into the *OPEN2LINKS_TO_RECORDER* state. If the attempt to open a media channel to the recorder box fails, then a transition is made to

the *END* state.

Just like in the *OPEN2LINKS_TO_PLAYER* state, in the *OPEN2LINKS_TO_RECORDER* state, the *ReceiveVMBoxFSM* attempts to open the media links between two ports, port *par* and *recorder*. If it is successful, the *ReceiveVMBoxFSM* will transition to the *RECORD_MESSAGE* state, in which the caller's message is to be saved. Before entering the *RECORD_MESSAGE* state, a *UserAddress* message and a *CallerAddress* message are sent out to notify the recorder box of the addresses of the subscriber and the current caller.

In the *RECORD_MESSAGE* state, if the caller finishes speaking his message and tears down the customer call, the *ReceiveVMBoxFSM* will transition to the *END* state.

Now let's go back to the *START* state. The *InitialTransparent2LinksFSMV2*'s entry into the *CALLEE_UNKNOWN* state indicating that the subscriber's LI box is closed and cannot be found, or the *InitialTransparent2LinksFSMV2*'s entry into the *CALLER_UNLINKED* state indicating that the subscriber is currently busy, will cause the *ReceiveVMBoxFSM* to transition to the *CONNECTED_PLAYER* state. Before entering the *CONNECTED_PLAYER* state, a *Setup* message is sent out through port *player*.

On entry into the *CONNECTED_PLAYER* state, the *ReceiveVMBoxFSM* creates a nested *ConnectedResourceFSM*, which attempts to set up a single media connection between two dual ports, port *par* and *player*. If this attempt is successful, the *ConnectedResourceFSM* will enter into the *CONNECTED* state. The super *ReceiveVMBoxFSM* will then enter into the *PLAY_GREETING* state. If this attempt fails,

the *ConnectedResourceFSM* will enter into the *CALLER_UNLINKED* or *CALLEE_UNKNOWN* state. The super *ReceiveVMBoxFSM* will then transition into the *END* state.

InitialTransparent2LinksFSMV2

InitialTransparent2LinksFSMV2 (DBox box, DualPort sub, DualPort par)

Figure 6.2 Statechart for **InitialTransparent2LinksFSMV2**

The *InitialTransparent2LinksFSMV2* (Figure 6.2) was modified from the *InitialTransparent2LinksFSM* (see Appendix A), which was developed by AT&T Research Labs. It just adds an extra state, *CALLEE_UNKNOWN*. Like the original FSM,

the *InitialTransparent2LinksFSMV2* is a FSM fragment that attempts to set up a single media connection between two dual ports, port *sub* and port *par*.

In the initial *START* state, if a *Setup* message arrives and the box appears in the subscriber's target zone, the FSM fragment initializes two ports, *caller* and *callee*, so that the port connected to the subscriber side of the box (port *sub*) is assigned to port *callee* and the port connected to the other side of the box (port *par*) is assigned to port *caller*. The FSM then transitions to the *LINK_CALLER* state. Two events are triggered on entry into the *LINK_CALLER* state: port *caller* signals an *Upack* message back to the sender of the *Setup* message, and port *callee* sends out a *Setup* message, with the command field in the *Setup* message being changed to *CONTINUE*.

Once the signaling channel has been established (*Upack 1.4a, 1.6c,d*), the FSM enters into the *TRANSPARENT* state. On entry into the *TRANSPARENT* state, the FSM creates a nested *TransparentFSM*, which transparently propagates messages between the two ports. This *TransparentFSM* was developed by AT&T Research Labs and reused here (see Appendix A). In the *TRANSPARENT* state, the arrival of an *Open* message from the caller causes the FSM to transition to the *OPEN_LINK* state.

After entering the *OPEN_LINK* state, the FSM creates a nested *Open2LinksFSM*, which attempts to establish a media connection between the two ports. The *Open2LinksFSM* was also developed by AT&T Research Labs and reused here (see Appendix A). If this fails, the FSM then reverts to the *TRANSPARENT* state and waits until the next *Open* message arrives from the caller. If this succeeds, then the FSM transitions to the *LINK_OPENED* end state.

The FSM transitions from the *LINK_CALLER* state to the *CALLER_UNLINKED* end state after the arrival of an *Upnack* message, indicating that the callee is busy currently. When an *Unknown* message is arrived, the FSM transitions from the *TRANSPARENT* state to the *CALLEE_UNKNOWN* end state, indicating that a routing error occurs, probably because the LI box of the callee cannot be found.

CallResourceFSM

CallResourceFSM (DualPort callPort, LocalSignalPathTermination callSPT)

Figure 6.3 Statechart for CallResourceFSM

The *CallResourceFSM* (Figure 6.3) attempts to open a media channel to a resource interface. It first attempts to set up a signaling connection, then a media connection to the resource interface. These two tasks are accomplished in the two nested FSMs, the *SignalCallFSM* and the *MediaCallNoAcceptFSM*. It ends in one of the three states: *CONNECTED*, *REJECTED* and *DISCONNECTED*. The former state indicates

that a connection is established, and the latter two states indicate that a connection cannot be established.

SignalCallFSM

Figure 6.4 Statechart for SignalCallFSM

The *SignalCallFSM* (Figure 6.4) attempts to set up a signaling connection with a remote line or resource interface. It was developed by AT&T Research Labs and reused here without any change (see detailed description in Appendix A). It ends in one of the two states: *AVAIL* and *DISCONNECTED*. The *AVAIL* state means that a signaling connection is established, and the *DISCONNECTED* state means that the signaling connection cannot be established.

MediaCallNoAcceptFSM

The *MediaCallNoAcceptFSM* (Figure 6.5) attempts to open a media connection to a line or resource interface. It is modified from the original FSM, *MediaCallFSM* (see Appendix A), which was developed by AT&T Research Labs, but does not handle the *Accept* message. The *Accept* message will be processed by the super FSM of this FSM to accomplish its own functionality. It ends in one of the three states: *CONNECTED*, *REJECTED* and *DISCONNECTED*. The *CONNECTED* state means that a media channel is established, and the *REJECTED* or *DISCONNECTED* state means that the media connection cannot be established.

MediaCallNoAcceptFSM (LocalSignalPathTermination callSPT)

Figure 6.5 Statechart for MediaCallNoAcceptFSM

ConnectResourceFSM

ConnectResourceFSM (DBox box, DualPort caller,
DualPort callee, LocalSignalPathTermination calleeSPT)

Figure 6.6 Statechart for ConnectResourceFSM

The *ConnectResourceFSM* (Figure 6.6) attempts to set up a single media connection between two dual ports. Its statechart is very much like the *InitialTransparent2LinksFSMV2*. One difference between them is that the *ConnectResourceFSM* does not handle *Setup* messages. Before creating this FSM, the

super FSM has just received a *Setup* message. So at the initial *START* state, this FSM will wait for the arrival of an acknowledge message (*Upack 2.2a 2.2b*, or *Upnack*).

The other difference is that this FSM explicitly defines a *CONNECTED* state, making sure that by the time the *CONNECTED* state is entered, the media connection is fully available. For the *InitialTransparent2LinksFSM*, when the *LINKED_OPENED* end state is entered, only the *OAck* message has been received. Its super FSM will have to keep on waiting for subsequent media status messages.

ReceiveVMBoxFSMControllerVersion

The *ReceiveVMBoxFSMControllerVersion* is the main FSM of the *ReceiveVMBoxControllerVersion* and is responsible for taking caller's messages while the subscriber is offline. Its statechart, as shown in Figure 6.7, is a simplified version of the *ReceiveVMBoxFSM*. The sequence numbers on Figure 5.6 (leaving message on offline) can be found on this diagram and the diagrams of its nested FSMs.

The initial state of the FSM is the *START* state. After entering the initial state, the FSM immediately creates a nested *InitialTransparent2LinksFSMV2*, which attempts to set up a single media connection between two dual ports, port *par* and *sub*.

When the *InitialTransparent2LinksFSMV2* enters into the *LINK_OPENED* state, it means that a media channel has been opened successfully between port *par* and *sub* (port *par* receives an *OAck* message). The *InitialTransparent2LinksFSMV2*'s entry into the *LINK_OPENED* state makes the super *ReceiveVMBoxFSMControllerVersion* transition from the initial *START* state to the *CALLER_CALLEE_CONNECTED* state.

Figure 6.7 Statechart for ReceiveVMBBoxFSMControllerVersion

In the *CALLER_CALLEE_CONNECTED* state, the FSM transparently propagates messages between the two ports, port *par* and port *sub*, and the caller or callee can open another media channel if he wishes. If the caller or callee tears down the customer call, a *Teardown* event occurs and the *END* state is entered.

The *CALLER_CALLEE_CONNECTED* state is very important in the lifetime of the *ReceiveVMBoxFSMControllerVersion*. When the subscriber is not offline, the end device part of the *ReceiveVM* feature will take callers' messages, and the network part will behave transparently. Its FSM, *ReceiveVMBoxFSMControllerVersion*, will stay in this state and transparently propagates messages between the caller and callee.

The *InitialTransparent2LinksFSMV2*'s entry into the *CALLEE_UNKNOWN* state means that the router encounters a routing error because the subscriber is offline, and an *Unknown* message is received from the routing error box. The *ReceiveVMBoxControllerVersion* will then spring into function. Thus in the subsequent states, its FSM will attempt to connect the player box to play a greeting message, and connect the recorder box to record the caller's message.

The *InitialTransparent2LinksFSMV2*'s entry into the *CALLER_UNLINKED* state means that an *Upnack* message has arrived on port *sub*. The *ReceiveVMBoxFSMControllerVersion* does not need to handle this situation, and it cannot send the *Upnack* message upstream because it has already sent *Upack*. Instead, it generates an *Unavail* message. After sending the message, the FSM transitions to the *END* state.

6.2.2 THE FSMs OF THE RETRIEVEVM AND RECORDGREETING FEATURE

BOXES

Figure 6.8 shows the main FSM of the *RetrieveVMBox: ReceiveVMBoxFSM*. The event sequence numbers in the brackets show the correlation between the FSM and the sequence diagram in Figure 5.9. This FSM also creates some nested FSMs which have been used by the *ReceiveVM* feature. In order to make the diagrams for the *ReceiveVM* feature easy to follow, the events sequence numbers relating to this FSM are not shown in those nested FSMs.

The initial state of the *RetrieveVMBoxFSM* is the *START* state. After entering the initial state, the FSM immediately creates a nested *InitialTransparent2LinksFSMV2*, which attempts to set up a single media connection between two dual ports, port *par* and *sub*. Port *sub* is connected to the subscriber side of the box, that is, the side of the player box. Port *par* is connected to the caller side of the box, that is, the side on which the *RetrievingMessages* request comes from. Depending on the result of the attempt, the *InitialTransparent2LinksFSMV2* may enter three possible states: *LINK_OPENED*, *CALLER_UNKNOWN*, and *CALLER_UNLINKED*.

When the *InitialTransparent2LinksFSMV2* enters the *LINK_OPENED* state, it means that a media channel has been opened successfully between port *par* and *sub*. The super *RetrieveVMBoxFSM* then transitions from the initial *START* state to the *LINKED_OPENED* state after saving the caller's address.

Figure 6.8 Statechart for RetrieveVMBoxFSM

In the *LINKED_OPENED* state, the *RetrieveVMBoxFSM* keeps on waiting for subsequent media status messages. In the mean time, the caller or the callee part of the call can open another media channel if he wishes. The arrival of the *Accept* message causes the FSM to transition to the *PLAYING_MESSAGES* state, in which the stored messages of the caller will be played back. In the transition, two actions are performed: the *Accept* message is passed upstream to the caller part, and a *RetrieveMessages* message with the caller's address as the parameter is sent out to the subscriber, that is, the player box. A *Teardown* event coming from the caller part causes the FSM to transition to

the *END* state.

When the *InitialTransparent2LinksFSMV2* enters the *CALLEE_UNKNOWN* state, it means that an *Unknown* message arrives from the subscriber, probably because the LI box of the player interface cannot be found and a routing error box is created which sends out the *Unknown* message. The super FSM *RetrieveVMBoxFSM* just passes the *Unknown* message upstream. When the *InitialTransparent2LinksFSMV2* enters the *CALLER_UNLINKED* state, it means that an *Upnack* message arrives from the subscriber, probably because the LI box of the player interface is busy. The *RetrieveVMBoxFSM* will generate an *Unavail* message and send it upstream.

The main FSM of the *RecordGreetingVMBox*, the *RecordGreetingVMBoxFSM*, is almost the same as the *RetrieveVMBoxFSM* except that in the transition from the *LINKED_OPENED* state to the *RECORDING_GREETING_MESSAGE* state (corresponding to the *PLAYING_MESSAGE* state in the *RetrieveVMBoxFSM*), a *RecordingPrompt* message is sent out instead of the *RetrieveMessages* message.

6.2.3 THE FSMs OF THE PLAYER/RECORDER BOX

Since in DFC the player/recorder interface box is a kind of resource interface box, it should extend the RI box, which will allow several callers to call simultaneously and dynamically allocate an idle port to each call. However, in the current ECLIPSE project, the RI box has not been implemented yet. Thus in this thesis the player/recorder box is implemented as a subclass of the LI box which only allows one caller to get through it at

a time. Once the implementation of the RI box has been finished by AT&T Research Labs, the current implementation of the player/recorder box can be moved onto the new RI box with a slight modification because every individual call to the player/recorder box follows the same procedure.

As mentioned before, there are two kinds of player/recorder interface boxes. One is used in the network, and the other is used in local end device. The *PlayerInterface/RecorderInterface* class is used by network player/recorder box to connect to the network VM server, while the *LocalPlayerInterface/LocalRecorderInterface* class is used by local player/recorder box to connect to both the network VM server and the local VM database. These classes have another import function, that is, they will create an instance of the *UDP* class, which is a simple class to allow communication on UDP sockets. For example, when the player box wants to play a message, it will call the play method in the *PlayerInterface* class and the *PlayerInterface* class will send out the message through the UDP socket.

In this section the FSMs of the network player box are described, while the FSMs of other player and recorder boxes are omitted since the fundamental behaviors of those boxes are almost the same as the network player box.

The player box has a box port, a dual port which is an internal port to connect to the ECLIPSE network, an external port which connects to the resource, and a player interface which is responsible for connecting to the VM server. Figure 6.9 shows the main FSM of the player box, *PlayerBoxFSM*. The *internalTermination* and the *externalTermination* are the signal path terminations associated with the specified type of media on the internal

port and the external port.

PlayerBoxFSM

PlayerBoxFSM (BoxPort boxPort,
DualPort internalPort, LocalSignalPathTermination internalTermination,
DBoxExternalPort externalPort, ExternalSignalPathTermination externalTermination,
PlayerInterface, playerInterface)

Figure 6.9 Statechart for PlayerBoxFSM

The initial state of the *PlayerBoxFSM* is the *OPEN_EXT_CHAN* state. On entry into this state, a nested *OpenExtChanFSM* (see Appendix A) is created to open an external media channel to the resource. The “*player*” parameter is the name of the external channel. The parameter, *udp*, obtaining its value from the *playerInterface*, will allow the resource to communicate with its remote peer through a UDP media channel.

If the *OpenExtChanFSM* enters the *END* state, it means that the external media channel has already been opened successfully. The super *PlayerBoxFSM* will transition from the *OPEN_EXT_CHAN* state to the *LINE* state. A nested *PlayerLineFSM* is created immediately, which is responsible for receiving the internal call, setting up the media links and channels, and playing appropriate messages.

PlayerLineFSM **Figure 6.10 Statechart for PlayerLineFSM**

The initial state of the *PlayerLineFSM* (Figure 6.10) is the *START* state. The FSM enters the *OPEN2LINKS* state from the *START* state through an internal transition. On entry into the *OPEN2LINKS* state, a nested *Open2LinksOnPortsFSM* is created immediately, which will open a bi-directional media links on the two ports, *internalPort* and *externalPort*. Once the links has been opened (the *Open2LinksOnPortsFSM* enters

the *LINKS_OPENED* state), the *PlayerLineFSM* transitions to the *IDLE* state, waiting for receiving internal calls.

In the *IDLE* state, if the box port receives a *Setup* message, the *BUSY* state is entered and a nested *PlayerBusyFSM* is created to open the signaling and media channels and play back messages. The *BUSY* is a nested history state, which means the nested *PlayerBusyFSM* will maintain its states between invocations (see Figure 3.10 for details). For example, if in any state, the *PlayerBusyFSM* receives a *Setup/Open* message, an *Upnack/ONack* message is returned and the previous state remains unchanged.

In the *BUSY* state, if the internal port receives a *Teardown* message, the Super *PlayerLineFSM* transitions to the *RESTART* state. If the nested *PlayerBusyFSM* enters the *DISCONNECTED* state indicating the call setup is failed, or the *PLAYING_WELCOME_MESSAGE* indicating a greeting message has just been played back, the super *PlayerLineFSM* will transition to the *RESTART* state too.

In the *RESTART* state, the FSM sends out the *Teardown* message to tear down its connection with the caller and reverts the *START* state, preparing to receive another internal call.

PlayerBusyFSM

The *PlayerBusyFSM* (Figure 6.11) starts with the initial *RECEIVING_CALL* state. Note that since the player box only receives calls and does not place calls, its FSM does not have a state in which it can initiate a call. On entry into the *RECEIVING_CALL* state, a nested *AnswerFSM* (see Appendix A) is created, which is responsible for setting up the

signaling and media channels. Two transitions are defined outside the *AnswerFSM* because the built-in *AnswerFSM* does not define them. One is: when the *internalTermination* receives a *Ready* message, it will immediately send back a *Wait* message, and the *AnswerFSM* transitions from the *OCK_WAIT* substate of the *MEDIA_ANSWER* superstate to the *LOCAL_ALERTING* substate of the *MEDIA_ANSWER* superstate. The other one is: the *AnswerFSM* takes an internal transition from the *LOCAL_ALERTING* substate of the *MEDIA_ANSWER* superstate to the *CONNECTED* substate of the *MEDIA_ANSWER* superstate. The *AnswerFSM* may end in one of the two states: the *CONNECTED* state indicating that a media connection has been set up successfully, and the *DISCONNECTED* state indicating that the attempt to set up the media connection is failed. The super *PlayerBusyFSM* will enter the corresponding *CONNECTED* or *DISCONNECTED* state according to the end state of the nested *AnswerFSM*.

In the *CONNECTED* state, the arrival of the *UserAddress* will make the *PlayerBusyFSM* transition to the *PLAYING_WELCOME_MESSAGE* state. On entry into that state, an action "*playerInterface.play(welcome_message)*" is invoked and the *playerInterface* will play back the appropriate welcome message according the user's address. After the welcome message has been played, a *Beep* message is sent out.

In the *CONNECTED* state, if the *RetrieveMessages* message is received, the FSM will transition to the *VALIDATING* state, in which the user-input password will be validated. The caller presses the buttons on the telephone, and the information will be sent to the player box through the *DTMF* messages. If the password is valid, the FSM

will enter the *PLAYING_PROMPT_FOR_RETRIVAL* state, in which a prompt message, such as “You have 5 new messages, 10 messages in total. Press 1 to listen to new messages; Press 2 to listen to all messages; Press * to exit”, will be played by the *playerInterface*. If the password is invalid, the FSM will enter the *DISCONNECTED* state.

Figure 6.11 Statechart for PlayerBusyFSM

In the *PLAYING_PROMPT_FOR_RETRIVAL* state, if the caller presses button 1 or 2, the FSM will enter the *PLAYING_SAVED_MESSAGE* state to play new or all stored messages of the caller one by one according to which button he has pressed. After playing one stored message, another prompt message will be sent to the caller, such as “Press 3 to keep the message; Press 4 to delete the messages; Press * to exit”. If the caller presses button 3, next message will be played. If he presses button 4, the current message will be deleted and next message will be played.

In the *VALIDATING* state, if the caller hangs up the phone, the call will be torn down. In the *PLAYING_PROMPT_FOR_RETRIVAL* and *PLAYING_SAVED_MESSAGE* states, if the caller hangs up the phone or presses the * button, the call will be torn down too. A transition is then made to the *DISCONNECTED* state. These transitions have not been shown on the diagram since they would make the diagram too cluttered to read.

6.3 COMPARISON WITH THE OTHER VOICE MAIL SYSTEM ON ECLIPSE

Another VM system was developed by Mr. Ruiguo Li in this thesis [33]. His VM system is based on the FDDP version of DFC, consisting of a VM feature and a VM center. But we find a significant problem with his design of the VM feature. The VM feature only transfers the call to the VM center. It is the LI box of the caller that performs much of the work, such as tearing down the current call, redirecting the call to the VM center, etc. Thus he modified the code of the LI box by adding a state and some

transitions. Actually all this work should be done by the VM feature. Feature box programmers should not modify the other part of the system; otherwise there would be no feature modularity in DFC.

There is another problem with his VM system. The ECLIPSE system cannot detect whether the end device is online or not, so the VM feature running on the router cannot run automatically. That is probably due to the limitations of the FDDP version of DFC, and not the VM system. In my modified architecture, if the end device is offline, the router will create a routing error box, which is the source of *Unknown* messages. The VM feature on the router will spring into function automatically upon receipt of an *Unknown* message.

6.4 TESTING

In order to evaluate the modified ECLIPSE architecture and the VM system, two test cases are generated and run. One is used to test the basic functionalities of the VM feature, and the other is used to test the general behaviours of the modified architecture by adding one more feature (Call Waiting).

Due to the limitation of the size of this thesis, only part of the test results is shown here. Screen snapshots are used to provide a better visualization of the test results. A couple of trace files are generated for each test case. Since some of the code is proprietary of AT&T Research Labs and we do not have access to it, some message

transmissions cannot be traced. Most of them are related to the Mbox [33]. Completed trace files can be found in Appendix C.

6.4.1 TEST CASE ONE – THE VM FEATURE

This test case is used to demonstrate the basic functionalities of the VM system. Two hosts are involved in this test. A primary router is started on one host. Two LI boxes, *Andrew* and *Lillian's* LI boxes, are running on this router inside the network. Another customer, *Eric*, uses the other host as his end system. His line interface controller is running on it. Eric has subscribed to the full package of the VM feature.

All aspects regarding the behaviours of the VM feature, including leaving messages on offline, busy, and no answer, retrieving saved messages, and recording greeting messages, are tested during this test case. The VM feature works very well and has reached our requirements. As an example, screen snapshots for leaving message on busy is shown here.

Step 1. Start the first connection:

Figure 6.12 shows that *Eric* places a customer call to *Andrew*. The status label on the GUI of *Andrew* shows that “*eric is calling*”.

Step 2. Start the second connection:

Figure 6.13 shows that a customer call is made successfully between *Eric* and

Andrew. The status labels on the GUIs of *Eric* and *Andrew* show “connecting”. *Eric* and *Andrew* are communicating through the media channel. Later, *Lillian* attempts to call *Eric*.

Figure 6.12 Snapshot of starting a connection between two customers

Figure 6.13 Snapshot of a second call attempt

Figure 6.14 Snapshot of playing a greeting message

Figure 6.15 Snapshot of leaving a message

Step 3. Playing the greeting message:

Since *Eric* is talking with *Andrew* and cannot accept the call from *Lillian*, his VM springs into function and redirects the call to the player. The player plays back a greeting message to *Lillian*, as shown in Figure 6.14.

Step 4. Leaving a message:

Figure 6.15 shows that *Lillian* begins to leave a message after hearing the greeting message of *Eric*. This message is saved, and will be retrieved by *Eric* at later time.

6.4.2 TEST CASE TWO – THE VM AND CW FEATURE

This test case is used to test the more general behaviors of the modified architecture than test case one. Two hosts are still involved in this test. A primary router, associated with two LI boxes, *Andrew* and *Lillian's* LI boxes, are running on one host. The line interface controller of another customer *Eric* is running on the other host. *Eric* has subscribed to the full package of the VM feature and a CW feature which will be running on his end system. CW has been set a higher priority to VM, so it will absorb busy signals earlier than VM.

Several usages are tested during this test. As an example, Figure 6.16 shows one scenario. At first *Eric* makes a successful call to *Andrew*. The usage passes transparently through CW because *Eric* subscribes to Call Waiting. Later, *Lillian* attempts to call *Eric*. This call passes through the *ReceiveVMBoxControllerVersion* and the *ReceiveVMBox*

transparently on its way to *Eric*, and finally an internal call is routed to CW, where it is accept at the third port. CW springs into action. It first signals back a *Wait* message upstream, and then sends a Call Waiting notification to *Eric*. Upon receiving the *Wait* message, the *ReceiveVMBox* will start the timer and passes the *Wait* message upstream too. If within the timeout period, *Eric* sends out a flash signal, CW will switch its voice channel from *Andrew* and *Eric* to *Lillian* and *Eric*. The *ReceiveVMBox* will cancel the timer and behave transparently. If *Eric* ignores the CW notification, the *ReceiveVMBox* will function and ask *Lillian* to leave a message.

Figure 6.16 One usage during test case two

This test case has proved that the modified architecture can handle situations in which line interface controllers appear in the source or target zone, and can deal with feature interactions just like the original architecture. Screen snapshots will not be shown here due to the space limitation of this thesis and the detailed trace files can be found in Appendix C.

CHAPTER 7 CONCLUSIONS AND FUTURE WORK

7.1 CONCLUSIONS

DFC is a service architecture that provides modularized features, separates signaling with media control, and offers support for managing feature interactions. ECLIPSE is a DFC implementation based on the IP platform. However, in the original ECLIPSE architecture, all features are deployed on routers inside the network. This feature deployment scheme wastes the strong processing and storage power of the end systems in Internet telephony. It may also cause the performance problem and the security problem. This thesis proposed a modified version to the original architecture.

A new component, the line interface controller, is added to the new architecture, which will allow features to be run on end systems. It also allows end users to download service blocks (feature boxes) from a third party without the need to ask the permission of the network. In the new architecture, end users have options to subscribe to features which should be run inside the network through their line interface controllers, and subscribe to features which will run on end systems through their line interface addresses.

In the new architecture, features are to be deployed according to their nature. Some features are liable or better to be put inside the network, such as the Three Way Calling

(3WC) feature. Some features are better to be deployed in end systems, such as the Incoming Call Screening (ICS) feature. And some features, such as the VM feature, have to be put in both places, with the network part being a backup to the end system part, because there is a need to keep state in the network even when an end system is not connected or accessible. While in the short term there will still contain a large amount of “dumb” telephone devices in Internet telephony, basic versions of features should be provided inside the network.

Compared to the original architecture, the modified architecture reduces the number of features residing inside the network, thus reduces the workload of the network node and saves the network’s storage space. It also eliminates the security risk caused by storing personal information inside the network. As to the feature interaction problem, it can deal with it just like the original architecture. Although in the modified architecture the number of the total signaling messages increases, the number of the messages transported over the network does not increase. Since transporting messages over the network costs network bandwidth and increases the transmission delay, it is very desirable to keep the number of this kind of messages as low as possible. Furthermore, compared to other modified architecture proposed previously, this architecture is symmetric and more flexible, as it allows both source features and target features residing on end systems.

There is another important characteristic of the new architecture: the router can automatically detect if the end device is online or not. This is the foundation of the implementation of split features, such as the VM feature, and makes the system provide

more reliable services.

The VM feature, which were designed and implemented in this thesis, consists of three feature boxes: the *ReceiveVMBox*, the *RetrieveVMBox*, and the *RecordGreeting-VMBox*. All three feature boxes are implemented as free feature boxes in the target zones. The *ReceiveVMBox* is designed as a split feature: one part is running on the end system and the other part is running inside the network. Both parts work together, providing a graceful solution for handling callers' messages. The implementation of this split feature covers all cases to store callers' messages when the subscriber is offline, online but busy talking to someone else, or online but does not answer the call. The other two feature boxes allow users to retrieve saved messages and record customized greeting messages.

All aspects regarding the behaviours of the VM feature are tested. The VM feature works very well and has reached our requirements.

7.2 FUTURE WORK

The modified ECLIPSE architecture presented in this thesis has several advantages over the original architecture. This thesis only gives intuitive and simple comparison between the original and new architectures regarding signaling messages. Deep performance analysis is needed to be performed on both architectures.

A new feature, the VM feature, was designed and implemented based on the modified architecture. It is proved that this VM feature provides a graceful solution for

taking and retrieving callers' messages. However, this thesis only discusses the deployment of several existing features, such as the VM feature and the CW feature on the new architecture. More features are needed to be developed based on the new architecture and more research is needed to be done into the deployment of more features on this architecture.

The new architecture inherits all mechanisms to manage feature interactions from the original architecture. With more and more features being added, the feature interaction problem will become more complex and severe. In DFC feature interactions can be analyzed by placing different precedence relations on features. Thus more investigation is needed to determine the desired precedence relations of various feature boxes.

REFERENCES

1. Jiri Kuthan, *Overview of the Internet Telephony*, 1998-2000, online at:
<http://www.iptel.org/info/>.
2. Mahbub Hassan, Alfandika Nayandoro and Mohammed Atiquzzaman, *Internet Telephony: Services, Technical Challenges, and Products*, IEEE Communications Magazine, vol.38, no.4, pp.96-103, April 2000.
3. Christos A. Polyzois, K. Hal Purdy, Pine-Fai Yang, David Shrader, Henry Sinnreich, Francois Menard and Henning Schulzrinne, *From POTS to PANS: A Commentary on the Evolution to Internet Telephony*, IEEE Network, vol.13, no.3, pp.58-64, May-July 1999.
4. Jonathan Lennox and Henning Schulzrinne, *Feature Interaction in Internet Telephony*, Proc. of Feature Interaction in Telecommunications and Software Systems VI, Glasgow, U.K., May 2000.
5. Dirk O. Keck and Paul J. Kuehn, *The Feature and Service Interaction Problem in Telecommunications Systems: A Survey*, IEEE Transactions on Software Engineering, vol.24, no.10, 1998.
6. Pamela Zave, *FAQ Sheet on Feature Interaction*, 1999, online at:
<http://www.research.att.com/~pamela/faq.html>.
7. Pamela Zave and Michael Jackson, *New feature interactions in mobile and multimedia telecommunication services*, in Feature Interactions in Telecommunications and Software Systems VI, pages 51-66, IOS Press, 2000.

8. Simon Tsang, Evan H. Magill, and Bryce Kelly, *The feature interaction problem in networked multimedia services – present and future*, British Telecom Technology Journal XV(1):235-246, January 1997.
9. Michael Jackson and Pamela Zave, *Distributed feature composition: A virtual architecture for telecommunication services*, IEEE Transactions on Software Engineering, vol.24, no.10, pp.813-847, October 1998.
10. Greg Bond, Eric Cheung, Michael Jackson, Hal Purdy, Chris Ramming and Pamela Zave, *Investigations of a new service architecture for the next generation of telecommunications*, April 2000, online at:
<http://www.research.att.com/~pamela/dfc.html>.
11. The ECLIPSE website, <http://www.research.att.com/projects/eclipse/>.
12. Mark Handley, Henning Schulzrinne, Eve Schooler, and Jonathan Rosenberg, *SIP: Session Initiation Protocol*, RFC 2543, Internet Engineering Task Force, March 1999.
13. International Telecommunication Union, *Visual Telephone Systems and Equipment for Local Area Networks which Provide a Non-guaranteed Quality of Service*, Recommendation H.323, Telecommunication Standardization Sector of ITU, Geneva, Switzerland, May 1996.
14. Henning Schulzrinne and Jonathan Rosenberg, *Signaling for Internet Telephony*, in International Conference on Network Protocols (ICNP), Austin, Texas, U.S.A., October 1998.
15. Henning Schulzrinne and Jonathan Rosenberg, *A Comparison of SIP and H.323 for Internet Telephony*, Proc. NOSSDAV, Cambridge, U.K., July 1998.
16. Ismail Dalgic and Hanlin Fang, *Comparison of H.323 and SIP for IP Telephony Signaling*, Proc. of Photonics East, Boston, Massachusetts, U.S.A., September 1999.

17. Bernard Pagurek and Tony White, *A Quick Evaluation of H.323/H.450*, Technical Report SCE-99-02, Department of Systems and Computer Engineering, Carleton University, April 1999.
18. Jonathan Rosenberg, Jonathan Lennox, and Henning Schulzrinne, *Programming Internet Telephony Services*, IEEE Network, vol.13, no.3, pp.42-49, May/June 1999.
19. Xiaotao Wu and Henning Schulzrinne, *Where Should Services Reside in Internet Telephony Systems?* Proc. of the IP Telecom Services Workshop 2000, pp.39-44, Atlanta, U.S.A, September 2000.
20. Greg Bond, Eric Cheung, Andrew Forrest, Michael Jackson, Hal Purdy, Chris Ramming, and Pamela Zave, *DFC as the basis for ECLIPSE, an IP communications software platform*, Proc. of the IP Telecom Services Workshop 2000, pp.19-25, Atlanta, U.S.A, September 2000.
21. Pamela Zave and Michael Jackson, *DFC Modifications I: Routing extensions*, AT&T Technical Memorandum HA1640000-990715-10TM.
22. Pamela Zave and Michael Jackson, *DFC Modifications II: Protocol extensions*, AT&T Technical Memorandum HA1640000-991119-16.
23. Michael Jackson and Pamela Zave, *Media implementation in ECLIPSE I: Principles of signaling/media separation*, AT&T Technical Memorandum HA1640000-000808-9TM.
24. Greg Bond, Eric Cheung, Andrew Forrest, Michael Jackson, Hal Purdy, Chris Ramming, Xiaotao Wu, and Pamela Zave, *The 12 Virtues of Modularity*, IP Telecom Services Workshop 2000, Atlanta, U.S.A, September 2000.
25. Michael Jackson and Pamela Zave, *The DFC Manual*, August 2001, online at: <http://www.research.att.com/~pamela/dfc.html>.

26. Java Remote Method Invocation (RMI) website,
<http://java.sun.com/products/jdk/rmi/>.
27. Tom Taylor, *Megaco/H.248: A New Standard for Media Gateway Control*, IEEE Communications Magazine, p.124, October 2000.
28. Greg Bond, Franjo Ivancic, Nils Klarlund, and Richard Trefler, *ECLIPSE Feature Logic Analysis*, 2nd International IP-Tel, New York City, U.S.A, April 2001.
29. Object Management Group, *OMG Unified Modeling Language Specification, version 1.4*, Object Management Group, September 2001, online at:
<http://www.omg.org/technology/documents/formal/uml.htm>.
30. Hassan Gomaa, *Designing Concurrent Distributed, and Real-Time Applications with UML*, Addison Wesley, 2000.
31. Martin Fowler and Kendall Scott, *UML Distilled*, Addison Wesley, 2000.
32. David Harel and Eran Gery, *Executable Object Modeling with Statecharts*, IEEE Computer, vol.30, no.7, pp.31-42, July 1997.
33. Ruiguo Li, *The Deployment, Design and Implementation of Voice Mail Feature based on DFC*, Master thesis, Department of Systems and Computer Engineering, Carleton University, September 2002.
34. Pamela Zave, *An Architecture for Three Challenging Features*, 2nd International IP-Tel, New York City, U.S.A, April 2001.
35. Dongyang Zhang, *The Deployment of Features in Internet Telephony*, Master thesis, Department of Systems and Computer Engineering, Carleton University, January 2002.

APPENDIX A BUILT-IN FSMs RELATING TO THE VM FEATURE

(1) *TransparentFSM*

TransparentFSM (LinkablePort port1, LinkablePort port2)

Figure AA.1 Statechart for TransparentFSM

The *TransparentFSM* (Figure AA.1) simply propagates *DFCProtocol* Messages between two linkable ports.

(2) *Open2LinksFSM*

The *Open2LinksFSM* (Figure AA.2) is an FSM fragment that sets up bi-directional media links between two local signal path terminations. This is a relatively common thing to do to in a feature box. The *openerSPT* is the signal path termination associated with the port that receives an *Open* message. The *openeesPT* is the signal path termination that the *Open* request is to be propagated to. The FSM first attempts to open a media channel with its peer who has sent the *Open* message, then attempts to open media links between its two SPTs. The FSM ends in one of two states: *LINK_OPENED* or *LINK_UNOPENED*. In the former state, all has gone well. In the latter state, an *ONack* message is received from the *openeesPT*, so the links cannot be opened. The event numbers shown on this figure correspond to those on Figure 5.5-5.8.

Open2LinksFSM (LocalSignalPathTermination openerSPT, LocalSignalPathTermination openeeSPT,
LinkablePort opener, LinkablePort openee,
OpenMessage openMessage)

Figure AA.2 Statechart for Open2LinksFSM

(3) InitialTransparent2LinksFSM

InitialTransparent2LinksFSM (DBox box, DualPort sub, DualPort par)

Figure AA.3 Statechart for InitialTransparent2LinksFSM

The *InitialTransparent2LinksFSM* (Figure AA.3) is an FSM fragment that sets up a single media connection between two dual ports, port *sub* and port *par*. This is a relatively common thing to do to start up a feature box. The fragment initializes the two ports so that the port connected to the subscriber side of the box (port *sub*) is distinguished from the port connected to the other side of the box (port *par*) regardless of

whether the box appears in the subscriber's source or target zone. Once the signaling channel has been established, the FSM transparently propagates messages between the two ports. When an *Open* message arrives from the caller, the FSM attempts to establish a media channel and links between the two ports. If this fails, then the FSM reverts to the transparent behavior until the next *Open* message arrives from the caller. If this succeeds, then the FSM transitions to its *LINK_OPENED* end state. While in this state, the FSM will propagate messages between the two ports and attempt to establish subsequent media channel and links between the two ports upon receiving *Open* messages. The arrival of an *Upnack* message causes the FSM to transition from the *LINK_CALLER* state to the *CALLER_UNLINKED* end state.

(4) Transparent2LinksFSM

The *Transparent2LinksFSM* (Figure AA.4) is an FSM fragment that sets up media connections between two linkable ports, port *sub* and port *par*. This is a relatively common thing to do in a feature box. The FSM transparently propagates messages between the two ports. When an *Open* message arrives on a port, the FSM attempts to establish media links between the two ports. Signal path terminations with the same name are created on both ports to establish the links. If an *ONack* message is received from the *openeePort*, then the link attempt fails and the FSM reverts to the transparent behavior until the next *Open* message arrives. Also the signal path terminations that were created to establish the links are destroyed. If the link attempt succeeds, then the FSM reverts to the transparent behavior until the next *Open* message arrives.

Transparent2LinksFSM (LinkablePort port1, LinkablePort port2)

Figure AA.4 Statechart for Transparent2LinksFSM

(5) *Open2LinksOnPortsFSM*

The *Open2LinksOnPortsFSM* (Figure AA.5) is an FSM fragments that open two links between the signal path terminations with the same name on port1 and port2. Signal path terminations with different names are ignored.

Open2LinksOnPortsFSM (ChannelPort port1, ChannelPort port2)

Figure AA.5 Statechart for Open2LinksOnPortsFSM

(6) *SignalCallFSM*

The *SignalCallFSM* (Figure 6.4) attempts to set up a signaling connection with a downstream feature box or line interface box. Before the creation of this FSM, its super FSM has already sent out the *Setup* message. Thus in the initial *Dialing* state, this FSM is waiting for the arrival of the acknowledgement messages. If an *Upack* message arrives, the FSM transitions to the *LINKED* state. Alternatively, if an *Upnack* message arrives, the FSM transitions to the *DISCONNECTED* state. In the *LINKED* state, if it receives an *Avail* message indicating that a signaling path to the target interface is available, the FSM enters into the *AVAIL* state. In the *LINKED* state, the arrival of a *Teardown* message indicating the interface's attempt to tear down the call, an *Unavail* message indicating

that the interface is temporarily unavailable, or an *Unknown* message indicating that a routing error occurs, will all make the FSM enter into the *DISCONNECTED* state.

(7) *MediaCallFSM*

The *MediaCallFSM* (Figure AA.6) attempts to open a media connection to a downstream feature box or line interface box. The initial *OPEN* state changes to the *AVAIL* state through an internal transition. During this transition, the call port sends out an *Open* message (could be the signal path termination of the call port. If sent out by the SPT of a port, then the type of media do not need to be specified since the SPT is only for a participant type of media).

In the *AVAIL* state, the FSM is waiting for the acknowledgement messages. If a negative acknowledgement message (*ONack*) arrives, it means that the attempt to open a media connection fails. The *DISCONNECTED* end state is then entered. If a positive acknowledgement message (*OAck*) arrives, it means that the media connection is established. Thus the *OPENED* state is entered. Before entering this state, a *Ready* message is sent out through the SPT of the call port.

In the *OPENED* state, the FSM will keep on waiting for subsequent media status messages. The arrival of the *Accept* message, indicating that the media connection is fully accepted, causes the FSM to transition to the *CONNECTED* end state. The arrival of the *Reject* message, indicating that the request to open a media connection has been rejected, causes the FSM to transition to the *REJECTED* end state.

MediaCallFSM (LocalSignalPathTermination callSPT) **Figure AA.6** Statechart for MediaCallFSM**(8) OpenExtChanFSM****OpenExtChanFSM** (ExternalSignalPathTermination externalTermination, UDP udp, String channelName) **Figure AA.7** Statechart for OpenExtChanFSM

The *OpenExtChanFSM* (Figure AA.7) is used to open an external media channel to the external device, such as a telephone or a resource. It has three simple states.

In the initial *START* state, the box sends an *OpenExtChan* message which contains a channel name and local media information (udp's port number and host address) to its Mbox through the external termination. The FSM then transitions to the *WAIT_EXT_ACK* state.

In the *WAIT_EXT_ACK* state, the FSM waits for an acknowledgement message from the Mbox. After receiving the *OpenExtChanAck* message which means that the external media channel has been opened successfully, the FSM extracts the remote media information about the external device and stores them. These media information will be used later when the external devices communicate with each other in real time. The FSM then transitions to the *END* state.

(9) AnswerFSM

The *AnswerFSM* (Figure AA.8) responds to an internal call's initiation and attempts to open a media channel to the caller. Before the creation of this FSM, the box must have already received a *Setup* message. Thus in its initial *START* state, the FSM sends out an *Avail* message through the dual port *answerPort* and transitions to the *UNOPENED* state.

In the *UNOPENED* state, it waits for further instructions about the media information. The arrival of the *Open* message from the caller makes the FSM transition to the *MEDIA_ANSWER* state, in which a media channel to the caller will be opened. The nested *MediaAnswerFSM* is responsible for that. If the media channel is opened

successfully, indicated by the *MediaAnswerFSM*'s entry into the *CONNECTED* state, the super *AnswerFSM* will enter the *CONNECTED* state. If the attempt to open the media channel is failed, a corresponding *DISCONNECTED* state is entered.

AnswerFSM (DualPort answerPort, LocalSignalPathTermination answerSPT)

Figure AA.8 Statechart for AnswerFSM

(10) *MediaAnswerFSM*

The *MediaAnswerFSM* (Figure AA.9) is a FSM fragment which attempts to open a media channel to the caller. In the initial *START* state, the FSM sends out an *OpenChan* message to its Mbox through the signal path termination *answerSPT*. If it receives an acknowledgement message *OpenChanAck* from the Mbox, indicating that a media channel has been opened successfully, it transitions to the *OACK_WAIT* state. During the transition, an *Oack* message is sent back to the caller. The FSM may end in one of the two states: *CONNECTED* or *DISCONNECTED*.

Figure AA.9 Statechart for MediaAnswerFSM

On entry into the *CONNECTED* state, the FSM immediately sends out an *Accept* message, indicating a full media channel has been established. On the other hand, a *Reject* message is sent out on entry into the *DISCONNECTED* state, indicating that the request to open a media channel has been rejected.

As how to reach the *LOCAL_ALERTING*, *CONNECTED*, and *DISCONNECTED* states, the FSM does not define them. This gives the user of this FSM the ability to define their own transitions in the parent FSM of this state, and will make the FSM more general to be used, since different boxes have different actions associated with the above states.

APPENDIX B A SAMPLE OF THE CODE OF A FSM

Definition of the *ReceiveVMBoxFSM* class

```
package com.att.eclipse.featureBoxes.receiveVM;
import com.att.eclipse.featureBoxes.fragments.*;
:
:

public class ReceiveVMBoxFSM extends FSM {

    // declare FSM state labels
    public static final FSMStateLabel START = new FSMStateLabel("START");
    public static final FSMStateLabel CALLEE_OPENED = new
        FSMStateLabel("CALLEE_OPENED");
    :
    :

    // declare private variables
    private UserDBox box;
    private BoxPort boxPort;
    private DualPort subscriber;
    private DualPort participant;
    :
    :

    // declare port evaluators for port aliases used in message transitions
    private FSMPortEvaluator boxPortEvaluator;
    private FSMPortEvaluator subscriberEvaluator;
    private FSMPortEvaluator participantEvaluator;
    :
    :

    public ReceiveVMBoxFSM(UserDBox pbox,
        DualPort psubscriber,
        DualPort pparticipant,
        DualPort pplayer,
        LocalSignalPathTermination pplayerSPT,
        DualPort precorder,
        LocalSignalPathTermination precorderSPT,
        OperData pfdata) throws Exception {

        box = pbox;
        boxPort = box.getBoxPort();
        subscriber = psubscriber;
        :
        :

        // define port evaluators for port aliases
        boxPortEvaluator = new FSMPortEvaluator() {
            public Port getPort() { return boxPort; };
        };
        subscriberEvaluator = new FSMPortEvaluator() {
```

```

        public Port getPort() { return subscriber; };
        :
        :

// add states
addState(new FSMNestedState(START, new FSMEvaluator() {
    public FSM createFSM() {
        return new InitializeTransparent2LinksFSMV2(box,
                                                    subscriber,
                                                    participant);
    }
}));
addState(new FSMState(CALLEE_OPENED) {});
:
:

setInitialState(START);

// add transitions
addInternalTransition(new FSMInternalTransition
    (new FSMNestedStateLabel(new FSMStateLabel[]
        {START, InitializeTransparent2LinksFSMV2.CALLER_UNLINKED_STATE}),
        CONNECT_PLAYER) {
    public void runAction() throws Exception {
        Object returnVal = (Object) getReturnVal();
        setup2 = (SetupMessage) returnVal;
        callee_address = setup2.getTargetAddress();
        caller_address = setup2.getSourceAddress();
        Address newTarget = new Address(player_address);
        setup3 = SetupMessage.updateSetup(setup2, newTarget,
            SetupMessage.UPDATE_TARGET);
        player.output(setup3);
    }
});
:
:

addMessageTransition(new FSMMessageTransition(
    participantSPTevaluator,
    ReadyMessage.class,
    CALLEE_OPENED,
    WAITING_CALLEE) {
    public void runAction(PortMessage pm) throws Exception {
        getSubscriberSPT().output(new ReadyMessage());
    }
});
:
:
}
}

```

APPENDIX C TRACE FILES FOR THE TWO TEST CASES

For each test case, there are two trace files. One trace file records all message transmissions of the primary router and its associated boxes (*Andrew* and *Lillian's* LI boxes, resource interface boxes, feature boxes) running on it. The other trace file records the message transmissions of the standalone line interface controller and its associated boxes (*Eric's* LI box, local resource interface boxes, feature boxes). Message transmissions inside *DualTeardownFSM* have not been traced since too many ports create this FSM. Some *SignalCallFSMs* are used by the phone GUIs and are marked. The numbers attached to the end of each line are message sequence numbers appearing in Figure 5.5-5.9.

(1) Trace files for test case one- the VM feature:

Trace file for the primary router:

Primary router: carleton is started...

```
/* TEST: LEAVING MESSAGE ON BUSY */
/* andrew receives a call from eric */
Lazy cleanup of precedence change
Lazy cleanup of precedence change
Receive: Hi, andrew, I'm eric. How are you? //media message received by andrew from eric
/* lillian places a call to eric */
Lazy cleanup of precedence change
*****ReceiveVMBoxControllerVersion is started...
InitializeTransparent2LinksFSMV2 box receives SetupMessage //1.1b
InitializeTransparent2LinksFSMV2 caller sends UpackMessage //1.3b
InitializeTransparent2LinksFSMV2 callee sends SetupMessage //1.2b
SignalCallFSM callPort receives UpackMessage //related to lillian's phoneGui FSM
InitializeTransparent2LinksFSMV2 callee receives UpackMessage //1.5b
TransparentFSM port1 receives AvailMessage //2.4b
TransparentFSM port2 sends AvailMessage //2.5b
SignalCallFSM callPort receives AvailMessage //related to lillian's phoneGui FSM
InitializeTransparent2LinksFSMV2 caller receives OpenMessage //2.6b
Open2LinksFSM openee sends OpenMessage //2.7 b
Open2LinksFSM openee receives OAckMessage //2.10b
Open2LinksFSM opener sends OAckMessage //2.11b
TransparentFSM port2 receives ReadyMessage //2.12b
```

```
TransparentFSM port1 sends ReadyMessage //2.13b
TransparentFSM port1 receives WaitMessage //2.16b
TransparentFSM port2 sends WaitMessage //2.17b
TransparentFSM port1 receives AcceptMessage //2.19b
TransparentFSM port2 sends AcceptMessage //2.20b
Receive: Hi, this is Eric. Please leave a message after the beep: //greeting message received by lillian
//from eric'VM
Send: Eric, this is Lillian. Please call me back when you have time. //lillian leaves a message
ReceiveVMBBoxFSMControllerVersion par receives TeardownMessage //3.11b
```

/* TEST: LEAVING MESSAGE ON NO ANSWER */

/* lillian places a call to eric */

*****ReceiveVMBBoxControllerVersion is started...

```
InitializeTransparent2LinksFSMV2 box receives SetupMessage //1.1c
InitializeTransparent2LinksFSMV2 caller sends UpackMessage //1.3c
InitializeTransparent2LinksFSMV2 callee sends SetupMessage //1.2c
SignalCallFSM callPort receives UpackMessage //related to lillian's phoneGui FSM
InitializeTransparent2LinksFSMV2 callee receives UpackMessage //1.5c
TransparentFSM port1 receives AvailMessage //1.8c
TransparentFSM port2 sends AvailMessage //1.9c
SignalCallFSM callPort receives AvailMessage //related to lillian's phoneGui FSM
InitializeTransparent2LinksFSMV2 caller receives OpenMessage //1.10c
Open2LinksFSM openee sends OpenMessage //1.11c
Open2LinksFSM openee receives OAckMessage //1.14c
Open2LinksFSM opener sends OAckMessage //1.15c
TransparentFSM port2 receives ReadyMessage //1.16c
TransparentFSM port1 sends ReadyMessage //1.17c
TransparentFSM port1 receives WaitMessage //1.20c
TransparentFSM port2 sends WaitMessage //1.21c
TransparentFSM port1 receives AcceptMessage //2.9c
TransparentFSM port2 sends AcceptMessage //2.10c
Receive: Hi, this is Eric. Please leave a message after the beep: //greeting message received by lillian
//from eric'VM
Send: Lillian again. Call me. //lillian leaves a message
ReceiveVMBBoxFSMControllerVersion par receives TeardownMessage //3.11b
```

/* TEST: CALLEE PICKS UP THE PHONE */

/* lillian places a call to eric */

*****ReceiveVMBBoxControllerVersion is started...

```
InitializeTransparent2LinksFSMV2 box receives SetupMessage //1.1d
InitializeTransparent2LinksFSMV2 caller sends UpackMessage //1.3d
InitializeTransparent2LinksFSMV2 callee sends SetupMessage //1.2d
SignalCallFSM callPort receives UpackMessage //related to lillian's phoneGui FSM
InitializeTransparent2LinksFSMV2 callee receives UpackMessage //1.5d
TransparentFSM port1 receives AvailMessage //1.8d
TransparentFSM port2 sends AvailMessage //1.9d
SignalCallFSM callPort receives AvailMessage //related to lillian's phoneGui FSM
InitializeTransparent2LinksFSMV2 caller receives OpenMessage //1.10d
Open2LinksFSM openee sends OpenMessage //1.11d
Open2LinksFSM openee receives OAckMessage //1.14d
Open2LinksFSM opener sends OAckMessage //1.15d
TransparentFSM port2 receives ReadyMessage //1.16d
TransparentFSM port1 sends ReadyMessage //1.17d
TransparentFSM port1 receives WaitMessage //1.20d
TransparentFSM port2 sends WaitMessage //1.21d
TransparentFSM port1 receives AcceptMessage //1.24d
TransparentFSM port2 sends AcceptMessage //1.25d
Receive: hi, I'm Eric. Are you lillian? //media message received by lillian from eric
```


ReceiveVMBoxFSMControllerVersion sub receives TeardownMessage

/* TEST: LEAVING MESSAGE ON OFFLINE */

/* lillian places a call to eric */

*****ReceiveVMBoxControllerVersion is started...

InitializeTransparent2LinksFSMV2 box receives SetupMessage //1.1a

InitializeTransparent2LinksFSMV2 caller sends UpackMessage //1.3a

InitializeTransparent2LinksFSMV2 callee sends SetupMessage //1.2a

SignalCallFSM callPort receives UpackMessage //related to lillian's phoneGui FSM

InitializeTransparent2LinksFSMV2 callee receives UpackMessage //1.4a

InitializeTransparent2LinksFSMV2 callee receives UnknownMessage //1.5a

ReceiveVMBoxFSMControllerVersion player sends SetupMessage //2.1a

Lazy cleanup of precedence change

ConnectResourceFSM callee receives UpackMessage //2.2a

TransparentFSM port2 receives AvailMessage //2.3a

TransparentFSM port1 sends AvailMessage //2.4a

SignalCallFSM callPort receives AvailMessage //related to lillian's phoneGui FSM

ConnectResourceFSM caller receives OpenMessage //2.5a

Open2LinksFSM openee sends OpenMessage //2.6a

Open2LinksFSM openee receives OAckMessage //2.7a

Open2LinksFSM opener sends OAckMessage //2.8a

TransparentFSM port1 receives ReadyMessage //2.9a

TransparentFSM port2 sends ReadyMessage //2.10a

TransparentFSM port2 receives WaitMessage //2.11a

TransparentFSM port1 sends WaitMessage //2.12a

ConnectResourceFSM callee receives AcceptMessage //2.13a

ConnectResourceFSM caller sends AcceptMessage //2.14a

ReceiveVMBoxFSMControllerVersion playerSPT sends UserAddressMessage //2.15a

Send: This is eric. Please leave a message after the tone: //greeting message of eric

ReceiveVMBoxFSMControllerVersion playerSPT receives BeepMessage //2.16a

ReceiveVMBoxFSMControllerVersion recorder sends SetupMessage //3.1a

ReceiveVMBoxFSMControllerVersion player sends TeardownMessage //2.17a

Receive: This is eric. Please leave a message after the tone: //greeting message received by lillian from eric's VM

Lazy cleanup of precedence change

SignalCallFSM callPort receives UpackMessage //3.2a

SignalCallFSM callPort receives AvailMessage //3.3a

MediaCallNoAcceptFSM callSPT sends OpenMessage //3.4a

MediaCallNoAcceptFSM callSPT receives OAckMessage //3.5a

MediaCallNoAcceptFSM callSPT sends ReadyMessage //3.6a

MediaCallNoAcceptFSM callSPT receives WaitMessage //3.7a

ReceiveVMBoxFSMControllerVersion recorderSPT receives AcceptMessage //3.8a

ReceiveVMBoxFSMControllerVersion recorderSPT sends UserAddressMessage //3.9a

ReceiveVMBoxFSMControllerVersion recorderSPT sends CallerAddressMessage //3.10a

Send: Hi, eric, come to meet around 1:00pm at the bookstore. //lillian leaves a message

Receive: Hi, eric, come to meet around 1:00pm at the bookstore. //message left lillian and received by eric's VM

ReceiveVMBoxFSMControllerVersion parSPT receives TeardownMessage //3.11a

Trace file for eric's end system:

Line interface controller: ericController is started...

load class com.att.eclipse.lineinterfaceBoxes.phoneGui.PhoneGuiBox from local drive... cannot find code locally.

load class com.att.eclipse.lineinterfaceBoxes.phoneGui.PhoneGuiBox from network... successful.

load class com.att.eclipse.resourceInterfaceBoxes.player.LocalPlayerBox from local drive... successful.

load class com.att.eclipse.resourceInterfaceBoxes.recorder.LocalRecorderBox from local drive... successful.

/* TEST: LEAVING MESSAGE ON BUSY */

/* eric places a call to andrew */

```

Lazy cleanup of precedence change
SignalCallFSM callPort receives UpackMessage // related to eric's phoneGui FSM
SignalCallFSM callPort receives AvailMessage // related to eric's phoneGui FSM
Send: Hi, andrew, I'm eric. How are you? // media message send by eric to andrew
/* lillian places a call to eric */
load class com.att.eclipse.featureBoxes.receiveVM.ReceiveVMBox from local drive... cannot find code locally.
load class com.att.eclipse.featureBoxes.receiveVM.ReceiveVMBox from network... successful.
*****ReceiveVMBox is started...
InitializeTransparent2LinksFSMV2 box receives SetupMessage //1.2b
InitializeTransparent2LinksFSMV2 caller sends UpackMessage //1.5b
InitializeTransparent2LinksFSMV2 callee sends SetupMessage //1.4b
InitializeTransparent2LinksFSMV2 callee receives UpackMessage //1.6b
ReceiveVMBoxFSM player sends SetupMessage //2.1b
Lazy cleanup of precedence change
ConnectResourceFSM callee receives UpackMessage //2.2b
TransparentFSM port2 receives AvailMessage //2.3b
TransparentFSM port1 sends AvailMessage //2.4b
ConnectResourceFSM caller receives OpenMessage //2.7b
Open2LinksFSM openee sends OpenMessage //2.8b
Open2LinksFSM openee receives OAckMessage //2.9b
Open2LinksFSM opener sends OAckMessage //2.10b
TransparentFSM port1 receives ReadyMessage //2.13b
TransparentFSM port2 sends ReadyMessage //2.14b
TransparentFSM port2 receives WaitMessage //2.15b
TransparentFSM port1 sends WaitMessage //2.16b
ConnectResourceFSM callee receives AcceptMessage //2.18b
ConnectResourceFSM caller sends AcceptMessage //2.19b
ReceiveVMBoxFSM playerSPT sends UserAddressMessage //2.21b
Send: Hi, this is Eric. Please leave a message after the beep: //greeting message of eric
ReceiveVMBoxFSM playerSPT receives BeepMessage //2.22b
ReceiveVMBoxFSM recorder sends SetupMessage //3.1b
ReceiveVMBoxFSM player sends TeardownMessage //2.23b
Lazy cleanup of precedence change
SignalCallFSM callPort receives UpackMessage //3.2b
SignalCallFSM callPort receives AvailMessage //3.3b
MediaCallNoAcceptFSM callSPT sends OpenMessage //3.4b
MediaCallNoAcceptFSM callSPT receives OAckMessage //3.5b
MediaCallNoAcceptFSM callSPT sends ReadyMessage //3.6b
MediaCallNoAcceptFSM callSPT receives WaitMessage //3.7b
ReceiveVMBoxFSM recorderSPT receives AcceptMessage //3.8b
ReceiveVMBoxFSM recorderSPT sends UserAddressMessage //3.9b
ReceiveVMBoxFSM recorderSPT sends CallerAddressMessage //3.10b
Receive: Eric, this is Lillian. Please call me back when you have time. //message left lillian and received
//by eric's VM
ReceiveVMBoxFSM parSPT receives TeardownMessage //3.12b

/* TEST: LEAVING MESSAGE ON NO ANSWER */
/* lillian places a call to eric */
*****ReceiveVMBox is started...
InitializeTransparent2LinksFSMV2 box receives SetupMessage //1.2c
InitializeTransparent2LinksFSMV2 caller sends UpackMessage //1.5c
InitializeTransparent2LinksFSMV2 callee sends SetupMessage //1.4c
InitializeTransparent2LinksFSMV2 callee receives UpackMessage //1.6c
TransparentFSM port1 receives AvailMessage //1.7c
TransparentFSM port2 sends AvailMessage //1.8c
InitializeTransparent2LinksFSMV2 caller receives OpenMessage //1.11c
Open2LinksFSM openee sends OpenMessage //1.12c
Open2LinksFSM openee receives OAckMessage //1.13c

```

Open2LinksFSM opener sends OAckMessage //1.14c
 ReceiveVMBoxFSM parSPT receives ReadyMessage //1.17c
 ReceiveVMBoxFSM subSPT sends ReadyMessage //1.18c
 ReceiveVMBoxFSM subSPT receives WaitMessage //1.19c
 ReceiveVMBoxFSM parSPT sends WaitMessage //1.20c
 ReceiveVMBoxFSM box sends SetTimerMessage //1.22c
 ReceiveVMBoxFSM box receives TimeoutMessage //1.23c
 ReceiveVMBoxFSM player sends SetupMessage //2.1c
 ReceiveVMBoxFSM sub sends TeardownMessage //1.24c
 SignalCallFSM callPort receives UpackMessage //2.2c
 SignalCallFSM callPort receives AvailMessage //2.3c
 MediaCallNoAcceptFSM callSPT sends OpenMessage //2.4c
 MediaCallNoAcceptFSM callSPT receives OAckMessage //2.5c
 MediaCallNoAcceptFSM callSPT sends ReadyMessage //2.6c
 MediaCallNoAcceptFSM callSPT receives WaitMessage //2.7c
 ReceiveVMBoxFSM playerSPT receives AcceptMessage //2.8c
 ReceiveVMBoxFSM parSPT sends AcceptMessage //2.9c
 ReceiveVMBoxFSM playerSPT sends UserAddressMessage //2.11c
 Send: Hi, this is Eric. Please leave a message after the beep: //greeting message of eric
 ReceiveVMBoxFSM playerSPT receives BeepMessage //2.12c
 ReceiveVMBoxFSM recorder sends SetupMessage //3.1b
 ReceiveVMBoxFSM player sends TeardownMessage //2.13c
 SignalCallFSM callPort receives UpackMessage //3.2b
 SignalCallFSM callPort receives AvailMessage //3.3b
 MediaCallNoAcceptFSM callSPT sends OpenMessage //3.4b
 MediaCallNoAcceptFSM callSPT receives OAckMessage //3.5b
 MediaCallNoAcceptFSM callSPT sends ReadyMessage //3.6b
 MediaCallNoAcceptFSM callSPT receives WaitMessage //3.7b
 ReceiveVMBoxFSM recorderSPT receives AcceptMessage //3.8b
 ReceiveVMBoxFSM recorderSPT sends UserAddressMessage //3.9b
 ReceiveVMBoxFSM recorderSPT sends CallerAddressMessage //3.10b
 Receive: Lillian again. Call me. //message left lillian and received by eric's VM
 ReceiveVMBoxFSM parSPT receives TeardownMessage //3.12b

/* TEST: CALLEE PICKS UP THE PHONE */

/* lillian places a call to eric */

*****ReceiveVMBox is started...

InitializeTransparent2LinksFSMV2 box receives SetupMessage //1.2d
 InitializeTransparent2LinksFSMV2 caller sends UpackMessage //1.5d
 InitializeTransparent2LinksFSMV2 callee sends SetupMessage //1.4d
 InitializeTransparent2LinksFSMV2 callee receives UpackMessage //1.6d
 TransparentFSM port1 receives AvailMessage //1.7d
 TransparentFSM port2 sends AvailMessage //1.8d
 InitializeTransparent2LinksFSMV2 caller receives OpenMessage //1.11d
 Open2LinksFSM openee sends OpenMessage //1.12d
 Open2LinksFSM openee receives OAckMessage //1.13d
 Open2LinksFSM opener sends OAckMessage //1.14d
 ReceiveVMBoxFSM parSPT receives ReadyMessage //1.17d
 ReceiveVMBoxFSM subSPT sends ReadyMessage //1.18d
 ReceiveVMBoxFSM subSPT receives WaitMessage //1.19d
 ReceiveVMBoxFSM parSPT sends WaitMessage //1.20d
 ReceiveVMBoxFSM box sends SetTimerMessage //1.22d
 ReceiveVMBoxFSM subSPT receives AcceptMessage //1.23d
 ReceiveVMBoxFSM parSPT sends AcceptMessage //1.24d
 ReceiveVMBoxFSM box sends CancelTimerMessage //1.26d
 Send: hi, I'm Eric. Are you lillian? // media message send by eric to lillian
 ReceiveVMBoxFSM subSPT receives TeardownMessage

```

/* TEST: RETRIEVING MESSAGES */
/* eric places a call to local player */
Lazy cleanup of precedence change
Lazy cleanup of precedence change
load class com.att.eclipse.featureBoxes.retrieveVM.RetrieveVMBox from local drive... cannot find code locally.
load class com.att.eclipse.featureBoxes.retrieveVM.RetrieveVMBox from network... successful.
*****RetrieveVMBox is started...
InitializeTransparent2LinksFSMV2 box receives SetupMessage 1e
InitializeTransparent2LinksFSMV2 caller sends UpackMessage 3e
InitializeTransparent2LinksFSMV2 callee sends SetupMessage 2e
SignalCallFSM callPort receives UpackMessage // related to eric's phoneGui FSM
InitializeTransparent2LinksFSMV2 callee receives UpackMessage 4e
TransparentFSM port1 receives AvailMessage //5e
TransparentFSM port2 sends AvailMessage //6e
SignalCallFSM callPort receives AvailMessage // related to eric's phoneGui FSM
InitializeTransparent2LinksFSMV2 caller receives OpenMessage //7e
Open2LinksFSM openee sends OpenMessage //8e
Open2LinksFSM openee receives OAckMessage //9e
Open2LinksFSM opener sends OAckMessage //10e
TransparentFSM port1 receives ReadyMessage //11e
TransparentFSM port2 sends ReadyMessage //12e
TransparentFSM port2 receives WaitMessage //13e
TransparentFSM port1 sends WaitMessage //14e
RetrieveVMBoxFSM subSPT receives AcceptMessage //15e
RetrieveVMBoxFSM parSPT sends AcceptMessage //17e
RetrieveVMBoxFSM subSPT sends RetrieveMessagesMessage //17e
Send: Please enter your password followed by # key: // password prompt sent by local player
Receive: Please enter your password followed by # key: //password prompt received by eric
RetrieveVMBoxFSM par receives DTMFMessage: 1 //18e
RetrieveVMBoxFSM sub sends DTMFMessage: 1 //19e
RetrieveVMBoxFSM par receives DTMFMessage: 2 //18e
RetrieveVMBoxFSM sub sends DTMFMessage: 2 //19e
RetrieveVMBoxFSM par receives DTMFMessage: 3 //18e
RetrieveVMBoxFSM sub sends DTMFMessage: 3 //19e
RetrieveVMBoxFSM par receives DTMFMessage: 4 //18e
RetrieveVMBoxFSM sub sends DTMFMessage: 4 //19e
RetrieveVMBoxFSM par receives DTMFMessage: # //18e
RetrieveVMBoxFSM sub sends DTMFMessage: # //19e
Send: You have 1 new messages, 6 in total. Press 1 to listen to new messages; Press 2 to listen to all messages;
Press * to exit //sent by local player
Receive: You have 1 new messages, 6 in total. Press 1 to listen to new messages; Press 2 to listen to all messages;
Press * to exit //received by eric
RetrieveVMBoxFSM par receives DTMFMessage: 1 //18e
RetrieveVMBoxFSM sub sends DTMFMessage: 1 //19e
Send: Message 1: lillian 05/23/02 19:26:16 Second local message. Press 3 to keep the message; Press 4 to delete
the message; Press * to exit
Receive: Message 1: lillian 05/23/02 19:26:16 Second local message. Press 3 to keep the message; Press 4 to
delete the message; Press * to exit
RetrieveVMBoxFSM par receives DTMFMessage: 3 //18e
RetrieveVMBoxFSM sub sends DTMFMessage: 3 //19e
Send: End of messages; Press * to exit //18e
Receive: End of messages; Press * to exit //19e
RetrieveVMBoxFSM par receives DTMFMessage: * //18e
RetrieveVMBoxFSM sub sends DTMFMessage: * //19e
RetrieveVMBoxFSM sub receives TeardownMessage //20e

/* TEST: RECORDING GREETING MESSAGES */
/* eric places a call to local recorder */

```

```

Lazy cleanup of precedence change
load class com.att.eclipse.featureBoxes.recordGreetingVM.RecordGreetingVMBox from local drive... cannot find
code locally.
load class com.att.eclipse.featureBoxes.recordGreetingVM.RecordGreetingVMBox from network... successful.
*****RecordGreetingVMBox is started...
InitializeTransparent2LinksFSMV2 box receives SetupMessage
InitializeTransparent2LinksFSMV2 caller sends UpackMessage
InitializeTransparent2LinksFSMV2 callee sends SetupMessage
SignalCallFSM callPort receives UpackMessage
InitializeTransparent2LinksFSMV2 callee receives UpackMessage
TransparentFSM port1 receives AvailMessage
TransparentFSM port2 sends AvailMessage
SignalCallFSM callPort receives AvailMessage
InitializeTransparent2LinksFSMV2 caller receives OpenMessage
Open2LinksFSM openee sends OpenMessage
Open2LinksFSM openee receives OAckMessage
Open2LinksFSM opener sends OAckMessage
TransparentFSM port1 receives ReadyMessage
TransparentFSM port2 sends ReadyMessage
TransparentFSM port2 receives WaitMessage
TransparentFSM port1 sends WaitMessage
RecordGreetingVMBoxFSM subSPT receives AcceptMessage
RecordGreetingVMBoxFSM parSPT sends AcceptMessage
RecordGreetingVMBoxFSM subSPT sends RecordingPromptMessage
Send: Please enter your password followed by # key: // password prompt sent by local player
Receive: Please enter your password followed by # key: //password prompt received by eric
RetrieveVMBoxFSM par receives DTMFMessage: 1
RetrieveVMBoxFSM sub sends DTMFMessage: 1
RetrieveVMBoxFSM par receives DTMFMessage: 2
RetrieveVMBoxFSM sub sends DTMFMessage: 2
RetrieveVMBoxFSM par receives DTMFMessage: 3
RetrieveVMBoxFSM sub sends DTMFMessage: 3
RetrieveVMBoxFSM par receives DTMFMessage: 4
RetrieveVMBoxFSM sub sends DTMFMessage: 4
RetrieveVMBoxFSM par receives DTMFMessage: #
RetrieveVMBoxFSM sub sends DTMFMessage: #
Send: Start recording your message: //sent by local recorder
Receive: Start recording your message: //receives by eric
Send: Eric is unavailable. Please leave a message after the beep: //new greeting message sent by eric
Receive: Eric is unavailable. Please leave a message after the beep: //new greeting message received
//by local recorder

RecordGreetingVMBoxFSM par receives TeardownMessage
RecordGreetingVMBoxFSM sub sends TeardownMessage

```

(2) Trace files for test case two – the VM and CW features

Trace file for the primary router:

Primary router: carleton is started...

/* andrew receives a call from eric. */

Lazy cleanup of precedence change

Lazy cleanup of precedence change

Receive: Hi, andrew. This is eric. How are you? //media message received by andrew from eric

/* lillian places a call to eric. */

Lazy cleanup of precedence change
 *****ReceiveVMBoxControllerVersion started...
 ReceiveVMBoxControllerVersion is in transparent state
 Receive: Eric is unavailable now. Please leave a message after the beep: //greeting message received by lillian
 //from eric'VM
 Send: Hi. eric. This is lillian. Call me when you have time. //lillian leaves a message

/* lillian places a call to eric again */
 *****ReceiveVMBoxControllerVersion started...
 ReceiveVMBoxControllerVersion is in transparent state
 Receive: Hi, eric is speaking. Are you lillian? //media message received by lillian from eric
 Receive: Andrew. I'm talking with lillian too. //media message received by andrew from eric

Trace file for eric's end system:

Line interface controller: ericController is started...
 load class com.att.eclipse.lineInterfaceBoxes.phoneGui.PhoneGuiBox from local drive... cannot find code locally.
 load class com.att.eclipse.lineInterfaceBoxes.phoneGui.PhoneGuiBox from network... successful.
 load class com.att.eclipse.resourceInterfaceBoxes.player.LocalPlayerBox from local drive... successful.
 load class com.att.eclipse.resourceInterfaceBoxes.recorder.LocalRecorderBox from local drive... successful.

/* eric places a call to andrew */
 Lazy cleanup of precedence change
 load class com.att.eclipse.featureBoxes.callWaiting.CallWaitingBox from local drive... cannot find code locally.
 load class com.att.eclipse.featureBoxes.callWaiting.CallWaitingBox from network... successful.
 *****CallWaitingBox is started...
 Send: Hi, andrew. This is eric. How are you? //media message send by eric to andrew

/* lillian places a call to eric */
 load class com.att.eclipse.featureBoxes.receiveVM.ReceiveVMBox from local drive... cannot find code locally.
 load class com.att.eclipse.featureBoxes.receiveVM.ReceiveVMBox from network... successful.
 *****ReceiveVMBox started...
 Lazy cleanup of precedence change
 ReceiveVMBox is in playing greeting message state
 Send: Eric is unavailable now. Please leave a message after the beep: //greeting message of eric
 Lazy cleanup of precedence change
 ReceiveVMBox is in recording caller's message state
 Receive: Hi. eric. This is lillian. Call me when you have time. //lillian leaves a message

/* lillian places a call to eric again */
 *****ReceiveVMBox started...
 CallWaitingBox receives a flash message
 ReceiveVMBox is in transparent state
 Send: Hi, eric is speaking. Are you lillian? //media message send by eric to lillian
 CallWaitingBox receives a flash message
 Send: Andrew. I'm talking with lillian too. //media message send by eric to andrew