Vrije Universiteit Brussel - Belgium

Faculty of Sciences

In Collaboration with Ecole des Mines de Nantes - France

2001

Aspect Oriented Programming for Features in Telecom Applications

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Victor Hugo Arroyo Ibañez

Promoter: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promoter: Rémi Dounce (Ecole des Mines de Nantes)

To the Memory of M. Fernanda Aguirre.

Contents

Chapter 1

Introduction

Phone systems have changed dramatically over the last years. Many people (including feature developers) are not aware of the underlying complexity in the concrete system and, as a way of simplifying the problem, often make incorrect assumptions based on their knowledge of the plain old telephone service.

The Plain Old Telephony Service (POTS in the rest of this thesis) cannot be always re-engineered for economical or technical reasons, because it is a huge service, working twenty four hours over twenty four, every day, in which is not possible shut-down the system, make the changes and turn-on again. The industry has invested great effort in developing software-switching system, many of which are extremely precarious and were developed without the advantages of the modern software and current technologies. The fundamental source of problems derives from the changing system requirements (concretely the features integration). Feature are additional services, which must interact with POTS and other features already integrated. In practical terms, featured interaction exists when a feature interact with another feature behavior and the quality of the services may be compromised. It make relevant that an adequate management must consider itself as be able to detect those interactions and resolve them in a suitable way.

This Thesis demonstrates how much difficult is definition, description and implementation of POTS.

This thesis follows a logical order in which, a first part will be exposed different approaches for modelling and describing POTS, features and features interaction, giving both informal description, formal description when it is possible, and uses cases. Next, in the second part it will try to give a new point of view for the feature implementation problematic using AspectJ. The methodology will follow the same logical sequence, that mean, modelling and building a POTS framework in a Java RMI environment, next modelling and building some features and finally implementing a the proper feature interaction.

Part I

State of the Art

Chapter 2

Plain Old Telephony Service

In this chapter we exposed different ways to specify POTS. These different approaches will be taken into account in order to build a simple but realistic POTS model in chapter 6.

2.1 Informal Description

POTS is the legacy software used to provide the basic service which is to allow the communications between two clients. The process of establishing the communication is not a simple process even if it is seem quite intuitive. In fact, in order to establish a simple communication between clients many conditions must be verified, many actions must be coordinated. The whole process can be seen as a sequence of hidden minor subprocess, and these hidden subprocess are responsible for the actual complexity.

These subprocess evolve not only the telephone central but the clients also. Additionally is necessary take in account the implicit complexity of a remote interaction. In the next subsection there is a way, at least, to avoid this complexity, resolving the interaction locally.

2.1.1 Enhance POTS

An agent-based service [HJ98] level is proposed in order to construct a uniform service environment enabling customized service control of network resources while maintaining strong separation of network and the service level. Formally at this part is not desirable see "services" as "features" in order to follow the logical order if this thesis, but in fact later will be seen that an enhanced POTS version is probably the best model to integrate "feature".

Services are implemented in agents, which encapsulate service significance (semantics), including specific data and the way to interpret the data. The agents operate in an agent environment which provides generic support as for creation, destruction and execution (in ideal cases) of agents. Both, the network primitives and the agent

environment provide an abstract remote interface, which separates services between the different levels at the network.

Assuming that the generic agent support, and the limited set of network primitives is the same across the network, service agents see a uniform service environment. The agents themselves represent autonomous entities that are mobile within the environment. Agents encapsulates the service semantics, the data and the way to interpret the data. An agent operates in a general purpose agent infrastructure in order to access network facilities through an interface to very basic primitives. The goal of a customer agent is to customize it for individual users. In particular, all user specific information like data, rules, etc is encapsulated in a corresponding customer agent.

2.2 Formal Description

In this section we expose how it is possible to define or at least describe the POTS Behavior. The two next subsection contain two approaches will be expose two approaches in details because is needed leave clear the basis of the Basic Process.

2.2.1 Behavior Description

[CR98] gives a formalization of the POTS behavior, described by an Non-Deterministic Automata (figure 2.1). It consists in the states which can be reached by the system. Each transition consists in input messages and output messages.

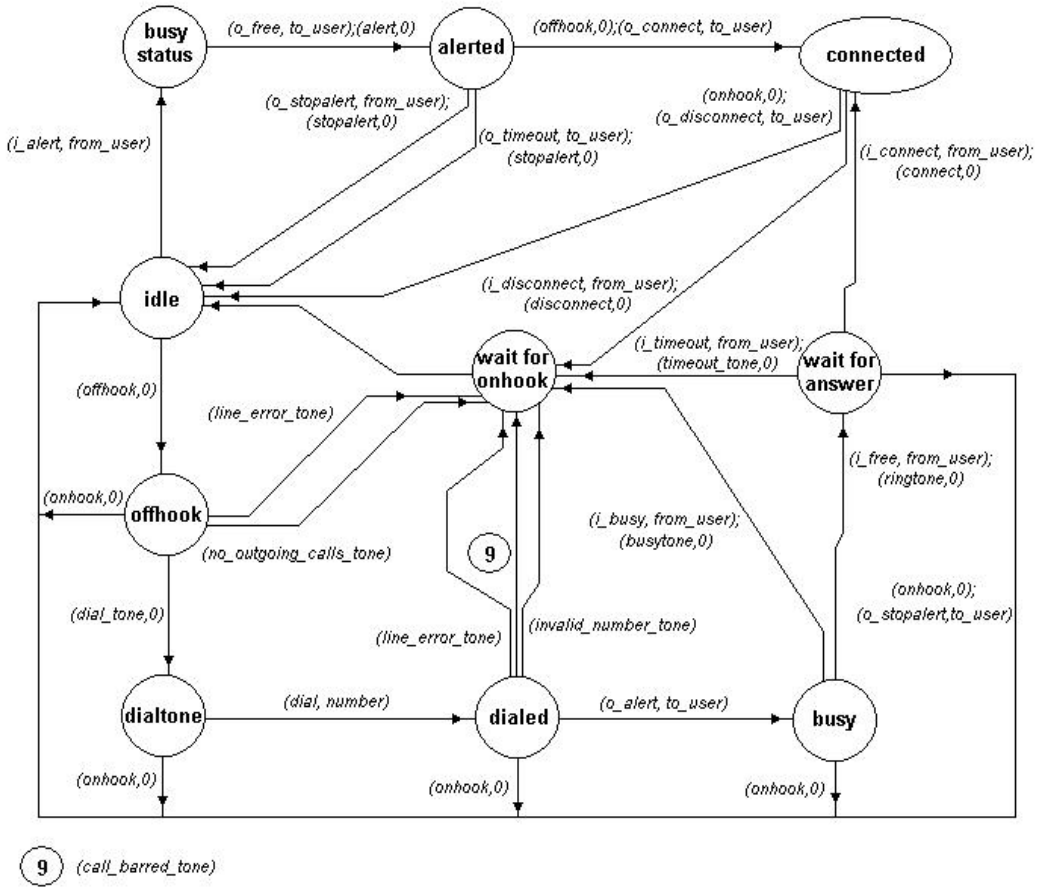

Figure 2.1: The Basic Call Process

The argument "*to_user*" is the user identifier or 0 if there is no argument needed (or not required). There are different messages separated by a semicolon , that means there is a sequential composition of messages. The state names enhance readability but they do not contain any relevant information. The whole behavior description is given by the input and output messages.

Let us consider the next example in which is exposed the process to connect then disconnect two clients. Let us suppose the clients are represented by 0 and 4. The client 0 is originally in the IDLE state and the client 4 is able to accept connection.

Example: The basic call software receive an ($offhook$,0) message to which it responds with (*dialtone*,0). Next a (*dial*,4) message trigger the sending of (o_alert,4), like the client 4 is able the message (i_free) is sent.Assume user 4 goes $offhook$ which leads to an ($i\_connect$,4) message being received, triggering (*connect*,0) being sent. Now the two users are connected. The Basic call software receive an (*onhook*,0) message indicating the connection is to be cleared down, triggering an ($o\_disconnect\_request$,4) message being sent. Then the basic call software is in its

initial state (IDLE) and a new call can be done.

2.2.2 Description of Basic Call Service in FI

This section exposes another formal description of POTS Behavior,(an algebraic definition). FI [CAdR99] is concretely a specification language for "Logic Description", FI defines a knowledge base for presenting a formal description of POTS Behavior. This is important because it defines a concrete interaction between clients.

As first step, let us define SUB as a finite set of indices which represent subscribers to the telephone system.

Atomics Concepts

Atomics Concepts are needed to express that a user u living in SUB ($u \in SUB$) is in a concrete state. Those concepts form the set ST_u of all possible states which can be reached by u.

- $idle_u$ the telephone has the receiver on hook and is silent.
- $dialtone_u$ the receiver is $offhook$ and emits a dial tone; a number can be dialed.
- $busytone_u$ the receiver is $offhook$ and emits a busy tone, indicating a failed call attempt or the party has hang up.
- $ringing_u$ the telephone is ringing with the receiver on hook.
- $ringback_u$ the receiver is off hook and emits a ring back tone (called party's phone is ringing).
- $engaged_u$ there is a connection also called speech path with another party.

Is necessary define some concepts to express a presence of the network, because a call proceeds through phases that are not directly observable by the user. In that sense is needed express the connections between users active. Then, for the subscriber u and v it is possible to define a set called IST_{uv} with the next states. $calling_{uv}$ the phone at v is ringing with u waiting for v to accept the call $path_{uv}$ u and v can communicate, there is a speech path.

Atomics Roles

Atomics Roles represents possible actions of each subscribers. Let us define the set called R as the union of all roles concerns for u, (with $u \in SUB$), where $roles_u$ is:

- $offhook_u$ represents the action of u lifting the receiver.
- $dial_{uv}$ represents the action of u dialing v (where $v \in SUB$)

- $onhook_u$ represents the action of u putting down the receiver

It is standard to assume that the system is deterministic, in another words, any action of a subscriber changes the state of the system to a unique and completely specified new state.

$TBox$ and $ABox$

Just at this part there are already defined atomics concepts and atomics roles, so it is enough to begin to specify POTS using "Logic Description". $TBoxes$ and $ABoxes$ are two kinds of description logics .$ABoxes$ contains assertible information, and $TBoxes$ terminological information:

- $TBox : T = \{man \sqsubseteq human \sqcap male,$
  $father \doteq man \sqcap \exists has - child.T\}$
- $ABox : A = \{m : man,$
  $e : male,$
  $(m, e) : has - child\}$

At the beginning the expression ($TBox$) connects the observable states of a telephone with the networks states. Of course taking the same consideration about u and v living in SUB ($u, v \in SUB$).

- $calling_{uv} \sqsubseteq ringing_v \sqcap ringback_u, u \neq v$
- $ringing_u \sqsubseteq calling_{vu}, u \neq v$
- $ringback_u \sqsubseteq callling_{uv}, u \neq vpath_{uv} \sqsubseteq engaged_u \sqcap engaged_v$
- $engaged_u \sqsubseteq path_{uv} u \neq vpath_{uv} = path_{vu}, u \neq v$

Next, there are statements specifying how a user and the network can change state.

- $idle_u \sqsubseteq \exists offhook_u.dialtone_u$ (if u is $idle$, it can go $offhook$ and accept digits).
- $dialtone_u \sqcap idel_u \sqsubseteq \exists dial_{uv}.calling_{uv}$ (if u has a $dialtone$ and $vidle$, u can dial v and establish a call).
- $dialtone_u \sqsubseteq \exists onhook_u.idle_u$ (if u has a $dialtone$, it can decide to hang up and go idle).
- $dialtone_u \sqcap -idle_v \sqsubseteq \exists dial_{uv}.busytone_u$ (if u party is busy, u will emits a busy tone).
- $busytone_u \sqsubseteq \exists onhook_u.idle_u$ (if u has a $busytone$, it can go $onhook$ to become $idle$).

- $calling_{uv} \sqsubseteq \exists offhook_v.path_{uv}$ (when u is calling v and v goes $offhook$, this will establish a speech path between them).
- $calling_{uv} \sqsubseteq \exists onhook_u.(idle_u \sqcap idle_v)$ (if u goes $onhook$, it and its party can go $idle$)
- $path_{uv} \sqsubseteq \exists onhook_u.(idle_u \sqcap busytone_v)$ (u can go $onhook$ when talking to v).

This definition makes it possible to define universal restrictions. The initial state of the whole system is defined by an $ABox$ statement:

$$s_0 : \bigcup idle_u,\ u \in SUB$$

Next we must specify that each subscriber is in one and only in one state at each moments:

$$T \sqsubseteq \neg(\bigsqcup_{s1,s2 \in ST_u, s1 \neq s2}(s1 \cap s2)) \sqcap \bigsqcup_{s \in ST_u} s.$$

Second, subscriber can change its state only by means of certain actions. As the subscribers and allowed transitions are finite sets it is possible to define (well define) the condition for these actions. Let us define:

$$D_{uv} \text{ as } \{idle_u, dialtone_u, busytone, ringing_u \sqcap calling_{vu}, calling_{uv}, path_u v\}$$

$(u, v \in SUB)$

Now, it is possible to define the mapping between Act_{uv} from D_{uv} to the subsets of $ROLES_u \bigcup ROLES_v$ as a function specifying the allowed actions:

- $Act_{uv}(idle_u) = offhook_u \bigcup dial_{vu} | v \in SUB, v \neq u$
- $Act_{uv}(dialtone_u) = onhook_u \bigcup dial_{uv} | v \in SUB, v \neq u$
- $Act_{uv}(busytone_u) = onhook_u$
- $Act_{uv}(ringing_u \sqcap callingv_u) = onhook_v$
- $Act_{uv}(calling_{uv}) = onhook_u, offhook_v$
- $Act_{uv}(path_{uv}) = onhook_u, onhook_v$

and this complete the POTS in FI.

2.2.3 Uses Cases

This section present two technics for POTS Uses Cases. The first one is an approach based on graphical representation (or maps). The second one follows the idea given in [CR98] and [DAW99] using Linear Temporal Logic.

Use Case Maps(UCM)

Use Case Maps are used to emphasize abstract sequencing (called causal paths) among the most relevant, interesting, and critical functionalities of reactive and distributed system, which are composed of responsibilities. $UCMs$ can represent specific scenarios, as well as abstract or generic ones and can cover multiples scenario instances. $UCMs$ are often highly reusable.

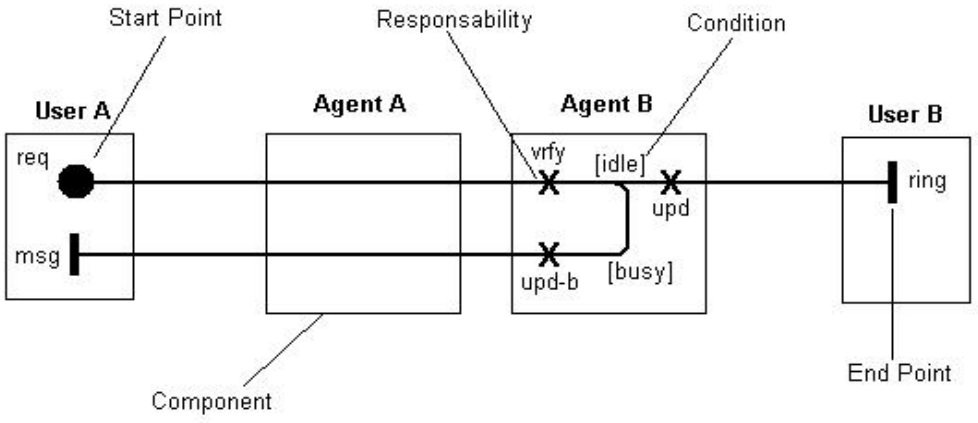

Figure 2.2: A Simple Use Case Maps

The last diagram (figure 2.2) shows a simple UCM where a user A attempts to establish a telephone communication with the user B through network using agents. User A sends a connection request (req) to the network through her agent. This request causes the called agent to verify ($vrfy$) whether the called party is idle or busy. If he is, then there will be some status updated (upd) and a ring signal will be activated over user B side with a $ring$. Otherwise, a different update will occur ($upd-b$) and an appropriate message (notifying that user B is not available) will be prepared and sent back to User A (by msg).

A scenario starts with a *start* point which represent a triggering event that can be associated with certain pre-conditions (the filled circle labelled req) and end with one or more *endpoint*, representing resulting events that can be associated with a certain post condition (bars), in this case $ring$ or msg. Intermediate responsibilities ($vrfy$,upd,$upd-b$) have been activated along the way. A *causalpath* goes from a start point to an end point. In this example, the responsibilities are allocated to abstract (or generic) components (boxes A and B, Agent A and Agent B), which could be seen as object, process, agents, databases or even roles, actors or persons. This last point becomes relevant at the moment of different features integration.

Use Case Maps can be refined in terms of Messages Sequence Charts (MSC) or UML diagrams. $UCMs$ do not explicitly define messages exchanges between components. In the following diagram it is possible to appreciate the correspondence between

Uses Case Maps and Message Sequence Charts. It is important to keep in mind this kind of transformation, in order to facilitate the implementation.

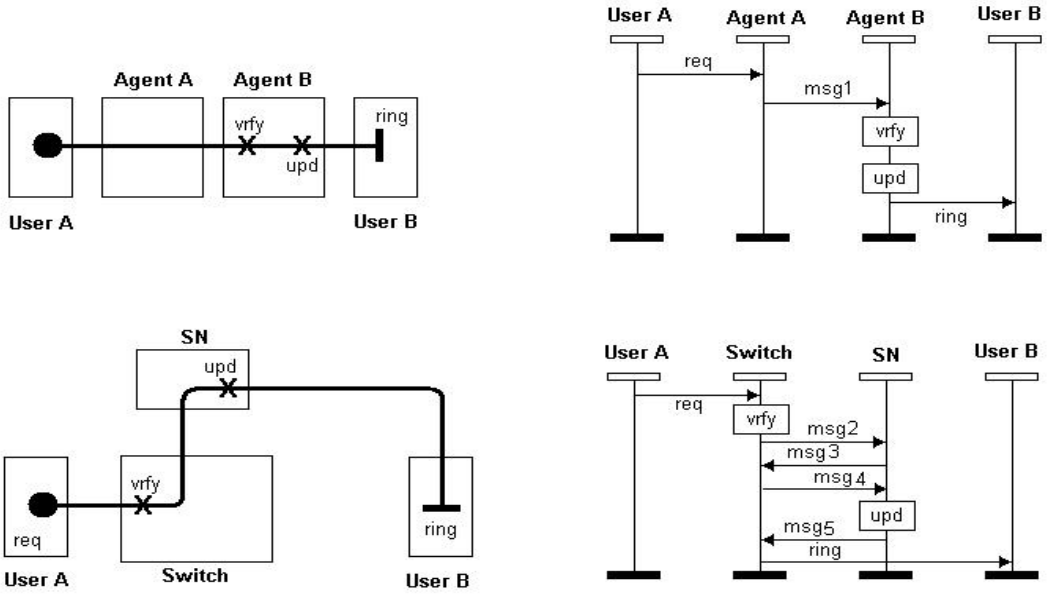

Figure 2.3: Use Case Maps and Message Sequence Charts

The causal path $< req, vrfy, upd, ring >$, which represents a successful scenario extracted from the first diagram (figure 2.2), at the top of the second diagram (figure 2.3), where implicit communication exists between users and their respective agents, and between agents. $UCMs$ allows designer to describe functionalities even when messages are not known in advance, for example when complex negotiations are involved between agents. $UCMs$ enable one to reuse the same paths on different architectural alternatives. Since they can be easily decoupled from structure, UCM paths improve the reusability of scenario and lead to behavior patterns that can be used across a wide range of applications. On many occasions $UCMs$ may provide helpful visual pattern that stimulate thinking and discussion about system issues and that may be reused.

Uses Cases using Linear Temporal Logic (LTL)

Linear Temporal Logic is a technique in which process are expressed in a chronological and ordered way, capturing the interdependencies between them. Let us consider the next operators:

operator $\Box$ means "always".

operator $\Diamond$ means "eventually".

The main point is reasoning about the system include the validations of the "expected behavior" versus the "observer behavior". The expected behavior is defined by abstract properties, like they are expressed in the automata exposed in the last section, (figure 2.1). And the observed behavior is expressed in terms of the observation of the system.
Now, let us see how some uses cases are expressed in order to respect (in this case) this two assumptions:

- if user A calls user A, then user A should receive a busy-tone.
- if there are no faults with the lines and user A is allowed to originate calls, user A calls B and user B is not busy, then eventually users A and B can be connected.

Given this two assumption the cases are:

Case 1 if ($dial$,1) is sent, then eventually a ($busy\_tone$,0) is sent.

$$\Box((dial,1) \rightarrow \Diamond(busy\_tone,0))$$

Case 2 if ($dial$,2) is sent, and neither of ($line\_error\_tone$,0), ($busy\_tone$,0), ($call\_barred\_tone$,0), ($no\_outgoing\_calls\_tone$,0) or ($i\_timeout\_request$,2), is sent then eventually a ($connect$,0) is sent.

$$\Diamond((dial,2) \wedge\neg\ (line\_error\_tone,0) \wedge\neg\ (busy\_tone,0) \wedge\neg(call\_barred\_tone,0) \wedge\neg(no\_outgoing\_calls\_tone,0)\wedge\neg(i\_timeout\_request,2)) \rightarrow \Diamond\ (connect,0))$$

This two uses cases have been based using the transitions defined in the automata showed in a previous section (picture 2.1).

2.2.4 Summary

In this chapter we have exposed two formal description about POTS, the first one (section 2.2.1) shows an automata which describe the switching system behavior; the second one (section 2.2.2) shows how model the POTS behavior using a knowledge base and refinements. Additionally some techniques to express uses cases.

Chapter 3

Features

In this chapter, first we introduce informally telecom features. Then, we present a few studies about feature formal specification.

3.1 Informal Description

In this section, we introduce features with the help of two examples [JC00]. Then, we provide a list of common features. Finally, we discuss semantics of features.

3.1.1 Two Features

Features provide additional functionalities to the POTS. For example, the Termination Call Screen (TCS) feature allows a client to specify telephone numbers in order to forbid some incoming calls. The figure 3.1 shows two clients. The client B uses the TCS feature in order to screen call from the client A. When the client A calls B, the client B sends back a reject message.

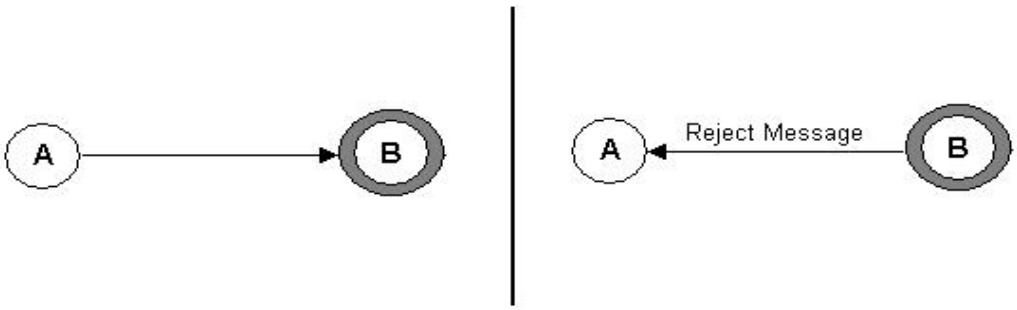

Figure 3.1: Termination Call Screen

In the POTS, a communication deals only with two clients. Some features deeply change the POTS functionalities and involve more than two clients. For instance, the

Call Forwarding (CF) feature allows a client to forward incoming calls to a third one. The figure 3.2 shows a scenario where the client B forwards its incoming calls to the client C. So, when A calls B, a communication is established between A and C.

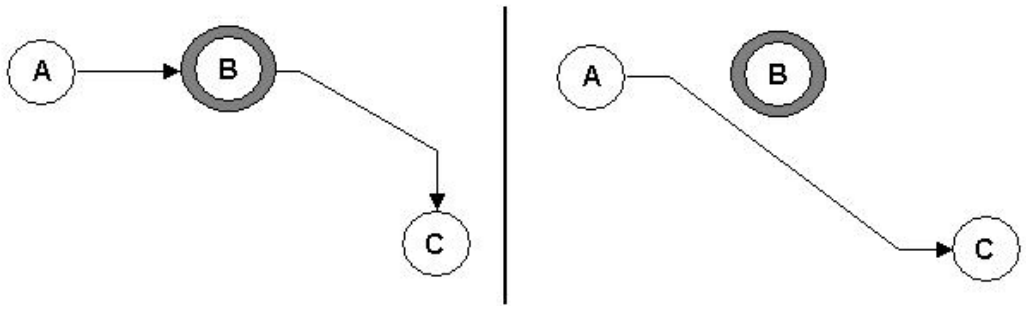

Figure 3.2: Call Forwarding

Note that the CF feature has two different versions. In the first version an incoming call is always forwarded (unconditional version). In the second version, an incoming call is forwarded when the client is busy (busy version).

3.1.2 Common Features

We now briefly review the most common features. This list is not complete, but it provides a good overview of features functionalities.

Call Waiting (CW) This feature allows a client to manage two different calls at the same time, but not simultaneously. When a client is already in communication and this client receives a second call, he can accept the second incoming call by leaving the first call in a waiting status. In that way a client can switch between two calls.

Three Way Calling (3WC) This feature allows a client to manage two different calls at same time and simultaneously. When a client is already in communication and this client receives a second call, he can accept the second incoming call by leaving the first call in a waiting status. When the second connection is established the first call will be integrated to the communication, then the three clients will share a common communication channel.

Answer Call This feature allows to the subscriber redirect all incoming call to an answer service, when he is busy.

Automatic Recall This feature allows the subscriber to triggers a new call if the first one has been rejected. That means, when an outgoing call has been rejected the number is stored to Recall as soon as the called side becomes free.

Originator Call Screen This feature allows the subscriber to specify some numbers in order to forbid all outgoing call to those already define numbers.

Operator Services This feature is an especial case, it may be handled in a remote switching element that have not access to the feature subscription of every customer who wishes to use the service. That mean every call made through Operator Services acts like an outgoing POTS call, except that it is assisted by the operator.

IN Teen Line Restrict outgoing calls based on the time of day.

Credit-Card Calling Instead of hanging up and dialing the long distant code (access code) some credits card calling services instruct to the caller to press the # key for placing another credit-card call.

Call Number Delivery This feature requires a special telephone that can show the caller's number before a call is answered. The user might decide not to answer a particular caller when the telephone rings.

Voice Mail Service This feature allows the subscriber to have access to voice mail messages. The subscriber calls to a service number and introduces his client's code in order to listen the messages.

Multiple Directory Number Line This feature allows to the subscriber have more than number associated with single line.

Automatic Call Back This allows the subscriber to call back an incoming call which has been previously rejected. That means that Call Back feature will store the last rejected incoming call in order call it as soon as become free.

Unlisted Number This feature allows to the subscriber to have a private number, which will not appears in the directory.

3.1.3 Features Semantics

Let us consider again the Call Forwarding feature previously exemplified 3.2. Its specification given in the last section was informal. This can generate unexpected behavior. For example, the figure 3.2 shows a scenario where the client B forwards its incoming calls to the client A. So, what is happening when A calls B? Either the client A gets a busy signal, or the system endlessly searches for the target client and loops forever.

This example proves that a simple feature can break the system down. So, the feature specification must be formally studied.

3.2 Features Specification

As exemplified in the previous section, a simple feature can dramatically modify the POTS behavior. So, it is necessary to have feature precise specifications. Such a specification could allow the feature designer to check consistency (e.g. a feature introduces no loop in the POTS) and can be used as a reference by the programmer.

First, let us focus on uses cases. Next, we present a formal specification based on logic.

3.2.1 Uses Cases

In this section we show how two Uses Case Maps techniques are used to integrate features. Note that this two techniques are not absolutely independents. Of course these are not the unique techniques but a similar model will be follow in the part of the implementation.

Uses Case Maps(UCM): Capturing Features

To introduce new features in the original model it is necessary to make some extension to the UCM notation. Root Map express the original model, but it is not enough to express a complete picture when a feature is working on. For this reason news structures called "plug-ins" are introduced. They are fact sub maps which, subsequently can be "plugged" in the root map.[RB99] [Tur00]

In the figure 3.3 is possible to appreciate that inside the Agents structures are "diamonds" draw, they are called "stub", and there are two kind of them:

Statics Stubs Represented as plain diamonds, they contain only one plug-in, hence enabling hierarchical decomposition of complex maps.

Dynamics Stubs Represented as dashed diamonds, they may contain several plugs-in, whose selection can be determinate at runtime according to a selection policy.

In the previous example we considered for instance the Termination Call Screening feature, but the next analysis is also validate for another feature.

Path segments coming in and coming out from stub have been identified on the root map. The Termination Call Screening plug-in improves the original UCM by allowing the update (upd) and the ring result to be accompanied, concurrently, by a ring-back signal to be prepared and sent back to the originating side (mrb). Concurrency is represented here by an AND-fork. The notation allows for alternative paths (OR-fork and OR-join, as in the Terminating plug-in), concurrent paths (AND-fork and AND-join). Shared responsibilities, exception paths, timers, failure points, error handling, and synchronous or asynchronous interactions between paths. This method of representation, using maps containing at different locations many types of stubs for

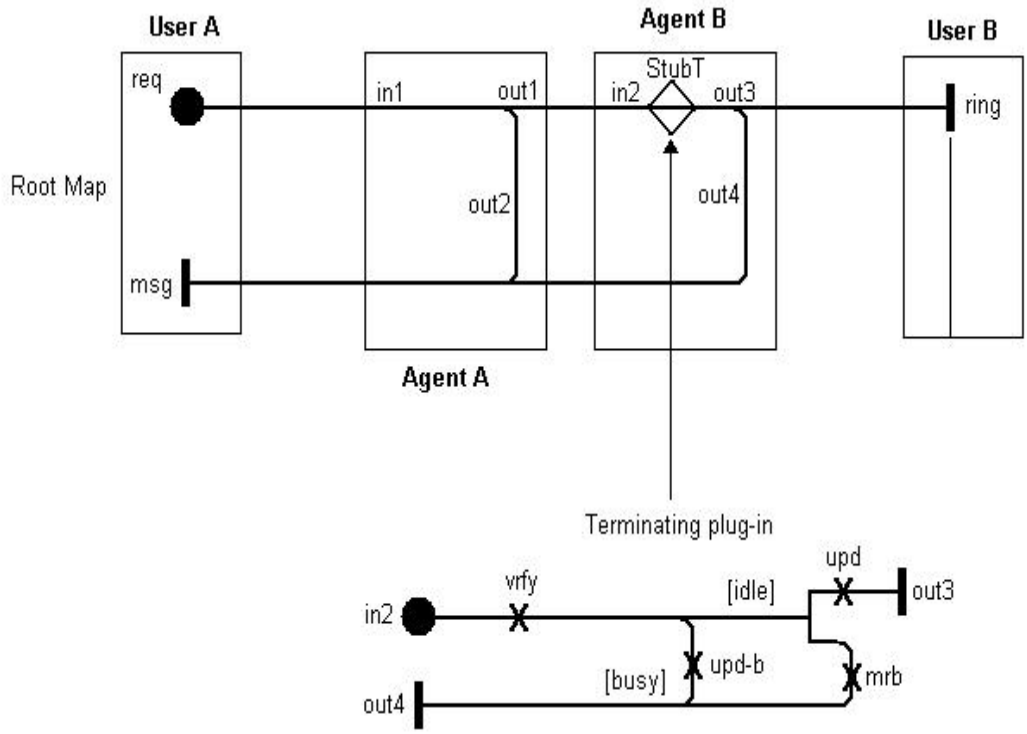

Figure 3.3: Stubs and Plug-ins

different features and other system characteristics, appears to be suitability general and flexible. [DAW99]

Integrating Features

As detailed in the last section, scenarios are useful for understanding the behavior of one feature, but they can also be integrated together to form a Global UCM. Integration helps to ensure early consistency between individual maps. In this sense, events or responsibilities that are labelled incorrectly or omitted at different level of abstraction could become hard to integrate, or effectible detects failures. A path segment that is a prefix to two different scenarios might imply the necessity for a way to decide which alternative to choose in a global scenario.[MF99] The figure 3.4 represent the global context in which sub-maps are plugged-in.

It is possible to construct a complete scenario by selecting appropriate plug-in for the stubs. Like in the las section, plug-in are bound to stubs by associating the entry and exit points of the stub with the start and end points of the plug-in map. The first stub in the room map, pre-dial, half one entry point (IN1), and two exit points (OUT1,OUT2). The pre-dial stub has two plugging, in which other features can be integrated is illustrated in the figure 3.4. The binding of the Feature (called FEATURE) plug-in is (FEATURE, IN1), (*goDial*,OUT1), (*goReject*,OUT2), figure

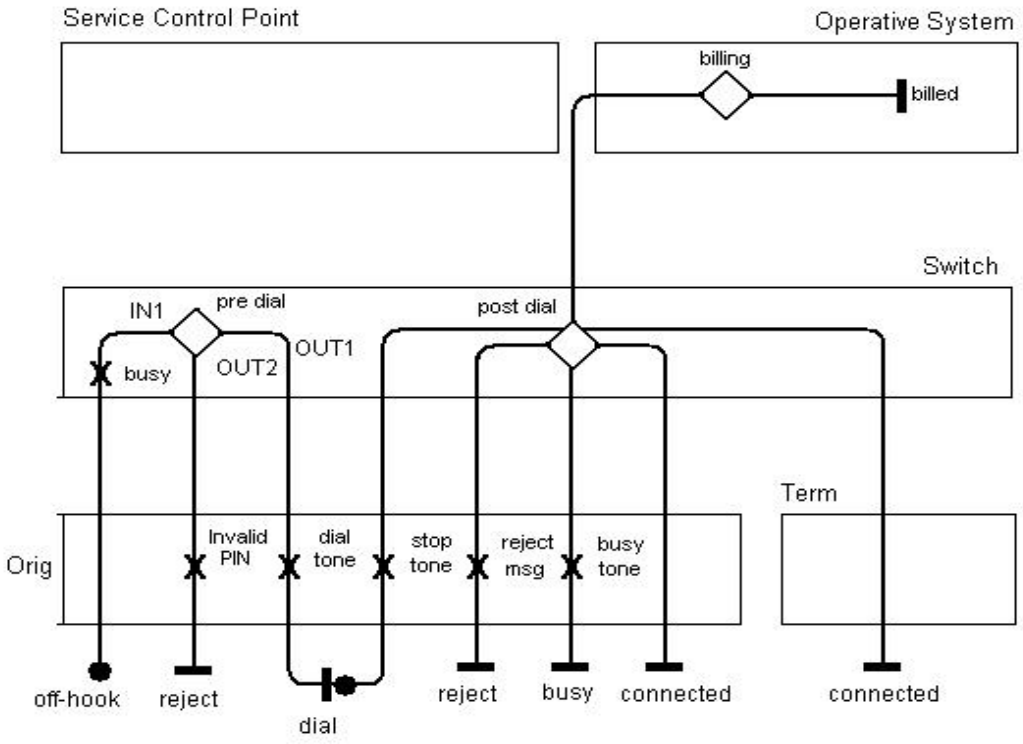

Figure 3.4: A global UCM Root Map

3.5.

The other plugging (default) for the post-dial stub is an empty path links IN1 to OUT1. In the case on IN Teen Line (INTL) when a subscriber uses it, the flattening of the root map with the INTL plug-in in the redial stub and default plug-in in the other stubs results in the individual UCM (figure 3.5).

3.2.2 Formal Definition

In the context of POTS modelled by a knowledge Base (see section 2.2.2), features are declared as extension of $POTS$ through refinements. This is important for the same reasons given in the section 2.2.2.

To clarify this idea, let us see some examples in [CAdR99] for Termination Call Screen (TCS) and Call Forward Unconditional (CFU).

TCS is a feature where a user u can put another user v on a screening list, that means all incoming calls from v will be not allowed to establish communication with the user u. A first attempt to formalize this behavior is by introducing a new concept called TCS_{uv} and simply refining $POTS$ by adding $TCS_{uv} \sqsubseteq \neg calling_{uv}$. Nevertheless this extension immediately interacts with itself in activation TCS_{uv}. This is defined by an expected interaction, because in fact, an extra feature could modify and hence

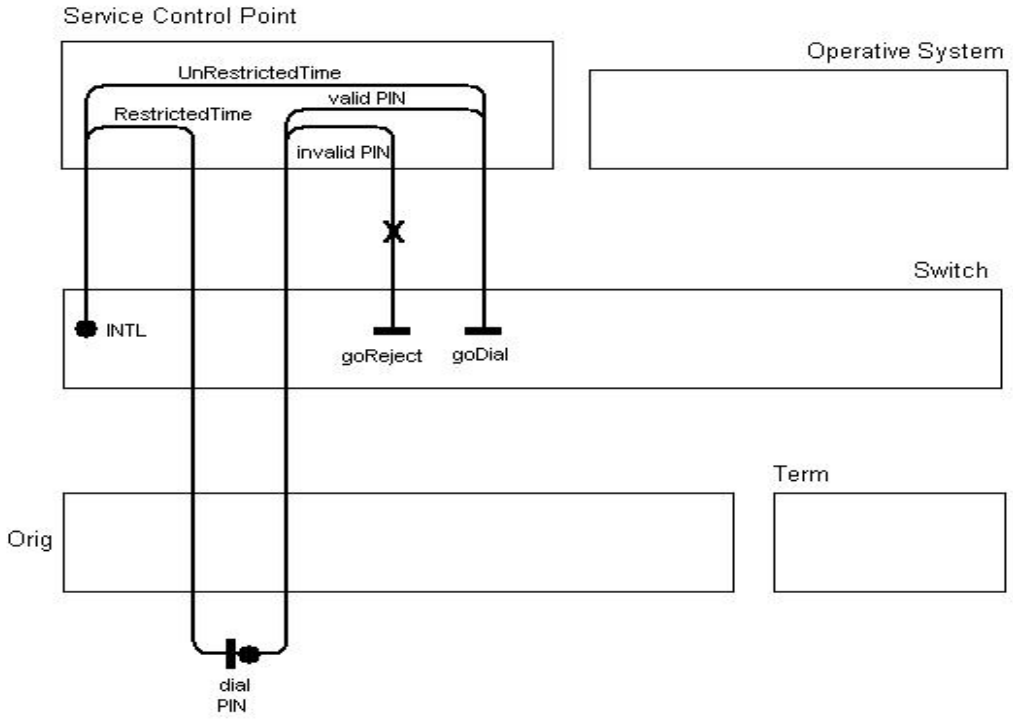

Figure 3.5: The INTL plug-in

contradictions, the basic system ($POTS$) (even if the basic system has been completely modelled). In order to obtain a correct definement, first is defined a set of activation concepts is defined for TCS: $TCS_{uv}u, v \in SUB, u \neq v$. The refinement δ_{tcs} is defined by replacing atomic Roles (see section 2.2.2), in any knowledge base.

Originally,$u, v \in SUB$, the role:

- $c1[dialtone_u \sqcap idle_v] \sqsubseteq \exists dial_{uv}.c2[calling_{uv}]$

is refined by:

- $c1[\neg TCS_{vu} \sqcap dialtone_u \sqcap idle_v] \sqsubseteq \exists dial_{uv}.c2[calling_{uv}]$,
- $c1[TCS_{vu} \sqcap dialtone_u \sqcap idle_v] \sqsubseteq \exists dial_{uv}.c2[busytone_u]TCS_{vu} \sqsubseteq calling_{uv}$

$C1$ and $C2$ are context, and in general $C[\phi]$ singles out an occurrence of phi as a sub formula of any formula χ.So in this case is import take care about δ_{tcs} be well defined by applying it only to knowledge bases where context single out unique occurrences.

CFU works as follow,let us consider CFU_{uv} means that whenever w call u, it will connected with v $(u, v, w \in SUB)$. So, for activation concepts the set$CFU_{uv}u, v \in SUB, u \neq v$. δ_{cfu} is formalized by replacing:

- $c1[dialtone_u \sqcap idle_v] \sqsubseteq \exists dial_{uv}.C2[calling_{uv}]$,

- $c3[dialtone_u \sqcap \neg idle_v] \sqsubseteq \exists dial_{uv}.c4[busytone]$

by

- $c1[\neg CFU_{vw} \sqcap dialtone_u \sqcap idle_v] \sqsubseteq \exists dial_{uv}.c2[calling_{uv}]$,
- $c1[CFU_{vw} \sqcap dialtone_u \sqcap idle_w] \sqsubseteq \exists dial.c2[calling_{uw}]$,
- $c3[\neg CFU_{vw} \sqcap dialtone_u \sqsubseteq \neg idle_v] \sqsubseteq \exists dial_{uv}.c4[busytone_u]$,
- $c3[CFU_{vw} \sqcap dialtone_u \sqcap \neg idle_w] \sqsubseteq \exists dial_{uv}.c4[busytone_u]$.

CFU affects also the definition of the Act function (see section 2.2.2), like if u has CFU_{uv}, so when w dial u (by $dial_{wu}$) it will not change its state because the call will be forwarded to v; modifying state (Act) appropriately is straightforward.

An interesting result from formal definition is the fact that specification "encodes" some policies, in this case, for example the number of forwards action is restricted to one. Forwarding event is implemented by changing the state $calling_{uv}$ to $calling_{uw}$ whenever CFU_{vw} is active, but this will bypasses the $dial_{uv}$ action, moreover is w has the —CFU_{wx} active (forwarding from w to x) this second forwarding will be not executed. In order to implement more multiple forwarding is necessary defined a new one refinement in $delta_{cfu}$, $forwarding_{uv}$. So it will replace:

- $c1[dialtone_u \sqcap idle_v] \sqsubseteq \exists dial_{uv}.C2[calling_{uv}]$,
- $c3[dialtone_u \sqcap \neg idle_v] \sqsubseteq \exists dial_{uv}.c4[busytone]$

by

- $c1[\neg CFU_{vw} \sqcap dialtone_u \sqcap idle_v] \sqsubseteq \exists dial_{uv}.c2[calling_{uv}]$,
- $c1[CFU_{vw} \sqcap dialtone_u] \sqsubseteq \exists dial_{uv}.c2[formarding_{uw}]$,
- $c3[\neg CFU_{vw} \sqcap dialtone_u \sqcap idle_v] \sqsubseteq \exists dial_{uv}.c4[busytone_u]$,
- $\neg CFU_{wx} \sqcap forwarding_{uw} \sqcap idle_w \sqsubseteq calling_{uv}$,
- $\neg CFU_{wx} \sqcap forwarding_{uw} \sqcap \neg idle_w \sqsubseteq busytone_v$,
- $CFU_{wx} \sqcap forwarding_{uw} \sqsubseteq forwarding_{ux}$.

With this new definition the number of "forwarding" actions is no more restricted, but is important take in account that infinite loops are also allowed.

In this kind of approaches is import remark that the new network states are not user state, that is important because the network model will be going to deal with multiple forwarding states until it can effectively reach a user which has not the call forwarding feature activated.

3.2.3 Summary

In this chapter we have exposed different features, giving an informal description about them (section 3.1); next, some specifications based in uses cases (sections 3.2.1 and 3.2.2). And finally it was showed a formal feature behavior expressed as knowledged base, taking the same nomenclature given in the previous chapter (section 2.2.2).

Chapter 4

Feature Interaction

Feature interaction are understood to be all interactions that interfere with the desired operation of the feature and that occur between a feature and its environment, including other features or other insurance of the same feature. Additionally, interference of a part of a feature, with another part of that feature is considered as Feature Interaction. The biggest problems is that the interaction is highly conflictive even if each feature involved in the interaction work properly. Let us see two particular cases:

- The picture 4.1 show a situation in which a client called A have both features, Originator Call Screen (OCS) and Call Forwarding Unconditional (CFU). Let us suppose Originator Call Screen deal with a telephone number X , ans it this number X has to be forwarded by A. In this case an autocall from A to A, to be automatically forwarded to X.

- The picture 4.2 show a situation in which a client called A have Automatic Call Back (ACB) feature, a client called B have the Automatic Recall (ARC), if B call A, and A is busy, B will store the $A's$ number for a recall, and A will store the $B's$ number to call back, so when A finish the previous connection will try to call back to B, but B at the same time will try to recall to A.

In both cases each feature work, fine in isolation, nevertheless the interaction produce a result not desirable.

In this chapter we expose existing approaches to study interaction. It is important to remark that all these studies will guide us in part II in order to implement the connect behavior (feature interaction).

4.1 Informal description

The problems of feature interaction has created difficulties in the process of service deployment. Recent and foreseeable changes in the telecommunications industry complicate the problem further. [JC00]

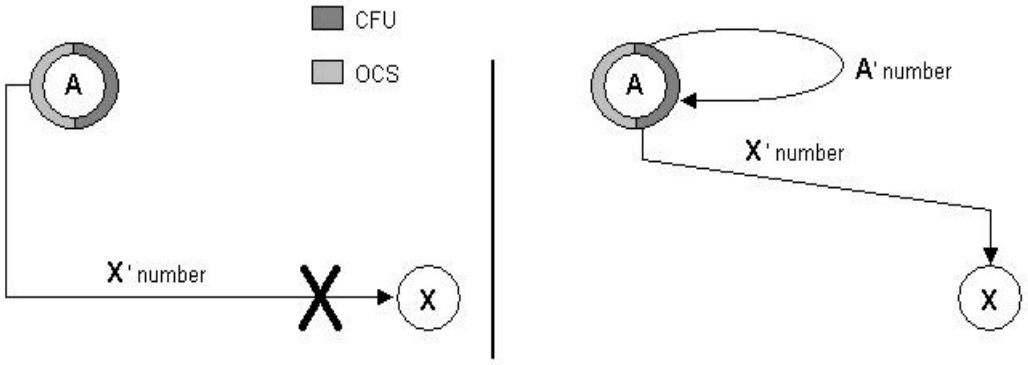

Figure 4.1: Basic Connect Model

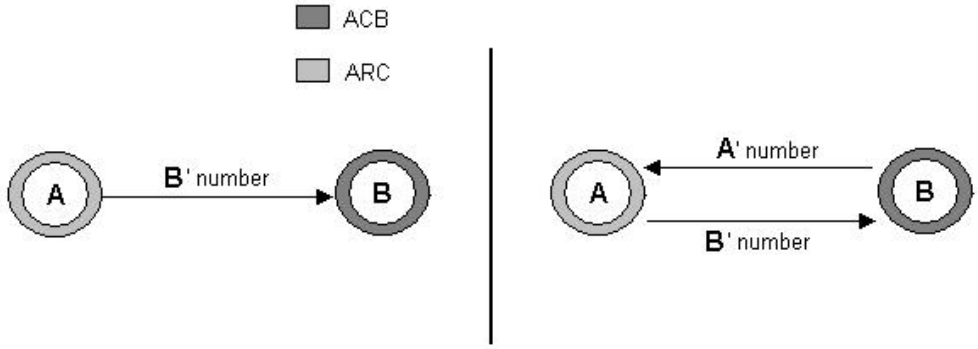

Figure 4.2: Basic Connect Model

- The process of service creation is not longer largely governed by a single organization.
- The platform for an intelligent network promotes independent and rapid deployment of customized features by operating companies and their associated suppliers and independent information providers
- Telecommunications network is becoming increasingly heterogeneous, with equipment and support software provided by several suppliers.
- The service logic for controlling call processing is becoming increasingly distributed, with service logic programs distributed among various network components

In that sense, while more feature supplier exist less homogeneous will be a possible classification of the problems with the interaction or even the features itself.

Given Features are observable behavior and are therefore a requirements specification problem [Gib98]. Most feature interaction problems can be (and should be)

resolved at the requirements capture stage of development. If there are no problems in the requirements specification then problems during the design and implementation will arise only through errors in the refinement process. Certainly the feature interaction problem have a tendency to the introduction of such errors because of the highly concurrent and distributed nature of the implementation domain, but this is for consideration after each individual features requirements have been modelled and validated; otherwise it will not be easy to identify the source of the interaction. Features are requirements modules and the units of increment as systems evolve[ZJ00]. Having features as the incremental units of development, the source of the complexity are:

Complexity explosion Potential feature interactions increase exponentially with the number of features in the system.

Assumption Problem Already developed features often rely on assumptions which are no longer true when later features are conceived. Consequently, features may rely on contradictory assumptions.

Independent Development Traditional approaches require a new feature developer to consider how the feature operates with all others already on the system. Consequently, it is not possible to concurrently develop new features: since how the new features work together will not be considered by either of the two independent feature developers.

Interface Problem User controls on traditional phones are very limited and hence the input and output signals become polymorphic. This is a major problem in requirements specifications as it can lead to the introduction of ambiguities in systems of features. Formal requirements models make explicit the mapping between abstract and concrete actions and the systems can be automatically verified to ensure an absence of ambiguity that could lead to interactions.

4.2 Technical Approaches

In this parts we expose some technics and point of view to study the interaction between feature. There are two different approaches which has been taken because they gives a concrete way to proceed. Nevertheless none of this approaches will be strictly followed as model in order to build the concrete implementation. The first one approaches refers to a way to express interaction using an Object Oriented language. However even if some pieces of codes are showed, the approaches maintain its high level description. The second approach show how to introduce incrementally features, verifying the interaction between them, and at the end creating some kind of hybrid model (POTS + Features).[Pre97b]

4.2.1 Composition

Let us consider the following conflict occurring in telephone connections: B forwards calls to his phone to C. C screens calls from A (TCS, Termination Call Screening). Should a call from A to B be connected to C? In this example, there is a clear interaction between forwarding (FD) and TCS, which can be resolved in several ways. The composition has been verified with the following set of features for this domain of connecting calls:

- CF Forwarding of calls (unconditional or busy)
- TCS (Termination call screening)
- OCS (Originator call screening)

Although TCS and OCS look very similar, there are small differences. For instance, they may interact differently with other features. The basic building block is a feature `Phone`, which provides a function `connect` for computing the phone reached by some dialed number. Here is used a simple technique for adding the actual services to the full object. Each feature adds its functionality by extending a function `dispatch`, which is used by `connect`. Because the authors use exceptions for modelling a busy signal, the function `dispatch` may throw an exception, as shown in the following Java interface description:

```
interface Phone {
    int connect(int dest) ;
    int dispatch(int dest) throws Busy ;
}
```

The implementation provides for a trivial connect functionality, just in order to set the stage for other features[Pre98]. In necessary an explicit constructor Ph here, which initializes the instance variable used for the origin of a call.

```
class Ph implements Phone
{
    // origin
    int o;
    // function for creating and initializing objects
    Ph(int orig) { o = orig;} ;
    // trivial dispatch
    int dispatch(int dest) throws Busy
        {return dest;};
    int connect(int dest)
    {
        try{ return dispatch(dest) ; }
```

```
        catch(Busy B) {println("Busy " ); return 0; }
    }
}
```

Forwarding Next a feature for forwarding is added with the following interface:

```
interface Forward { boolean cf_check(int i);
int forward(int dest);}
```

The code is again as simple as possible, just forwarding selected numbers to the next number via an auxiliary function `fd_check`.

```
feature CF implements Forward
{
    // aux. function
    // a check if forwarding is desired
    boolean cf_check(int i)
    {
        return (i == 5 || i == 7 || i == 10 || i == 11 || i == 12);
    };
    int forward(int i)
    {
        return i+1;
    };
}
```

To integrate the service, a `dispatch` function is raised in the following `lift` feature:

```
feature CF lifts Phone
{
    int dispatch(int dest) throws Busy
    {
        if (cf_check(dest))
            // recursive forwarding
            return connect(forward(dest));
        else return super.dispatch( dest);
    }
}
```

The above either calls `super.dispatch` in order to invoke (possibly) other features, or calls connect to attempt a recursive connect attempt. (This recursive forwarding is not limited, for simplicity.) For Termination Call Screening is used a simple check for each call and raise the exception busy if ICS disallows a call. The structure of the following code is as above.

```
interface TcsI { int tcs(int dest) throws Busy ; }
feature Tcs implements TcsI
{
    Tcs(int orig)
    {
        super(orig);
    };
    //aux. functions
    boolean tcs_check(int i)
    {
        // no calls from 5 to 8\\
        return (o == 5 and i == 8);
    };
    int tcs(int dest) throws Busy
    {
        if (tcs_check(dest)) throw new Busy();
        else
            return dest;
    };
}
feature Tcs lifts Phone
{
    // lift Phone
    int dispatch( int dest) throws Busy
    {
        // add TCS service
        return super.dispatch( tcs(dest) ) ;
    };
}
```

To resolve the interaction between forwarding and TCS, the `forward` function is raised to TCS. The lifting for forward is done as follows:

```
feature Tcs lifts Forward
{
    // lift forward
    int forward(int dest)
    {
        // update origin (also ok if not forwarded!)
        o = dest; return super.forward(dest);
    };
}
```

In case of forwarding over several jumps, TCS is checked for each jump the next jump. If only a check the origin of the call is desired, one just has to adapt the lifter and not update the origin of the call. So, lifting allows for a modular resolution the interaction between two features. For Originator Call Screening OCS is similar to TCS, but model interaction with forwarding differently is a little bit different: OCS should always be checked from the initial phone to the final destination. With this choice OCS can be modelled similar to TCS, but with a different interaction code. An even stronger condition would be to check OCS for each intermediate hop. For this, one needs an extra data structure for storing these.It is now possible to create objects with any subset of the three features TCS, OCS and CF. For instance, with the object con created by:

```
con = new Tcs (CF (Ph 5))) ;
```

CF and TCS are selected and the originating phone is set to 5. With the above code and settings, obtaining the following examples for a connect call from phone 5 (the number 5) with all features:

```
// just gives 6
println( con.connect(5));
// just gives 6
println( con.connect(6));
// gives 8, as forwarded to 8, which is allowed by ICS
println( con.connect(7));
// gives Busy, as forwarded to 8
which is not allowed by TCS println( con.connect(8));
// gives 13, as forwarded twice
println( con.connect(11));
```

It is easy to imagine other features and also variations of the above features. This approach allows to compose such features in a flexible way. This provides for a clear structure of their dependencies, which is needed if the number of complementary or alternative features grows. The last pieces of code shows that FOP easily accommodates exceptions as in Java. Java requires to declare an exception for any method whose body may raise one, and also in interfaces for which one implementation raises an exception. This rigid and fully explicit declaration of exceptions in Java can be used for feature combination in FOP, as all affected methods are marked. On the other hand, this restrictive scheme may be a bit too inflexible. For instance, in the above example, in needed to declare an exception for the function dispatch, although a concrete feature composition may not use exceptions. In such a case, it might be desirable to handle raised exceptions, such that a feature can be adapted to a context where an exception is not declared and handled.

4.2.2 Feature Classification

In [Gib98] there is an attempt at defining a set of categories of features, these categories are not independent and are intended to introduce some rules for integration and composition.

Function Type In the set of telephone services each feature falls into one of three function types, corresponding to billing, routing and endpoint. Routing features are those which are concerned with setting up connections between users at the network level. The set of telephone services routing features are: Call Forward Unconditional (CFU), Call Waiting (CW), Call Forward on Busy (CFB), Termination Call Screening (TCS), and Originator Call Screen (OCS). These features cannot be specified by changing the telephone model alone. Such features are called "Endpoint features". Endpoint features are the easiest to specify as they require changes only at the telephone level of the requirements model.

Connection Type Connection type partitions features into those which are concerned with two users at one time (single connections) and those which allow more than two users on the same line (multiple connections).

Refinement Type Most features in set of telephone services are intended to alter or extend POTS functionality. Such features are said to be of refinement type POTS. However, other features are not designed to change the functionality of POTS but are intended to alter or extend other features in the system. These are said to be of refinement type Feature.

Impact Type Impact Type classifies where in our requirements models there is a need to make changes in order to incorporate the new functionality. A local impact is one in which only the telephone model of the service requester has to be changed. A global impact is one in which all the telephone models have to be changed. A network impact is one in which the network model (between users) has to be changed.

Composition Type Composition type corresponds to the way in which a feature has been specified to configure with POTS. Each configuration is specified as an ensemble of compositions. Many such configurations are common to different features. Thus, composition type identifies the structure that exists between POTS and the feature being classified.

Point of view Type There are two different models for most features: the caller and the called. Some features change the behavior of the feature subscriber as a caller, as a called, or as both. The role of the point of view classification is to make this property an explicit part of the requirements model.

In [Gib98] these categories a fundamental for his algebra. For the purposes of this thesis, his Algebra is not interesting, but categorization of features allows enface in a better way the interaction problematic, as we will see in the second part of this thesis.

4.2.3 Interactions via refinement

A feature is defined as a refinement of an existing component by a sequence of refinement steps above refinement calculus. Usually, new states and new transitions are introduced. Refinement ensures that the new component satisfies all propositions of the original component as well as the newly added feature. Given an existing system S, denoting the refinement step introducing a new feature F and leading to an enhanced system S' by $F(S) \rightsquigarrow S'$. Given an existing system S, calling two features F' and F'' conflicting, if is possible apply independently $F(S') \rightsquigarrow S'$ and $F''(S) \rightsquigarrow S''$ but there exists no $\widehat{S}$ such that.[KPR97][Pre97a]

$$F''(S') \rightsquigarrow \widehat{S}$$
$$F'(S'') \rightsquigarrow \widehat{S}$$

In other words, it is not possible to find a common refinement of S incorporating both features. The problem of checking if a set of features $F_1, ...F_n$ can be integrated into an existing component S can therefore be reduced to the problem of finding refinements such that.

Note that feature interactions as defined above occur only if two features F' and F'' can be applied independently of each other. If one feature depends on the other, for example using a state introduced by the other, a conflict may not occur in the above sense. The advantage of this specification approach is that it abstracts from irrelevant implementation details at first place. This allows to concentrate on the main issues of feature interaction, and considerably enhances the probability that two feature specifications have a common refinement.

It is possible study the example showed in the picture 4.3 using refinement. Services are added via refinements. Refinement is an associative operation, so several features to a given specification. "Refinement" is not formalized but modelled by replacing some tables. The example models the process of connecting a call.

Beginning from the diagram showed in the picture 4.3 the refinements are justifies by the rules of the last section, but only give informal proofs. For the following development, the underlying call processing model has to be quite general, as e.g. it has to encompass the separation of subscriber and phone. This is needed to enable the addition of features. In practice, this means that it is possible to have to backtrack to earlier design stages, in order to accommodate for later steps.

Step 0: The STD in picture 4.3 shows a very abstract model of a switching unit. A call is invoked by a message "call" and finally ends either in state "busy" or "alerting" (i.e. ringing the phone). The variables "sub" and "ph" are for subscriber and phone, respectively.

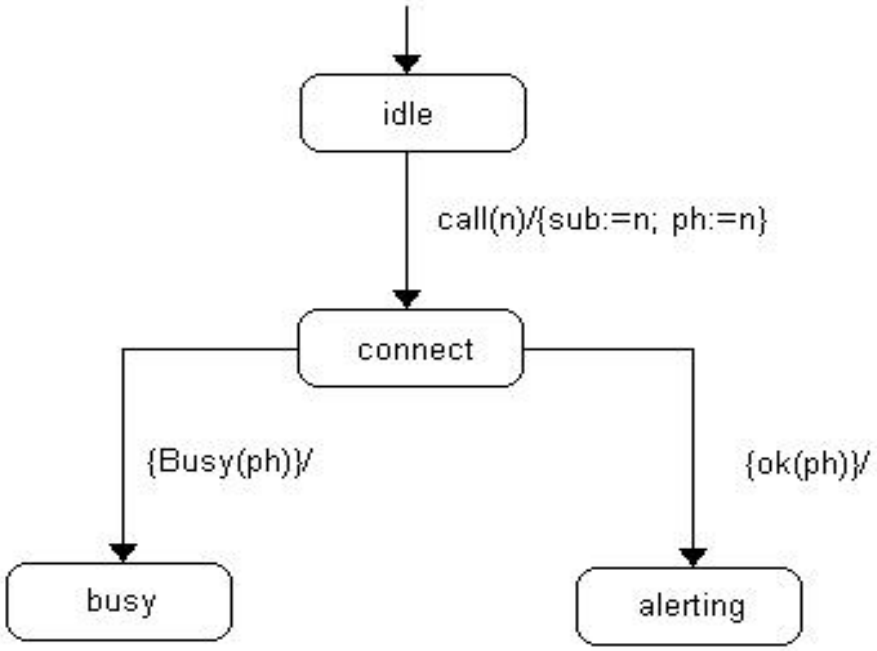

Figure 4.3: Basic Connect Model

The predicates *busy*: indicates if a DN is busy, and *ok* , which denotes successful cases. For refinement, it will be important to assume that ok is disjoint from the predicates on all further added transitions.

Step 1: The first refinement step adds a state abandoned, shown in picture 4.4 , where the newly added states and transitions are drawn with bold lines. This new state models the case of a hookup. The newly added transitions are invoked by the new message *abandon*. Note that this is a proper refinement step, since in figure 4.3, the behavior of the STD with the message abandon is unspecified and fully unpredictable.

Step 2: In the second step, a nontrivial refinement step is performed, namely splitting the state connector into two states, "findsubscriber" and "findphone". This is shown in picture 4.5. Note that the new transition between the new states has a condition with a new predicate "oks", which is unspecified. For refinement, it is not strictly needed to add a transition from findsubscriber to abandoned. After generalizing the structure in the last step, is possible add features, beginning by a structure as is showed in the picture 3.3

Step 3:The picture 4.6 shows additional transitions and a new state timeout for the forwarding features. The new state "timeout" is needed for the delegation over "no answer" features. Note that the condition {else} (picture 4.6 and picture 4.8) is a shorthand for the negation of all other conditions. In the new transition to alerting, assuming an external timer, which is set by *setquickalerttimer*. It is assumed to

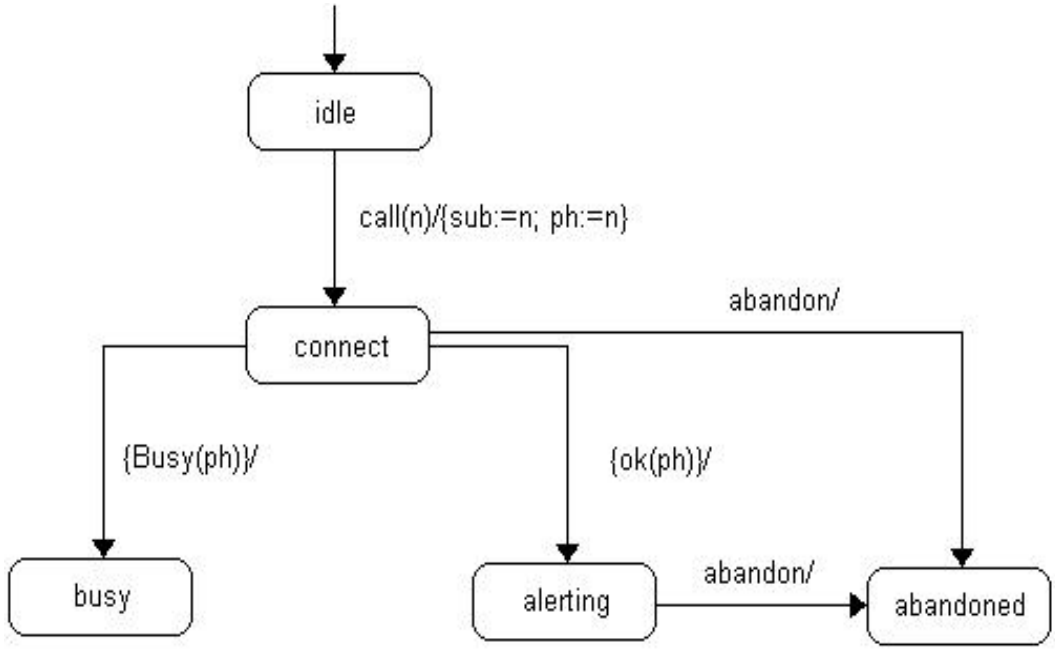

Figure 4.4: Adding State Abandoned (step1)

send the message *timeoutalarm*, if the call is not answered. To show that the added transitions give a proper refinement, assume that the *oks* condition on the transition from the state findsubscriber is disjoint from the new transitions. In other words, these transitions model cases are not handled in the previous $STDs$.

Step 4: The picture 4.7 shows how the blocking features are added to the STD in picture 4.5. Introducing a new state named "blocked". Assume here an additional variable *org* for the origin of the call. The conditions in the new transition assume new predicates, which can be defined in terms of the data structures of these features as follows:

$Block - Sub(origin, sub) = DNR(sub) \vee CNDB(orign, sub) \vee ACR(orign, sub)$
$BlockPhone(origin, sub) = VP(sub)$
$BlockRoute(origin, sub) = OCS(origin, sub) \vee TCS(origin, sub)$

Step 5: The last step, integrating the added features of Step 3 and Step 4 is showed in picture 4.8. As in the above steps, it is easy to see that this is a refinement. Since the features are not conflicting, the order in which the blocking and forwarding features are added does not matter. It could be achieved by a simple syntactic device, such as adding an ordering on the overlapping transitions. Such an ordering can however easily be translated by adding negations of other conditions, in order to exclude certain transitions. It is a different problem to organize such orderings like using refinements, independent of the features themselves.

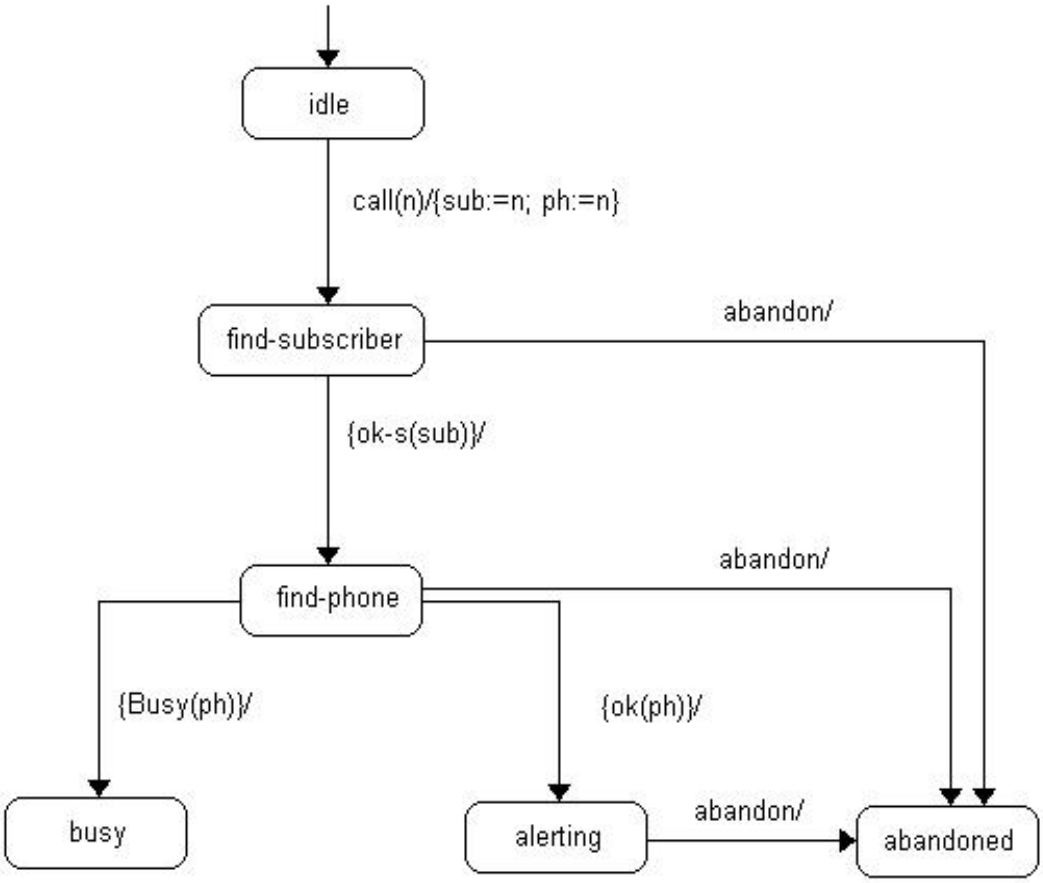

Figure 4.5: Splitting State Connect (step2)

4.2.4 Summary

In this chapter we have exposed feature interaction problems. First just exposing informal description using some conflict interaction samples (section 4.1), and next, two technical approaches which shows different ways to deal with interaction.

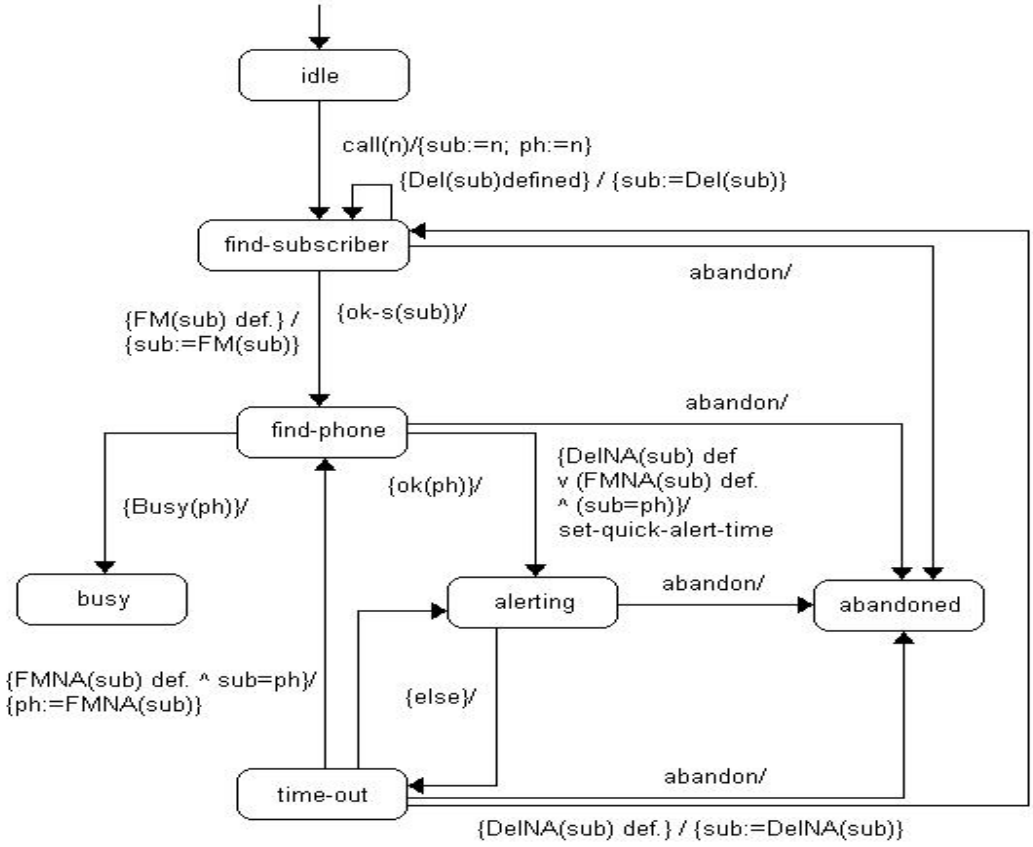

Figure 4.6: Adding Forwarding (step3)

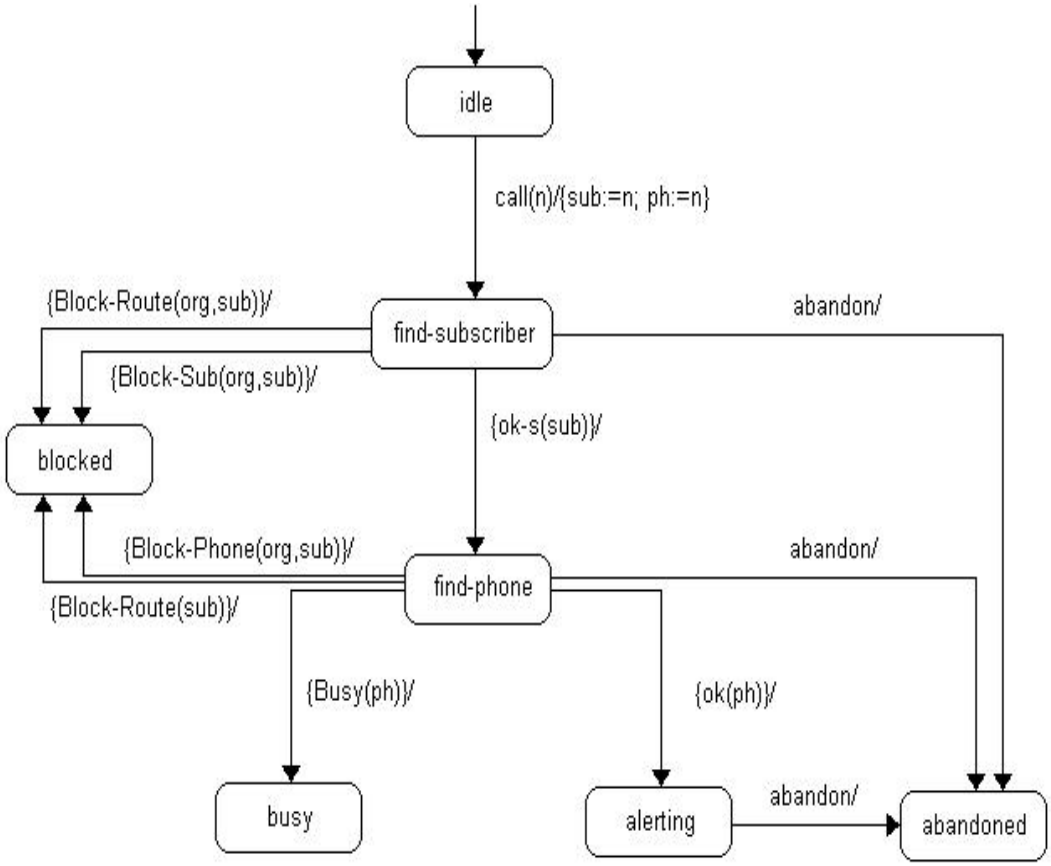

Figure 4.7: Adding Blocking (step4)

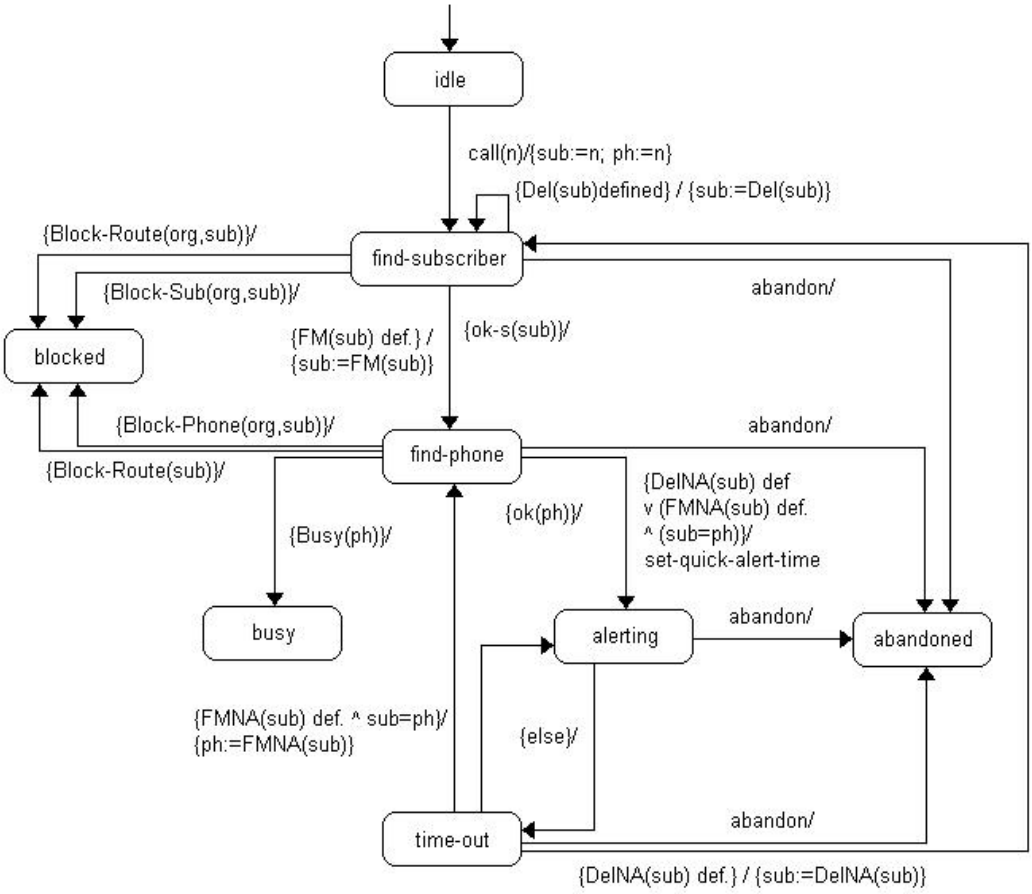

Figure 4.8: Adding Forwarding and Blocking together(step5)

Chapter 5

AspectJ

5.1 Aspect Oriented Programming

Object Oriented Programming is actually considered as a good programming paradigm because the object model provide a better fit with real domain problems. Nevertheless there are problems in where the Object Oriented paradigm is not enough to capture all important design decision. Software design processes and programming languages exist in a mutually support relationship. Design processes break a system down into smaller pieces. Programming languages provide mechanism that allow the programmer to define abstraction of system sub-units, and then compose those abstractions to produce the overall system. A design process and a programming language work well together when the programming language provides abstraction and composition mechanism that cleanly support the kinds of units the design process breaks the system info. Many existing programming languages, including object oriented languages, procedural language and functional languages can be seen as having a common root in their key abstraction and composition mechanisms are all rooted in the same form of generalized procedure. The design methods that have evolved to work with generalized procedure languages tend to break systems down into units of behavior or function.[KLM+97]

A new unit of software modularity, called an aspect, appears to provide a better handle on managing crosscutting concerns. Like objects, aspects are intended to be used in both design and implementation [EAW]. During design the concept of aspect facilitates thinking about crosscutting concerns as well-defined entities. During implementation, aspect-oriented programming languages make it possible to program directly in terms of design aspects, just as object-oriented languages have made it possible to program directly in terms of design objects. Capture the tracing, debugging and instrumentation support for a complex system in a few aspects, rather than as multiple code fragments tangled throughout the classes. Localize the implementation of some design patterns in a few aspects, rather than spreading the fields and methods of those patterns throughout the classes. Capture error handling protocols

involving several classes in a single aspect, rather than as multiple code fragments tangled throughout the classes. Capture distribution and distributed configuration concerns within a set of aspects rather than in numerous code fragments and setup code. Capture resource sharing algorithms involving several classes in a single aspect, rather than as multiple code fragments tangled throughout the classes.

The central idea in AOP is that while the hierarchical modularity mechanism of object oriented languages are extremely useful, they are inherently unable to modularize all concern of interest in complex systems. Instead, in the implementation of any complex system, there will be concerns that one would like to modularize, but for which the implementation will instead be spread out. This happens because the natural modularity of these concern crosscut the natural modularity of the rest of the entire system.[XDA]

Let us consider the next case, there is application which draws points, so there is a class to define each point with its setters. Now, the application must be enhanced in order to draws the shadow of each point. For this, the aspect "shadow" is defined as aspect of each point instance. The "shadow" aspect will "crosscut" some action like point setters in order to define each shadow. In the next sections will be exposed how an aspect is implemented.

5.2 AspectJ

AspectJ extends Java by overlaying a concept of join points onto the existing Java semantics and by adding adds four kinds of program elements to Java: [AWS]

Join points are well-defined points in the execution of a program. These include method and constructor calls, field accesses and others described below.

An aspect is a crosscutting type. By type, we mean Java's notion: a modular unit of code, with a well-defined interface, about which it is possible to do reasoning at compile time. Aspects are defined by the aspect declaration.

A pointcut is a set of join points that exposes some of the values in the execution context of those join points. There are several primitive pointcut designators, new named pointcuts can be defined by the pointcut declaration.

Advice is code that executes at each join point in a pointcut. Advice has access to the values exposed by the pointcut. Advice is defined by before, after, and around declarations.

An introduction is code that may change the type structure of a program, by adding to or extending interfaces and classes with new fields, constructors, or methods. Introductions are defined through an extension of usual method, field, and constructor declarations.

An aspect is a crosscutting type defined by the aspect declaration. The aspect declaration is similar to the class declaration in that it defines a type and an implementation for that type. It differs in that the type and implementation can cut across

other types (including those defined by other aspect declarations), and that it may not be directly instantiated with a new expression.

Let us continue with points and shadows (section 5.1). A "shadows" would looks like the next declaration:

```
aspect Shadow of eachobject (intancesof(Point)
{
    private static final int offset=10;
    protected int x, y;
    ...
}
```

The complete aspect declaration can be found in Appendix A.

5.2.1 Join Points

While aspects do define crosscutting types, the AspectJ system does not allow completely arbitrary crosscutting. Rather, aspects define types that cut across main points in a program's execution. These main points are called join points.

A join point is a well-defined point in the execution of a program. Some of these join points defined by AspectJ are:

Initializer execution Static initialize execution join points occur when static initializers for a class.

Method call When a method is called, not including super calls.

Method call reception When a non-static method is dynamically selected–based on the run-time type of the called object–and executed.

Constructor reception When a constructor is selected and executed based on the type of object created.

5.2.2 Reception and execution join points

The difference between a reception and an execution join point is subtle, has to do with decisions made in the design of the Java programming language relating to method dispatch. In Java, there are two ways to execute the bodies of non-static, non-private methods. One is through a normal method call. In the expression:

```
this.m(3);
```

The actual method of the object bound to `this` is found at run time, through dynamic method lookup. The actual class of `this` (which may be a subtype of the enclosing class) is found, and a method with signature `m(int)` is looked for starting

in that class and going up the inheritance chain. When a method with that signature is found, it is executed. This is exactly the same lookup mechanism that is used for expressions that look like `foo.m(3)`, where `foo` is a variable whose declared type is Foo: First, `foo` actual class is found, then lookup for a method with the signature `m(int)` begins with that class. The other way to execute the body of non-static, non-private methods is with the `super` keyword. In the expression:

```
super.m(3);
```

The compiler determines statically which `m(int)` to execute (the nearest one, starting with the superclass of the current class), and at run-time, that method is simply executed. There is no lookup step at run time. Therefore, super method dispatch does not include reception join points, only execution join points for a superclass's particular implementation of the method. Reception join points and Execution join points capture the points where a method with a particular interface or signature is run. Concretely in the next piece of code the join point to be receipted is the Point constructor:

```
after(Point p) returning(): instanceof(p) && receptions(new(..))
{
    x=p.getX() + offset;
    y=p.getY() + offset;
}
```

So, after Point instantiation the Shadow attributes `x` and `y` will be initialized.

5.2.3 Pointcuts

A pointcut is a set of join points that exposes some of the execution context of those join points. Pointcuts may be defined in classes or aspects.

Pointcut naming and declaration

Named pointcut declarations may include the access modifiers public, private, or protected. Pointcuts are defined in terms of pointcut designators that are composed with the algebraic operators and, or, and not, which are spelled

```
&&, || , and !.
```

for example:

```
pointcut setY(Point p):instanceof(p) && reception (void setY(int));
after(Point p) returning(): setY(p)
{
```

```
        y=p.getY() + offset;
        p.printPosition();
        printPosition();
    }
```

In addition to named pointcut designators, the join points are picked out by primitive pointcut designators. Some of these pick out only one kind of join point. Others pick out all join points during which methods of particular types of objects are executing. Others deal with more lexical issues; when the code defined in particular classes or methods executes.

5.2.4 Aspect precedence

Each join point in a system may have different pieces of advice from different aspects operating on it. In such cases, the advice is ordered:

- First, any around advice is run. Most specific around advice is run first, and each piece of around advice can continue the process by calling proceed().
- Then, all before advice is run, most specific first.
- Then, the computation of the actual join point is executed.
- Lastly, all after advice is run, least specific first.

Whether a particular piece of advice is more specific than another determined by aspect precedence. When an aspect A's advice is more specific than another aspect B's, then A is said to have precedence over B.

In many cases, these different pieces advice are independent, and as such it does not particularly matter whether the code for one runs before the code for another.

In many other cases, the default precedence rules for aspect inheritance is enough to constrain the behavior: Advice in a sub-aspect has default precedence over advice in its super-aspect.

Part II

Implementation

Chapter 6

Modelling Pots

In this chapter we expose in detail the general design of POTS simulation and the two models resultant. In first part will be described informally but detailed the whole system behavior. Next, the two resulting model will be explained, showing some example code.

6.1 Base Model

As said in chapter 2, the functionality of POTS seems quite simple, it just try to establish communication between two clients. But during this document has been unmasked those disguised process behind the scene. let us sequentially and quickly overview about them. At the beginning there is a call request from a client which let us call A, trying to establish communication with a client B, the central receives this request and will search for the client B, if B does not exists a error message must be sent to client A, if B effectively exists the central must asks if it is free to accepts a call. If B cannot accepts the call a busy message must be sent to A, if B effectively can receives a call a message is sent to B to inform that A is trying to establish communication with it. If B does not accepts the call a reject message must be sent to A, if B accepts the call a accept message is sent to A, and, somehow a channel between A and B is established.
As was seen above there are many conditions to be verified before connecting two clients, and there many actions to take in account according to the current condition is verified or not. From a practical point of view there are two kind of entities, clients and a telephone central. The telephone central can be seen as an entity which coordinate the different actions between condition and messages to be sent, but again behind the scene there is a new kind of entity, because the telephone central must be left free to attends new calls request is necessary left the message administration "from client to client", to someone. So, the new entity which represents channel between client will be called as Common Space.

To model the Telephone Central is not difficult because the only task involved

is how deal with the messages, given that each message is triggered by a condition verification. It is enough to define clearly each condition.

For Common Space, it is more simple, because the administration between messages does not represent complication (just modelling POTS, with some features already included that is not so clear) because it only needs the address of the correct direction each message.

The client model is much more complicated, because each client have different states, and the actions to be taken depend of the current state, same thing for reception and remittance of messages. At this point just has been showed the scene from the request call from a Client, let us see what happens before. When the physical user (a real human) take the hook (an offhook action) that is a internal request from the client to the Telephone Central for dial tone (this can be appreciated when the lines are congested). When the dial tone is received is now possible to dial any number. When the user is dialing any number it is another state.

As a formal definition of the user behavior a diagram of state has been build based in the automata showed in the section 2.1.

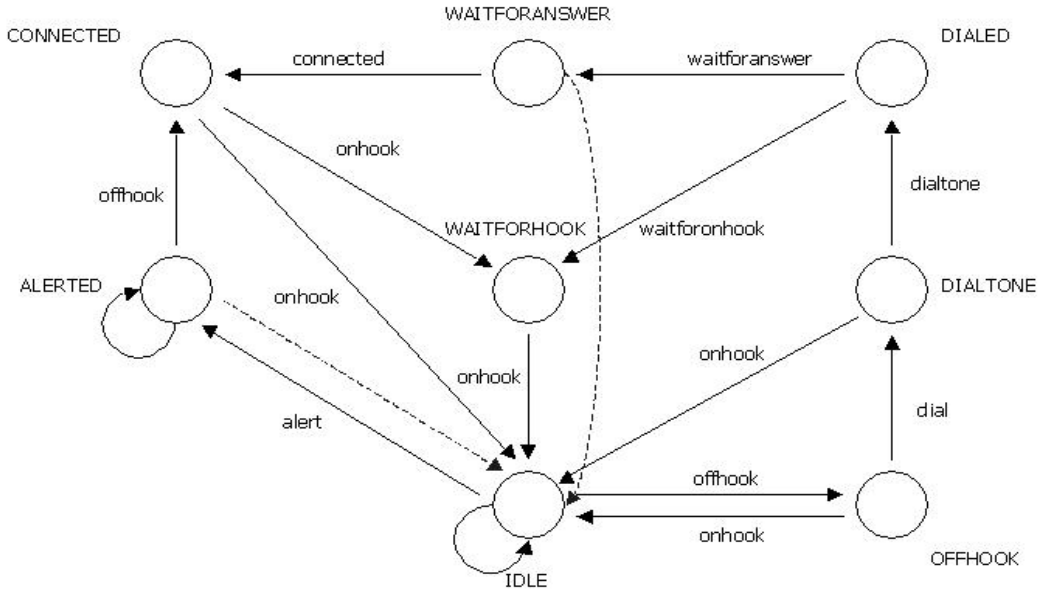

Figure 6.1: The Basic Call Process

This behavior is respected rigourously by the two model.

6.2 First Implementation Details

In this section we show the concrete implementation of each model, with each entity modelled as objects.

The first model take the entities already described just above:

- Client
- Telephone Central
- Common Space

In order to respect the reality the remote environment has been built using Java RMI, because it provides a set of desirable properties for the propose of the model built in this thesis, like internal synchronization. Additionally it provides an API much simpler to use than other currently available distributed application programming frameworks.

In this way there are two different sides: client side and server side. Client side is composed by Client entities, and the server side by Telephone Central and Common Space entities. Each entity is declared as an remote object.

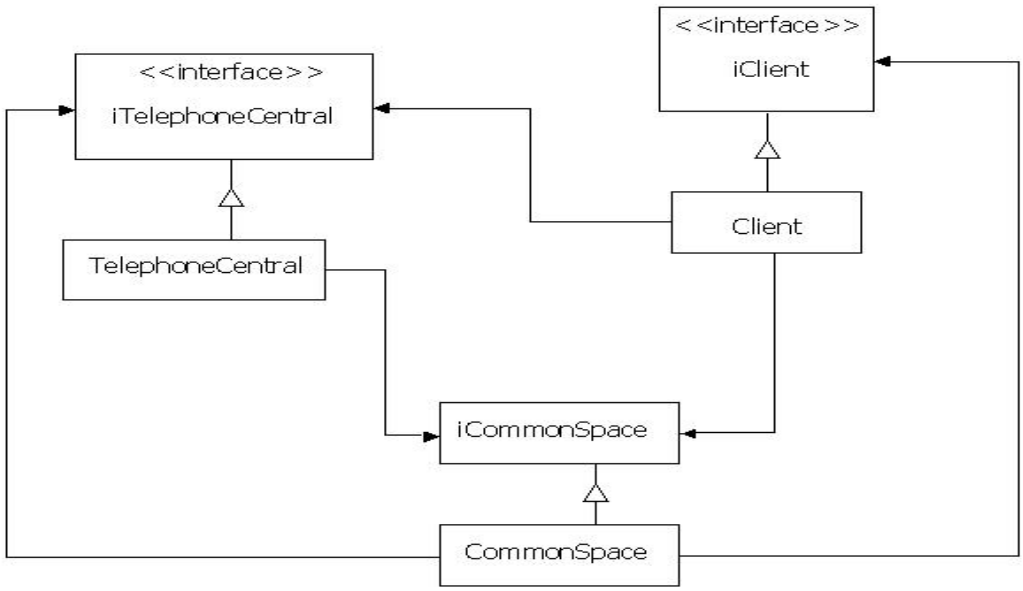

Figure 6.2: First POTS Model

6.2.1 Telephone Central

The Telephone Central entity is defined by *TelephoneCentral.java*, for simplicity all physical defects of line have not been modelled, e.g. "no line tone", "line congestion", etc. The telephone Central stores a collection of clients, concretely a vector of *iClient*, in order to know all clients. The next piece of code shows how to deal with the different situations and messages to be sent.

```
public void dial(iClient callerClient,String phoneNumber)
throws RemoteException
{
    try{
    iClient calledClient=searchClient(phoneNumber);
    if (calledClient!=null)
    {
        if (calledClient.isAvailable(callerClient.getPhoneNumber()))
        {
            callerClient.waitforanswer();
            aCommonSpace=new CommonSpace(callerClient,null);
            callerClient.setCommonSpace(aCommonSpace);
            calledClient.alerted();
            calledClient.setCommonSpace(aCommonSpace);
        }
        else
            callerClient.waitforonhook();
    }
    else
        callerClient.waitforonhook();
    }
    catch(Exception e){callerClient.connectionBreakDown();}
}
```

The method `dial` receives as parameters a remote reference of requester client and the number. The number is searched, if the target client does not exists the message `waitforonhook` is sent to the requester client. If the target client exists, it is checked if it is available to accept calls. If it is not available the message `waitforonhook` is sent to the caller client. If the target client is available a new Common Space is created. And finally the target client is notified with an `alerted` message. The implemented process is not exactly the same as the one detailed in section 2.1; the Common Space is created before the target client accept the call, indeed Telephone Central is not responsible if the target client accepts or does not accepts the call. Additionally in each client (caller and called) there is a timeout running in order to do not wait eternally.

6.2.2 Common Space

The Common Space entity is defined by `CommonSpace.java`. The Common Space knows both caller and called clients. The Common Space task is to coordinate those messages between clients and take measurement when the communication is terminated by some side.

```
public void receiveMsg(iClient aClient,String msg)
throws RemoteException
{
    try{
    if (aClient.getId().equals(callerClient.getId()))
        calledClient.receiveMsg(msg);
    else
        callerClient.receiveMsg(msg);}
    catch(Exception e){aClient.connectionBreakDown();}
}
```

This last piece of code show as the Common Space receive the message and the originator client, verifies if the originator coincide with the caller or called client and sends the message to the right address.

```
public void endConnection(iClient aClient)
throws RemoteException
{
    if (aClient.getId().equals(callerClient.getId()))
        calledClient.connectionBreakDown();
    else
        callerClient.connectionBreakDown();
}
```

When the Common Space receives a `endConnection` means that a side has abandoned the connection. In this case, the Common Space notifies to the other side that connection must be finalized.

6.2.3 Client

The Client entity is defined by `Client.java`. Client behavior is described in the section 2.1; we now explain with the terms of section 2.1 how the client behavior is defined, based in automata showed in picture 6.1. The initial state correspond to IDLE, in order to ask for dial tone, the `offhook` is sent, and the state changes to OFFHOOK, when the `dialtone` message is received from the Telephone Central the state changes to DIALTONE. executing the `dial` message the state change to DIAL in which state dial a number is allows, otherwise is not possible. Depending of the response from the Telephone Central, the state will change to WAITFORONHOOK (is the target client was not found or not available) or WAITFORANSWER. When target client accept a call request, the Telephone Central calls `connect()` message and the connection is established, changing the state to CONNECTED. In another hand, when a client being in the IDLE state receive a call for `isAvailable` means the Telephone Central has receive a request call for it (the current client), if the answer

is TRUE a `alerted` message call will be received, changing the state to ALERTED. For establish the communication the `offhook` message must be executed and `connect` message will be triggered, and finally the state will change to CONNECTED. In order to finish a communication is need that one side call the `onhook` message, and the Common Space will manage the rest. The next piece of code show the `dial` method:

```
public void connect() throws RemoteException
{
    if (getState()==ALERTED)
    {
        try{changeState(CONNECTED);
        myCommonSpace.connect(this);}
        catch(Exception e){setState(IDLE);}
    }
    else
        changeState(CONNECTED);
}
```

The RMI environment is built as normally, declaring the TelephoneCentral Object as Server and Client Objects as clients.

6.3 Second Implementation Details

The second model includes Agents as new entities, as was exposed in the section 2.1. Agents provides local independency in this concrete case means that the Agent, as internal representation of clients, and deals with Telephone Central actions. The Agent contains all client states, leaving the client as a remote message emitter entity which is more close to the reality. Now, the server side is composed by Agent, Telephone Central and Common Space entities.

Both, Telephone Central and Common Space does not suffer a significant change, just they will know agents and no more client. As is showed in the next piece of code, now the Telephone Central has a vector of Agents.

```
public synchronized Agent searchClient(String aNumber)
{
   Agent aux;
   Enumeration numberEnumeration;
   for(numberEnumeration=agentVector.elements();
   numberEnumeration.hasMoreElements();)
   {
        aux=(Agent)numberEnumeration.nextElement();
        if(aNumber.equals(aux.getPhoneNumber()))
```

```
            return aux;
        }
        return null;
    }
```

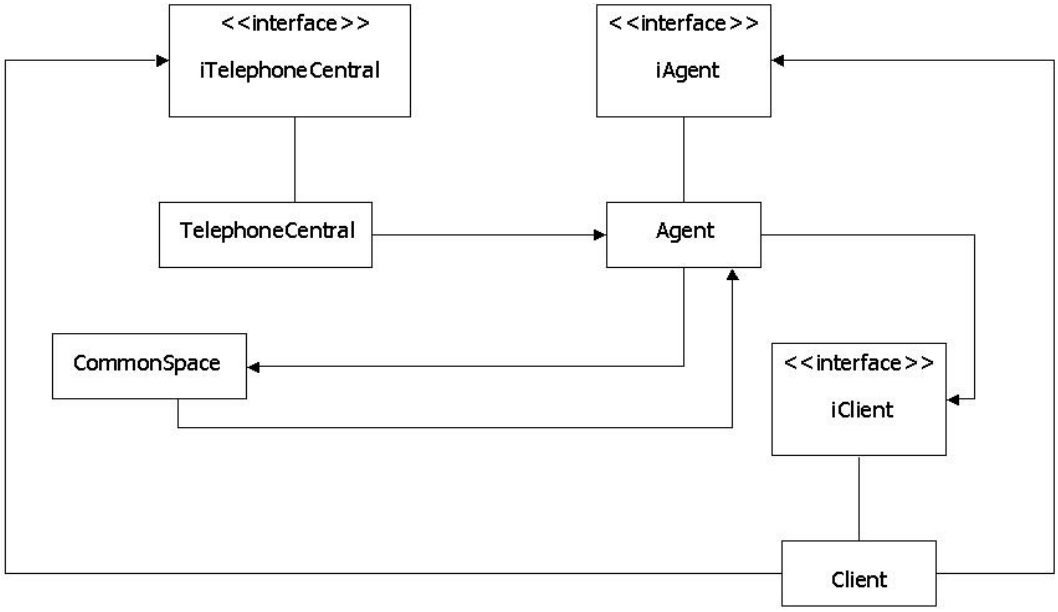

Figure 6.3: First POTS Model

6.3.1 Client

For the client there is no more state, it is just a entity that sends and receives messages. As result code of the whole system is much more simple, because Agents are entities defined in the server side, and remote relationship is between Clients and Agents. Consequently all remote exceptions, defined in the `TelephoneCentral` and `CommonSpace` are now useless. Additionally the remote Interface for the Clients entities is also more simple.

6.3.2 Agent

As was said before Agents are internal representation of clients, and the the state diagram will represent now the Agent behavior. The only relevant change is that the client is still dials numbers, but DIAL state is still in the Agents. The rest of the system remains the same.

6.4 Discussion

Both Model represent a simplified reality of POTS. The main difference between them is the location of problem's resolution. For the first POTS model, the resolution is shared by client side and server side. In the second model, the responsibility of dealing with the events is in the server side. The advantage of the second POTS model is the simplification of the code in remote essential files, that mean Stub, Skeletons and Interfaces. Additionally, there is a bigger transparency for the whole system given the responsibility pass by the server side, giving a better representation of the reality. In the following this thesis will focus in the second model.

Chapter 7

Modelling Features

In this section we expose which and how features are built. The Features modelled in this thesis are:

- Originator Call Screening.
- Termination Call Screening.
- Automatic Call Back.
- Automatic ReCall.
- Call Forwarding Unconditional.
- Call Waiting.

These features has been selected because there are quite common and they provide a interesting number of conflicts between them. In this part we explain informally but detailed each feature implementation.

7.1 Feature Implementation Description

In this section we gives just an informal description of the features implementation, shared by both implementations: object and aspect. Additionally for each POTS model there are not same feature implementation. These explanations follows the POTS specifications given in the previous chapter (Modelling POTS)

7.1.1 Originator Call Screening (OCS)

This feature allows the subscriber to forbid some outgoing calls. The Originator Call Screening entity requires a vector of forbidden numbers. When the `dial` message be executed by the Agent the OCS check its vector. If the number appears, the outgoing call will be aborted.

7.1.2 Termination Call Screening (TCS)

This feature allows to forbid some incoming calls. The Termination Call Screening entity requires a vector of forbidden numbers. When the `isAvailable` message be received by the Agent the TCS will check its vector. If the number appears the incoming call will be aborted.

7.1.3 Automatic Call Back (ACB)

This allows to the subscriber call back an incoming call which has been previously rejected. The Call Back feature stores the last rejected incoming call in order call it as soon as become free. The Automatic Call Back entity stores all incoming number's call in a queue when the `isAvailable` message is called, in order to call them back in the correct order. When `onhook` message is called or `endConnction` is received (both, being in the CONNECTED state) a `dial` message is triggered using the first number in the queue, and removing that number from the queue.

7.1.4 Automatic ReCall (ARC)

This feature allows the subscriber to trigger a new call if the first one has been rejected. When an outgoing call has been rejected the number is store to Recall as soon as the called side become free. The Automatic Recall Entity will store a number when `dial` is called. If the WAITFORONHOOK state is reached via `waitforonhook` message from Telephone Central (and assuming the target client exists) a new `dial` message will be triggered using the stored number.

7.1.5 Call Forwarding Unconditional (CFU)

This feature allows the subscriber to redirect (forward) all incoming call. There are two variants of this feature, if the Call Forwarding is specifically when the subscriber is busy, or if the incoming call are forwarded always (unconditional). The Call forwarding entity stores the forward number. When a `isAvailable` message be received a new `dial` message is sent to the Telephone Central, the old Agent reference is kept but the old number is replaced by the forward number.

7.1.6 Call Waiting (CW)

This feature allows the subscriber to manage two different calls at the same time, but not simultaneously. When the call waiting subscriber receive is already communicates and he receives a further call, he can choose accepts a second incoming call, leaving in a WAIT status the previous call. In that way the subscriber can "switch" between calls. The Call Waiting entity is particulary complex, because introduce two new states to Agent behavior:

- CONNECTEDAVAILABLE: the Agent have one line busy.
- CONNECTEDALERTED: the Agent has received a second call request from the Telephone Central.

Let us examine a call sequence. when the subscriber Agent has just establish a communication is current state is CONNECTEDAVAILABLE, that means when a `isAvailable` message is received it will return TRUE. If this is the case, after return true, a `alerted` will be received changing the state to CONNECTEDALERTED. At this point the first one communication has not suffers any alteration. When the second incoming call is accepted (via a "switch" command) a `connected` is executed, but this time the Call Waiting entity will execute a `changeContext` message, in which the Common Space references will be changed, in order to leave in a WAITFORANSWER state to the Agent from the previous connection. Behind of the scene, the `changeContext` message inform to the current Common Space (means the `CommonSpace` from the current connection) to leave in a wait state the current communication. And finally the state of the subscriber Agent will be CONNECTED. So, each time that a "switch" command be activated `changeContext` will work. To finalize connection the Call Waiting entity will works when a `onhook` message be sent, or `endConnection` message be received, in order to check the correctness of the state, that means if the previous state was CONNECTED the current state must be CONNECTEDAVAILABLE, and so on.

7.1.7 Observations

As can be appreciate in the previous features descriptions there are two key methods:

boolean isAvailable(aNumber) Receives incoming calls

void dial(aNumber) Generates outgoings calls

As will be exposed in the next chapter, these two methods are key for address any solution for feature interaction.

7.2 Integrating Features to POTS

In this Thesis has been used 2 different techniques approaches to integrate features and POTS, they are:

- Inheritance.
- Aspects (AspectJ).

For space reasons these two techniques are explained using as example the Call Waiting case.

7.2.1 Inheritance

Looking at in previous features definition it is possible to see the actions taken for each feature according to message reception or message execution for Agents.

In this approach, for each feature we create a new class which extends from Agent. This feature class overrides all these method used by the feature. Let us see in details for Call Waiting definition given it is the hardest feature implementation.

Call Waiting

List of methods to override:

- *iCommonSpace getMyCommonSpace()*
- *void setCommonSpace()*
- *boolean isAvailable()*
- *void alerted()*
- *void connect()*
- *void endConection()*
- *void connectionBreakDown()*
- *void sendMsg()*

The next piece of code details the class declaration and the constructor:

```
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;
import java.io.*;

public class CWAgent extends Agent implements iAgent
{
    static final int CONNECTEDALERTED=8;
    static final int CONNECTEDAVAILABLE=9;
    static final int WAITING=10;
    CommonSpace my2ndCommonSpace,myCurrentCommonSpace;

    public CWAgent(String _id,String _myPhoneNumber,
    TelephoneCentral _remoteCentral) throws
    RemoteException
    {
```

```
        super(_id,_myPhoneNumber,_remoteCentral);
        my2ndCommonSpace=null;
        myCurrentCommonSpace=null;
    }
    ...
```

As example of method override, this piece of code show the new `connectionBreakDown`.

```
public void connectionBreakDown() throws RemoteException
{
    if (getState()==CONNECTED)
    {
        setState(CONNECTEDAVAILABLE);
        changeContext();
        setCommonSpace();
    }
    else
                waitforonhook();
    }
}
```

This is valid for both POTS models, but these pieces of code correspond to the feature integration of the first POTS model. For the second model of POTS there is not relevant changes, in fact will be just a feature class extending from Agents.

7.2.2 AspectJ

Another way to see the additional functionality of each feature is that a feature implementation "crosscut" the POTS implementation. Using inheritance feature implementation override methods, here feature implementation will create "joinpoints" with these method. Let us continue looking at Call Waiting Case.

Call Waiting

The list of methods is already given in the last subsection.

Aspect declaration:

```
import java.rmi.*;
import java.io.*;
aspect CallWaiting of eachobject(intanceof(Client))
{
    static final int CONNECTEDALERTED=8;
    static final int CONNECTEDAVAILABLE=9;
```

```
    static final int WAITING=10;
    iCommonSpace my2ndCommonSpace,myCurrentCommonSpace;
    ...
```

As is possible appreciate, the aspect for Call Waiting is associated to each Client instance.

```
pointcut connectionBreakDown(Client client)
:instanceof(client)&& receptions(void connectionBreakDown());
around(Client client) returns void
:connectionBreakDown(client)
{
    if (client.getState()==CONNECTED)
    {
        client.setState(CONNECTEDAVAILABLE);
        changeContext(client);
        setCommonSpace(client);
        my2ndCommonSpace=null;
    }
    else
        proceed(client);
}
```

This is valid for both POTS models, but these pieces of code correspond to the feature integration of the first POTS model. For the second models there are no significant changes, because now the aspect will be aspect for Agent instances.

7.3 Discussion

In this section we analyze the advantages and weakness of the two techniques used. There are two main problems related to remote environment and feature integration.

7.3.1 Remote Environment

As was already mentioned the Remote Environment was build using RMI, which means : Stubs, Skeletons and Interfaces. Using inheritance to integrate features makes the new Agent object as remote, and of course extending from the remote interface too. For the second POTS model this changes have a direct impact in the Telephone Central object definition and the Common Space objects definitions, because these entities must know the type of the Agent objects, same thing for Client. That means for each new feature added the definition for Telephone Central, Common Space and Client object. For the first model of POTS that problems does not exists, given the

new Client object will extends from the original interface.
In other hand, for each POTS model the feature definition as Aspect does not present the problematic of create new objects, given is possible integrate aspects keeping the original definitions untouched.

7.3.2 Feature Integration

At this moments all previous description about techniques for integrating feature have solved the problem when every Agent have the same feature. Nevertheless this is not completely enough to create an environment in which exists different Agents with different features associated, without consider the case in which a same Agent have more than one feature associated. Even for an environment between normal Agents and one feature integrated Agents. This happens in both POTS models.

Using inheritance is necessary create a new object definition for each feature integration, for the first POTS model it will have more transparency in side of the Server (Telephone Central and Common Space) given the remote interface will be the same for every remote Client. For the second POTS definition this problem is particulary awful, because each Agent object definition must live in the server side, that mean the Telephone Central object and Common Space object must knows, somehow, the Agent type. The inclusion of reflection could give a great help in this case. Contrarily to the intuition, using aspects the problems have not a better solution. Given each aspect have an object type associated, each object from that type will have this aspect. So, will be necessary create a different object for each feature to integrate, in order to associated a different aspects to different objects.

Along this Thesis has been considered two big assumption at the time of consider the different solutions, in order to respect the realism:

- Is not allowed re-engineers the Basic System.
- Is not allowed alter the features behavior.

That means is not possible to alter the POTS design in order to improve feature integration, and is not possible create features with a different behavior described in the chapter 2. For that reason the problematic of feature integration will be enfaced using aspects and the second model of POTS. Aspect allows increments the model without modify previous designs, because, as will be exposed in the next chapter, there is yet another solution using aspects; and the second models allows a desirable independence between client and server sides, in which the Client is just a message transmitter entity, and that is closer to the reality.

Chapter 8

Modelling Interaction

The goal of this chapter and this Thesis is not to give a definitive solution to the feature interaction problematic, but give an addressing about it. Three approach has been proposed to enface the problems taking in account the next constraints:

- A POTS model.
- 6 Features.
- AspectJ.
- 2 big assumptions

As was mentioned in the last chapter, these two big assumptions are:

- Is not allowed reengineers the Basic System.
- Is not allowed alter the features behavior.

Along of this thesis has been detected two king of interaction, there are:

- when 2 or more feature interacts between them inside a same Agent.
- when 2 features interacts between living in different clients.

For time reason only the first problematic was considered.

8.1 Addressing Classifications

Before addressing the possible solutions it is necessary to go deeper in the problem's causes. For the six features we found eight problematic cases of interaction, always considering interaction between two features. In all conflict cases the problems is originated by a "bad" precedence. In this context let us say the feature A have precedence over the feature B when A take actions first than B. If A and B are concurrent in the

pointcut definitions.

The conflictive interactions are:
Call Waiting & Call Forwarding Unconditional: When an incoming call arrives it is accepted then forwarded, producing a incoherence.

Call Forwarding Unconditional & Termination Call Screening: When an incoming call arrives CFU forwards it, then, this incoming call is checked by TCS. In which case the TCS actions is useless.

Call Forwarding Unconditional & Originator Call Screening:When an incoming call arrives CFU forwarded it, then, the outgoing call generated by CFU is checked for OCS. In which case the OCS actions is pointless.

Automatic Call Back & Call Forwarding Unconditional: If the current Agent state is not IDLE and a incoming call arrive, ACB stores the number in order to call back as soon as the state becomes IDLE, later the call will be redirected by CFU. Finally when the state becomes IDLE an outgoing call will be generated.

Automatic Call Back & Termination Call Screening: If the current Agent state is not IDLE and a incoming call arrives, ACB stores the number in order to call back as soon as the state becomes IDLE, later the incoming call will be checked by TCS, and finally when the state becomes IDLE an outgoing call will be anyway generated even if TCS did not allow the previous incoming call.

Automatic Call Back & Originator Call Screening: If the current Agent state is not IDLE and an incoming call arrives, ACB stores the number in order to call back as soon as the state becomes IDLE, when the state becomes IDLE an outgoing call will be anyway generated and sent, and later OCS will check it.

Automatic ReCall & Originator Call Screening: If an outgoing call is rejected ARC will store that number in order to re-call as soon as the target client state become IDLE. If that number is added to the screening list of OCS just before the re-call, the new outgoing call is sent and later cheked, leaving OCS actions useless.

8.1.1 Feature Precedence

In all these problems are solved when the precedence is inverted. Building a table is possible appreciate the problem more graphically.

X	CW	CFU	TCS	OCS	ACB	ARC
CW		X				
CFU			X	X		
TCS						
OCS						
ACB		X	X	X		
ARC				X		

The "X" character denote a conflict. So now it is possible to define a feature hierarchy in order to establish a "good" precedence order. The criterion used was looking for features less conflictive being second and more conflictive being first. The result in mayor to minor order was:

- OCS
- TCS
- CFU
- ARC
- ACB
- CW

That means, if two features interacts between them inside a same client, and the precedence respect the hierarchy there is not conflicts.

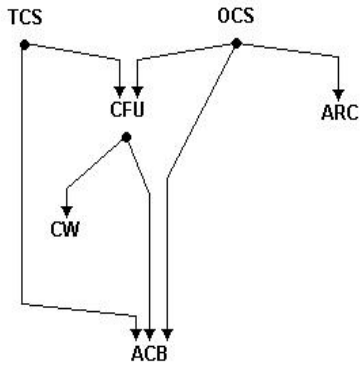

Figure 8.1: Correct Precedence

The top of the hierarchy display two feature quite similar, in fact both have the same behavior: they forbid (incoming or outgoing) calls. TCS and OCS respective. Next, CFU which does not forbid anything but alters the capability of accepting calls by redirect them. Finally, the three last features does not forbid anything or change some capabilities, replacing for anothers. In fact these features enhance the original capabilities without altering them.

Features Classification

Given the last point, it is possible to create categories inside the hierarchy depending of the feature behavior.

- Restrictive: TCS and OCS.
- Excluders: CFU.
- Enhancers: ARC, ACB and CW.

The next picture show graphically the internal behavior of these categories when they are already integrated to POTS.

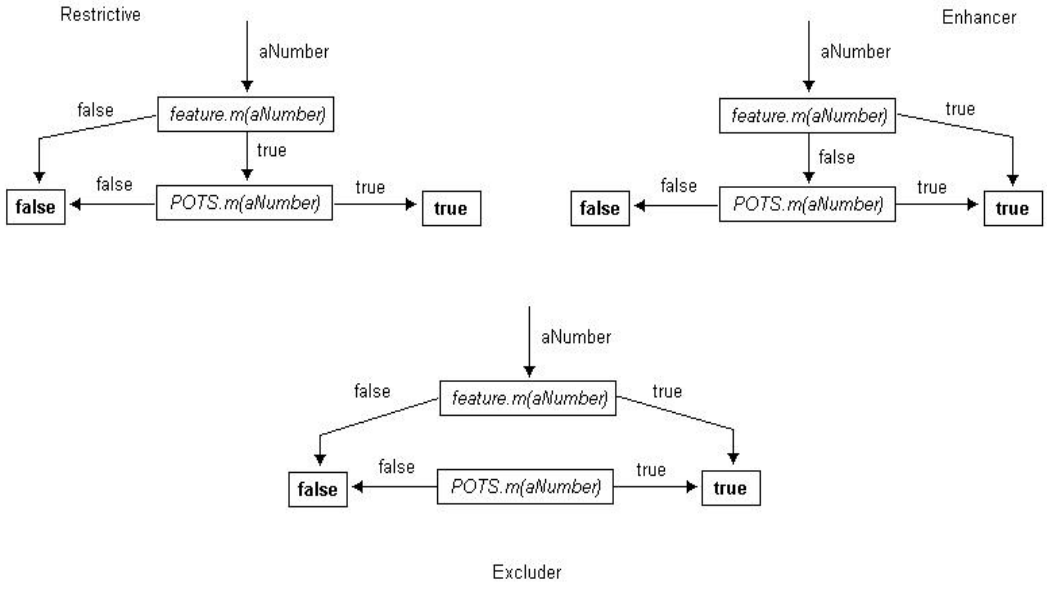

Figure 8.2: Internal Behavior Categories

The figure 8.2 represents a general situation in which a POTS method `m`, which receives as parameter `aNumber`, is crosscuted by s feature method `m`. Feature method `m` also receives as parameter `aNumber`.

Conflicts Classification

In the same way is possible distinguish categories of problem depending the feature classification.

- Precedence: Between restrictive features and excluder features; and between restrictive features and enhancer features.
- Incompatibility: Between excluder features; and between restrictive features and enhancer features.

8.2 Addressing Solutions

As was exposed in the previous chapter there are two key methods in the POTS definitions:

isAvailable(aNumber) Receives incoming calls

dial(aNumber) Generate outgoings calls

These two methods are recurrently "crosscuted" by features, and in all cases the inconsistences or incoherence in the interactions is located there.

In this thesis we explored three possible solution, for the interactions between two features inside an Agent. These three solutions deal with concurrence in pointcut definitions.

8.2.1 Spontaneous Interaction

This solution rest on the aspect precedence definition provided by AspectJ, as exposed in the chapter 5.
The normal precedence between aspects is given by the natural precedence in the advice definition given in the pointcut definition. That is:

- AROUND.
- BEFORE.
- AFTER.

That means this is aspect precedence at the level of pointcuts.
Additionally there is a way to "force" the precedence but this time at the levels of aspect. This is reached using the AspectJ keyword DOMINATE.
Let us consider aspect A and aspect B, in order to specify that A will have precedence over B the definition of aspect A must be:

aspect A dominate B

This is presented as solution when two aspect are concurrent in at least one pointcut definition.
The next piece of code show deal with the concurrence over the pointcut defined by reception of `isAvailable` methods. In this case the aspect TCS will "dominate" to aspect ACB:

```
//OCS aspect

    pointcut isAvailable(Client client,String aNumber)
```

```
    :instanceof(client) && receptions(boolean isAvailable(aNumber));
    around(Client client,String aNumber) returns boolean
    :isAvailable(client,aNumber)
    {
        if (!fNumberExists(aNumber))
        {
            if (proceed(client,aNumber))
                return true;
            else
                return false;
        }
    else
        return false;
    }

//ACB aspect
    pointcut isAvailable(Client client,String aNumber)
    :instanceof(client) && receptions(boolean isAvailable(aNumber));
    around(Client client,String aNumber) returns boolean
    :isAvailable(client,aNumber)
    {
        if (proceed(client,aNumber))
                return true;
        else
        {
            setWNumber(aNumber);
            return false;
        }
    }
```

Discussion

This solution has two main problems: First, this solution present the same problem exposed by integrate features as aspects, because each feature is defined as Agent aspect, in which case all agents will have all features working.

Second, let us suppose only three features, each one represented by aspect A, B and C. A will dominate C, B will dominate C too. So what about interaction between A and B, given that the dominate precedence is defined at compile time?, AspectJ solved this problem internally giving precedence to the alphabetically minor aspect name. In this case A will dominate B, obviously there is a problem when this is not the correct precedence, and the solution will pass by rename each features in order to reach the "correct precedence". This is a dangerous solution because for each new

feature introduction could mean rename all rest of feature. Additionally Agents with less than two features associated are not allowed.

8.2.2 Forcing Interaction

This solution consist in creating, manually, hybrid aspects which will contain functionalities of two features. As is showed in the next pieces of code in which both TCS and ACB are concurrent in the *isAvailable* pointcut definition.

```
pointcut isAvailable(Client client,String aNumber)
:instanceof(client) && receptions(boolean isAvailable(aNumber));
around(Client client,String aNumber) returns boolean
:isAvailable(client,aNumber)
{
    if (!fNumberExists(aNumber))
    {
        if (proceed(client,aNumber))
            return true;
        else
        {
            setWNumber(aNumber);
            return false;
        }
    }
    else
        return false;
}
```

This pieces of code solve the problems of precedence between TCS and ACB. *fNumberExists(aNumber)* is a boolean method which verifies if the number given as parameter exists in the screening list. If not, the number can be stored by using setWNumber(aNumber). Note ACB will store the number only if the Agent is not available to accepts incoming calls.

Discussion

This solution is also not good, it has two problems. First, as exposed in the previous solution in which each aspect is an Agent aspect.

Second, AspectJ does not provide a high level solution for associating aspect to objects dynamically, that mean, for this approach, for each interactions we must be created a new one hybrid aspect. Concretely, in this thesis has been studied six features, and found eight problems, that means the quantity of hybrid aspect to build will be twenty two. Also as in the second problems exposed in the last solution, this

will present problems at the moment of receive new features, in which must be create news hybrid aspects. Additionally Agents with less than two features associated are not allowed.

8.2.3 Ruling Interaction

This solution explore a new way to integrate features in POTS. That new technique come from AspectJ developers and it allows to use aspect as object this is called *Explicit Composition.*[AWS]
This explicit composition consists in associating two objects in order to simulate the association between object and aspect. That association in made by an aspect, which contains a hash table in which we store tuples of object.

The Main idea in this solution is to extend the explicit composition in order to associate more than one features to an Agent.

The first step is to create an object definition for each feature, for that a generic feature definition is created. It is an Abstract class, from which all features extend. The complete code of the generic feature is showed in the appendix C.1. The next piece of code show how are defined all these methods which are at least one time crosscuted by a feature. Each feature definition overrides these methods.

```
boolean isAvailable(Agent client,String aNumber) {return true;}
boolean dial(Agent client,String aNumber) {return true;}
void sendMsg(Agent client,String words) {}
void onHook(Agent client) {}
boolean endConnection(Agent client) {return true;}
void waitforonhook(Agent client) {}
CommonSpace getMyCommonSpace(Agent client) {return null;}
boolean setCommonSpace(Agent client,CommonSpace aCommonSpace){
    return true;
}
String stateToString(Agent client) {return null;}
boolean alerted(Agent client) {return true;}
boolean connect(Agent client) {return true;}
boolean connectionBreakDown(Agent client) {return true;}
```

Additionally an empty feature has been created, which does not override any method.

The next step is to create the composer. It contains two hash tables in order to support two features, and implements the methods necessary to create the associations and recover these associations. The complete piece of code is showed in the appendix C.2 .

There is a generic feature,so it is possible to give to the composer a "feature independence" in order to allow dynamic composition. The pointcut definition are

still quite static, however there is no needed to know the type of the feature, as is shows in the next piece of code.

```
pointcut isAvailable(Agent client, String aNumber)
:instanceof(client) && receptions(boolean isAvailable(aNumber));
around(Agent client, String aNumber) returns boolean
:isAvailable(client,aNumber)
{
    Feature feature1=(Feature)getAspect1OfAgent(client);
    Feature feature2=(Feature)getAspect2OfAgent(client);
    if (feature1.isAvailable(client,aNumber))
    {
        if (feature2.isAvailable(client,aNumber))
            return proceed(client,aNumber);
        else
            return false;
    }
    else
        return false;
}
```

Features must be specify at some point. This is done when an Agent object is created. It contains the type of each feature. This does not violate the first assumption (see at the begin of this chapter), because this is not re-engineering the system. So, when an Agent object is created the composer creates the corresponding concrete features, as is showed in the next piece of code.

```
Feature addFeature(char feature)
{
    if (feature!='N')
    {
        if (feature=='F' || feature=='f')
            return(new CFU());
        else if (feature=='T' || feature=='t')
            return(new TCS());
        else if (feature=='O' || feature=='o')
            return(new OCS());
        else if (feature=='B' || feature=='b')
            return(new ACB());
        else if (feature=='R' || feature=='r')
            return(new ARC());
        else if (feature=='W' || feature=='w')
```

```
            return(new CW());
        else
            return(new None());
    }
    else
        return (new None());
}
```

`None` correspond to the empty feature, this is useful when a Agent has less than one feature associated.

Discussion

The cost of deal manually is the high order that implies the solution, it means if there are n features, the quantity of features as aspects to build is $O(n^2)$, both cases *Spontaneous Interaction* and *Forcing Interaction*. The cost of deal dynamically is the lack of precision at the time of define the correct precedence in the concurrent pointcut definitions. But it provides a more high level solution giving a more reusable model.

Even if the composer is still being and aspect associated to Agent objects there is not problem, because the association is dynamical for each Agent object. Additionally the existence of the `None` feature allows Agents with less that two features associated. In that way the problems of the static association between object and aspect is solved.

```
pointcut isAvailable(Agent client, String aNumber)
:instanceof(client) && receptions(boolean isAvailable(aNumber));
around(Agent client, String aNumber) returns boolean
:isAvailable(client,aNumber)
{
    Feature feature1=(Feature)getAspect1OfAgent(client);
    Feature feature2=(Feature)getAspect2OfAgent(client);
    if (feature1.isAvailable(client,aNumber))
    {
        if (feature2.isAvailable(client,aNumber))
            return proceed(client,aNumber);
        else
            return false;
    }
    else
        return false;
}
```

Nevertheless is not demonstrated that pieces of code, as the previous one, be the solution for all precedence problems for all features, in fact there is particular case,

contemplated in the eight problems mentioned (section 8.1) in which the CFU can forward numbers which are forbidden by OCS, this two features do not share any joinpoint.

Chapter 9

Final Conclusions

In this thesis we expose, in the first part, approaches related about how express: POTS, Features and Features Interaction. Giving formal and informal descriptions, also uses cases.

In the second part, we expose implementation about POTS. How Features have been implemented as aspects, using AspectJ, Features integration on POTS implementation. And finally it was given three different techniques to introduce solutions for Features Interaction.

9.1 Toward to better feature interaction declaration

The main difficulty with the Feature Interaction problem is that complexity of the interaction is not related with the complexity of each feature involved in the interaction. The interaction between two features can be conflictive even if both features works fine in isolation.

In this thesis the problematic of express interaction has been addressed with the help of classification for features. This classification is based on the behavior of each feature studied. A classification for features, allow us to categorize each problems, and given a categorization for each problem allow us to address a categorization for solutions, which will enface each problem category.

The feature interaction problem is difficult: having formal requirements models makes it manageable. A formal model of requirements is needed because it will allows only one correct way to interpret the behavior being defined. Although the model must still be mapped onto the real world (i.e. validated by the customer), this mapping is in essence more rigorous than in informal approaches. Building a formal model requires a better understanding of the problem domain and a better understanding of how the problem domain is viewed by the customer.

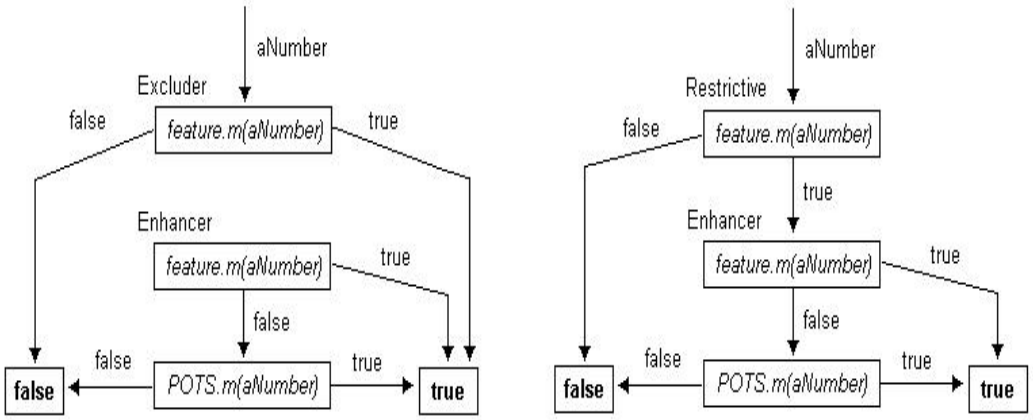

Figure 9.1: A tentative solution for *Spontaneous Interaction* and *Forcing Interaction*

9.2 Further Works

Particular problems is still without solution, and the "Feature Composer" is not a direct solution.Concretely this is produced between two features which produce a conflictive interaction, but they do not share any joinpoint. It was discovered that features in a same category have the same way to define joinpoints, but they are not necessarily concurrent in the joinpoints definition. In the picture 9.1 is possible appreciate a tentative solution for the "Incompatibility Problem" and "Precedence Problem" (section 8.1.1), in which each pointcut is checked first by the feature who has more high precedence. And the main goal of the "Features Composer" is express these solution at level of pointcut definitions.

For this king of conflictive is necessary build feature implementations which consider other feature specifications manually, and the problems of build features definitions manually al are already exposed in this thesis. Even if the addressed solution given by *Ruling Interaction* looks like as a good way, it is just the first step, there are many things to solve and refine as:

- Give a better classification for features, having in account more quantity of feature.
- Improving feature specification.

Appendix A

Explicit composition in AspectJ

The example used in the chapter 5 in order to introduce some concepts of AspectJ is here complete exposed. This Aspect-Object composition is called "Automatic Composition"

```
aspect Shadow of eachobject (intancesof(Point)
{
    private static final int offset=10;
    protected int x, y;

    after(Point p) returning(): instanceof(p) && receptions(new(..))
    {
        x=p.getX() + offset;
        y=p.getY() + offset;
    }

    after(Point p) returning(): instanceof(p) && receptions(int setX(int))
    {
        x=p.getX() + offset;
        p.printPosition();
        printPosition();
    }

    after(Point p) returning(): instanceof(p) && receptions(int setY(int))
    {
        y=p.getY() + offset;
        p.printPosition();
        printPosition();
    }

    public void printPosition()
```

```
    {
        System.out.println("Shadow at("+x+", "+y+")");
    }
}
```

Taking the same Shadow-Point example, is possible appreciate "Explicit Composition". Now, Shadow is an Object:

```
class Shadow
{
    public static final int offset=10;
    protected int x, y;

    public Shadow(int _x, int _y)
    {
        x=_x;
        y=_y;
    }

    public void printPosition()
    {
        System.out.println("Shadow at("+x+", "+y+")");
    }
}
```

Now is necessary define a protocol between objects, this protocol will be defined as an Aspect:

```
aspect PointShadowProtocol
{
    private static Hashtable associations= new Hashtable();

    public static void associated(Point p, Shadows s)
    {
        association.put(p,s);
    }

    public static Shadow shadowAspectOf(Point p)
    {
        return (Shadow)associations.get(p);
    }

    static after(Point p, int _x, int _y) returning ()
    : instanceof(p) && receptions(new(_x,_y))
```

```
    {
        Shadow s=new Shadow(_x,_y);
        associate(p,s);
    }

    static after(Point p) returning (): instanceof(p) && receptions(void setX(int))
    {
        Shadow s=shadowAspectof(p);
        s.x=p.getX() + Shadow.offset;
        p.printPosition();
        s.printPosition();
    }

    static after(Point p) returning (): instanceof(p) && receptions(void setY(int))
    {
        Shadow s=shadowAspectof(p);
        s.y=p.getY() + Shadow.offset;
        p.printPosition();
        s.printPosition();
    }

}
```

Appendix B

Features Crosscuting

As follows, a list with all POTS methods crosscuted by features; Ordered by features.

B.1 Originator Call Screening

```
boolean dial(Agent client,String aNumber)
void sendMsg(Agent client,String words)
```

B.2 Termination Call Screening

```
boolean isAvailable(Agent client,String aNumber)
void sendMsg(Agent client,String words)
```

B.3 Automatic Call Back

```
boolean isAvailable(Agent client,String aNumber)
void onHook(Agent client)
boolean endConnection(Agent client) {return true;}
```

B.4 Automatic ReCall

```
boolean dial(Agent client,String aNumber)
void waitforonhook(Agent client)
```

B.5 Call Forwarding Unconditional

```
boolean isAvailable(Agent client,String aNumber)
void sendMsg(Agent client,String words)
```

B.6 Call Waiting

```
CommonSpace getMyCommonSpace()
void setCommonSpace(CommonSpace aCommonSpace)
boolean isAvailable(String aNumber)
void alerted()
void connect()
void endConection()
void connectionBreakDown()
void sendMsg(String words)
```

Appendix C

Code

C.1 Feature Code

Abstract Feature Code described in the section 8.2.3. From this abstract class will extend all features.

```
abstract class Feature
{
    static final int IDLE=0;
    static final int OFFHOOK=1;
    static final int DIALTONE=2;
    static final int DIALED=3;
    static final int WAITFORONHOOK=4;
    static final int WAITFORANSWER=5;
    static final int ALERTED=6;
    static final int CONNECTED=7;

    void setFwdNumber(String _fNumber){}
    void setFNumber(String _fNumber){}
    String getFwdNumber(){return null;}
    void addFwdNumber(Agent aClient){}
    void addFNumber(Agent aClient){}
    boolean numberExists(String aNumber){return true;}
    void setWNumber(Agent client,String _fNumber){}
    boolean acbNumberExists(String aNumber){return true;}

    void setLNumber(String _lNumber){}
    String getLNumber(){return null;}
    void setCommonSpace(Agent client){};
    void changeContext(Agent client){};
```

```
    void welcome(){}
    boolean isAvailable(Agent client,String aNumber){return true;}
    boolean dial(Agent client,String aNumber){return true;}
    void sendMsg(Agent client,String words){}
    void onHook(Agent client){}
    boolean endConnection(Agent client){return true;}
    void waitforonhook(Agent client){}
    CommonSpace getMyCommonSpace(Agent client){return null;}
    boolean setCommonSpace(Agent client,CommonSpace aCommonSpace){return true;}
    String stateToString(Agent client){return null;}
    boolean alerted(Agent client){return true;}
    boolean connect(Agent client){return true;}
    boolean connectionBreakDown(Agent client){return true;}
}
```

C.2 Composer Code

Code of the composer aspect, described in the section 8.2.3

```
    import java.util.*;
    import java.rmi.*;
    import java.io.*;
    aspectComposer
    {
        Hashtable Aspect1=new Hashtable();
        Hashtable Aspect2=new Hashtable();

        /** Binding between Agents and Features */

        void associate(Agent agent, Feature feature1, Feature feature2)
        {
            Aspect1.put(agent,feature1);
            Aspect2.put(agent,feature2);
        }

        Object getAspect1OfAgent(Agent agent)
        {
            return Aspect1.get(agent);
        }

        Object getAspect2OfAgent(Agent agent)
```

```
{
        return Aspect2.get(agent);
}

/** Features Instantiation */

Feature addFeature(char feature)
{
    if (feature!='N')
    {
        if (feature=='F' || feature=='f')
            return(new CFU());
        else if (feature=='T' || feature=='t')
        return(new TCS());
        else if (feature=='O' || feature=='o')
            return(new OCS());
        else if (feature=='B' || feature=='b')
            return(new ACB());
        else if (feature=='R' || feature=='r')
            return(new ARC());
        else if (feature=='W' || feature=='w')
            return(new CW());
        else
            return(new None());
    }
    else
        return (new None());
}

/** Pointcut Definitions */

pointcut welcome(Agent agent)
:instanceof(agent) && (receptions(new(..)));
after(Agent agent) returning():welcome(agent)
{
    System.out.println(agent.features[0]+" "+agent.features[1]);
    Feature feature1=addFeature(agent.features[0]);
    Feature feature2=addFeature(agent.features[1]);
    associate(agent,feature1,feature2);
    feature1.welcome();
    feature2.welcome();
}
```

```
pointcut stateToString(Agent client)
:instanceof(client) && receptions(String stateToString());
around(Agent client) returns String:stateToString(client)
{
    Feature feature1=(Feature)getAspect1OfAgent(client);
    Feature feature2=(Feature)getAspect2OfAgent(client);
    if(feature1.stateToString(client)==null)
    {
        if(feature2.stateToString(client)==null)
                return proceed(client);
        else
                return feature2.stateToString(client);
    }
    else
        return feature1.stateToString(client);
}

pointcut alerted(Agent client)
:instanceof(client) && receptions(void alerted());
around(Agent client) returns void:alerted(client)
{
    Feature feature1=(Feature)getAspect1OfAgent(client);
    Feature feature2=(Feature)getAspect2OfAgent(client);
    if(feature1.alerted(client))
        if(feature2.alerted(client))
            proceed(client);
}

pointcut getMyCommonSpace(Agent client)
:instanceof(client) && receptions(void getMyCommonSpace());
around(Agent client) returns CommonSpace:getMyCommonSpace(client)
{
    Feature feature1=(Feature)getAspect1OfAgent(client);
    Feature feature2=(Feature)getAspect2OfAgent(client);
    if(feature1.getMyCommonSpace(client)==null)
    {
        if(feature2.getMyCommonSpace(client)==null)
            proceed(client);
        else
            return feature2.getMyCommonSpace(client);
    }
```

```
    else
        return feature1.getMyCommonSpace(client);
}

pointcut setCommonSpace(Agent client,CommonSpace aCommonSpace)
:instanceof(client) && receptions(void setCommonSpace(aCommonSpace));
around(Agent client,CommonSpace aCommonSpace) returns void
:setCommonSpace(client,aCommonSpace)
{
    Feature feature1=(Feature)getAspect1OfAgent(client);
    Feature feature2=(Feature)getAspect2OfAgent(client);
    if(feature1.setCommonSpace(client,aCommonSpace))
    {
            if(feature2.setCommonSpace(client,aCommonSpace))
                    proceed(client,aCommonSpace);
    }
}

pointcut isAvailable(Agent client, String aNumber)
:instanceof(client) && receptions(boolean isAvailable(aNumber));
around(Agent client, String aNumber) returns boolean
:isAvailable(client,aNumber)
{
    Feature feature1=(Feature)getAspect1OfAgent(client);
    Feature feature2=(Feature)getAspect2OfAgent(client);
    if (feature1.isAvailable(client,aNumber))
    {
        if (feature2.isAvailable(client,aNumber))
                return proceed(client,aNumber);
        else
                return false;
    }
    else
        return false;

}

pointcut connect(Agent client)
:instanceof(client) && receptions(void connect());
around(Agent client) returns void:connect(client)
{
    Feature feature1=(Feature)getAspect1OfAgent(client);
```

```
    Feature feature2=(Feature)getAspect2OfAgent(client);
    if(feature1.connect(client))
    {
        if(feature2.connect(client))
            proceed(client);
    }
}

pointcut dial(Agent client, String aNumber)
:instanceof(client) && receptions(void dial(aNumber));
around(Agent client,String aNumber) returns void:dial(client,aNumber)
{
    Feature feature1=(Feature)getAspect1OfAgent(client);
    Feature feature2=(Feature)getAspect2OfAgent(client);
    if(feature1.dial(client,aNumber))
    {
        if(feature2.dial(client,aNumber))
                    proceed(client,aNumber);
    }
}

pointcut onHook(Agent client)
:instanceof(client) && (receptions(void onHook())
||receptions(void endConnection()));
after(Agent client) returning():endConection(client)
{
    Feature feature1=(Feature)getAspect1OfAgent(client);
    Feature feature2=(Feature)getAspect2OfAgent(client);
    feature1.onHook(client);
    feature2.onHook(client);
}

pointcut endConection(Agent client)
:instanceof(client) && (receptions(void endConnection()));
around(Agent client) returns void:endConection(client)
{
    Feature feature1=(Feature)getAspect1OfAgent(client);
    Feature feature2=(Feature)getAspect2OfAgent(client);
    if(feature1.endConnection(client))
    {
        if(feature2.endConnection(client))
            proceed(client);
```

```
        }
    }

    pointcut connectionBreakDown(Agent client)
    :instanceof(client) && receptions(void connectionBreakDown());
    around(Agent client) returns void:connectionBreakDown(client)
    {
        Feature feature1=(Feature)getAspect1OfAgent(client);
        Feature feature2=(Feature)getAspect2OfAgent(client);
        if(feature1.connectionBreakDown(client))
        {
            if(feature2.connectionBreakDown(client))
                proceed(client);
        }
    }

    pointcut waitforonhook(Agent client)
    :instanceof(client) && receptions(void waitforonhook());
    after(Agent client) returning():waitforonhook(client)
    {
        Feature feature1=(Feature)getAspect1OfAgent(client);
        Feature feature2=(Feature)getAspect2OfAgent(client);
        feature1.waitforonhook(client);
        feature2.waitforonhook(client);
    }

    pointcut sendMsg(Agent client,String words)
    :instanceof(client) && receptions(void sendMsg(words));
    around (Agent client,String words) returns void:sendMsg(client,words)
    {
        Feature feature1=(Feature)getAspect1OfAgent(client);
        Feature feature2=(Feature)getAspect2OfAgent(client);
        feature1.sendMsg(client,words);
        feature2.sendMsg(client,words);
        proceed(client,words);
    }
}
```

Bibliography

[AWS] http://aspectj.org AspectJ Web Site.

[CAdR99] Wiet Bouma Carlos Areces and Maarten de Rijke. Feature interaction as a satisfiability problem. 1999.

[CR98] Muffy Caldel and Stephan Reiff. Modeling legacy telecommunications switching systems for interaction analysis. 1998.

[DAW99] L. Charfi N. Gorse L. Logrippo J. Sincennes B. Stepien D. Amyot, T. Gray and T. Ware. Feature description and feature interaction analisys with use case maps and lotos. 1999.

[EAW] http://trese.cs.utwente.nl/ ECOOP'99 AOP Workshop.

[Gib98] J. Gibson. Towards a feature interaction algebra. 1998.

[HJ98] Gisli Hjalmtysson and A. Jain. An agent-based approach to service management - towards service independent network architectura. 1998.

[JC00] Yow-Jin Lin Margaret E. Nilson William K. Schnure Jane Cameron, Nancy Griffeth. A feature interaction benchmark for in and beyond. 2000.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, 1997.

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature specification and refinement with state transition diagrams. In *Fourth IEEE Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, 1997.

[MF99] L. Logrippo M. Faci. Specfying feature and analysing their interaction in a lotos enviroment. 1999.

[Pre97a] Christian Prehofer. From inheritance to feature interaction. 1997.

[Pre97b] Christian Prehofer. An object-oriented approach to feature interaction. In *Fourth IEEE Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, 1997.

[Pre98] C. Prehofer. Feature-oriented programming: A fresh look at objects. 1998.

[RB99] M. Elammari D. Quesnel T. Gray S. Mamkovski R.J.A. Buhr, D. Amyot. Feature-interaction visualization and resolution in an agent enviroment. 1999.

[Tur00] Kenneth J. Turner. Formalising the chisel feature notation. 2000.

[XDA] http://www.parc.xerox.com/csl/projects/aop/ Xeros Design Area.

[ZJ00] Pamela Zave and Michael Jackson. New feature interactions in mobile and multimedia telecomunication service. 2000.