METHODS FOR DESIGNING INTERNET TELEPHONY SERVICES WITH FEWER FEATURE INTERACTIONS

by

Ken Yat-Wan Chan

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

Master of Applied Science, Electrical Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering
School of Information Technology and Engineering
University of Ottawa

May 2003

Abstract

This thesis describes a practical formal approach to designing Internet (IP) Telephony services with fewer feature interactions using SDL and Message Sequence Charts (MSC). Feature interactions are undesirable side effects caused by interactions between features and/or their environment. These undesirable side effects are known to affect the construction of reliable software systems. The IETF 'Session Initiation Protocol' (SIP) is chosen as the IP telephony signaling protocol for the case study of this thesis. Although IP Telephony services do not have some of the traditional feature interactions, the service designers are confronted with new feature interaction problems. We have proposed an extension to the classical feature interaction classification system. The main contribution of this thesis is our formal SDL model of SIP and its sample services that are derived from informal SIP specification. We apply use case analysis to some informal extent in our design, and specify SIP services more precisely by creating an "Abstract User interface". Detecting feature interactions is a verification and validation exercise in which automated tools, such as Telelogic Tau, play an important role. The published sample SIP services are used as use case scenarios and are rewritten in the form of message sequence charts against which the corresponding SDL model is validated. Nevertheless, most feature interactions can be described as distributed system properties and be detected best using Observer assertions in Tau. A feature negotiation approach for resolving feature interactions is also described. Furthermore, we discuss the advantages and shortcomings of using a formal language such as SDL to model an IETF application signaling protocol like SIP. Evaluating the feasibility of using CASE tools such as Telelogic Tau in modeling a complex protocol like SIP is also the interest of this study. Finally, the results of this research show IP telephony services can be implemented with fewer feature interactions by incorporating preventive measures in the design and by resolving feature interactions using our feature negotiation approach.

Keywords: Internet Telephony, SIP, Feature Interaction, SDL, MSC, UML, Use Case, Telelogic, Software Specification, Validation & Verification, and Software Engineering.

Table of Contents

List of figures

Acknowledgments

I wish to thank my supervising professor Dr. Gregor v. Bochmann for his support and guidance in my research. He has given me not only the freedom to think creatively but also the insights to software engineering and formal methods (e.g. SDL) and valuable opinions on how to conduct proper academic research. I would also like to thank two other professors, in particular Dr. Luigi Logrippo from Université du Québec en Outaouais and Dr. Daniel Amyot from University of Ottawa, for their insights to the domain of feature interactions and telephony service specification. Without the suggestions of Dr. Luigi Logrippo, the author would be lost in the world of feature interactions.

In addition, I would like to thank my colleagues in my associated research groups, namely Bassel Daou, Abdel Maach, Jacques Sincenes, Eric Zhang and others, for listening to my ideas and complaints. Of course, I thank my parent May and John Chan, my uncle Tim-Hung Chan, my brothers Paul and Eric, and my sister Sandi Chan for their support during my study.

Also, this project would not be possible without the support of Communications and Information Technology Ontario (CITO) and Natural Science and Engineering Research Council (NSERC), Canada. The tools and the support I have received from Telelogic Inc. are appreciated.

Introduction

The Session Initiation Protocol (SIP), which was standardized under RFC 2543 by the Internet Engineering Task Force (IETF), has become one of the most important Internet telephony signaling protocols in the recent years [44]. Because of its emerging importance, SIP is used as the case study of Internet Telephony services in this thesis. Although RFC 2543 gives a detailed protocol specification for SIP, there is no formal service specification of SIP. Thus, the main motivation of this thesis is to describe our methods for formally specifying SIP services with minimal feature interactions. Feature interactions, which are undesirable side effects caused by interactions between features (services) and/or their environment, are a well-known problem for deploying telephony services. Therefore, our formal model of SIP services that addresses feature interactions is an innovative way for specifying telephony services more precisely.

The following list describes the contributions of our work in decreasing order of importance:

1. We provide a formal SDL model as the specification of SIP services. We have used message sequence charts to characterize user interaction scenarios and used these scenarios to verify the SDL model [2].
2. We discuss the suitability of SDL/MSC tools (e.g. Telelogic Tau) for modeling IETF signaling protocols (e.g. SIP), and to demonstrate the benefits of formal tools in their abilities of validating and verifying the model [2].
3. We explain a number of new schemes for preventing (resolving) feature interactions in SIP [1]. The feature negotiation extension to SIP is particularly interesting.
4. We describe our investigation in applying known methods for detecting feature interactions with the Tau tool [1].
5. We propose an extended classification of feature interactions for SIP services [1]. This taxonomy can be generalized for any telephony services.
6. We identify some new feature interactions that are unique to SIP and their mapping to the extended feature interaction classification (see also [1]).

This document is organized as follows: Chapter 1 gives an overview of Internet (IP) telephony, the IETF Session Initiation Protocol, and feature interactions. This chapter presents the problem domain that we investigate. It also discusses the importance of SIP and

feature interactions, and the reasons for developing Internet telephony. Chapter 2 gives an overview of verification and validation techniques that are significant for both detecting feature interactions and verifying the correctness of the model. This chapter gives an overview of the Specification and Description Language (SDL) and Message Sequence Charts (MSC). The reasons for selecting SDL and MSC as the modeling languages will also be explained. Chapter 3 introduces an enhanced taxonomy for classifying feature interactions among SIP services. This taxonomy is applicable to Plain Old Telephony Services (POTS). Also, this enhanced classification system is useful for associating relationships between different types of feature interactions and the corresponding detection and resolution (prevention) schemes. Chapter 4 gives a detailed description of our SDL specification of SIP services. This chapter also describes the formal design methodology that we used, which involves experimenting with generating use case scenarios from use case diagrams, specifying the structural and behavioral definition, and performing verification and validation on our model. Chapter 5 discusses the semi-automated process of verifying traditional feature interactions that may exist in SIP. Evaluating the feasibility of verifying Message Sequence Charts versus applying Observer Process assertions for detecting feature interactions is the main objective of this chapter. Chapter 6 presents methods for preventing traditional feature interactions in SIP. Preventive (resolution) measures to the feature interaction types (e.g. livelocking, deadlocking, unfair, unexpected non-deterministic, and incoherent) that are discussed in Chapter 3 are examined. Most importantly, the feature negotiation extension to SIP is discussed. Chapter 7 shows new feature interactions that are potentially introduced by SIP and their mapping to the extended classification system. Finally, the document ends with a conclusion and discussion of future work in Chapter 8. Complete information about this thesis project including the complete SDL model of SIP can be found on the Web [43].

1 Overview of Internet Telephony, SIP, and feature interactions

In this chapter, we will discuss the problem domains that are related to the main objective of this thesis which is to come up with methods for designing Internet telephony services with fewer feature interactions. The chapter is divided into two main subsections: overview of IP Telephony and SIP, and overview of feature interactions.

1.1 Overview of IP Telephony and SIP

The emergence of the Internet and the deregulation of the telecommunication industry in the early 1990 have given rise to competition among Internet service providers, traditional and start-up carriers. The main competition is in the prices of voice services, particularly long distance calls. Voice traffic can now be routed over IP networks with a tariff structure based on leased bandwidth instead of per call or per minute-call. Although voice over IP (VoIP) has yet to offer guaranteed quality of service (QoS), the ability of startup service providers to offer voice services by circumventing toll charges has accelerated the development of telephony applications on the Internet, which is also known as IP telephony. In addition to lower the operating cost, IP telephony is intended to allow the integration of voice, multimedia, and data applications by taking advantage of the open nature of the Internet. Although the concepts of a software application and a service are commonly known to be different in the software industry, the term application and service are considered the same and are used interchangeably throughout this thesis for simplicity reason. Whether it is a voice service running on a central switch or a telephony application running on customer premise equipment, developing an IP telephony service/application is more than porting a traditional telephony application from a proprietary carrier network to an IP network. IP telephony is built on the decentralized paradigm of the Internet where IP is the end-to-end transport protocol and IP-enabled devices can address other devices using IP addresses. The possibility of decentralizing IP telephony services is one of the key differentiating factors in developing applications for traditional (POTS) and IP telephony, and poses new challenges to researchers.

Traditional telephony services, sometimes called "Plain Old Telephony Services" or POTS, tend to have application processing centralized among telephone switches. The user terminals tend to be dumb terminals with very limited intelligence. A user has limited control over how the voice services are run, for example, a user tends to subscribe to one service provider which usually only permits all their call forward subscribers to enter their call forward numbers. IP telephony along with the well-known Voice over IP (VoIP) signaling protocols can level the playing field for the end-users and small service providers in this new competitive voice market. In this research, our case study on IP telephony services focuses on SIP [4]. Although H.323 [28] is still the most predominant VoIP signaling protocol suite in IP Telephony, SIP is backed by companies such as Cisco, Microsoft, Nortel Networks, Sonus, and DynamicSoft, and is gaining momentum as one of the de facto VoIP signaling protocols in the industry. The basics of SIP are to be discussed but the emphasis is on services not on the intricate details of signaling.

SIP (Session Initiation Protocol) [is an application-layer multimedia control protocol [35] standardized under IETF RFC 2543 [4]. Similar to most World Wide Web protocols, SIP has ASCII-based syntax that closely resembles HTTP [34]. SIP can establish, modify and terminate multimedia sessions including multimedia conferences, Internet telephony calls, and similar applications. SIP can also initiate multi-party calls using a multipoint control unit (MCU) or fully meshed interconnections instead of multicast [4]. Companies have developed Internet telephony gateways that interface between the Public Switched Telephone Network (PSTN) parties and Internet Telephony applications using SIP. There are additional IETF drafts that describe other important extensions to SIP in efforts to realize VoIP deployment. However, they are beyond the scope of this thesis.

SIP distinguishes five facets of establishing and terminating multimedia communications:

User location: mechanism to determine the end-system to be used for communication;

User capabilities: mechanism to determine the media and media parameters to be used;

User availability: mechanism to determine if the called party is willing to engage in communications;

Call setup: establishment of call parameters at both called and calling party;

Call handling: ability to manage mid-call and third-party call control such as call transfer and call waiting, after the initial call has been setup, and to manage termination of these calls.

Although the five signaling facets mentioned above are all important to SIP, we focus only on call setup and call handling in this research because they are the central functions of any telephony service and offer interesting behaviors for formal modeling.

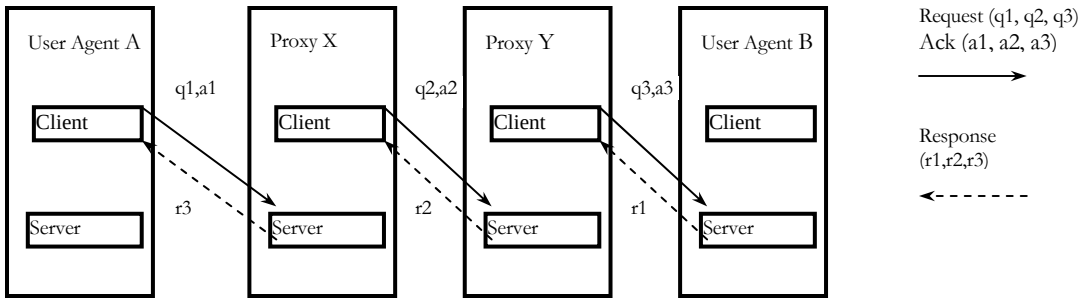

Figure 1: User Agent A calls B via Proxies X and Y

Under the SIP model, signaling parties communicate with each other through asynchronous messaging. Since SIP is a transport-independent signaling protocol, SIP messages can be transferred via UDP, TCP, or other transport protocols. There are three types of messages: request, response, and acknowledgement. In a basic two-party call setup as illustrated in Figure 1, the initiator 'A' sends an "INVITE" request to the called party 'B' to establish a unicast multimedia session (a conference call would require multicast sessions). The "METHOD" field in a SIP request message indicates the action to be performed, and in this case, "INVITE" is the typical call setup and handling request method. Then, the called party would generally reply with one or more appropriate response messages, and the caller would finally acknowledge the final "OK" response by sending an "ACK" message (which is considered as a special request message) to the called party. Upon reception of the "ACK" message, the voice media would be established between the two parties [4].

To cancel a previous request, the request initiator may send a "CANCEL" request message to the called party. To terminate an existing session, the member of the session who desires to terminate the session can send a "BYE" request message to the other party. In an N-party call session, the initiator of a SIP request does not necessarily have to be a member of the session to which it is inviting. Media and participants can be added to or removed from an existing session by sending "INVITE" message(s) with new parameters in the SDP [5] after the initial

call setup. A SIP message consists of a header and a body. The body or the payload of the message typically contains an SDP which conveys information about media streams in multimedia sessions to allow the recipients of a session description to participate in the session [5]. To register the location of the session member, the member can send a "REGISTER" message to a SIP registrar that serves as a lookup directory. There are other SIP methods that have been defined, for example, "INFO". Additional SIP methods can also be added to the standard [4]; however, these are beyond the scope of this thesis.

In SIP terminology, a call consists of all participants in a conference invited by a common source. A SIP call is identified by a globally unique call-id. Thus, if a user is, for example, invited to the same multicast session by several people, each of these invitations will be a unique call. However, after the multipoint call has been established, the logical connection between two participants is a call leg, which is identified by the combination of "Call-ID", "To", and "From" header fields. A point-to-point Internet telephony conversation maps into a single SIP call. In a call-in conference using a multiparty conference unit [4], each participant uses a separate call to invite himself to the unit. The sender of a request message or the receiver of a response message is known as the client, whereas the receiver of a request message or the sender of a response message is known as a server. A user agent (UA) is a logical entity, which contains both a user agent client and user agent server, and acts on behalf of an end-user for the duration of a call. A proxy is an intermediary program that acts as both a server and a client for the purpose of making requests on behalf of other clients. A proxy may process or interpret requests internally or by passing them on, possibly after translation, to other servers (Figure 1). A reader should not confuse the SIP client and server with the initiator (caller or originator) and callee.

A user agent or a proxy can act as either a client or a server, but not both simultaneously in the same transaction [4]. A SIP transaction occurs between a client and a server and comprises all messages from the first request sent from the client to the server up to a final (non-1xx that is a response code outside the range of 100 and 199) response sent from the server to the client. The ACK for a 2xx response to an INVITE request is a separate transaction [4]. The SIP message exchange for establishing a new session is analogous to the three-way handshake of TCP, but is different from the traditional transaction concept in computer science. In fact, the word "transaction" is not appropriate in the context of SIP because it has a different meaning in the world of databases.

A traditional handshake for establishing a TCP connection/session requires three-way handshake between the client and the server [35]. Such three-way handshake between the client and the server involves three types of messages: request, response, and acknowledgement. Unlike SIP, the acknowledgement message is always a part of the handshake. Therefore, not only the word "transaction" is misused in SIP, but also a SIP transaction is not equivalent to a typical client-server three-way handshake known in the field of distributed computing.

What makes SIP interesting and different from other VoIP protocols are the message header and body. Like HTTP, a SIP message, whether it is a request, response, or acknowledgement message, consists of a header and a body. A sample "INVITE" request message body is shown in Figure 2:

```
INVITE sip:ken@ee.uottawa.ca SIP/2.0
Via: SIP/2.0/UDP gtwy1.uottawa.ca;branch=8348
;maddr=137.128.16.254;ttl=16
Via: SIP/2.0/UDP gtwy.ee.uottawa.ca
Record-Route: gtwy.ee.uottawa.ca
From: Bill Gate <sip:bill@Microsoft.com>
To: Ken Chan <sip:ken@uottawa.ca>
Contact: Ken Chan <sip:ken@site.uottawa.ca>
Call-ID: 56258002189@site.uottawa.ca
CSeq: 1 INVITE
Subject: SIP will be discussed, too
Content-Type: application/sdp
Content-Length: 187

v=0
o=bill 53655765 2353687637 IN IP4 224.116.3.4
s=RTP Audio
i=Discussion of .Net
c=IN IP4 224.2.0.1/127
t=0 0
m=audio 3456 RTP/AVP 0
```

```
OK 200 SIP/2.0
Via:SIP/2.0/UDP gtwy1.uottawa.ca;branch=8348
;maddr=137.128.16.254;ttl=16
Record-Route: gtwy.ee.uottawa.ca
From:  Bill Gate <sip:bill@Microsoft.com>
To: Ken Chan <sip:ken@uottawa.ca>
Contact: Ken Chan <sip:ken@site.uottawa.ca>
Call-ID: 56258002189@site.uottawa.ca
CSeq: 1 INVITE
Content-Type: application/sdp
Content-Length: 187
```

Figure 2: INVITE request from Bill to Ken (left) and OK response without SDP from Ken to Bill (right) captured at gtwy.ee.uottawa.ca

The top portion of the message is the message header. The first line is the request line, which contains the Method name 'INVITE', the Request-URI (e.g. ken@site.uottawa.ca), and the SIP Version. The Request-URI names the current destination of the request. It generally has the same value as the "To" header field but may be different if the caller is given a cached address that offers a more direct path to the callee through the "Contact" field. The "From" and "To" header fields indicate the respective registration address of the caller and of the callee. They usually remain unchanged for the duration of the call. The "Via" header fields are

optional and indicates the path that the request has traveled so far. This prevents request looping and ensures replies take the same path as the requests, which assists in firewall traversal and other unusual routing situations. Only a proxy may append its address as a "Via" header value to a request message. When the corresponding response message arrives at a proxy, the proxy would remove the associated "Via" header from the response message header. The "Record-Route" request and response header fields are optional fields and are added to a request by any proxy that insists on being in the path of subsequent requests for the same call leg. It contains a globally reachable Request-URI that identifies the proxy server. "Call-Id" represents a globally unique identifier for the current call session. The Command Sequence ("CSeq") consists of a unique transaction-id and the associated request method. It allows user agents and proxies to trace the request and the corresponding response messages associated to the transaction. The "Content-Type" and "Content-Length" header fields indicate the type of the message body's content and the length of the message body measured in bytes.

The sample response, excluding the associated SDP body in Figure 2, indicates a success 2XX response returned by the callee. The first line of a response message is the status line that includes the response string, code, and the version number. It is important to note that "Via" header fields are removed from the response message by the corresponding proxies on the return path. When the calling user agent client receives this success response, it will send an "ACK" message that has a very similar format as the "INVITE" request message, except the Method name would be "ACK" instead of "INVITE" in the request line. Note that SDP is not required in the body of the "ACK" message.

We have discussed only some of the key header fields in SIP. There are many header fields available in SIP and can become very complex. The associated RFC [4] is recommended for further details on SIP. So far, we have discussed only the basic signaling of SIP. However, there are a few IETF drafts [6,7] which describe the telephony features that extend the basic signaling of SIP. Like H.323/H.450, SIP features are based on a distributed feature model. Although SIP supports intelligent terminals, a true transactional model for signaling between features has not been standardized. Nevertheless, we can apply the same feature categories that exist in POTS and H.323 (e.g. local and network based features like authorization and address resolution, and supplementary services) to SIP [51].

Although H.323 and SIP are similar in certain ways, SIP started with a different approach as compared to H.323 and traditional telephony in defining supplementary services, such as Call Transfer, Call Forward. Whereas traditional supplementary services are explicitly standardized, SIP provides several elements in its message header (e.g. Method) to allow the addition of new services. SIP features such as Call Hold are realized by inserting feature specific information in the header or adding parameters in the SDP (payload of the message). The key characteristic of the H.323 service architecture is its explicit definition of separate state machines for each feature independent from the basic call state machine. On the other hand, the SIP community has not yet standardized these supplementary or advanced telephony services and separated the core signaling from these supplementary features [51].

In addition, the SIP community has specified a variety of traditional telephony services for SIP in [6,7,8], for example, Call Forward Busy (CFB), Call Waiting (CW), Originator Call Screening (OCS), Terminating Call Screening (TCS), etc. In the case of CFB, user 'A' calls user 'B' who is busy and replies with a busy response message. Then, the proxy will forward the call to user 'C'. While we have implemented a number of Internet telephony services, for the reason of simplicity, we will use CFB as the sample SIP service to describe our modeling approach in the subsequent sections.

1.2 Overview of Feature Interactions

Features in a telecommunication system are packages of incrementally added functionality that provide services to subscribers or administrators. In this document, we do not make any distinction between feature and service; the two terms would be used interchangeably. In the community of intelligent network (IN), the idea of building traditional telephony services from SIBs (service independent building blocks) has been studied extensively. The SIBs can be considered as reusable features or service logics that are used to compose complex services [8]. However, the challenges in building reliable and reusable services have always been a part of on-going software engineering researches. These challenges are not only unique to IN and traditional telephony but also to IP telephony. One of these challenges is feature interaction (FI). There are many definitions of feature interactions. However, feature interactions may be defined as "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 instances of the same feature" [3]. In other words, feature interactions are undesirable side effects caused by interactions between features and/or their environment.

The research community of feature interactions has generally categorized feature interactions by the nature and the cause of the interaction. This traditional classification system or feature interaction taxonomy was first proposed in [3]. The system categorizes feature interactions by two aspects: nature and cause. In the following subsections, we will present an overview of the traditional feature interaction classification.

1.2.1 The Nature of Feature Interactions

The traditional categorization by nature includes three dimensions: by kind, by the number of users, and by the number of network components. The first dimension distinguishes subscriber/customer features from system features. This distinction is mainly based on roles (subscriber versus system) of the key actors involved in the system, thus definitely consistent with use case analysis. The second dimension distinguishes feature interactions by the number of users. Users and subscribers are not the same; a user uses a service while a subscriber pays for the service but may or may not use the service. Since the user is the main actor type of a telephony system, interactions of interest must revolve around the users, thus the number of users is an important factor. The number of network components is also a logical factor because a network component usually hides its internal states and data from other components, thus likely contributing to feature interactions and making feature interactions harder to detect and handle.

1.2.1.1 Single User Single Component (SUSC)

SUSC interactions occur when a user simultaneously invokes incompatible features that are assigned to a single network component, such as a switching element. This type of interaction is generally easy to detect because features are constrained to a single actor (user) and a single network component, which usually has access to the internal states of this type of invocation

(or call). An example of SUSC interactions would be the classical example of Call Waiting (CW) and Three Way Call (TWC) [3]. Suppose that both user A and user B are engaged in a two party call conversation. Since user B has subscribed to both services, when another user (user C) calls B, the call waiting tone would be presented to B. If user B has flashed the hook, should the flash hook be considered as the response signal of Call Waiting (putting A on hold and connecting to C) or initiating signal of Three Way Call (user B attempts to "conference in" user C)?

1.2.1.2 Single User Multiple Components (SUMC)

SUMC interactions occur when a user invokes incompatible features that are assigned to more than one network component simultaneously or sequentially in the same call session. An interaction arises when the service designer of one service is not aware of the other service that is designed and deployed by another service designer on another network component. An example of SUMC interactions would be the classical example between Operator-Assisted Call (OAC) and Originating Call Screening (OCS) [3]. Originating Call Screening aborts attempts of making outgoing calls to any number that is on the restricted list. Supposed A is subscribed to Originating Call Screening and makes an operator-assisted call. The operator–assisted call destination may be one of the numbers on the originating call screening list. Since the network component that executes the operator-assisted service does not have access to the originating call screening list that resides on another network component, the call would be made successfully.

1.2.1.3 Multiple User Single Component (MUSC)

MUSC interactions arise when more than one user invoke incompatible features that are assigned to one network component at the same time. A classical example would be Call Forward Busy (CFB) and Originating Call Screening (OCS) [3]. Call Forward Busy allows an incoming call to be redirected to another number when the local callee is busy. Suppose that user A is subscribed to Originating Call Screening service that forbids any outgoing call to user

C, and user B is subscribed to Call Forward Busy on the same network component. If user A makes a call to user B, the network component would forward all incoming calls for user B to user C when user B is busy. Therefore, the call made by user A to user B would be forwarded successfully to user C because the forwarded call consists of two separate call requests (A to B) and (B to C), which do not violate the Originating Call Screening filter (no calls to C) subscribed by user A.

1.2.1.4 Multiple User Multiple Components (MUMC)

MUMC interactions arise when more than one user invoke incompatible features that are assigned to more than one network component in the same call session. A classical example would be Automatic Callback (AC) and Call Waiting (CW) [3]. Automatic Callback is triggered when the local callee (B) is busy. The caller (A) would receive a busy signal and when the callee (B) becomes idle again, the Automatic Callback service would initiate a call to the original caller (A) on behalf of the original callee (B). Suppose that user B is also subscribed to Call Waiting, which is running on another network component. Depending on the ordering of the network components, the busy signal may not be passed back to the network component that operates the Automatic Callback Service. As a result, no callback would be activated.

1.2.1.5 Customer System (CUSY)

CUSY interactions are interactions between customer features that are user-oriented and system features that are related to operation, administration, and maintenance services. A classical example mentioned in [3] is between Long Distance Call (LDC) and Message Rate Charge Call (MRC). Each long distance call consists of at least three call segments: two local accesses and at least one inter-exchange carrier access. The question is whether a customer should be billed for each successfully connected segment or not even when the long distance call has not been completed. Also, should it be billed as one unit towards the total local units allowed per month by Message Rate Charge Call? The question that is raised in [3] may be of

concern in IP telephony (e.g. SIP), as discussed in the next section, but it is not an issue in the PSTN world where calls that have not been completed are never charged.

1.2.2 The Cause of Feature Interactions

Categorizing FI's by the cause has been suggested in [3]. The suggested causes are violation of feature assumptions, limitations of network support, and intrinsic distributed system problems. Instead of discussing these general causes, we consider below four detailed causes that are also mentioned in [2] and are specializations of the above general causes.

1.2.2.1 Resource Contention (RSC)

Resource Contention (RSC) is definitely a well-known distributed system issue. It is defined as the attempt of two or more nodes to access the same resource [3]. In the context of FI's, an accessing node may be the feature running on a network component. Resource is an abstract term; it needs not be a physical entity. For example, Call Waiting (CW) and Three Way Calling (TWC) in POTS may be considered as contending for the flash hook signal simultaneously.

1.2.2.2 Resource Limitation (RSL)

Resource Limitation (RSL) is also mentioned in [3], and is definitely a common cause of FI's in POTS, because most of the traditional POTS end-user devices (e.g. basic phones) have limited user interfaces and computational power. If the classical example of resource limitation were revised (e.g. Call waiting and Three Way Call), the confusion of the flash hook signal can also be attributed to the lack of separate buttons for the two features; thus that feature interaction can be also classified under resource limitation.

1.2.2.3 Violation of Feature Assumptions (VFA)

Violation of Feature Assumptions (VFA) is defined as a set of assumptions that telecommunication features, much like any software features, operate under, and are designed on. There are many types of assumptions. One example from [3] that is particularly worth examining is assumptions about data availability. Features such as Terminating Call Screening (TCS) cannot function properly without the caller-id of the caller. If the caller-id were made unavailable in the case of Private Call (PC), TCS would permit the call request. In the case of Operator Assisted Call (OAC) and OCS, the hiding of the original caller-id by OAC allows the call to bypass the OCS restriction.

1.2.2.4 Timing and Race Conditions (TRC)

Timing and Race conditions (TRC) are also mentioned in [3], and are common, particularly in distributed real-time systems like POTS running on PSTN. A race condition is defined as a condition where two or more nodes have non-mutually exclusive read and modify access to a shared resource. As a result, the value or status of the shared resource may be undefined (different values in the same context) depending on the timing of the accesses between the nodes. A classical example is between Automatic Callback (AC) and Automatic Recall (AR). The AR feature makes a call on behalf of the original caller when the callee becomes idle again. Both AC and AR features depend on the busy signal of the callee. The timing of the busy signal received by the two features would either allow one of the calls to go through on the single line of the POTS phone, or reject both call requests because both ends are calling each other simultaneously.

1.2.3 Existing Research in Feature Interactions

Feature interaction is not a new problem and has been around for a long time. Many researchers have developed many solutions to tackle the feature interaction problem in POTS [13,18,19,20,21,22,23,24]. Some of these techniques may be applied to IP Telephony, but

further investigations are required to deal with the new feature interactions that appear in IP Telephony [9]. According to J. Lennox, feature interaction is a more serious problem in IP Telephony than in traditional telephony [9] because: a) the process of service creation is no longer governed by a single organization, b) the network is becoming more heterogeneous, c) service logic may be provisioned or created by subscribers, service designers, service operators, or network administrators over the Internet [3], and d) address allocation on the Internet is less regulated than in PSTN. Chapter 6 discusses the new feature interaction challenges to be faced in IP Telephony.

In general, research interests in feature interactions may be classified into the following areas [19]: 1) classification of feature interactions [3], 2) methods for detecting feature interactions, manually, semi-automatically, or automatically, at design-time [13,23,24] or run-time, 3) methods for avoiding feature interactions at design-time [21], and 4) methods for resolving interactions at run-time [18,20]. Detecting and avoiding feature interactions at design-time and resolving feature interactions at run-time have been the most active areas of research [19]. Detecting feature interactions is a computationally intensive process that is feasible usually at design time. Automated detection scheme is always the most preferable choice but the degree of automation is not the most important criteria for choosing a feature interaction detection scheme. Factors such as the coverage of feature interactions and the ease of expressing features and their interactions are other important criteria as well. In our work, we emphasize the ease of expressing features and their interactions because we would like to come up with a detection scheme (see Section 5) that uses as many facilities offered by the design tool (Tau [12]) as possible. This way, we do not have to export our design model from one tool to another tool, thus eliminate the chance of introducing errors during the translation process. The coverage of feature interactions is also important because we would like to discover as many feature interactions as possible (see Section 3). Ideally, when a feature interaction is detected, the service designer would incorporate some schemes in the design to avoid the interaction. However, this is not always possible because the interaction may occur only when features are bound to dynamic data at run-time. Thus, resolving the interaction at run-time is the only solution though the process is still resource intensive.

In the subsequent subsections, we will give an overview of significant detection and resolution schemes which were proposed by various feature interaction researchers. Each approach has its advantages and disadvantages. In the subsequent chapters, we will show our

hybrid detection and resolution approach is most appropriate and feasible given our problem statement and the design restrictions we face.

1.2.4 Automatic Filtering of Incoherences at design-time using Prolog

N. Gorse proposed the use of Prolog to detect feature interactions by expressing telephony features and their interactions as predicate logic formula [13]. He has also proposed a scenario-based validation approach for detecting feature interactions. However, he seemed to favor the Prolog or logic programming approach because it can be fully automated. Thus, we will only discuss his logic programming approach.

In general, there are four important concepts in his formalism: the feature description, the properties, the interaction rules, and contradiction pairs. A feature is represented by a feature description which is expressed as a predicate; the head of the predicate contains four lists: the name, pre-conditions, trigger events, and results. The pre-conditions, trigger events, and results are presented by properties. A property which typically represents the condition or state of a call (e.g. call(A,B), busy(A,B)) is expressed as a clause.

Given a set of feature descriptions, it is possible to identify incoherences manually by applying each interaction rule to each possible feature pair. Thus, for n feature descriptions, n^2 feature pairs must be considered. The body of the predicate contains constraints, stated as relations between variables. Contradictions between two properties are known as the "contradiction pair" which is stated as a fact. An example of a contradiction pair is that a user X cannot be busy and idle at the same time. This is expressed by the following Prolog fact:

- contradiction_pair(busy(X), idle(X)).

To check for feature interactions, a feature incoherence, which is a form of feature interaction, is expressed in the form of a predicate; it is also known as a rule. The signature of the predicate is the following:

- fi_check(R, [F1, Fx], [F2, Fy], PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2).

The variables of this predicate respectively denote the name of the rule, the names (split in two pairs) of the two features to be considered, the pre-condition, triggering events, and results of the first features, and the pre-conditions, triggering events and results of the second one. The corresponding definition of the rule is the following:

- fi_check(R, [F1, Fx], [F2, Fy], PcnF1, TrgF1, ResF1, PcnF2, TrgF2, ResF2) :- feature ([F1, Fx], [F2, Fy], PcnF1, TrgF1, ResF1), !, feature([F2, Fx], [F1, Fx], PcnF2, TrgF2, ResF2).

Given a specific rule, the strategy followed for the identification of an incoherence is to bind the variables of properties to a proper combination of users such that the two features satisfy the rule. Prolog excels in this context because a Prolog interpreter always tries to satisfy the predicates by binding variables to all combinations until a combination that satisfies the predicates is found or all combinations of users have been tried. Thus, given two feature descriptions, a single query to one of the rules allows finding all incoherences that can be identified by the rule [13].

The advantage of this approach is that the designer can express feature interactions as known incoherent rules and reuse these rules to find new feature interactions among new features. The Prolog interpreter generally computes faster than state-space based validation tools because feature descriptions are associated to a finite set of facts whereas the state-space of a model tends to increase exponentially as the number of users and features (processes) increases. The disadvantages of this approach are that 1) Gorse described how he could detect feature interactions between two features (feature pair) at a time, but he did not elaborate on how, or whether it is possible, to detect feature interactions among a combination of more than two features, and 2) describing telephony features, which are state-oriented and highly concurrent, as logic formulas is inconvenient. Prolog is a goal-oriented language designed to solve problems as rules. It does not offer language facilities to conveniently express concurrency, like languages such as SDL [10] and Java [31].

1.2.5 Automatic Conflict Detection Using a LTL-based Model Checker

A. Felty and K. Namjoshi presented a formal feature specification language and a method of automatically detecting feature conflicts ("undesirable interactions") at the specification stage in [23]. They showed early conflict detection could help prevent costly and time-consuming problem fixes during implementation. Features were specified in linear temporal logic; two features conflict essentially if their specifications were mutually inconsistent under axioms about the underlying system behavior. They showed how this inconsistency check might be

performed automatically with existing model checking tools. The model checking tools could also be used to provide witness scenarios, both when two features conflict as well as when the features were mutually consistent. Both types of witnesses were useful for refining the specifications. They implemented a conflict detection tool, FIX (Feature Interaction eXtractor), which used the model-checker COSPAN for the inconsistency check. Furthermore, they described their experience in applying this tool to a collection of feature specifications derived from the Telcordia (Bellcore) standards [23].

This approach is somewhat similar to Gorse's logic based approach [13]. However, it is based on linear temporal logic and uses a model checker to check for feature interactions. The advantage of this approach is that it can express reactive properties in linear temporal logic (LTL) and the model checker could check whether the formulas are sound. However, model checkers are known to be computationally slow. Another disadvantage of this approach is that telephony features tend to be state-oriented and expressing features in LTL is less convenient than expressing them in formal languages such as SDL [10].

1.2.6 Goal-Oriented Feature Interaction Detection in the IN Model

In their research, Kamoun and Logrippo specified the Intelligent Network call model and services as a LOTOS model [24]. The functional entities involved in the establishment of call connection and invocation of services were formally specified. They detected violation of feature properties which were considered as feature interactions between IN services. The approach was based on stating feature properties, on deriving goals satisfying the negation of these properties, and on the use of Goal Oriented Execution developed by other members of their research group, to detect traces satisfying these goals. A trace satisfying a goal shows that an interaction exists between the specified features. In fact, it describes a scenario violating one of the properties of the features. Details on their work can be found in [24].

In short, their approach is very reasonable. They have used a formal language called LOTOS [45], which is excellent for describing reactive systems, to specify their model. They use CTL, a branching time logic, to describe feature properties. We believe this is an efficient feature interaction detection scheme at design-time. We decided to use SDL as our modeling language because we believe expressing SIP services in extended communicating finite state

machines is more convenient. One of our objectives is to evaluate the feasibility of using a commercial formal language tool such as Tau [12], thus LOTOS which lacks commercial supports was not selected in our study for this reason.

1.2.7 Negotiating Agent Approach for Resolving Feature Interactions

One of the first feature interaction researchers, N. Griffeth proposed a negotiating agent approach to resolve conflicts between features of one user and of different users [20]. The basis of negotiation is a goal hierarchy whose lowest-level (basic) goals are operations on a given service platform. Higher level goals correspond to possible goals or intentions of a user. Examples of basic goals are *get-key(t,u)* and *connect-station(u,s,v,t)*. The operation *get-key(t,u)* asks user *u* to enter the key for station *t*. It succeeds if *u* responds with the correct key. The operation *connect-station(u,s,v,t)* connect stations *s* and *t* on behalf of users *u* and *v*. New goals can be formed by combining other goals in one of two ways; this is a composition relation. For example, the goal of *call(u,v)* is an abstraction of the goals *connection-station(u,s,v,t)* and *key-connect(u,s,v,t)*. The goal *call(u,v)* denotes a call between two users u and v [20].

Whenever a goal needs to be achieved, that goal can also be achieved by achieving any of the specializations of that goal or by achieving all of its composite subgoals. A *specification* of a goal is derived by recursively replacing an abstract goal by one of its specializations and a composite goal by the set of its components. By definition of a specification, whenever all goals in the specification of a goal are achieved, then that goal itself is achieved [20].

They used specifications as the proposals and counter proposals in their negotiation mechanism. They proposed a negotiating system that works on behalf of *entities* (e.g. users), which have policies restricting what operations they are willing to perform. It includes agent objects that represent the various entities of the system and try to carry out their policies, and negotiators, which help the agents to reach agreement. Users are represented by user agents which may negotiate directly with other agents or through one or more negotiators. The functions of an agent are to produce proposals to initiate or modify a call and to evaluate proposals from other agents, possibly generating *counter-proposals*. *Proposals* specify the desired operations on calls. Whenever a proposal is received, the hierarchy can be used to infer what

goals it is trying to achieve. If a proposal is not acceptable to an entity, the agent constructs a counter-proposal [20].

In general, they distinguished between three quite different organizations for negotiation: 1) *direct negotiation:* In this case, agents negotiate with each other without the assistance of a mediator, 2) *indirect negotiation:* A negotiator serves as a dedicated entity (mediator) who recognizes which agents have to approve a proposal and to route proposals and counter-proposals to appropriate agents, 3) *arbitrated negotiation:* An *arbitrator* takes the complete script of each agent and has sole responsibility for finding a resolution of a conflict. Thus in this type of negotiation, the agents do not need to generate and evaluate any proposals [20].

Their approach used a three-phrase negotiation: generating a proposal, then determining acceptability, and possibly generating a counter-proposal. We find this negotiation approach very much applicable to Internet Telephony services such as SIP services. We will show our similar negotiation approach for resolving SIP feature interactions in Chapter 6.

2 Overview of Formal Software Specification

Over the years, developing software has evolved from building executable directly based on informal textual documents to modeling system using graphical notation. Although software designers are still reluctant to adopt automated modeling tool to generate the complete implementation, they cannot deny their desire of having a tool to capture the structure and the behavior of the system in the graphical format. The main approaches to high-level system modeling have been structured analysis/structured design (SA/SD), and object analysis and design (OOAD). OOAD has become the design methodology of choice [25]. Most notably, the Unified Modeling Language (UML) supported by the Object Management Group (OMG) is the modeling language that has generated the most excitement among the developer community [27]. On the other hand, Specification and Design Language (SDL) is the more mature language that is built on a well-developed formalism and has been used in many real-time software development projects [12].

Formal methods, based on sound and provable mathematical theories, differ from non-formal methods because they offer designers the ability to prove the correctness of the software through validation and verification. Formal methods may be used to specify and model the behaviors of a system and to mathematically verify that the system design and implementation satisfy certain functional and safety properties [17]. Although the software industry favors the non-formal approach that steers away from spending too much time and resource in software specification, formal methods emphasize precise specifications thus potentially reducing bugs and facilitating accurate tests and code generation. One of the most well-know formal languages is the Specification and Description Language (SDL) standardized by the ITU (International Telecommunication Union) as Recommendation Z.100 [10]. The key features of the language are: the ability to be used as a wide spectrum language from requirements to detailed design allowing automatic code generation; suitability for real-time, stimulus-response systems; presentation in a graphical form; a model based on communicating processes (extended finite state machines); object oriented description of SDL components. SDL was conceived for use in telecommunication systems, but is also suitable for all kinds of real-time communicating systems. It describes the co-operation of these systems with their environment and the internal structure of a system.

In a typical software development cycle, verification and validation (V&V) are the two processes involved in ensuring the correctness of the software. Verification is defined as the process of checking whether the implementation of the software satisfies the design specification or not. The goal of verification is to answer the question: "Have we done the thing right?" A verification process typically comprises the executions of interactive or automated test cases. These test cases determine whether the execution of the implemented software, given a certain set of input data, would produce output data that match those that are generated by the conceptual (design) model. Checking for distributed system properties, such as deadlock, memory leak, and etc, is generally a part of the verification process. On the other hand, the goal of validation is to check whether the implementation satisfies the user requirements [38]. The user requirements are typically expressed as use cases which may be expressed in various notations , such as MSC [11], HMSC [49], URN [39] which includes Use Case Map [40] and GRL [41], Use Case Diagram [27], and others. By validating the model, we can be more confident that we might have done "the right thing" [42].

In our work, we emphasize validation because our formal specification must be compatible to the call flows described in various IETF documents [4,6,7] which are the basis for our use requirements. We have chosen UML's Use Case Diagram [27] to characterize use cases and MSC to characterize use case scenarios over other notations because: 1) Use Case Diagrams are one of the most popular notations among commercial tools to capture use cases, and we would like to investigate how practical it is to capture use cases with use case diagrams and to realize these diagrams with use case scenarios, 2) MSC is a proven notation for capturing execution traces (scenarios) and can be used for validation, 3) we could deduce use case diagrams from the informal SIP service specification described in [4,6,7] and infer the need for an "Abstract User interface" to enrich our SIP service specification. We also discovered that we could describe the use cases more precisely as use case scenarios written in the form of MSCs, 4) MSCs are well integrated with SDL, which is our chosen modeling language, and supported by a popular development tool called Telelogic Tau [12]. Regardless of the development methodology, or language tools being used, software engineers have acknowledged that the ability to validate and verify the intermediate and final products is an integral part of a successful software engineering process which enables efficient development of reliable software [25].

In our work, SDL was chosen over UML [27] or other modeling languages as the modeling language for this research because: 1) a number of well-known commercial tool development companies, such as Telelogic [12] and Cinderella [26], support SDL and offer flexible and powerful validation and verification features, 2) an asynchronous signaling protocol like SIP is state-oriented, thus expressing its services in extended finite state machines would be fitting and convenient. Also, SDL offers the ability of validating the design against messaging sequence charts. Since most informal protocol and service specifications for SIP in the IETF drafts describe the functional behaviors in graphical message sequence chart-like format (call flows), this ability of SDL allows us to express these call flows in the form of message sequence charts and to validate the model against these charts.

In this chapter, we will discuss the background material on formal software specification. The first subsection will describe the basics of the Specification and Description Language (SDL). Then, the second subsection would give an overview of Message Sequence Charts (MSC), which are used as our language to describe scenarios. Finally, the chapter ends with an overview of various verification and validation (V&V) techniques using SDL and MSCs.

2.1 Overview of SDL

An SDL specification is a formal model that defines the relevant properties of an existing system in the real world. The specification is divided into two parts: the system and its environment. Everything outside the system belongs to the environment. The SDL specification defines how the system reacts to events in the environment that are communicated by signals sent to the system. The behavior of a system is the joint behavior of all the process instances contained in the system, which communicate with each other and the environment via signals. All process instances exist in parallel with equal rights. A process instance may create other process instances, but once created, these new process instances have equal rights. SDL process instances are extended finite-state machines, a class of automata from automata theory. An extended finite-state machine (EFSM) is based on finite-state machine (FSM) with the addition of variables to the explicit states of the process instance. SDL process instances are communicating extended finite-state machines which run

concurrently and influence each other by exchanging signals that may carry information via parameters [16].

In general, a user may view an SDL specification as a top to bottom hierarchy, starting with systems, then blocks, and finally processes. From SDL-92 version onwards, the language allows the user to specify types in the global spaces to be reused at that level and the enclosed units. The concept of libraries or packages in SDL terminology is introduced to include types or data definitions that can be reused by more than just one system. Thus, a system can enclose block instances, block types, process instances, or process types. Similarly a block may contain other block instances, block types, process instances, or process types (see Figure 3). Although the channels connected between systems and blocks show the static behaviors of the model, the specifications of process instances and types that include state transitions and input and output signal processing contain the dynamic behaviors of the model.

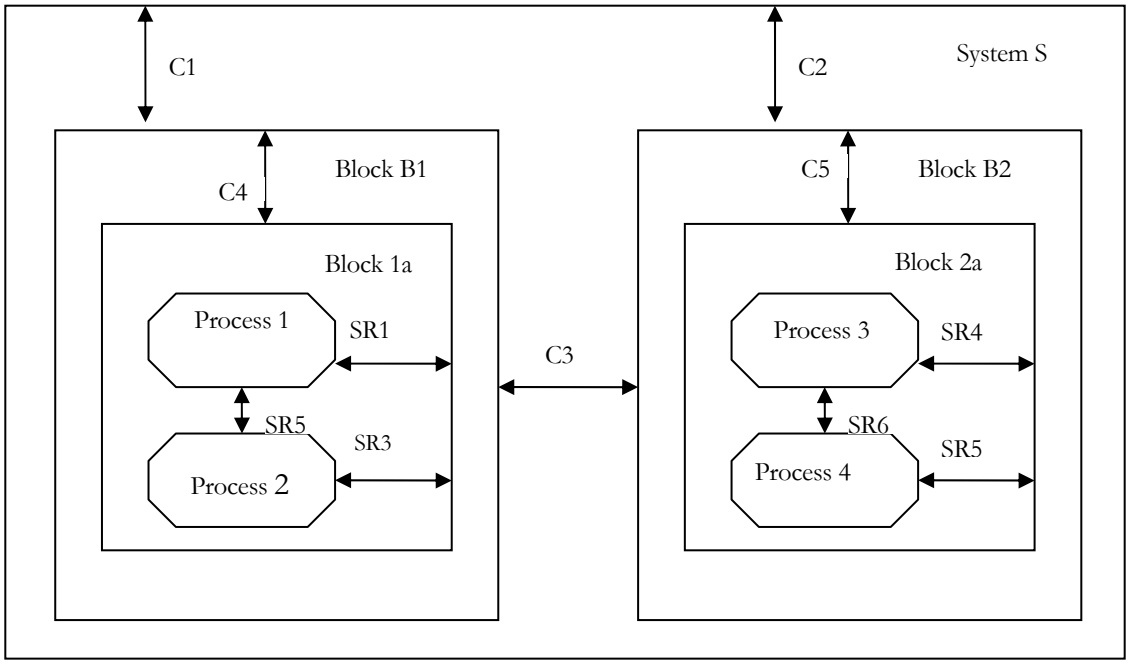

Figure 3: Basic SDL specification hierarchy

The interconnections between a system and the environment, system and blocks, and a block with another block, are called channels. Similarly, the interconnections between two processes and between a process and its enclosing block are called signal routes (see Figure 3).

2.2 Overview of MSC

A Message Sequence Chart (MSC) is a graphical way of describing asynchronous communication between processes. A chart does not describe the total system behavior, but is rather an execution trace. Such an execution trace shows the messages/signals being passed between communicating processes at different points of the execution. While specification and design languages like SDL describe both the structure and behavior of the system, a message sequence chart describes only the actions of each entity and the interactions between entities (or components) in a system. An MSC can capture one or more use case scenario of the system; it is only a small portion of the total system behavior [15].

From the previous section, we understand the environment and the system form the basis of an SDL world. Within a system, there are different type of entities that interact with each other and the environment through signal passing. So, how do these entities relate to the MSC instances that we represented by a vertical line within an MSC (see Figure 4)? An instance in an MSC can be an SDL system, an SDL block instance, or an SDL process instance [11].

In an MSC, an entity that communicates with other entities via message passing is called an "instance". An instance consists of an instance head and an instance end, which are connected by a timeline. The instance head and the instance end represent the start and end of events on the instance timeline within the MSC. Analogous to SDL, the creation and termination of instances may be specified in MSCs. An instance may be created by another instance. No message events before the creation can be attached to the created instance. The instance stop is the counterpart to the instance creation; an instance can only stop itself whereas an instance is created by another instance. The timeline of an instance contains a sequence of events. The most basic events are output and input of messages. Each message has exactly one output event and one input event. Messages are communicated between instances or between an instance and the environment. The events are ordered along each timeline, but events on different timelines are not ordered. This means that MSC cannot describe absolute time. Also, the distances between events on the timeline have no significance [11]. A MSC describes the possible sequences of events, but says nothing about the underlying causes of the events.

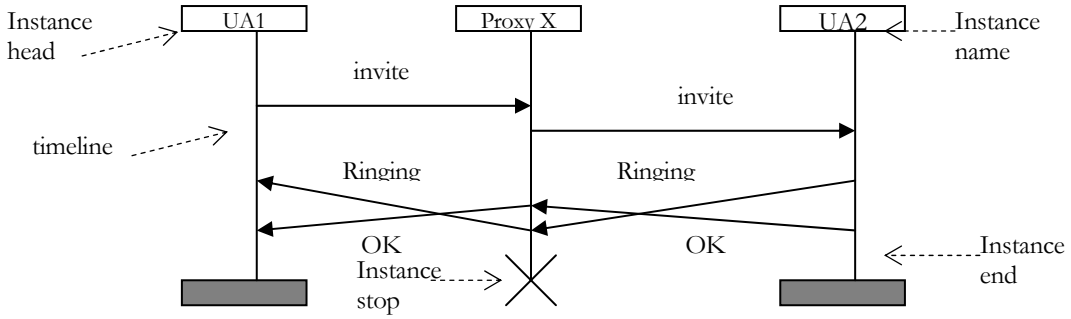

Figure 4: A sample SIP MSC

A requirement engineer can specify the setting of a timer and the subsequent time-out due to the timer expiration, or the setting of a timer and the subsequent timer stop. Setting and guarding conditions can be used to restrict the traces that a MSC can take. General ordering is used to impose additional orderings upon events that are not defined by normal ordering given by the MSC semantics (see Figure 4). For example, to specify that an event on one instance must happen before an otherwise unrelated event on another instance, or to specify ordering on events contained within a co-region. A co-region is used to specify that the consumptions of the contained input events are not ordered in time. Sample SIP service call flows have been converted into MSCs in our SDL model [43]. Please refer to the ITU specification of MSC [11] for more detailed explanations on MSC.

2.3 Overview of validation and verification techniques using SDL and MSC

In general, there are five common types of validation and verification (V&V) techniques for modeling a system using SDL and MSCs. If we take Tau as our example of an SDL/MSC development platform, Tau offers many V&V reachability analysis features: bit-state, exhaustive-state, random walk state space exploration, and MSC verification, plus Observer Process Assertions.

Bit-state space exploration is based on the bit-state hashing algorithm first proposed by G. Holzmann [14]. It is a variation of reachability analysis. The search algorithm is still based on the depth-first search algorithm, but it speeds up the lookup of the already explored states by storing these visited states in one or more hash tables. Since collisions can occur with any

hashing function, the size of the hash table(s) usually affects the performance of the algorithm; the larger the hash table, the fewer the number of collisions, the faster the search would complete. However, we would also need more memory for the search. Thus, if one were to choose an optimal size of the hash table(s) for a test, one must understand the tradeoff between the speed and the memory requirements for the simulation.

On the other hand, exhaustive-state space exploration is simply a depth-first-search reachability analysis of the model without hashing. It is typically inefficient and thus is not a useful approach. Random walk state space exploration is a variation of bit-state space exploration; the algorithm randomly picks the next branch for searching. As with any heuristic -based algorithm, this approach may offer a good best-case performance but cannot guarantee a better (or worse) average-case performance. If one of these exploration schemes is selected, Tau may report two types of results: deadlock, or unreachable states [12].

If the option of 'Verify MSC" is selected in Tau, Tau would verify the model against a given MSC if there exists an execution path in the SDL model such that the scenario described by the MSC can be realized. Thus, an MSC in Tau is considered as an existential quantification of the scenario. The searching algorithm for a matching execution trace is the bit-state hashing algorithm. When "Verify MSC" is selected, Tau may report three types of results: verification of MSC, violation of MSC, and deadlock.

Another validation alternative is to use Tau's Observer Process Assertions [12]. Observer Processes with powerful access to signals and variables of various processes are provided by Tau as a tool which is known to be useful for feature interaction detection. However, it can also be used for writing test cases. The way the Observer Process feature works is that one or more observer processes can be included in an SDL system to observe the internal state of other processes during validation [12]. When the validation engine is invoked to perform state exploration, the developer could activate the observer processes via the command interface of the validation engine. If the developer chooses to activate the observer processes, the assertions in these observer processes would be evaluated during the validation. The observer processes remain idle until all the observed processes have made their transitions. Then, each observer process would make one transition. All observer processes have access to the internal states of all the observed processes via the Access operators. Therefore, the conditions (e.g. the continuous signals, decisions and enabling condition) in which the observer process makes a transition may be considered for defining assertions for the observed processes. If the

observer process finds that the assertion is violated, it will generate a report in the middle of the validation. The validation process would be stopped immediately and the report would be presented to the user. Obviously, we can express our test cases in the form of Observer Process Assertions. A violation of an assertion would mean the failure of the corresponding test case. In Section 5.2, we will show how we characterize various feature interactions in the form of Observer Process Assertions.

Unless Tau is set to perform 100% state space coverage (by selecting exhaustive state space exploration), partial state space coverage is generally performed and reported in most cases [12]. Verification of an MSC in Tau is apparently achieved by using the MSC to guide the simulation of the SDL model. As a result, the state space of the verification becomes manageable [15].

However, as the complexity of a model increases (e.g. our SDL model for SIP), the state space of the model also increases exponentially; the resource and computational speed required by all V&V techniques also increase exponentially. As a result, no V&V tools (including Tau) can perform 100% state space coverage for average-size specifications in a reasonable time. In addition, Tau selects a small range of test values for each message and entity (e.g. block, process) parameters during the validation process; for example, three test values (e.g. 55, 0, -55) are used for integer type [12]. This is far from complete test coverage. Thus, if Tau does not report any failures (violations of message sequence charts) or faults (deadlock) during verification and validation, it does not mean that we have tested all possible execution scenarios; thus if we cannot find any failures or faults with Tau's Validator, we can say the model is relatively robust but cannot guarantee the model to be 100% correct [12]. Nevertheless, our objective is simply to validate our SIP services against the success scenarios of the published call flows [4,6,7], therefore we believe the functionality offered by Tau's Validator is sufficient for our purpose.

3 Extended Classification of Feature Interactions

Although the research community of FI's has generally categorized FI's by the nature and the cause of the interaction, as explained in Section 1.2, we believe the latter categorization (by cause) may be useful in classifying FI's, but the classification does not lead explicitly to a general scheme for detecting or resolving FI's. A classification process (taxonomy) is most useful if it leads to methods of detecting, preventing, or resolving FI's. While understanding the causes of interactions could give us some ideas on how to detect feature interactions, the causes of interactions are too vague for formulating precise design rules to prevent feature interactions, and policies to resolve them; for example, a resource contention scenario like presenting a voice greeting and call waiting tone to the same user may or may not be considered as a FI. Also how does a service designer know which "resource" to declare for checking resource contention? Thus, the causes cannot be construed as the most efficient means to derive methods for detecting FI's. We propose adding another type of categorization to the taxonomy of FI's, which is categorization *by symptom or effect*. This new categorization enables FI's to be associated to some of the well-known distributed system properties. As a result, we believe existing verification and validation techniques or tools for detecting these distributed system properties may be used to facilitate detection of FI's, especially in IP Telephony such as SIP.

The following new categorization of feature interactions facilitates detecting these feature interactions. The categorization focuses on the effects or symptoms of the problems that are more specific and can be used to narrow down the search for detecting feature interactions. Common distributed system properties such as livelock, deadlock, fairness, and non-determinism have a well-defined meaning and tools such as Telelogic Tau [12] can verify some of these properties in a given system. The category called incoherent interactions is also introduced in the following subsection.

3.1 Livelocking (LLCK) and Deadlocking (DLCK) interactions

Livelock is a well-defined computer science concept that occurs when the affected processes enter coordinated state transition loops and make no progress. For example, in the case of Automatic Callback (AC) and Auto Recall (AR), if both features receive the busy signal from the callee and initiate the call simultaneously, both calls would not go through on single-line phones because both ends are busy making calls. Both features are in a livelock situation because they may always get the busy tone when they repeat the process and never complete the call. A livelocking interaction needs not be permanent; the affected features may not be able to reach certain states for an undetermined duration, though they would eventually reach those states. In most cases, a trigger event like a "No Answer" tone could eventually break the livelocking interactions between two features. However, we still consider such situation as a livelocking interaction because the users have experienced undesirable effects of the interacting features in their call.

Deadlock is another well-known distributed system concept that occurs when two or more processes are in a blocked state because they require mutual exclusive access to a shared resource that belongs to the other, or wait for a message from the other process that will never be sent. An example is described in Figure 5 involving Call Waiting and CFB at A communicating with Call waiting and CFB at B.

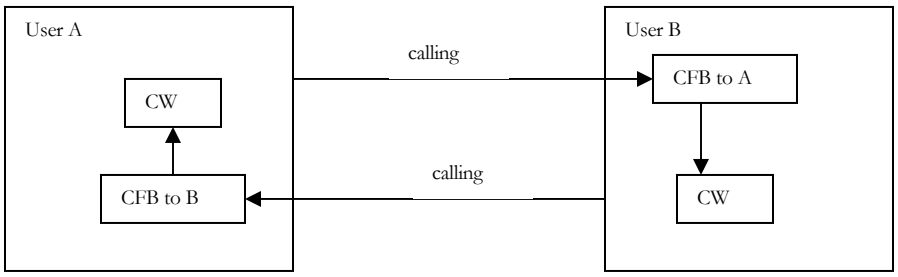

Figure 5: Call waiting and CFB at A versus Call waiting and CFB at B (deadlock)

If A and B call each other simultaneously and are programmed to forward to each other on busy, then both callers would keep hearing the phone ringing at the other end (and perhaps also hear the call waiting tone). They would be deadlocked at the ringing state and no progress would be made until one of them hangs up. If both ends do not subscribe to CW service, the call would be forwarded to each other on busy (or a call loop). Both users would be presented with a busy tone instead.

3.2 Incoherent (ICOH) interactions

An incoherent interaction is a form of a violation of feature assumptions (or properties). This term was first introduced in [13] to describe "the identification of specific incoherence properties" between the affected features. Furthermore, an incoherent interaction can be caused by resource contention and resource limitation. The two properties from the classical case of Call Forward Busy and OCS (see Section 1.2.1.3) is a good example; user A would successfully call user C via CFB even though an OCS entry at A is supposed to forbid any outgoing call to C. The CFB and OCS are programmed with contradictory assumptions.

3.3 Unfair (UFR) interactions

Fairness is also a well-known distributed system concept in which processes of equal priority should be scheduled fairly so that they eventually proceed and have fair access to shared resources. A novel case of unfair feature interaction occurs between Pickup (CP) and Auto Answer (AA) (also known as Call Forward to Voicemail). If one of the Call Pickup destinations has subscribed to AA that would unconditionally forward any incoming calls to the voicemail, the call would always be answered by the destination with AA first.

3.4 Unexpected Non-deterministic (NDET) interactions

Unexpected non-determinism is introduced here because it is an observable feature interaction. It occurs when a feature with non-deterministic behavior triggers other features that have normally deterministic behaviors, but due to this triggering behave in a non-deterministic fashion. The users are usually confused by the behavior of the affected features because they do not expect such non-deterministic behaviors. This is usually caused by timing and race conditions among the features. A case of this type of feature interaction is between Automatic Call Distribution (ACD) and Call Pickup (CP) (see Figure 6). The ACD feature allows incoming calls to the subscriber be redirected to one of the pre-programmed

destinations (e.g. destination A or B). The redirecting policy can be a random selection or a deterministic algorithm. The CP feature allows the subscriber of the service to inform a list of destinations (destination A and B) that an incoming call has been put on hold, and is ready to be picked up by one of the destinations. It is conceivable that the switching element Y sends a call pickup message to all destinations and then another switching element called X with ACD would redirect the call to a particular destination (e.g. B). When destination A decides to pick up the call, the incoming call is no longer available because the call has already been redirected to destination B. In summary, the non-deterministic call redirecting policy of ACD affects the first-come-first-serve call pickup policy of CP.

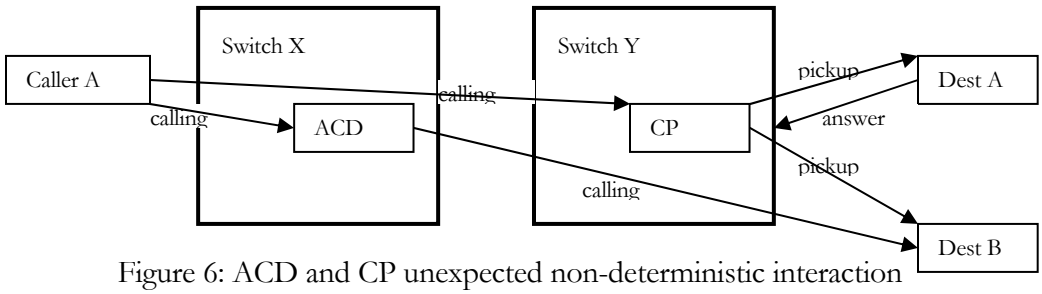

Figure 6: ACD and CP unexpected non-deterministic interaction

3.5 Feature Interaction Tree (FIT)

The following diagram (Figure 7) illustrates the graphical representation of the extended taxonomy, which is called the "feature interaction tree" (FIT). Since a feature interaction can be associated to just one kind of interaction, but one or more cause-effect category pairs, a feature interaction may be defined as a spanning sub-tree within the following feature interaction tree. Two or more FI's may overlap with each other on the FIT (Figure 7). In addition, each effect-type interaction (e.g. livelocking (LLCK)) is associated to one or more preventive measures (not shown in the diagrams).

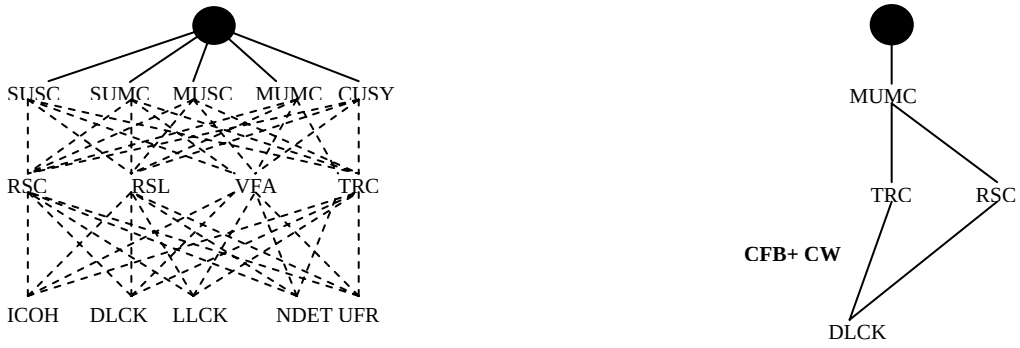

Figure 7: Feature Interaction Tree (FIT) and CFB+CW example

Although the FIT appears to be a graphical notation of the extended taxonomy at the moment, it is more than a simple graph (tree); it does provide a view of the interactions among features. By examining the patterns (e.g. frequency of feature occurrences) in the FIT, we can infer an appropriate feature interaction detection, prevention, or resolution strategy for a feature by examining the interaction patterns of the feature. In this sense, a FIT can be transformed into individual graphs with edges being associated to a feature and the frequency of the feature. Figure 8 is an example of a FIT for the Call Forward Busy (CFB) feature. In this example, the CFB feature has incoherent interactions with both the Originating Call Screening and Terminating Call Screening feature. Since CFB has been associated to the category of incoherent interactions, we can derive a FIT for CFB with the CFB edge being assigned with a frequency value of 2.

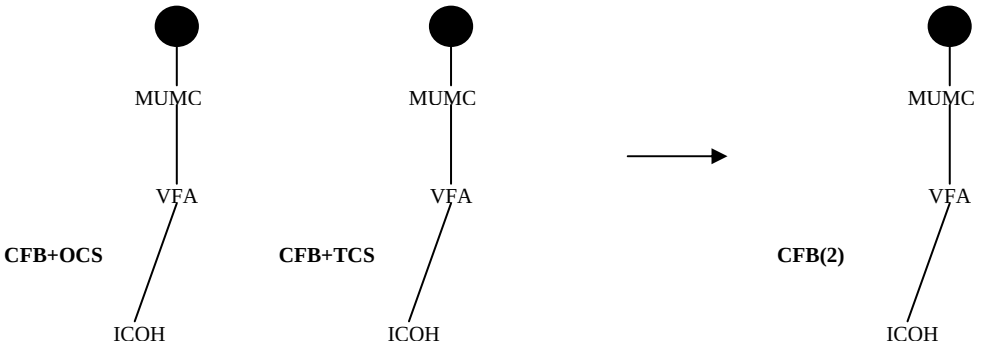

Figure 8: Deriving a FIT for CFB

As a result, we can derive a FIT for each feature. By comparing the frequency values of the interacting features, we could prioritize the tasks for feature interaction detection, prevention,

or resolution. For example, if a feature like CFB appears frequently in both deadlocking and incoherent interaction categories, we would develop more test cases to detect deadlocking and incoherent interactions for CFB and other features. On the other hand, we have not found any cases in which features such as OCS and TCS cause deadlocking and livelocking interactions. Therefore, since testing could be a time consuming exercise, one could presume writing test cases for detecting deadlocking and livelocking interactions among OCS, TCS and other features as a lower priority task. Also, since features such as CFB and ACD seem to cause many livelocking interactions with other features, one could presume CFB and ACD as the likely livelocking interaction candidates with new features.

In addition to the feature occurrences, we have not explored other interesting properties that the "FIT" may offer. For example, if deadlocking interaction could happen in the CFB+CW pair and the Call Transfer (CT) + CW pair separately, the question would be: could the feature interaction leading to a deadlock be transitive? That is, could deadlocking interaction also happen in the CFB+CT pair? The transitive property is always an interesting property to look at first because we may deduce new feature interactions from existing feature interactions. In this case, CFB could have deadlocking interactions with CW because the CFB calling party continues ringing the forwarding proxy with the busy signal being blocked by the CW at the forwarding party. Likewise, unattended CT blindly transfers the call to the remote party whose busy signal is blocked by its CW feature. However, we find that CFB and CT would not have deadlocking interaction because they do not hold and compete for the same resource (busy signal). From these specific examples, it seems to us that this feature interaction is not transitive. Further examination in formalizing the definition of feature interaction tree (FIT) would be an interesting future work item, but it is beyond the scope of this thesis.

4 Service Specification of SIP in SDL

This chapter describes the main contribution of our research, that is, to come up with a formal service specification of SIP in SDL. This contribution is significant because the current specification documents for SIP maintained by IETF are informal and protocol-oriented. Using our formal service model of SIP, we can examine the problem of feature interactions. Thus, we can detect and resolve (prevent) feature interactions in our specification which would describe SIP services more precisely than those described in [4,6,7]. In this chapter, we will first discuss our general design methodology which follows the typical software development lifecycle, starting from requirement analysis, then design/implementation, to testing. Then, we will describe our informal use case analysis, which consists of use case diagrams and use case scenarios. Next, we will explain the structural model and then the behavior model of the service specification. Finally, we end the chapter with a discussion on the verification and validation of our model.

4.1 Modeling Approach

Although the end result of this formal modeling exercise is to produce an SDL model that serves as a formal specification of SIP and some of the sample services, the modeling exercise fits into a bigger picture, that is, the overall development process to develop reliable SIP implementation. Before we dive right into the details of the SDL model, we present the overall design approach that we use in this project. Not surprisingly, the process is highly iterative by nature because the given description of SIP services is a collection of informal textual service descriptions and sample call flows [6,7] that do not offer all the detailed requirements up front. The process is best described by Figure 9, as a typical software development lifecycle, except it has incorporated formal modeling.

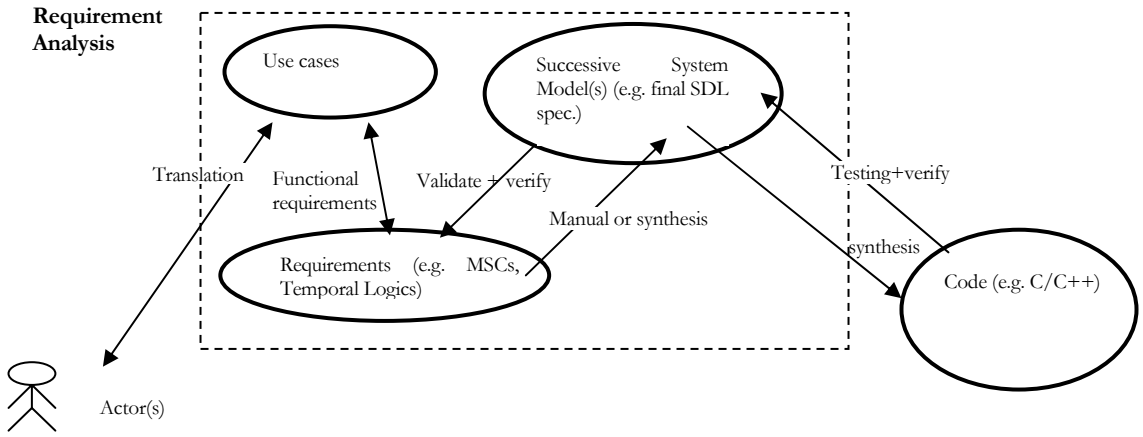

Figure 9: Typical Software Development Lifecycle

Tools that are based on formal methods can be used to ensure that the final SDL model would meet these requirements described in [4,6,7]. In a large project, we believe a system modeling team takes the responsibility of building system models and verifying that all requirements, particularly performance requirements, can be met. The system model serves as the final specification resulting from the requirement analysis. The design and implementation teams may use tools to synthesize the implementation or manually translate the specification to code. In this paper, the focus is on the generation of message sequence charts as both our use case scenarios and test cases, and the modeling of SIP and its services in the form of an SDL system. Synthesis is beyond the scope of this thesis.

We choose UML's Use Case Diagrams [27] as the graphical notation for capturing user and system requirements, because we want to find out whether we can capture the high level view of the relationships between the use cases and actors using a popular notation like Use Case Diagrams. Also, we would like to examine if it is possible to write our user case diagrams in such a way that we can realize use case scenarios from these diagrams. A High-level Message Sequence Chart (HMSC) describes how other message sequence charts may be combined to represent more complex cases [49]. In an HMSC, the MSCs are represented by MSC references that may be composed in sequence, in parallel or as alternatives. An HMSC does not depict instances or messages. It may contain start and end symbols, restrictive conditions, MSC references, connecting nodes, and connecting lines between nodes. In a sense, since we emphasize the use of MSCs to describe our scenarios and to validate our model, HMSC is a far better candidate than Use Case Diagram for linking these MSCs together to describe complex

scenarios. Nevertheless, we decided to experiment with Use Case Diagrams, and to determine whether it is possible to write these diagrams in such a way that we can generate use case scenarios manually from these diagrams.

After the sample service scenarios are translated into message sequence charts, an SDL model of SIP and selected services is created. We found that it is useful to define three sets or versions of the SDL models: the initial model, the version that complies with the published specification (the standard model), and the refined (final) version that includes the preventive measures for feature interactions (the refined model). In Figure 10, we show that the initial models involve validation of each feature against MSC test cases. Then, the standard model is checked for feature interactions. Finally, the refined model is produced.

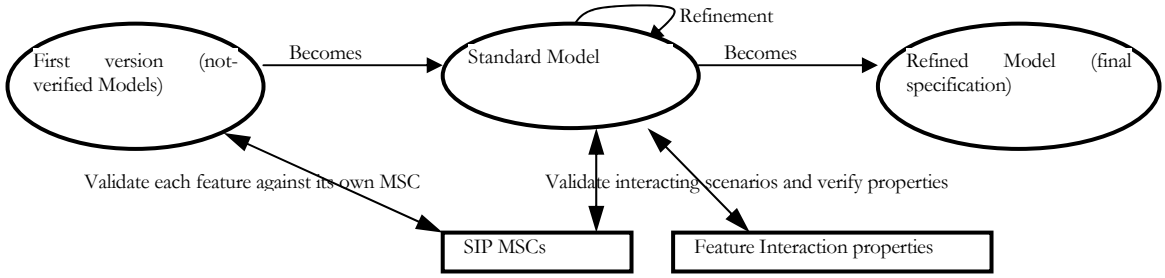

Figure 10: Iterative specification process

The approach to model SIP and its services begins with modeling the core signaling functionality of SIP; we call this the basic (telephony) service of SIP. The basic service includes establishing a two party call, terminating a two party call, suspending the call, presenting dial tone, busy signal, no answer signal, ringing, and alerting signal. The user agent and proxy are the only SIP entities that are modeled. After we complete the basic service, we add additional SIP services such as multi-party call signaling features (call redirection, forwarding, and holding (suspending)) to the model. These protocol features are essential to enabling the development of more complex telephony features such as CFB, CW, OCS, TCS, etc. Advanced Internet telephony features such as Call Forking (CF) and Auto-callback (ACB) will be added at the later stage. The complete SDL model [43] consists of over 60 pages of diagrams, thus only selected diagrams will be presented in this thesis.

4.2 Defining Use Cases as Use Case Diagrams

Use case diagrams [27] are one of the most popular notations for describing use cases, and most commercial UML-based tools support this notation. Many software developers like to use these diagrams because they have a simple syntax and convey the high level view of structural and behavioral relationships between use cases and actors. In our research, we would like to investigate whether it is possible to derive use cases from the given call flow diagrams and to describe them abstractly as use case diagrams, and more precisely as MSCs. The use case diagrams are based on the informal call flow diagrams and textual service specification described in various IETF documents [4,6,7]. Each actor in our use cases has a specific role in a service; Figure 11 shows the use case diagram of Call Forward Busy (CFB).

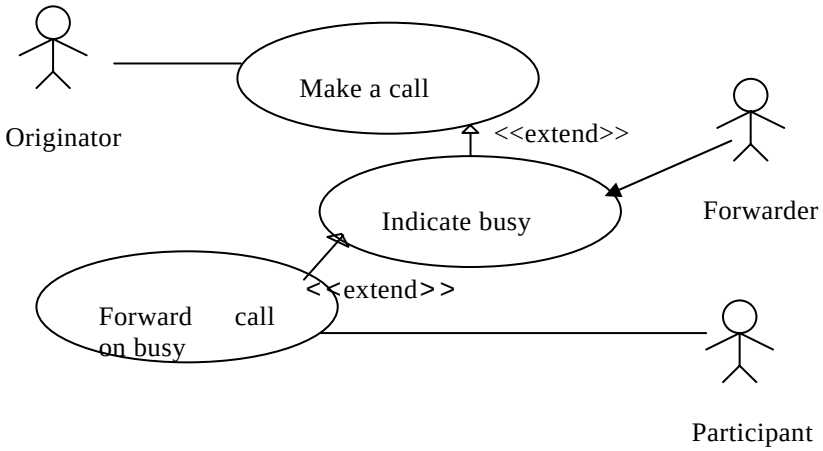

Figure 11: Use Case Diagram of Call Forward Busy

In this example, there are three distinct actors (originator, forwarder, and participant/forwardee). We can map these actors to actual MSC instances and each use case to a portion of the corresponding test scenario (execution trace). In Section 4.4, we show a test scenario (combined service and protocol scenario) of Call Forward Busy. In that scenario, each MSC instance corresponds to an actor in Figure 11; for example, the "SipUserAgent1_1" user agent process corresponds to the originator. The "SipUserAgent1_2" user agent process corresponds to the forwarder. The "SipUserAgent1_2" user agent process corresponds to the forwardee. Furthermore, we show the "indicate busy" use case extends the "make a call" use

case in Figure 11. The "extend" relationship is used to describe additional functionality that the extended use case has. In our research, we have purposely defined our use case diagrams such that they can be mapped easily to use case scenarios in the form of MSCs; each use case in a use case diagram corresponds to one partial trace in a use case scenario. We will show how we realize use case diagrams and the corresponding use case scenarios in the next section.

4.3 Realizing Use Case Diagrams with corresponding Use Case Scenarios

By definition, a use case scenario describes only the interactions between the actors. In our perspective, the end users are the only actors in a service specification. Therefore our use case scenario of SIP services must not show interactions between actors and supporting classes (e.g. proxy, location server). These interactions are the protocol (SIP) messages exchanged between SIP entities and are considered as "internal" interactions. They are not part of our use case scenario; instead, they will be added in Section 4.4.

In the previous section, we showed how we came up with use case diagrams. Now, we would like to describe these use cases more precisely using use case scenarios because we could be more specific on the functional behavior of our model with use case scenarios and we could also validate our model against these scenarios. Since the IETF drafts have provided a call flow diagram or success scenario for each sample service in graphical notation [6,7], we translate these scenarios into message sequence charts. However, we note that these call flow diagrams only include the sequence of exchanged SIP messages at the protocol level. They do not represent service scenarios in the sense of use cases. So we deduce the interactions between the actors and the system based on our use cases, the textual service description in [8], and the SIP messages that users can observe (e.g. ringing, invite (dialing)) described in the call flow diagrams [6,7]. The deduction process is very intuitive and we have successfully defined use case scenarios for various SIP services (e.g. basic call, CFB, OCS, TCS, CT, AC, AR, CW).

Following standard practice of software engineering, we think that it is important to define service usage scenarios at the interface between the user and the system providing the communication service. We have therefore associated each use case (e.g. "make a call", or "indicate busy", or "forward call on busy") described in our use case diagrams to a partial trace of a certain call flow scenario. Although we have obtained just one scenario from our call

forward busy use case diagram, we are not restricted to one scenario per one use case diagram; we could in fact obtain one or more service usage scenarios from a use case diagram. As an example, Figure 12 shows a use case scenario (service usage scenario) of CFB, which corresponds to the use case diagram in Figure 11 and also to the call flow diagram of CFB found in [6,7].

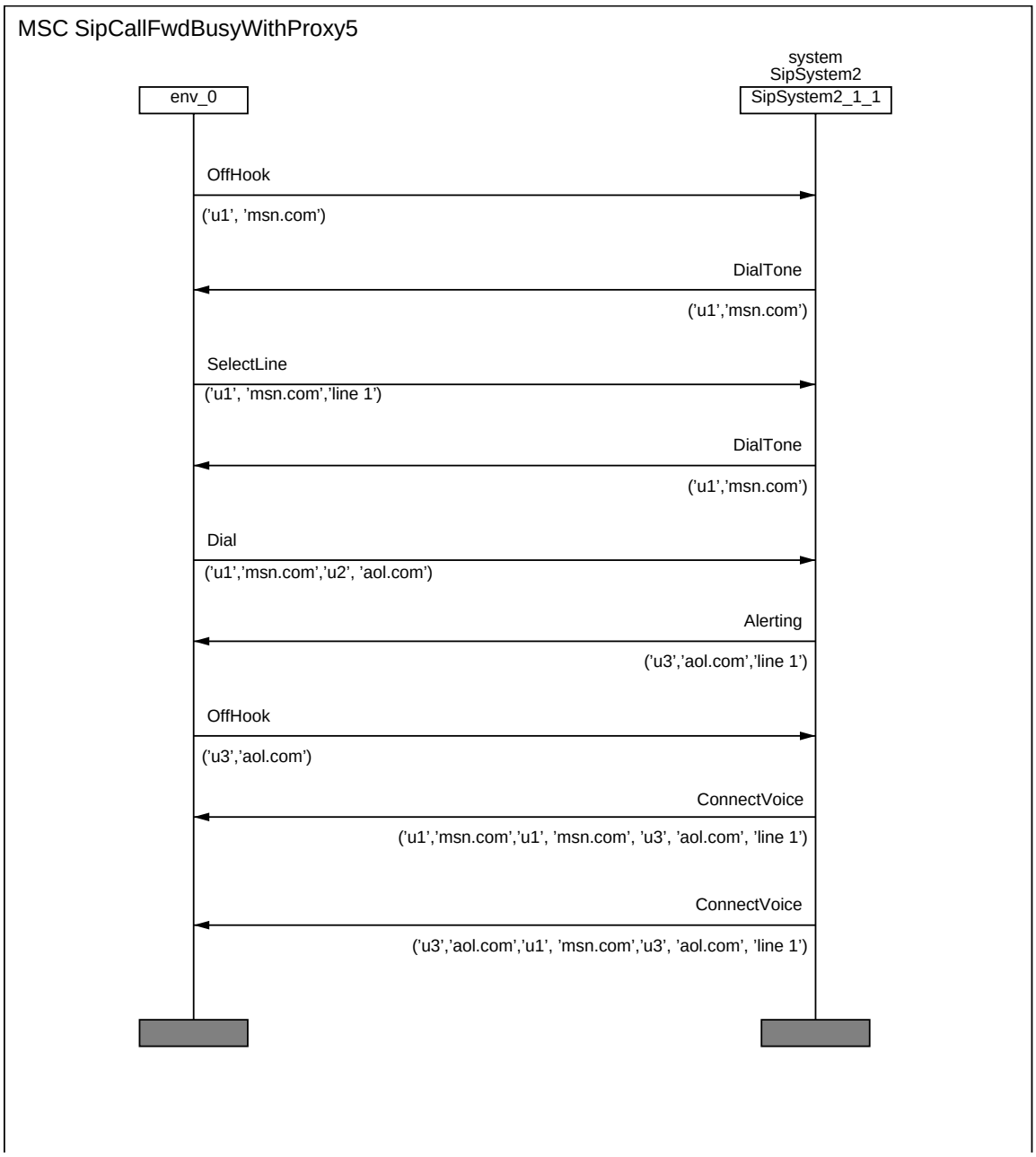

Figure 12: Call Forward Busy Use Case Scenario

We have also defined an abstract user interface (see Section 4.5.3) which represents the interactions at the service level. These interactions between the users and the SIP system can be used to describe use case scenarios of SIP services. The users are the actors of the use case scenarios and are represented by the environment "env_0" in SDL. Since the actors are all represented by the environment which is represented by a single process called "env_0" and

the SIP system hides all the user agent processes from the environment, it is confusing to see who the actual sender or receiver of each "Abstract User" signals is in Figure 12. Therefore, we purposely add two extra leading parameters ('u1', 'msn.com'), which stands for (u1@msn.com) indicating the address of the sending or receiving user agent hidden by the system, to all signals for distinguishing the originator or receiver of the signals inside the system. Note that, u1@msn.com represents the originator; u2@aol.com represents the forwarder; u3@aol.com represents the forwardee in Figure 11.

In an "extend" relationship shown in Figure 11, the derived use case inherits the functional behaviors of the parent use case, and may also include substitutions and additions to the inherited behaviors. For example, the "make a call" use case is mapped to a use case scenario similar to Figure 12 except the "ConnectVoice" signals would be sent to user "u2@aol.com" instead of user "u3@aol.com". With the "indicate busy" use case that extends the "make a call" use case, the corresponding use case scenario which is not shown here is similar to the scenario of the "make a call" use case except the "ConnectVoice" signals would be replaced by a "Busy" signal to user "u1@msn.com". Finally, the "forward call on busy" use case extends the "indicate busy" use case. The resulting use case scenario would have the "Busy" signal replaced by the two "ConnectVoice" signals as shown in Figure 12.

We conclude that it is possible to generate use case scenarios from use case diagrams, provided that these diagrams are written under our mentioned guidelines. However, we do not believe the use case diagram community intends to restrict their notation to this extent. Therefore, other notations, such as HMSC [49] and URN [39], should be used to describe user requirements. Applying these notations in our work is beyond the scope of this thesis.

4.4 Converting call flows to MSC as test case scenarios

The combination of the use case scenario with the corresponding scenario of exchanged SIP messages from [6,7] is represented in Figure 13. We may call such a combined scenario a service and protocol scenario. It can be used as a test case for validating the SDL specification of the SIP protocol, as explained in Section 4.7.

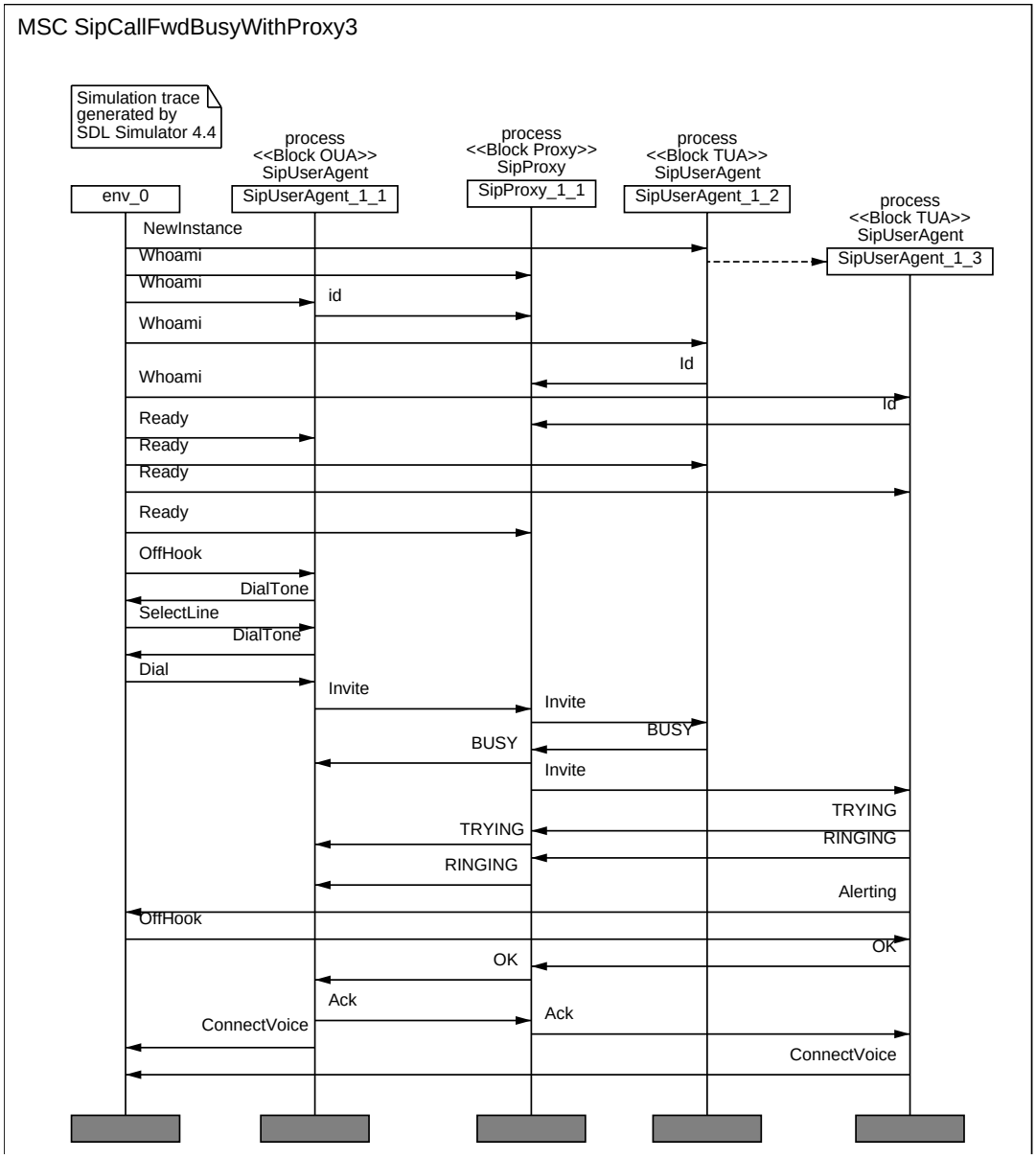

Figure 13: Call Forward Busy Test Scenario in IETF SIP Service Examples draft

The sequence chart shown in Figure 13 represents the Call Forward Busy "service and protocol scenario". We also call this the test scenario of CFB for normal operation. This sequence chart is the combination of the use case scenario with the corresponding scenario of exchanged SIP messages from [6,7]. We use this test scenario of CFB to demonstrate our process of converting informal requirements to message sequence charts. In addition, the test scenario of CFB together with the test scenarios of other services are written as message

sequence charts against which we use the Telelogic Tau's Validator to verify our SDL model. In Figure 13, user 'A' calls user 'B' who is busy and replies with a busy response message. Then, the proxy forwards the call to user 'C'. Since Figure 13 can be used as a test scenario MSC against which the model is verified, such test scenario must be a complete trace which includes the initialization phase of the simulation. However, the initialization phase (signal "NewInstance", "Whoami", "Id", and "Ready" are all operational management signals, see Section 4.5.3) is not part of the service specification; it is our invention for facilitating the simulation and verification of the model.

The call flow diagram and the service requirements of call forwarding described in [6,7] can be used in conjunction for creating a formal call forward busy specification. Before we begin developing the SDL model, we need to articulate the requirements in formal notations. The first step is to convert the call flow diagram to a syntactically correct message sequence chart. In Figure 13, the method name of the request message (e.g. 'INVITE') is converted to an SDL signal (or MSC message). The SIP message parameters would become the signal parameters. However, Figure 13 is written with response codes as message names and without message parameters so that the chart can fit on a single page. Also, we could use co-regions to express the general ordering of messages because the SIP RFC indicates that the arrival of a call setup response message, such as 'TRYING' or 'RINGING', could be out of order. However, we decided that we would not use co-regions in our message sequence charts because Tau does not support co-regions in the validation process. The symbol of instance end is used to end all instances in our message sequence charts because it defines the end of the description of an instance in a message sequence chart; it does not define the termination of the instance. An action that describes the interaction between a SIP entity and a non-SIP entity is written as a signal/message from a SIP entity to the environment in our message sequence charts. For example, "ConnectVoice" is a signal that is sent from the user agent to the media controller to establish RTP voice streams. Similarly, the "SelectLine" signal is used to select the line on the phone. Evidently, our use case scenarios written as message sequence charts together with the SDL model give a more precise formal service specification of SIP services than the IETF drafts.

After we have developed the SDL system model for SIP, we would verify the model against these message sequence charts which serve as the basic SIP compliant test cases for the system model. The final system model that includes all the preventive measures for feature

interactions is the final specification of the SIP protocol entities. The SDL model may also be used to synthesize the code base, but it is beyond the scope of this thesis.

The complete MSC can be used as a test case for validating the SDL specification of the SIP protocol, as explained in the next subsection. As a matter of fact, we have written one or more message sequence charts (service and protocol scenario) as test cases for each service because we use the Telelogic Tau's Validator to verify our SDL model against the combined scenarios.

4.5 Defining the Structural Model

In this section, the structural definitions of the SIP entities are discussed. The relationship between the modeled entities, their interfaces, and attributes are considered parts of the structural definition. A SDL system represents static interactions between SIP entities. The channels connected between various block instances specify the signals or SIP messages that are sent between user agents and/or proxies. Block and process types (e.g. SipUserAgentType, SipProxyType) are used to represent SIP entity types such as user agent and proxy. In addition, the Tau tool Organizer component [12] allows the service designer to partition the specification into a number of modules, called SDL Packages. SDL Packages are used to package SDL entities for reuse.

4.5.1 Core SIP entities

The following subsections describe the two main SIP entity types in the model: User Agent and Proxy.

4.5.1.1 User Agent

A SIP User Agent contains both, what is called in SIP, a user agent client (UAC) and a user agent server (UAS). Since a user agent can only behave as either a UAC or UAS in a SIP

transaction, the user agent is best represented by the inheritance of UAC and UAS interfaces. The inheritance relationship is modeled using separate gates (C2Sgate and S2Cgate) to partition the user agent process and block into two sections: client and server. The "Envgate" gate manages the sending and receiving of "Abstract User" signals between the user agent and the environment (see Figure 14). An instantiation of a block type represents an instance of a SIP entity such as user agent or proxy, and contains a process instance that describes the actual behavior of the entity. The process definition file contains the description of all the state transitions or behaviors of the features to which the SIP entity has subscribed. In addition, each SIP entity must have a set of permanent and temporary variables for its operations. In the case of a user agent, the permanent variables store the current call processing state values of the call session (e.g. To, From). The temporary variables store the values of the consumed messages for further processing.

4.5.1.2 Proxy

Similar to a user agent, a proxy consists of client and server portion. It tunnels messages between user agents but also intercepts incoming messages, injects new messages, or modifies forwarding messages on behalf of the user agent(s). A proxy is also a favorite entity to which features are deployed. If a proxy needs to keep track of the states of a session for a feature, it is considered a stateful proxy; otherwise it is called a stateless proxy. A SIP Proxy has four gates that interact with user agents or proxies: client-to-proxy (C2Pgate), proxy-to-client (P2Cgate), server-to-proxy (S2Pgate), and proxy-to-server (P2Sgate). The "Envgate" gate manages the sending and receiving of operational management signals (part of the "Abstract User" interface) between the user agent and the environment (see Figure 14).

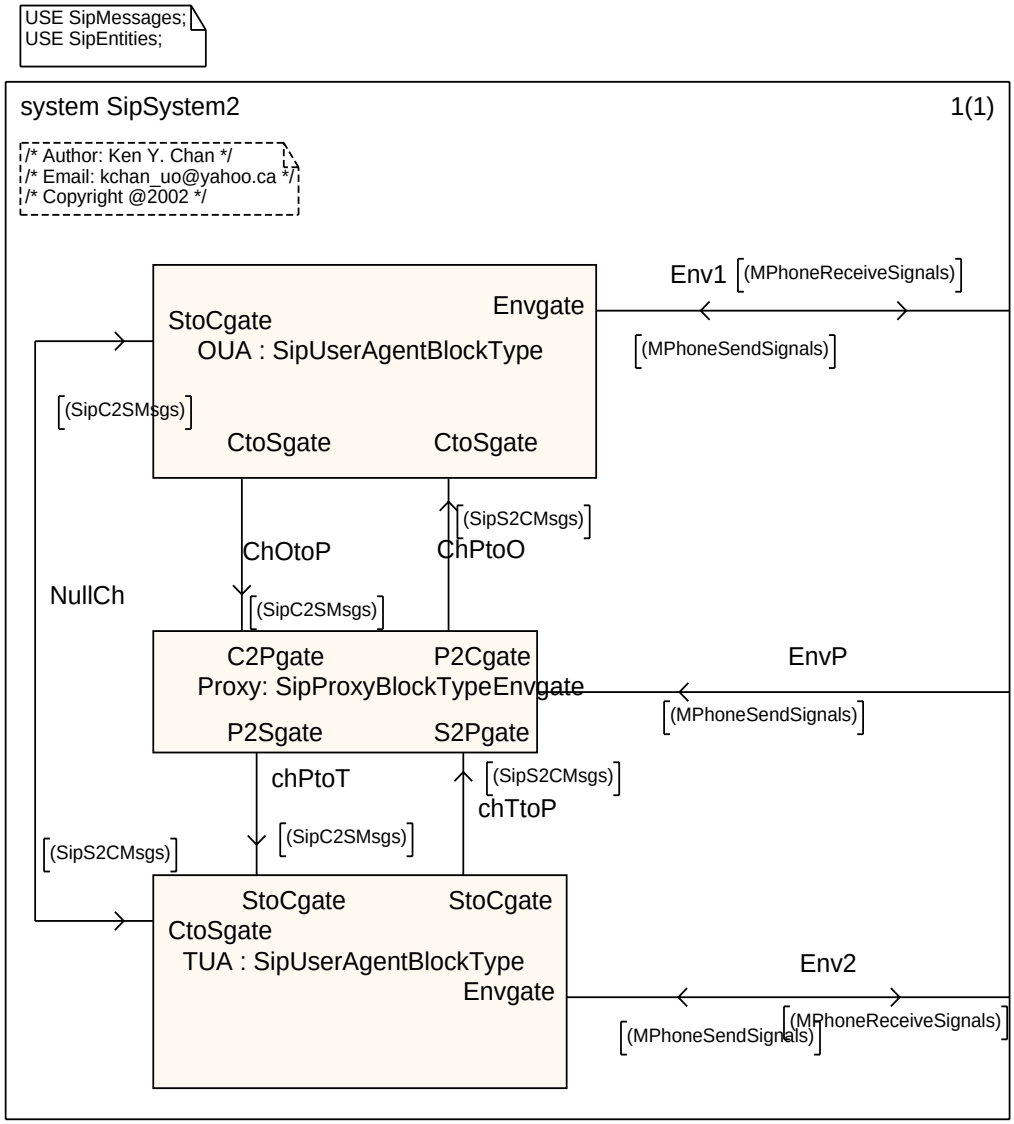

Figure 14: System Diagram of Proxy and User Agent

In our SDL model, we use different SDL systems to represent different structural bindings between SIP entities and to simulate a particular set of call scenarios. The most complex system in our telephony model (see Figure 14) realizes the concept of originating and terminating user endpoints. It contains an originating user agent block, a proxy block, and a terminating user agent block. The originating block contains all the user agent process instances that originate SIP requests while the terminating block contains all the user agent process instances that receive these requests. Upon receiving a request, a terminating user

agent would reply with the corresponding response messages. It is important to note that only the originating user agent and proxy instances can send SIP requests (including acknowledgements).

In the system shown in Figure 14, the originating user agent block has a channel called "NullCh" which is connected to the terminating user agent block's "CtoSgate". Signals are not supposed to be exchanged during the simulation in that system configuration because the originating user agent block would send requests from its "CtoSgate" (Client to Server) gate to the proxy block's "CtoPgate" (Client to Proxy) gate. A proxy process would then send the request from its "PtoSgate" (Proxy to Server) gate to the terminating user agent block's "StoCgate" (Server to Client) gate. As the diagram has illustrated, the response signals/messages would travel in the reverse direction starting from the "StoCgate" at the terminating user agent block. In our design, each gate in a block serves a purpose and may be used, or not used, in different system configurations. Thus, our SDL model offers great flexibility for different simulation tests.

All blocks are initialized with one process instance. During the simulation, a 'NewInstance' "Misc User" signal can be sent to a process instance to create a new process instance. Signals such as "NewInstance", "Whoami", "Id", and "Ready" are not "Abstract User" signals. They are created for the purpose of operational management, configuration and administration. We grouped all non-SIP signals (abstract user and management signals) to the "Envgate" gate. We believe it was an oversight. We should have created two separate gates, one for the abstract user interface and one for the management interface. This would be a part of our future work.

Before the first "INVITE" message is sent, the environment must initialize each user agent and proxy instance with a unique Internet address by sending them a message called "Whoami". The user agent instances would in turn send an "Id" message along with its Internet address and process id (PId) to the proxy. Thus, the proxy can establish a routing table for routing signals to the appropriate destinations during simulation. Similar to the user agent, a proxy has a set of permanent and temporary variables for its operations. In Figure 15, we have a block interaction diagram that describes the relationship between the process set and its block.

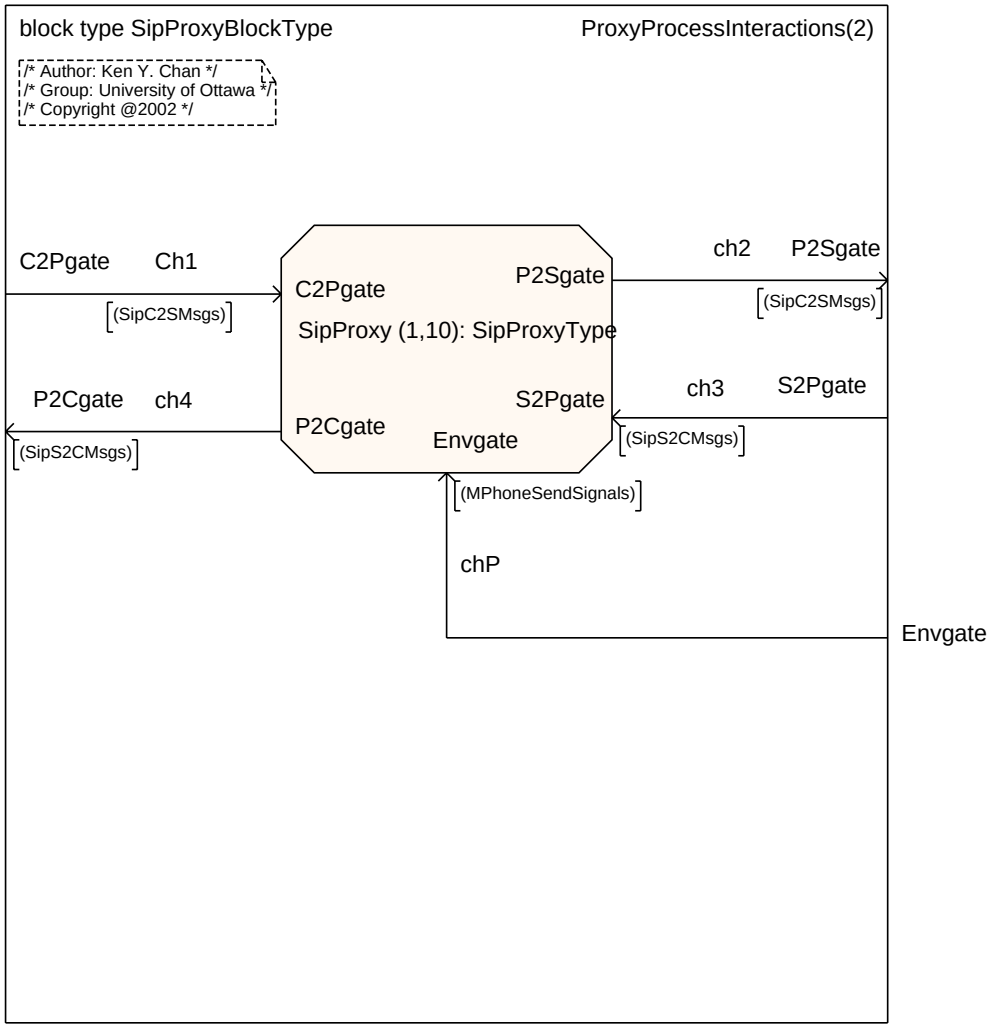

Figure 15: Block-Interaction Diagram of Proxy Type

The SDL system described in Figure 14 is only one such configuration. It is defined as a system instance in this SDL specification; we could define many other system instances to represent different network configurations (topologies). The system shown in Figure 14 characterizes a SIP network in which there are one block representing the originating user agent processes, one block of proxy processes, and one block of terminating user agent processes. Since processes are dynamically created in a block, we could not find a way to establish signal routes between processes within the same block dynamically. As a result, processes within the same block cannot exchange signals directly with each other. A process can communicate with processes from another block but not with those within the same block. Obviously, we cannot create a system configuration, in which an originating user agent and a terminating user agent are connected by series of proxies, with only one proxy block. If

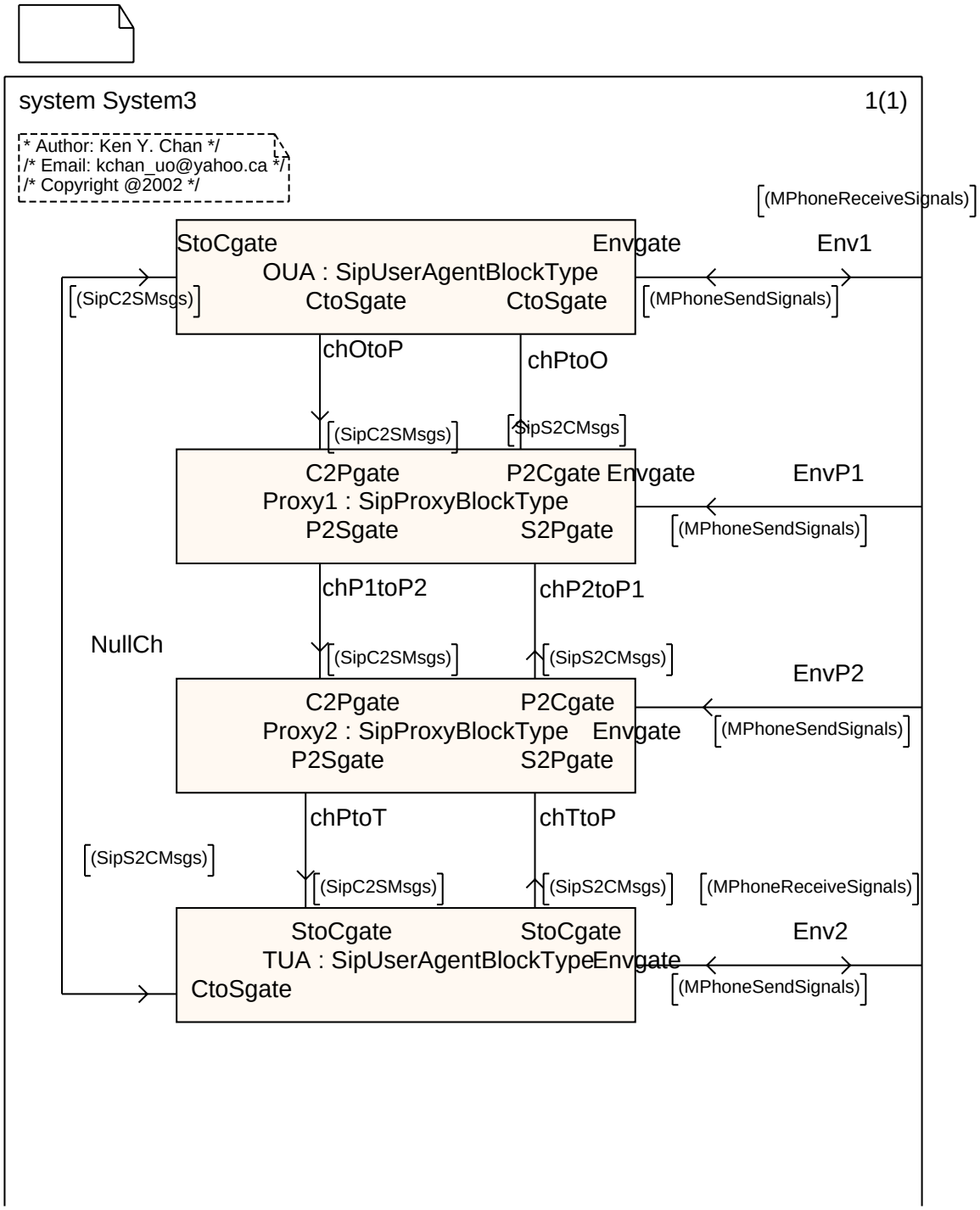

Figure 16: A System with Two Proxies Connected in Series

In Figure 16, we have a dedicated outbound proxy block named "Proxy1" connected to the originating user agent block called "OUA", and a dedicated outbound proxy block named "Proxy2" connected to the terminating user agent block called "TUA". The way the blocks are connected is very similar to the simple configuration illustrated in Figure 14, except for the extra proxy block. Additional proxy blocks can be connected in series in the similar fashion. Thus, we have shown that our SDL model supports all kinds of network configurations.

4.5.2 SIP Messages

SIP messages are defined as SDL signals in the SIPMessage package. We have defined only the main header fields of the SIP header because we are interested in only the operation (method) and the endpoints of the call session. These fields in a SIP message are represented by the corresponding signal parameters. Since we could not find any supports for *linked-list*-like data structures in SDL, the number of variable fields such as 'Via' and 'Contact' must be fixed in our model. Complex data and array types of SDL have been tried for this purpose, but were dropped from the model because they may cause the Tau validation engine to crash. Instead, we used a fixed number of parameters to simulate these variable SIP header fields.

4.5.3 Abstract User Interface

As mentioned in Section 4.5.3, a set of user signals (see Figure 17), that is not part of the SIP specifications [4,6,7], has been defined here to facilitate simulation and verification. They represent the interface between the user and the local IP-telephony equipment in an abstract manner; we call this interface the "Abstract User interface". The modeling of these user-observable behaviors is essential to describe feature interactions; however, the SIP protocol messages described in the SIP standard do not describe these user interactions properly. For example, an INVITE message in SIP may play different roles. It is more than just a call setup message; for instance, it may be used for putting the call on hold in the middle of a call (mid-call features) [4]. To make the simulation as realistic to real phone calls as possible, the "Abstract User interface" includes signals such as Offhook, Onhook, Dial, SelectLine,

CancelKey, RingTone, AlertTone, TransferKey which relate to the actions that are available on most telephones units on the market (see Section 4.3 and Figure 12).

In the SIP community, some researchers have attempted to specify telephony services as call flow diagrams [6,7] and as Java APIs (e.g. JAIN [50]). JAIN stands for Java APIs for Intelligent Networks. In general, the JAIN community has specified a framework for developing call control and telephony features in Java. Their framework is protocol-independent but has a mapping to SIP. In addition, their specifications are API (Application Programming Interface) driven; they are close to the implementation, in the form of Java source code. Although they have specified the interactions in the two-party and three-party call control scenarios, we could not find any detailed specifications for complex telephony services like Originating Call Screening, Auto-callback, Call Forward on Busy under their standardization process. Therefore, we believe our research, particularly in the "Abstract User interface", is complementary to their work.

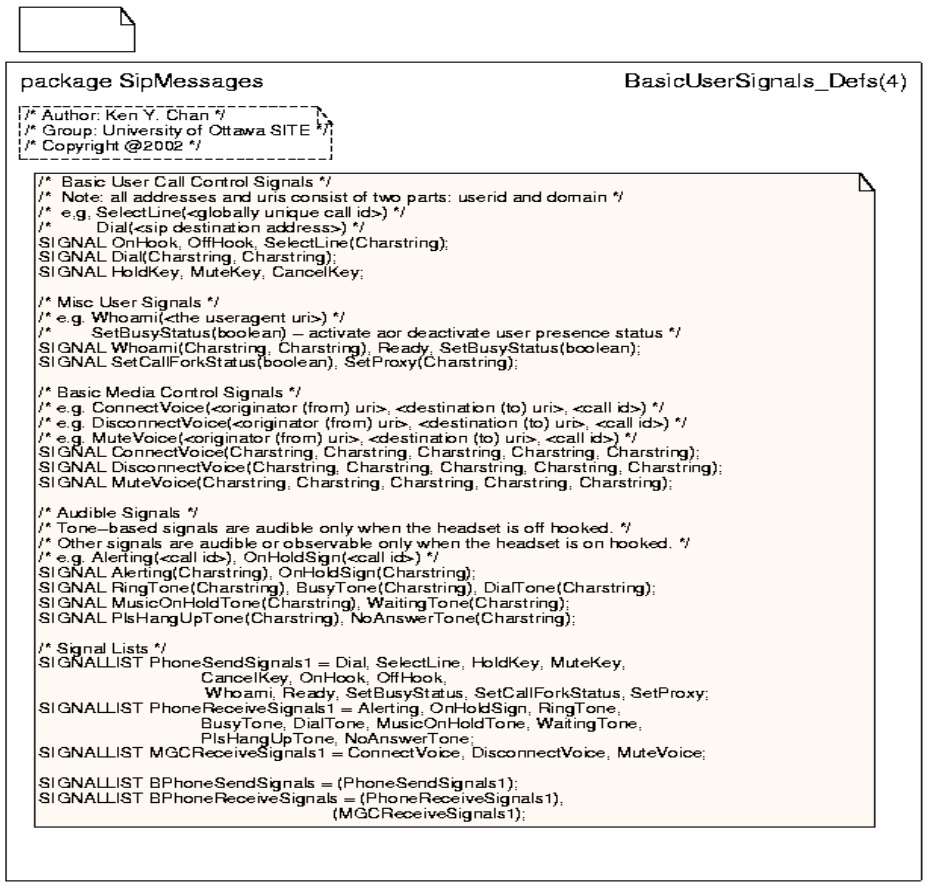

Figure 17: "Abstract User interface" (Signals)

The parameters of each "Abstract User" signal are designed to give an unambiguous semantics to a user action. For example, The 'ConnectVoice' signal has five parameters: From (user and domain), To (user and domain), and call-id. These parameters mark the logical endpoints of a call segment, which is used by a media controller to establish a voice stream between the two hops. All the output signals have the call-id as their parameter representing the line number of the call.

Finally, one may ask how these high-level "Abstract User" signals are connected to the protocol's low-level primitives (the SIP messages). We initially considered having a dedicated

process in each SIP entity block (SipUserAgentBlockType and SipProxyBlockType) to map these user signals to SIP signals which are managed by another dedicated process (SipUserAgent and SipProxy). However, we found it would introduce unnecessary complexity to the design so we abandoned this approach. Instead, each block type has one process which contains all the behavior of the block type. The mapping between "Abstract User" signals and SIP signals is simple. In Section 4.3, Figure 18 shows when the user agent receives a "HoldKey" "Abstract User" signal at the "Connect_Completed" state, it updates its state variables, sets the timer, and sends the corresponding "Invite" message to the destination. Then, the user agent would make a state transition from the "Connect_Completed" state to "SentOnHold" state.

Similarly, when the originating user agent receives a success response from the destination, the user agent would cancel the timer and send the corresponding "OnHoldSignal" signal to the environment (e.g. the caller phone alerts the user with an On-hold tone). Finally, the user agent would make a state transition from the "SentOnHold" state to the "OnHold" state. In general, the mapping between "Abstract User" signals and SIP signals is based on a relationship of trigger events and actions. For example, an "Abstract User" signal can trigger the sending of a SIP signal, and vice versa. We will explain this in more detail in Section 4.6.

4.6 Defining the Behaviour Model

In general, a SIP feature or service is represented by a set of interactions between users and the user agent processes, and possibly the proxy processes. Each process instance plays a role in a feature instance. A process instance contains state transitions which represent the behaviors that the process instance plays in a feature instance. In our model, we capture the behavior of a SIP entity in an SDL process type (e.g. UserAgentType). The first feature we model is the basic SIP signaling functionality, also known as the basic service. Each process type has state transitions that describe the basic service.

In general, the number of state transitions and input/output signals varies among SIP features. We have not made any effort to reduce the number of states or signals used in our specification, but rather tried to write well-structured, easily readable SDL models. Our main interest is to develop methods for designing IP telephony services with fewer feature

interactions. Thus, we emphasize on the functional design and verification aspect in our model.

4.6.1 User Agent Process Specification

In our model, we capture the behavior of a SIP entity in an SDL process type (e.g. UserAgentType). The first feature we model is the basic SIP signaling functionality, also known as the basic service. Each process type has state transitions that describe the basic service. In the case of a user agent, the process includes the UAC and UAS behavior. We define a feature role as the behavior of a SIP entity that makes up a feature in a distributed system. A feature role is invoked or triggered by trigger events. The entry states of a feature role in a SIP entity are the states of the basic service for which the trigger events are defined as inputs. For example, the on-hold feature can be triggered in the 'Connect_Completed' (entry) state by an INVITE message with 'ONHOLD' as one of its parameters (Figure 18). After the invite message is sent, a response timer is immediately set. Then, the user agent is waiting for the on-hold request to be accepted. When the other user agent responds with an OK message, the requesting agent would cancel or reset the response timer and inform the device to display the on-hold signal on the screen.

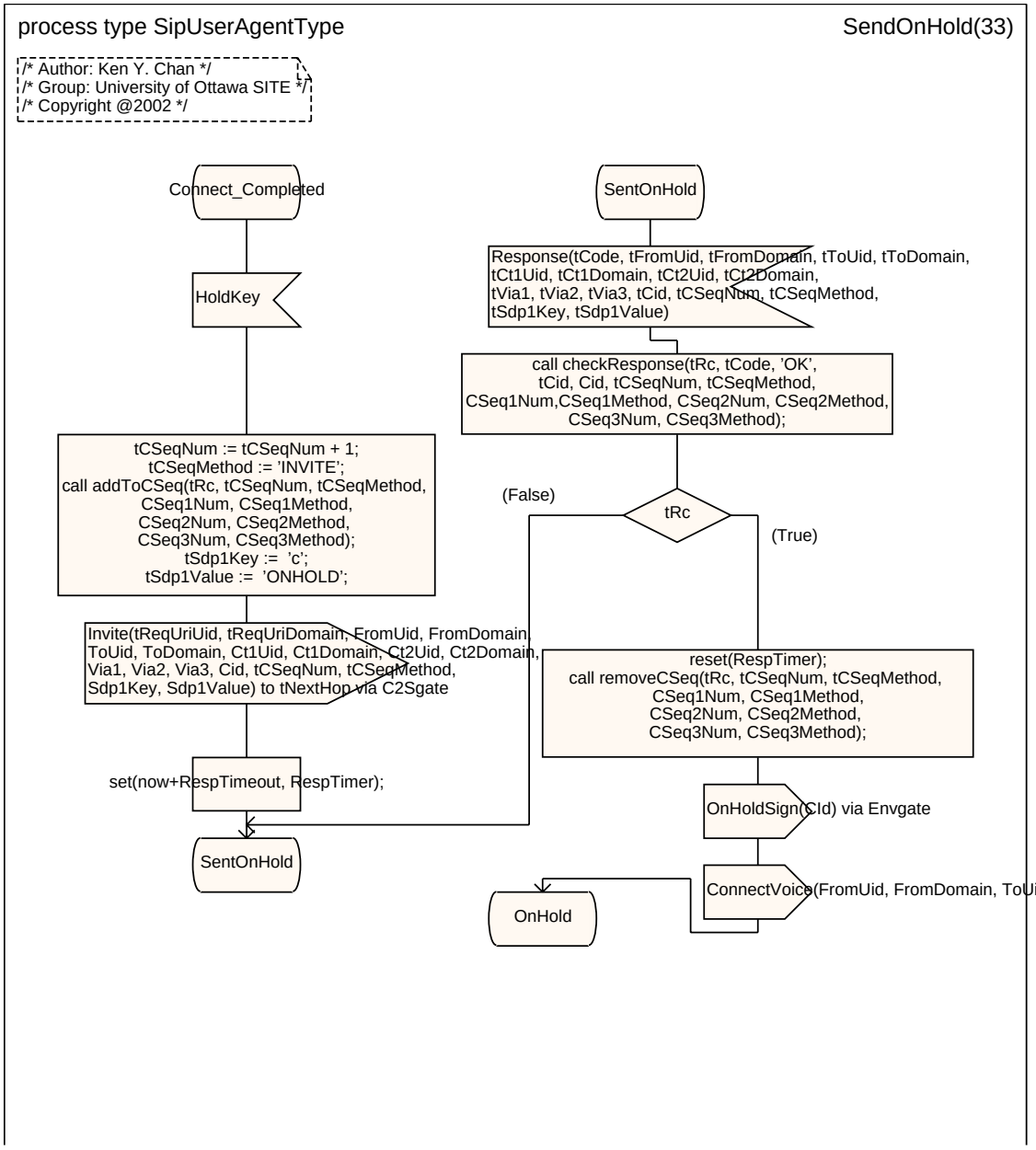

Figure 18: SipUserAgent Sending OnHold

In general, trigger events are expressed as incoming signals; pre-conditions, post-conditions, and other constraints are expressed as enabling conditions or decisions. Actions are tasks, procedure calls, or output signals. As the user agent process consumes a trigger event, the parameters of the event may be examined along with the pre-conditions of the feature. Then, actions such as sending out a message and modifying the internal variables may be

executed. Post-conditions and constraints on the action may also be checked. Finally, the process progresses to the next state.

Although we have described the skeleton of the model (systems, blocks, data variables in processes), we have not defined the essential behavior of the model, which are the state transitions of the processes. The question is how to come up with the state transitions for the "initial" models. Since we have a message sequence chart describing the success scenario of a two party call, we assign a unique state to each input and output of the instance type (user agent or proxy type). For example, if 'Connect_Request' state is the current state and we have received a 'HoldKey' signal, we would send an 'Invite' request to the other user agent. We do not put any timers in the 'initial' model because we would like to produce a simple rapid prototype model that we can experiment with. However, we add timers (e.g. response timer) to the 'standard' and 'refined' models.

A state transition occurs when 1) an "Abstract User" signal is received from the environment, 2) a request or response message is received, or 3) a continuous signal is enabled. The so-called Continuous Signal in SDL is used to model the situation in which a transition is initiated when a certain condition is fulfilled. For example if the UAS is busy, the boolean 'isBusy' would be true in the 'Server_Ring' state. The UAS would immediately send the BUSY response to the caller. This way, we would not have to worry about the timer expiration because we do not need to send a busy toggling signal to simulate busy during a simulation. An asterisk '*' can be used in a state and signal symbol to denote any state or any signal; its semantics is equivalent to a wildcard. A state symbol with a dash '-' means the current state. Error handling, such as response timer expiration, can easily be modeled with SDL timers and a combination of '*' and '-' state symbols. For example, when the response timer expires, a timeout message would be automatically sent to the input queue of the user agent process. The expiration of the response message is generalized as the situation in which the receiving end does not answer the request in time. Thus, a 'NoAnswer' signal is sent to the environment. Finally, the process can either return to the previous state or go directly to the idle state through the 'Jump_Idle' connector in this case.

Moreover, we can add additional features or services such as CFB, OCS, and other services, to the system. To add behaviors of additional features to a process type, we can subtype a "basic" process type such as UserAgentType. The derived type has the same interfaces and additional state transitions corresponding to the added features [16]. We do not

want to add new interfaces (signal routes) to the process types because we do not want to change the interfaces of the block types. If a feature requires new SIP methods and response codes, we would not need to change the interfaces because method names and response codes are simply values of signals parameters in our model. Thus, we avoid the need to add new interfaces whenever a new feature is defined.

4.6.2 Proxy Process Specification

A proxy is different from a user agent because a proxy processes messages exchanged between user agents or proxies. We model the proxy also as an entity to which telephony features may be deployed. The proxy performs a predefined set of actions based on the header information (e.g. originator and destination addresses) of the incoming message(s). In Figure 19, a proxy listens for response messages from the terminating user agent and forwards the response message to the originating agent based on the routing information that was set during the initiation phase of the simulation.

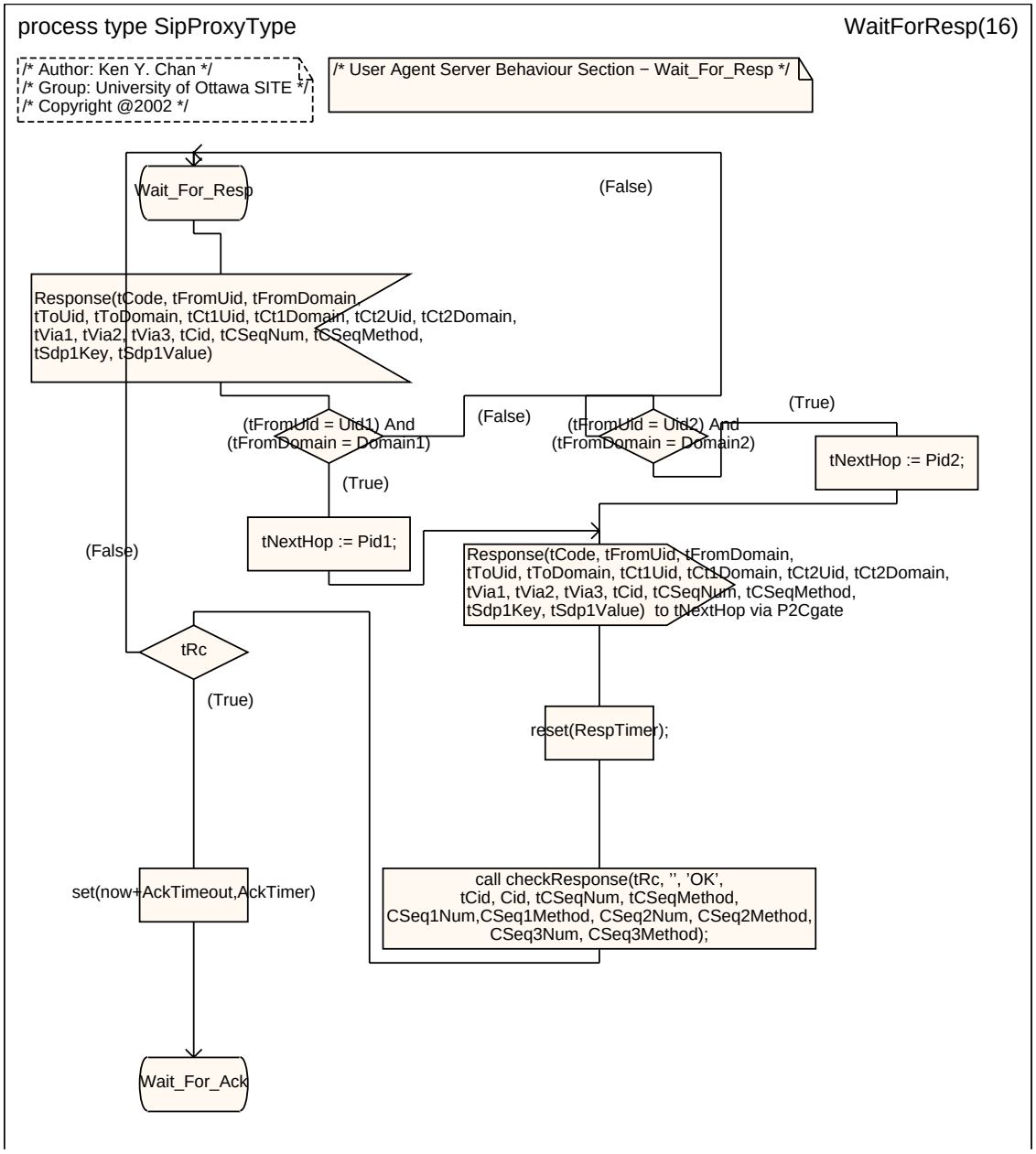

Figure 19: Proxy Waiting for Response

In summary, a state model of SIP and its sample services can be derived from message sequence charts. Many protocol features such as timer expiration, parameter checking, sending and receiving messages can be mapped to appropriate structures in SDL. The object-oriented extension to SDL allows specialization of telephony services. Thus, SDL is suitable in modeling telephony services. In the next section, we will examine the validation and verification process.

4.7 Verification & Validation

The SDL specification that has been discussed in the previous subsection was constructed using the Telelogic Tau tool version 4.3 and 4.4 [12]. Tau offers many verification or reachability analysis features: bit-state, exhaustive-state, random walk bit state space exploration and MSC verification (see Section 2.3). Bit-state space exploration is particularly useful and efficient [14] in checking for deadlocking interactions because it checks for various reactive system properties without constructing the whole state space like the exhaustive approach does. We have tried exhaustive-state space exploration but its performance is generally poor (required a lot more memory and time for verification). Furthermore, we have tried random walk bit state space exploration in our verification but no noticeable performance speed-up was observed.

Since Tau uses a predetermined set of test values for each data type (e.g. test values for integer type are: 0, 55, -55) and does not always perform 100% state space coverage, we cannot assume the above state space exploration techniques guarantee the model to be free of faults. We used these techniques to find potential faults in the model. We are more confident in our model being correct by validating the model against the test scenarios. This is achieved by using the "Verify MSC" feature of Tau.

We verified our SDL model of SIP mainly by checking whether the model would be able to realize specific interaction scenarios which were described in the form of Message Sequence Charts (MSCs) [11,12] (see Section 5.1). In fact, we used the scenarios described informally in [6,7], and rewrote them in the form of MSCs. Then we used the Tau tool to check that our SDL model was able to generate the given MSC. An MSC is verified if there exists an execution path in the SDL model such that the scenario described by the MSC can be realized. Thus, an MSC in Tau is considered as an existential quantification of the scenario. When "Verify MSC" is selected, Tau may report three types of results: verifications of MSC, violations of MSC, and deadlocks. After a few iterations of test and refinement, we were able to come up with a final model that realized all test scenarios (no MSC violations).

Although we could use MSCs as the test cases of our SDL model, MSCs have limitations in terms of expressing quantification of instances and their behaviors. For example, we could not find a way to write an MSC to verify the following condition: For all user agents that

receive an "OK" response, they must reply the sender of the response with an "ACK" message. We could use Observer Process assertions to verify this condition (see Section 2.3). For example, we could create one observer process for each user agent in our model. We would then create one assertion for each observer process; each assertion would contain a Boolean expression that check the reception of an "OK" response must be followed by the sending of an "ACK" message to the response sender. Clearly, writing test scenarios in this manner is a tedious task but it is the only viable option, particularly for detecting feature interactions which would be discussed in the next chapter.

In general, we derive the test scenarios for our features indirectly from the call flow diagrams described in [6,7]. Since the researchers have generally specified one scenario for each feature that they described, we have written only one scenario (success path) per feature in our model, except for the two-party call for which we have tested the busy tone, ring tone, "No Answer" time out, and invalid SIP address scenarios. We have also written one test scenario for each known traditional feature interaction to confirm whether certain feature interactions exist in SIP. We will discuss detecting feature interactions in Section 5.

Another alternative would be to use Live Sequence Charts (LSC) [30], which has not been discussed much in this thesis. They appear to be a promising extension to MSCs in this context. However, Tau does not support LSC at the moment. Further research on the application of LSC would be an interesting future work item.

5 Methods for Detecting Feature Interactions

One of the objectives of our research was to explore the feasibility of using Tau to detect known and new FI's. Although SIP is fundamentally different from traditional PSTN signaling protocols, most of the traditional POTS FI's still exist in SIP, if the SIP services are designed and implemented with the mindset of POTS. (Note: Alternate design approaches to SIP services are discussed in the next section). We believe that it is neither feasible nor practical to come up with an automated feature interaction detection scheme within the Tau environment because we could not find a way to express the existing and new feature interactions (see Section 3) as general test cases (e.g. temporal logic formula or sequence charts that can be quantified over instances as explained in Section 4.7). Furthermore, Tau has limited validation features and Tau's Validator frequently crashes. Instead, we write specialized test cases for each SDL process instance in the form of either MSCs or Tau's Observer Process Assertions to verify whether traditional POTS FI's that were discovered by other feature interaction's researchers still exist in SIP. Obviously, it is a tedious task.

In the previous sections, many FI's were identified as violations of distributed system properties: deadlock (a form of violation of safety) and livelock (a form of violation of liveness). The Tau tool offers various automated reachability analysis of SDL models and reports any deadlocks found during the analysis. Let us revisit the previous deadlock example on CFB and CW (see Section 1.3). If we have an SDL system with the corresponding SDL blocks or SIP entities which has a user agent process running with CFB and CW behaviors, we can select the "bit state exploration" option to ensure no deadlock would be found. Tau's Validator does not seem to offer reports on the liveness of the analyzed system in any of its reachability analysis functions. Thus, we use Tau's Observer Process features (see Section 2.3) to detect live-locking FI's, as explained in Section 5.2.

Furthermore, incoherent interactions are not easy to check with the Tau tool because certain interactions that involve contradictory properties cannot be expressed in a straightforward manner. There are two ways to approach this problem: (1) Use an MSC to specify incoherent properties, or (2) use an observer process offered by Tau to specify assertions. By examining the validation report (which shows the simulation trace in the form of MSC), we could trace back to the locations where the problem occurs; thus we could

identify the interacting features. We have developed the following test scenarios to confirm if certain feature interactions exist in SIP. The following table (Figure 20) describes the results of our experiment and the features we have selected for detecting their feature interactions:

	CW	**OCS**	**TCS**	**CFB**	**ACB**	**AR**
CW	-	No	No	No	No	No
OCS	No	-	No	ICH	No	No
TCS	No	No	-	No	No	No
CFB	No	ICH	No	-	No	No
ACB	No	No	No	No	-	LCK
AR	No	No	No	No	LCK	-

Figure 20: Results of Feature interaction detection

In Figure 20, the symbol "-" means we did not conduct any test for that feature pair. The cell value of "No" indicates that we had conducted one of the feature interaction tests (livelocking, deadlocking, or incoherent) but did not discover any feature interactions for that feature pair. We did not conduct all possible feature interaction tests for all feature pairs because we could deduce intuitively that most of these feature pairs would not have any interactions. The abbreviation "ICH' denotes the presence of incoherent interaction and "LCK" denotes the presence of livelocking interactions. We wrote test scenarios as MSCs for detecting known incoherent interactions such as CFB and OCS. We wrote Observer Process Assertions to detect incoherent interactions caused by OCS and TCS, and livelocking interactions caused by AR and ACB. We did not catch any livelocking interaction between CW and CFB, which occurs in POTS. We believe we have eliminated this livelocking interaction because we have implemented CW such that it does not intercept the busy signal, which is a trigger event on which CFB depends.

In general, both approaches (MSCs and Observer Process Assertions) have their advantages and disadvantages. However, the Observer Process approach (see Section 2.3) is the only practical approach in the current Tau release and we will discuss the reasons in the following subsections.

5.1 Specifying incoherences as MSC

Many service designers like to use MSCs to capture important scenarios of a feature. If the sets of message sequence charts describing two features can be compared/verified against each other, then certain obvious incoherent FI's may be detected in this informal requirement specification stage. However, capturing key behaviors of a feature in MSC is not always possible. For example, the success scenario of OCS cannot be expressed directly as an MSC in the case of OCS and CFB interaction. How can we express in an MSC that user A cannot call user C? The concept of something that can "never happen" is usually expressed using universal quantification. Since verification of MSCs in Tau is based on an existential quantification of the scenario, a property that something should "never" happen can be expressed by negation. If ***m*** is a scenario that should never happen, then we could use the Tau tool to check whether the SDL model can satisfy this MSC ***m***. If the result is that ***m*** cannot be satisfied by the model, this verifies the property.

If this concept is applied to OCS, "verifying user A can call user C" being false is equivalent to the truth of "user A can never call user C". Thus, the successful verification of the success call scenario from user A to user C concludes the violation of the OCS specification (see OCS MSC diagram). Since features like OCS apply to all calls including mid-call, the pre-enable condition of the MSC must be specified. The preamble condition must include all possible situations in which a call can be made to user C. There are very many possibilities; clearly, detecting incoherent interactions using MSC with the current version of Tau is almost impossible.

However, an extension to MSCs, called Live Sequence Chart (LSC) [30], seems to address the above concerns. First of all, an LSC allows the designer to specify messages that cannot be sent in a scenario. This is useful for specifying OCS type of services. Secondly, an LSC can be specified with either the universal or existential quantification of a scenario, which offers great flexibility in verifying scenarios that are contradictory to each other. Last but not least, a universal LSC allows an associated pre-enable condition which defines the scope of the scenario. This is essential to modeling services because such pre-enable conditions can be used as the triggering condition of a feature. Unfortunately, Tau does not support LSC but we believe LSC is a promising tool in this context. Details on LSC are beyond the scope of this thesis and will be considered in future work.

5.2 Specifying incoherences as Assertions in an Observer Process

Tau provides Observer Processes as a tool for detecting feature interactions (see Section 2.3). Since an observer process can observe the internal state of other processes during validation [12], we can express various types of feature interactions as assertions in these observer processes. The syntax of an assertion comprises the SDL conditions (e.g. the continuous signals, decisions and enabling condition) of the internal states of the observed processes. Although these SDL conditional constructs have different semantics, they are evaluated much like boolean expressions (e.g. true or false). Typically, if an assertion is evaluated to false, the assertion would be followed by a procedure call to the violation report. Thus, the validation process would be stopped immediately and the report would be presented to the user.

We have experimented with assertions that verify certain liveness properties of the system. For example, if we have an integer counter for each user agent to keep track of the number of times each user agent has been in 'Ringing' state, we can monitor whether a user agent has been looping at 'Ringing' state or not. More specifically, an assertion, which verifies user agent UA1 and UA2 are not simultaneously in 'Ringing State' for more than three times ensures a certain level of liveness in the system. In this case, we were able to detect livelocking interaction between Auto-Recall (AR) and Auto-Callback (AC), and incoherent interaction between Call Forward Busy (CFB) and Originator Call Screening (OCS). The following diagram (Figure 21) illustrates an "Observer Process Assertion" for OCS.

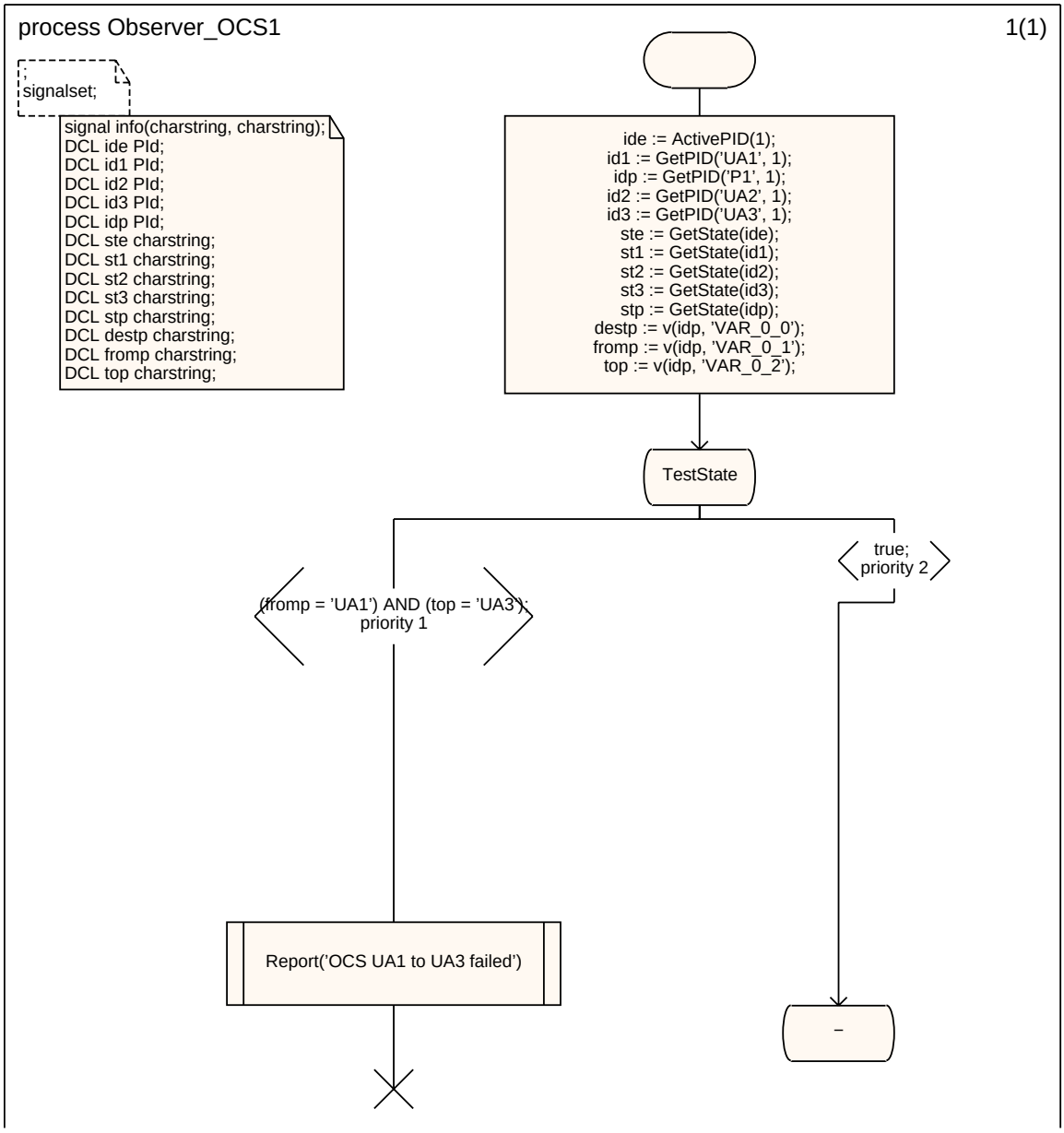

Figure 21: Sample Observer Process Assertion for OCS

Figure 21 shows the definition of a sample observer process for the OCS feature. The first task in an observer process is to read in all the relevant internal variables and state information of the process that we like to observe. Then, we can express our assertion using those data. To ensure that the intent of our OCS feature is not violated, we wrote the assertion as an enabling condition which checks that the internal variables that keep track of the "From" and "To" addresses of the session in the proxy are assigned with "UA1" and "UA3", respectively. If the answer is yes, we presume the originator has attempted to establish a

connection with a screened destination (e.g. UA3) and a violation report (e.g. "OCS UA1 to UA3 failed") would be generated. Otherwise, the observer process would perform no action and make a state transition; the simulation would continue. Since one may access many types of internal data and state information using the operators supported by Tau's Observer Process facility, one could construct complex assertions using this facility. We have yet to experiment with converting well-known liveness detection algorithms into Observer Processes Assertions. However, since we are not aware that Tau supports temporal logic, we believe specifying liveness properties without temporal logic would be non-trivial.

In general, we use our intuition to come up with useful assertions for each feature interaction category. Although this is not an automated process, we believe it is a practical approach to a subjective problem such as detecting "undesirable side-effects" between features (FI's) in the Tau environment. We believe the current features offered in Tau are very limited for detecting new FI's, but Tau meets our key objective; we could verify whether the well known FI's still exist in SIP or not. Thus, we could apply preventive measures (resolution schemes) to make SIP services more robust, as explained in the next section.

6 Methods for Preventing Feature Interactions

After a feature interaction is detected, the next natural step is to change the design to prevent it. In this section, we discuss corrective measures that can be incorporated in the design and implementation of a system to prevent FI's. These corrective measures map directly to the causes and effects of FI's. Preventing FI's may be done at design-time or at run-time. Run-time prevention, which is known as *resolution* in the feature interaction community, is more difficult to exercise because the features may be running on different nodes and it is not easy to coordinate the actions that should be performed when a feature interaction is detected. Design-time prevention and run-time resolution strategies for different feature interaction categories are presented in the following subsections. First, we will give an overview of caller preferences [36], and feature set [47] proposed by the IETF SIP community. Then, we will present a feature negotiation framework based on the negotiating agent approach [20], and SIP caller preference [36] for resolving feature interactions at run-time.

In addition, the feature interaction examples described in Section 3 will be revisited again. The goal of this discussion is to provide a catalog of feature interactions to service designers, such that a designer can associate a feature with a set of prevention schemes that would reduce potential interactions with any unknown features. We will also see that it is easier to prevent feature interactions in SIP than in POTS, because SIP has extra call information (e.g. 'Via', 'Record-Route', 'Contact') in the message headers.

6.1 Overview of IETF Caller Preferences & Feature Set

J. Rosenberg has proposed an IETF draft which describes a caller preference extension to SIP [36]. It describes a set of extensions to the Session Initiation Protocol (SIP) [4] which allow a caller to express preferences about request handling in servers. These preferences include the ability to select which Uniform Resource Identifiers (URI) a request gets routed to, and to specify certain request handling directives in proxies and redirect servers. It does so by defining three new request header fields, Accept-Contact, Reject-Contact, and Request-Disposition, which specify the caller's preferences. The extension also defines new parameters

for the Contact header field that describe the capabilities and characteristics of a User Agent (UA) [36].

In this proposal, when a Session Initiation Protocol (SIP) [4] server receives a request, there are a number of decisions it can make regarding processing of the request. These include:

- whether to proxy or redirect the request,
- which URIs to proxy or redirect to,
- whether to fork or not,
- whether to search recursively or not,
- whether to search in parallel or sequentially.

The server can base these decisions on any local policy. This policy can be statically configured, or can be based on programmatic execution or database access.

However, the administrator of the server is not the only entity with an interest in request processing. There are at least three parties which have an interest: (1) the administrator of the server, (2) the user that sent the request, and (3) the user to whom the request is directed. The directives of the administrator are embedded in the policy of the server. The preferences of the user to whom the request is directed (referred to as the callee, even though the request may not be INVITE) can be expressed most easily through a script written in some type of scripting language, such as the Call Processing Language (CPL) [46]. However, no mechanism exists to incorporate the preferences of the user that sent the request (also referred to as the caller, even though the request may not be INVITE). For example, the caller might want to speak to a specific user, but want to reach them only at work, because the call is a business call. As another example, the caller might want to reach a user, but not their voicemail, since it is important that the caller talk to the called party. In both of these examples, the caller's preference amounts to having a proxy make a particular routing choice based on the preferences of the caller [36].

This extension allows the caller to have these preferences met. It does so by specifying mechanisms by which a caller can provide preferences on processing of a request. There are two types of references. One of them, called request handling preferences, are encapsulated in the Request-Disposition header field. They provide specific request handling directives for a server. The other, called feature preferences are present in the Accept-Contact and Reject-Contact header fields. They allow the caller to provide a feature set [47] that expresses its

preferences on the characteristics of the UA that is to be reached. These are matched with a feature set carried in the Contact header field of a REGISTER request, which describes the capabilities of the UA represented by the Contact URI. The extension is very general purpose, and not tied to a particular service. Rather, it is a tool that can be used in the development of many services.

Indeed, the feature sets uploaded to the server in REGISTER requests can be used for a variety of purposes, not just meeting caller preferences. We will show how we could encode feature information in the feature set of SIP messages so that we could resolve feature interactions in the later subsections.

There are a few important definitions within the context of the caller preferences draft [36] that are relevant to our proposed negotiation extension; they are:

- Caller: It refers to the user on whose behalf a UAC is operating. It is not limited to a user whose UAC sends the INVITE method.
- Feature: As defined in RFC 2703[48], a piece of information about the media handling properties of a message passing system component or of a data resource. For example, the SIP methods supported by a UA represent a feature.
- Feature Tag: As defined in RFC 2703[48], a feature tag is a name that identifies a feature. An example is "methods".
- Feature Set: As defined in RFC 2703 [48], a feature set contains information about a sender, recipient or other participant in a message transfer which describes the set of features that it can handle. While a 'feature' describes a single identified attribute of a resource, a 'feature set' describes a full set of possible attributes.
- Feature Preferences: Caller preferences that described desired properties of a UA that the request is to be routed to. Feature preferences can be made explicitly with the Accept-Contact and Reject-Contact header fields.
- Request Handling Preferences: Caller preferences that describe desired request treatment at a server. These preferences are carried in the Request-Disposition header field.
- Feature Parameters: A set of SIP header field parameters that can appear in the Contact, Accept-Contact and Reject-Contact header fields. The feature parameters

represent an encoding of a feature set. Each set of feature parameters maps to a feature set predicate.

- Capability: As defined in RFC 2703 [48], a capability is an attribute of a sender or receiver (often the receiver) which indicates an ability to generate or process a particular type of message content.
- Target Set: A target set is a set of candidate URI that a proxy or redirect server can send or redirect a request to. Frequently, target sets are obtained from a registration, but they need not be.
- Explicit Preference: A caller preference indicated explicitly in the Accept-Contact or Reject-Contact header fields.
- Implicit Preference: A caller preference that is implied through the presence of other aspects of a request. For example, if the request method is INVITE, it represents an implicit caller preference to route the request to a UA that supports the INVITE method.
- Predicate: A boolean expression.
- Feature Set Predicate: From RFC 2533 [47], a feature set predicate is a function of an arbitrary feature collection [36] value which returns a Boolean result. A TRUE result is taken to mean that the corresponding feature collection belongs to some set of media feature [36] handling capabilities defined by this predicate.
- Contact Predicate: The feature set predicate associated with a URI registered in the Contact header field of a REGISTER request. The contact predicate is derived from the feature parameters in the Contact header field.

This extension defines a set of additional parameters to the Contact header field, called feature parameters. Each parameter name is an encoded feature tag, as defined in RFC 2703 [48], that defines a capability for the UA associated with the Contact header field value. For example, there is a parameter for the SIP methods supported by the UA. Each feature parameter has a value; that value is the set of feature values for that feature tag. Put together, the entire feature parameters specify a feature set that is supported by the UA associated with that Contact header field value [36]. The following is a sample caller preference's feature predicate:

```
(& (audio=TRUE)
    (video=TRUE)
    (msgserver=TRUE)
    (automata=TRUE)
    (attendant=TRUE)
    (mobility=fixed)
    (| (methods=INVITE) (methods=BYE) (methods=OPTIONS) (methods=ACK)
      (methods=CANCEL))
    (uri-user="user")
    (uri-domain=host.example.com) )
```

These would be converted into feature parameters and included in the REGISTER request:

```
REGISTER sip:example.com SIP/2.0
From: sip:user@example.com;tag=asd98
To: sip:user@example.com
Call-ID: hh89as0d-asd88jkk@host.example.com
CSeq: 9987 REGISTER
Max-Forwards: 70
Via: SIP/2.0/UDP host.example.com;branch=z9hG4bKnashds8
Contact: <sip:user@host.example.com>;audio="TRUE";video="TRUE"
 ;msgserver="TRUE";automata;attendant;mobility="fixed"
 ;methods="INVITE,BYE,OPTIONS,ACK,CANCEL"
 ;uri-user="<user>"
 ;uri-domain="host.example.com"
Content-Length: 0
```

When a caller sends a request, it can optionally include new header fields which request certain handling at a server. These preferences fall into two categories. The first category, called request handling preferences, are carried in the Request-Disposition header field. They describe specific behavior that is desired at a server. Request handling preferences include whether the caller wishes the server to proxy or redirect, and whether sequential or parallel search is desired. These preferences can be applied at every proxy or redirect server on the call signaling path [36].

The second category of preference is called feature preferences. These preferences are carried in the Accept-Contact and Reject-Contact header fields. These header fields also contain feature sets, represented by the same feature parameters that are used in the Contact header field. Here, the feature parameters represent the caller's preferences. The Accept-Contact header field contains feature sets that describe UAs that the caller would like to reach. The Reject-Contact header field contains feature sets which, if matched by a UA, imply that the request should not be routed to that UA [36].

Proxies use the information in the Accept-Contact and Reject-Contact header fields to select amongst contacts in their target set. When neither of those header fields is present, the proxy computes implicit preferences from the request. These are caller preferences that are not explicitly placed into the request, but can be inferred from the presence of other message components. As an example, if the request method is INVITE, this is an implicit preference to route the call to a UA that supports the INVITE method [36].

Both request handling and feature preferences can appear in any request, not just INVITE. However, they are only useful in requests where proxies need to determine a request target. If the domain in the request URI is not owned by any proxies along the request path, those proxies will never access a location service, and therefore, never have the opportunity to apply the caller preferences. This makes sense; typically, the request URI will identify a UAS for mid-dialog requests. In those cases, the routing decisions were already made on the initial request, and it makes no sense to redo them for subsequent requests in the dialog [36].

In subsequent subsections, we will discuss how we would incorporate functions offered by the negotiating agent approach [20], caller preferences [36], and feature set [47] to resolve feature interactions.

6.2 Feature Negotiating Extension to Caller Preferences

We propose an extension to the caller preferences draft [36] for resolving feature interactions. The current caller preferences draft [36] allows a caller to express preferences about request handling in servers. We believe we could append feature parameters to the header fields used by the caller preferences extension in the caller request message. We understand that not all calling parties are interested in allocating computational resources to resolve feature interactions. Therefore, we propose these parameters to be made optional. The downstream (the direction in which the request messages travel, away from the caller) proxies or user agents have the option to process these parameters for resolving feature interactions. The caller preferences draft leaves much room for the types of values that can be put in the header fields; for example, one can place quality of service (QoS) parameters like delay, bandwidth, media type capabilities, or request methods in the header. The caller preferences draft also took the SDP Offer/Answer approach [37] into account for the negotiation of the multimedia session.

In [37], one participant offers the other a description of the desired session from their perspective, and the other participant answers with the desired session from their perspective. However, for the purpose of resolving feature interactions, we should be concerned with the header fields; we believe we have to be more specific on the types of feature parameters that are relevant to the resolution algorithm. Thus, we shall formalize the types of feature parameters that would be required for resolving most feature interactions, at least for those feature interactions that we have proposed in Section 3.

In Section 3, we have shown that most feature interactions are related to reactive system properties (e.g. livelock, deadlock, fairness, non-determinism), or incoherence (or inconsistency) between conflicting user/feature goals. Our feature interaction tree (FIT) shows that various reactive-system types of feature interactions are usually caused by resource contention and timing problems. In addition, N. Gorse also shows that feature goals can be expressed as first order predicates [13] in the context of automatically filtering feature incoherences. We decided to apply his concept of predicates and feature incoherences in the context of feature interaction resolution. We believe if the session participants (e.g. user agents and proxies bound to the session) place the name of the features which they manage, along with the resources, conditions (expressed as predicates), and the participants that these features depend on, in the request messages, the messages would have the basic information needed by any feature interaction resolution algorithms.

However, the feature set predicate defined in [47] currently expresses the relationship between a session participant and its desired set of features in the form of a tuple (e.g. <user, feature set>). Capabilities of a feature are currently assigned as boolean values (e.g. TRUE or FALSE) indicating whether the capability should be present or not. This is insufficient for describing a feature for the purpose of resolving feature interactions. In fact, many researchers such as Gorse [13] and Kamoun [20] have described a feature as a predicate comprising pre-conditions, triggering events, post-conditions (results) (see Section 1.2.4). An example would be the following:

- feature-name ([Preconditions], [TriggerEvents], [Results]).

Gorse used his formalism for detecting feature interactions. However, we could apply his definition of a feature for our feature negotiation extension. The difference is that we do not use his predicate form as the complete description of a feature. Instead, we add the feature

participants (user agents and proxies bound to the feature) to this predicate form, which is then used as the signature of a feature.

- feature-name ([Participants], [Preconditions], [TriggerEvents], [Results]).

In our work, the complete specification of a feature is described in our SDL model (see Section 4). We may use synthesis tools to generate target executables or scripts (e.g. CPL [46]) from our SDL model, but this is beyond the scope of this thesis. With participants, pre-conditions, triggering events, and post-conditions as part of a feature signature, the session participants would have sufficient information for deducing potential feature interactions. If a server needs more information about the features, the server could always consult the registry for the complete feature description (e.g. CPL scripts [46]) if desired. However, comparing scripts during a session invitation is generally too computationally intensive, thus impractical for our consideration.

Therefore, we propose to apply our "new" feature signature as the extension to the schema of the feature set predicate, as described in the form of BNF grammar in Figure 22:

```
<feature-tag> := any string

<parameter-type> := "node" | "precondition" | "postcondition" | "trigger" | "priority"

<parameter-tag> := "in" | "out" | "caller" | "callee" | "forwarder" | "destination"| "screen"

<list-of-variables> := <parameter-tag> | any string  [ "," < list-of-variables>]

<parameter-value> := any string | True | False

<parameter-clause> :=  ["!"] <clause-name> "(" <list-of-variables> ")"

<parameter-expression> := <parameter-clause> |
                          { <parameter-tag> "#"<parameter-value> } |
                          <parameter-value>

<parameter-description> := <parameter-type> ":" <parameter-expression>

<feature-predicate> :=   "(" <feature-tag> "=" "\""
                                              <parameter-value> |
                                              <parameter-description>
                                              "\"" ")"
```

Figure 22: Our extension to the schema of the feature predicate

In Figure 22, which shows the extended schema of the feature predicate, a feature-predicate typically comprises a feature tag (the name of the feature), and either a parameter

value or a parameter description. A parameter description is defined by the type of the parameter and the expression that describes the parameter. There are four parameter types: the node (address of a session participant), precondition, post-condition, trigger events, and priority level of the feature. The parameter expression contains the context sensitive information of the parameter of the feature; it may contain a clause that describes the condition for the feature, or a parameter tag and a string that indicates the value of the parameter. The parameter tag identifies a keyword that has a special meaning depending on the type of the parameter, for example, the "caller" keyword represents the originator of the call and the "in" keyword indicates the event is a received event. Since it is impossible to determine the parameter tags needed by all possible features, additional parameter tags may be defined in the future when they are needed. The symbol "!" is the negation operator for a parameter clause which has a special meaning to the feature (e.g. !Connect(caller, callee) means the voice path between caller and callee is not connected). Furthermore, a parameter value can be a boolean value (e.g. true of false), or an ASCII string. If a feature predicate comprises nothing more than a feature tag and a boolean value, then the boolean value would indicate whether the feature is available or not. Let us take the Auto-Recall (AR) and Originator Call Screening (OCS) features which are collocated in the same server as our example. We would have the following sample feature set predicate:

```
(& (audio=TRUE)
    (video=TRUE)
    (msgserver=TRUE)
    (automata=TRUE)
    (attendant=TRUE)
    (mobility=fixed)
    (| (methods=INVITE) (methods=BYE) (methods=OPTIONS) (methods=ACK)
      (methods=CANCEL))
    (uri-user="user")
    (uri-domain=host.example.com)
   (orig-call-screen="node:caller#user1@example.com")
   (orig-call-screen="node:screen#user3@example.com")
   (orig-call-screen="precondition:!CONNECT(caller,screen)"
   (orig-call-screen="postcondition:!CONNECT(caller,screen)")
   (auto-recall="node:caller#user1@example.com")
   (| (auto-recall="node:callee#user2@example.com")
     (auto-recall="node:callee#user3@example.com"))
   (auto-recall="trigger:in#BUSY(callee)")
   (auto-recall="precondition:IDLE(caller)")
   (auto-recall="postcondition:CONNECT(caller,callee)"))
```

So, how do the caller preferences [36] related to the negotiating agent approach [20] at this point? The answer to this question is simple; a proposal generated by the caller, as explained in

[20], is analogous to the query for capabilities generated by the caller, as described in [36]. Thus, we can embed a proposal in the corresponding INVITE request message, as described below:

```
INVITE sip:example.com SIP/2.0
From: sip:user1@example.com;tag=asd98
To: sip:user2@example.com
Call-ID: hh89as0d-asd88jkk@host.example.com
CSeq: 1 INVITE
Max-Forwards: 70
Via: SIP/2.0/UDP host.example.com;branch=z9hG4bKnashds8
Contact: <sip:user1@host.example.com>;audio="TRUE";video="TRUE"
 ;msgserver="TRUE";automata;attendant;mobility="fixed"
 ;methods="INVITE,BYE,OPTIONS,ACK,CANCEL"
 ;uri-user="<user>"
 ;uri-domain="host.example.com"
 ;orig-call-screen=”node:caller#user1@example.com”
 ;orig-call-screen=”node:screen#user3@example.com”
 ;orig-call-screen=”precondition:!CONNECT(caller,screen)”
 ;orig-call-screen=”postcondition:!CONNECT(caller,screen)”
 ;auto-recall=”node;caller#user1@example.com”
 ;auto-recall=”node:callee#user2@example.com,node:callee#user3@example.com”
 ;auto-recall=”trigger:in#BUSY(callee)”
 ;auto-recall=”postcondition:!CONNECT(caller,callee)”
 ;auto-recall=”postcondition:CONNECT(caller,callee)”
Content-Length: 0
```

When a downstream entity (proxy or user agent) in a session receives the proposal, it would determine the acceptability of the proposal [20]. If the downstream entity do not detect any feature interactions that could be caused by the proposal, the entity could return a response indicating the caller preferences are acceptable, as described in [36]. Figure 23 illustrates this negotiation scenario:

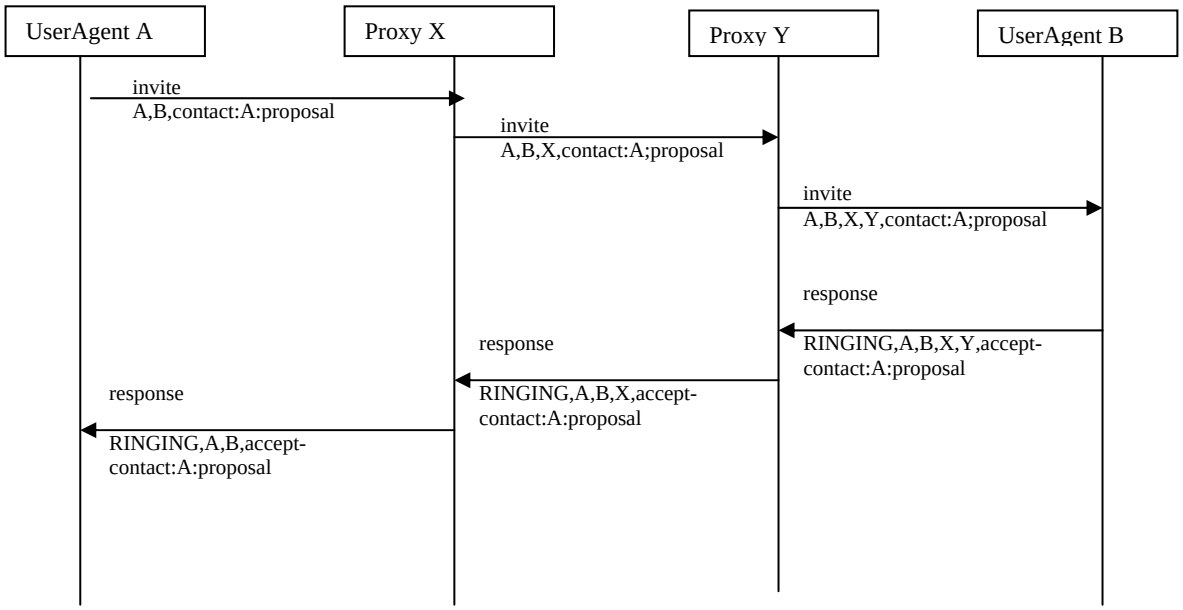

Figure 23: The callee accepted the caller's proposal with no feature interactions detected

In addition, the signatures of the features that are managed by the downstream entity may optionally be appended to the response header. This way, the caller has the option to determine any potential feature interactions which may be caused by the downstream entity. If the caller detects a feature interaction, it may initiate a new proposal in a re-invite request message to the callee(s). Figure 24 illustrates this re-proposal negotiation scenario.

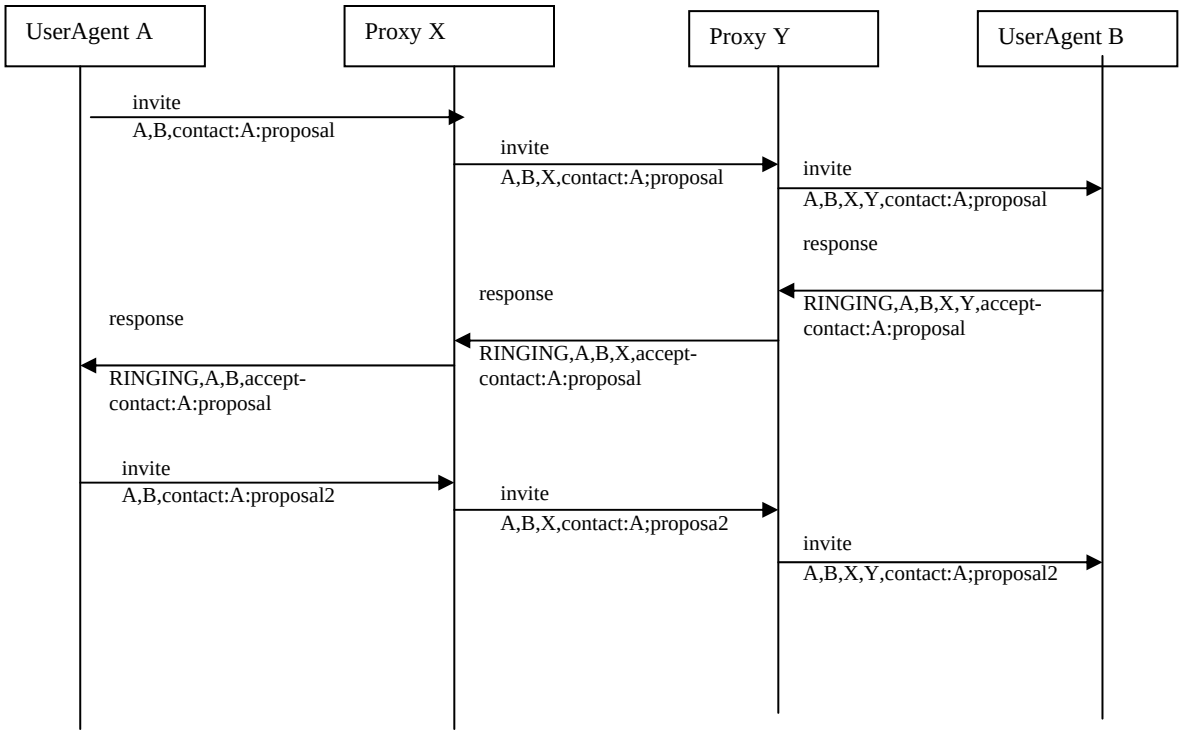

Figure 24: The caller detects feature interaction(s) and re-proposes

However, if the downstream entity were to detect any feature interaction(s) and decide to reject the proposal, it may reply with a response code and Reject-Contact header(s) as described in [36], along with the signature(s) of the interacting feature(s), indicating the reason for the rejection. The downstream entity has the option not to generate a counter-proposal. However, if it chooses to do so, we propose the downstream entity would initiate an "INFO" request to the caller with the same call-id [4] after it has replied the caller with a reject response. An "INFO" request is used instead of a response for the counter proposal because the counter proposal is considered as a different request and requires more time to be generated; thus the downstream entity should reply to the originator with a reject response which is eventually followed by a counter-proposal request to the originator. An "INFO" request needs not be closed by a response message but it must have the same call-id as the original request did. Figure 25 illustrates this counter-proposal negotiation scenario.

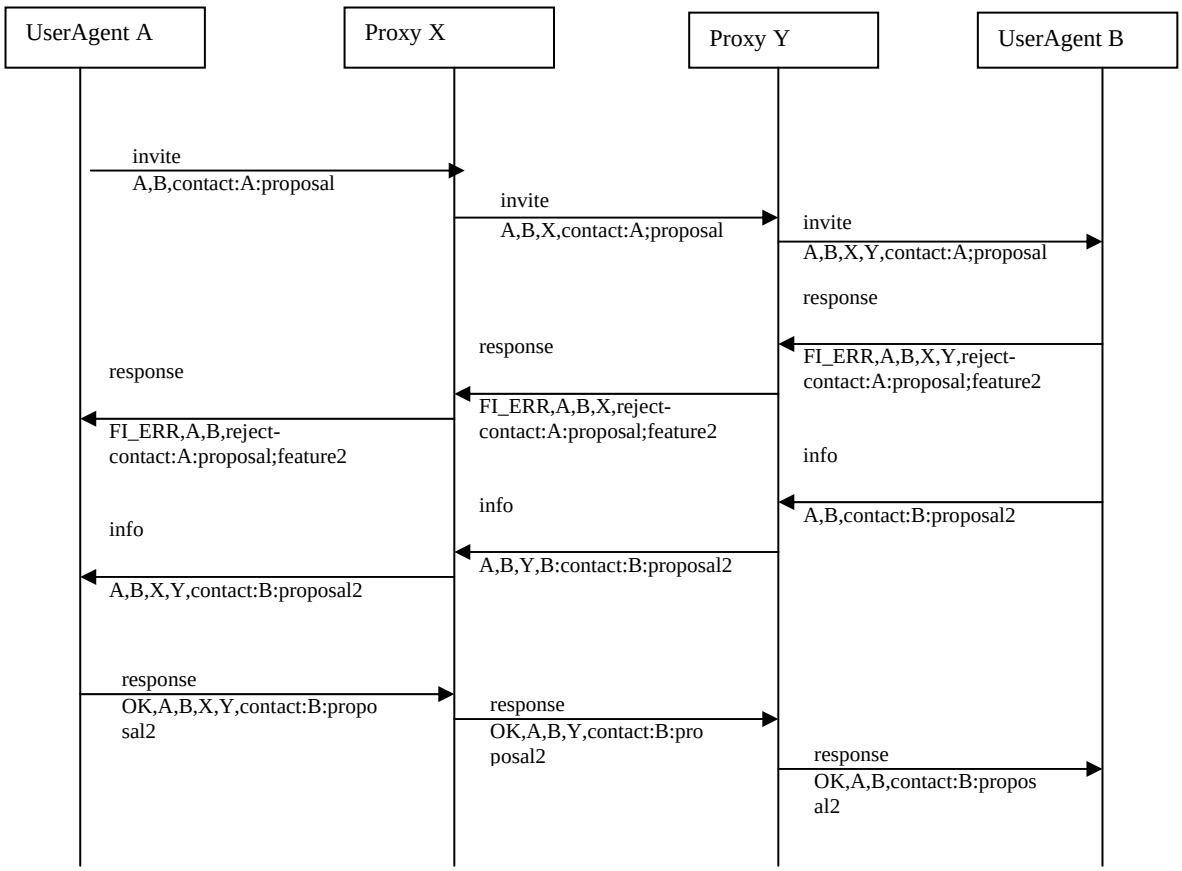

Figure 25: The callee rejects the caller's proposal and makes a counter-proposal

To prove that our feature negotiation extension to the caller preferences is feasible, we will apply this scheme to the incoherent interaction example of CFB and OCS in the subsequent section (see Section 6.8).

6.3 Preventing Resource Contention and Limitation - (CW & TWC)

The natural prevention strategy for resource limitation is to increase the number of available resources. IP Telephony services, particularly SIP, should face fewer resource limitation problems than POTS because SIP end-user devices tend to be more powerful: fast processor, a lot of memory, and flexible user interfaces and displays. For example, the semantic ambiguity of flash hook at the POTS user interface, which is the cause of both resource contention and resource limitation in the case of CW and TWC in the POTS service, should never happen with SIP phones. SIP phones should be able to display the choice of two actions, namely

putting the current call on hold and switching to the incoming call, or conferencing in the third party. The phone may assign one or more soft buttons for this purpose. However, if interactive user intervention is not possible, priority schemes (e.g. fuzzy policies [18]) may be used to resolve resource contention. Clearly, these strategies can be applied in both, design-time and run-time.

6.4 Preventing Deadlocking interactions – (CW & CFB)

Since a deadlock between two features is caused by a mutual dependency on each other's resources, removing the cyclic dependency would typically resolve this problem. The dependency in the case of CW and CFB is on the Busy signal. If user "A" makes a call to user "B" who is already busy with another call, the CFB feature subscribed by user "B" may be programmed to forward the call originated at "A" back to user "A". The CFB and CW feature may be implemented differently by different service developers. However, if the CW feature is implemented to intercept the busy signal and the CFB feature does not check for looping among calls, the CFB feature would not be triggered and the calling parties at both ends would continuously ring each other. If the CW feature is designed such that it does not intercept the busy signal, loop detection can prevent this deadlocking interaction. Loop detection and prevention is a mandatory feature of the core SIP protocol [4]. However, the loop detection in SIP is mandatory only within the same session [4], not across different sessions. We believe SIP user agents or proxies, particularly those with multi-party call features such as CFB, may avoid deadlocking interactions by examining the {From, To, Contact, Via, Record-Route} headers of all active and idle calls to determine whether a loop would occur or not. If the answer is yes, the subsequent forward would not be allowed and a response message with a response code of LOOP_DETECTED would be returned.

6.5 Preventing Livelocking interactions – (ACB & AR)

Livelock interactions, like deadlock interactions, also imply cyclic dependency on each other's resources. However, the result is that the system would fail to proceed. To break the lock or

the loop, a randomized timer can be introduced to the triggering of the affected features. For example, ACB and AR may be stuck in a livelock if ACB and AR would initiate the callback and the recall simultaneously on single-line phones. If the triggering of one of these features was delayed, one of the calls would go through. However, this livelocking interaction is unlikely to happen because SIP phones usually have more than one line. Instead, ACB and AR would result in an incoherent interaction because both users would be presented with two calls between the same endpoints. Since two different SIP user agents at both ends handle the calls, loop detection in a user agent would normally not detect this interaction at run-time. However, the user agents that are running in the same terminal may be designed such that they would share their route information with all their local user agents. As a result, this incoherent interaction could be prevented.

6.6 Preventing Unfair interactions – (CP & AA)

An unfair interaction represents the violation of a fairness property. There might be tools that can perform reachability analysis on the system and verify that all processes are treated fairly. Such properties can be specified in temporal logic. We are not aware that Tau has the ability to deals with such properties. An unfair interaction can be avoided by assigning relative priorities between the two features or by introducing randomization in the selection process (e.g. if a feature often answers an incoming call faster than other features, that feature may be assigned with a random delay timer so that other features are given a fair chance to compete for answering the incoming calls.).

6.7 Preventing Unexpected Non-determinism – (ACB & CP)

Unexpected non-deterministic interactions must be caused by at least one feature that has some non-deterministic behavior. The non-deterministic action of one feature typically triggers the other feature. Therefore, if we were to remove the triggering dependency of the second feature on the first feature, the problem would be solved. In the case of ACD and CP, an incoming call may trigger the non-deterministic invitation of ACD and the pickup invitation of

CP to the same or different destination(s). If concurrent triggering of ACD and CP was detected and only one feature was activated, the problem would be solved.

6.8 Preventing Incoherent interactions – (OCS & CFB)

Incoherent interactions can either be direct or transitive [13]. Direct incoherences are present when two features are associated with the same trigger signal but lead to different or contradictory results. A transitive incoherent interaction occurs when one feature triggers another feature, and the latter has results that are contradictory to the former feature. The word 'transitive' refers to the transitivity with respect to features that may lead to loops. Gorse has suggested a scheme to detect this type of interaction [13]. However, no prevention scheme has been presented. Since incoherences lead to contradictory results, allowing only one of the features to be triggered would prevent the incoherence. However, transitive incoherences such as OCS and CFB are difficult to prevent at run-time because the interaction may involve indirect triggering of a remote feature. How could we know that the incoming call has triggered CFB running on server Y which contradicts the assumptions of the OCS feature running on server X? Obviously, if the triggering condition of the first feature were made available to the other feature and vice versa, the two features could take advantage of the extra information and negotiate a settlement. This negotiation approach was discussed in Section 6.2. We have simulated this negotiation approach for this interaction scenario in our SDL model. Figure 26 illustrates an application of this scheme.

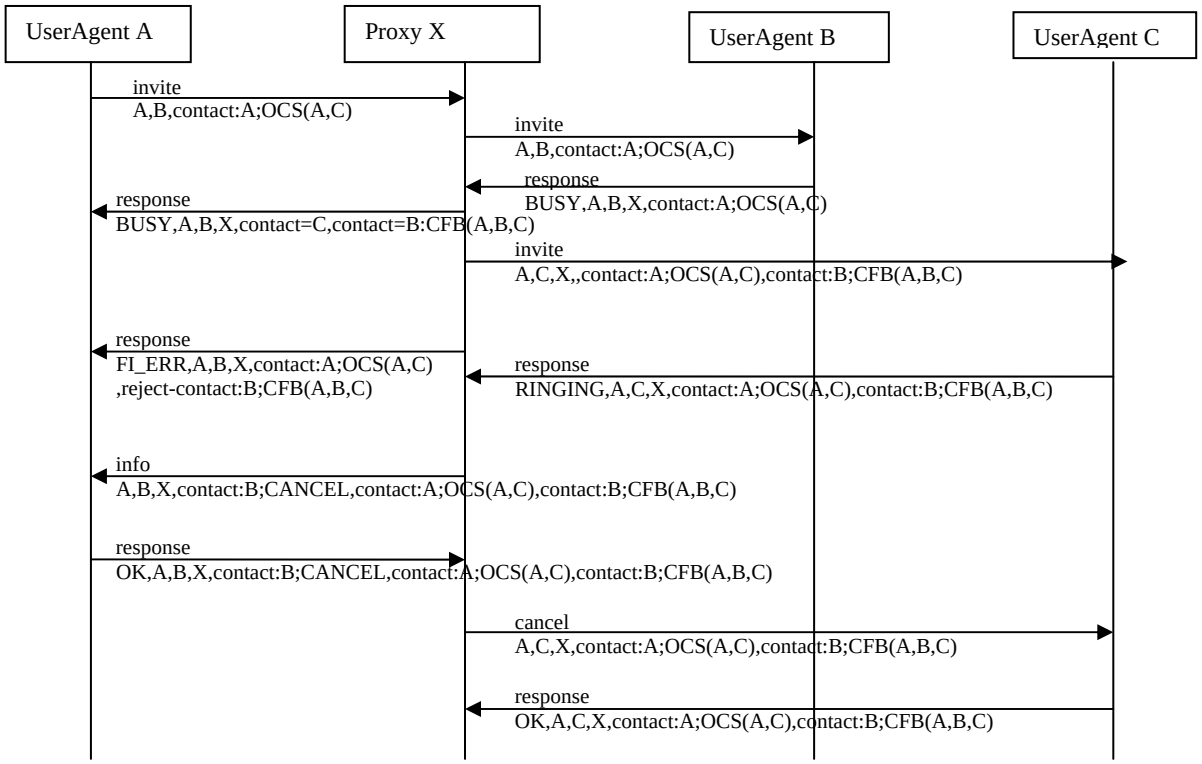

Figure 26: Resolving CFB and OCS incoherent interactions in SIP

In SIP, the 'Via' header indicates the path the request has traveled and can be used to prevent loops. The 'Record-Route' header indicates the proxy path that subsequent requests must follow. The formal specification of OCS and CFB can take advantage of header values in {From, To, Via, Record-Route, Contact} to determine whether a request has been forwarded to restricted destinations or not. In Figure 26, User Agent A, B, and C correspond to: user1, user2, user3 in the sample SIP messages described below. If OCS is activated at user agent A, the user agent A could add the feature predicate of OCS in the contact header field of the all outgoing invitation requests, indicating that OCS is present at A (see Figure 26); Both the precondition and post-condition of the OCS feature indicate the voice path between user1 and user3 must not be connected. The following is a sample of the request:

```
INVITE sip:example.com SIP/2.0
From: sip:user1@example.com;tag=asd98
To: sip:user2@example.com
Call-ID: hh89as0d-asd88jkk@host.example.com
CSeq: 1 INVITE
Max-Forwards: 70
Via: SIP/2.0/UDP host.example.com;branch=z9hG4bKnashds8
```

```
Contact: <sip:user1@host.example.com>;audio="TRUE";video="TRUE"
  ;msgserver="TRUE";automata;attendant;mobility="fixed"
  ;methods="INVITE,BYE,OPTIONS,ACK,CANCEL"
  ;uri-user="<user>"
  ;uri-domain="host.example.com"
  ;orig-call-screen=”node:caller#user1@example.com”
  ;orig-call-screen=”node:screen#user3@example.com”
  ;orig-call-screen=”precondition:!CONNECT(caller,screen)”
  ;orig-call-screen=”postcondition:!CONNECT(caller,screen)”
Content-Length: 0
```

When a downstream proxy or a back-to-back user agent that is running the CFB feature receives the invitation, it would add the ‘Record-Route’ header field with its proxy address and append all its features to the caller preferences of its response. The following is our sample response (reject) from the forwarding proxy to the caller:

```
FI_ERR sip:example.com SIP/2.0
From: sip:user1@example.com;tag=asd98
To: sip:user3@example.com
Call-ID: hh89as0d-asd88jkk@host.example.com
CSeq: 1 INVITE
Max-Forwards: 70
Via: SIP/2.0/UDP host.example.com;branch=z9hG4bKnashds8
Contact: <sip:user1@host.example.com>;audio="TRUE";video="TRUE"
  ;msgserver="TRUE";automata;attendant;mobility="fixed"
  ;methods="INVITE,BYE,OPTIONS,ACK,CANCEL"
  ;uri-user="<user>"
  ;uri-domain="host.example.com"
  ;call-forward-busy=”node;caller#user1@example.com”
  ;call-forward-busy=”node:forwarder#user2@example.com”
  ;call-forward-busy=”node:destination#user3@example.com”
  ;call-forward-busy=”trigger:in#BUSY(forwarder)”
  ;call-forward-busy=”postcondition:CONNECT(caller,destination)”
Contact: <sip:user2@host.example.com>;audio="TRUE";video="TRUE"
  ;msgserver="TRUE";automata;attendant;mobility="fixed"
  ;methods="INVITE,BYE,OPTIONS,ACK,CANCEL"
  ;uri-user="<user>"
  ;uri-domain="host.example.com"
Content-Length: 0
```

Upon the reception of the response from either the proxy or the destination, the OCS feature at the user agent A may deduce the call forward path and could determine whether a violation of its screening list occurs or not. In our case, the post-condition of the OCS feature (e.g. !CONNECT(user1@example.com, user3@exmaple.com) contradicts the post-condition of the CFB feature (e.g. CONNECT(user1@example.com, user3@exmaple.com). Therefore, a feature interaction occurs at the downstream proxy ”X”. As a result, a reject response (e.g. FI_ERR, a response code in 500-599 range, indicates the occurrence of feature interactions.) is sent to the originator. Then Proxy “X’ sends a counter proposal, which is a “CANCEL”

request to be sent to cancel all outstanding invitations, via an "INFO" message [4], to the originator. If the proposal is accepted, then the cancel operation would commence. The following forwarded invitation is our example:

```
INVITE sip:example.com SIP/2.0
  From: sip:user1@example.com;tag=asd98
  To: sip:user3@example.com
  Call-ID: hh89as0d-asd88jkk@host.example.com
  CSeq: 2 INVITE
  Max-Forwards: 70
  Via: SIP/2.0/UDP host.example.com;branch=z9hG4bKnashds8
  Contact: <sip:user1@host.example.com>;audio="TRUE";video="TRUE"
    ;msgserver="TRUE";automata;attendant;mobility="fixed"
    ;methods="INVITE,BYE,OPTIONS,ACK,CANCEL"
    ;uri-user="<user>"
    ;uri-domain="host.example.com"
    ;call-forward-busy="node;caller#user1@example.com"
    ;call-forward-busy="node:forwarder#user2@example.com"
    ;call-forward-busy="node:destination#user3@example.com"
    ;call-forward-busy="trigger:in#BUSY(forwarder)"
    ;call-forward-busy="postcondition:CONNECT(caller,destination)"
  Contact: <sip:user2@host.example.com>;audio="TRUE";video="TRUE"
    ;msgserver="TRUE";automata;attendant;mobility="fixed"
    ;methods="INVITE,BYE,OPTIONS,ACK,CANCEL"
    ;uri-user="<user>"
    ;uri-domain="host.example.com"
    ;orig-call-screen="node:caller#user1@example.com"
    ;orig-call-screen="node:screen#user3@example.com"
    ;orig-call-screen="precondition:!CONNECT(caller,screen)"
    ;orig-call-screen="postcondition:!CONNECT(caller,screen)"
  Content-Length: 0
```

Furthermore, a feature may be assigned with a priority value. The use of feature priority has been suggested in [18]. The priority value may be used to determine the appropriate actions for resolving a feature interaction. For example, if the OCS feature were to have a higher priority than the CFB feature, the OCS user agent could send a cancel request instead of the final acknowledgement message, in order to cancel the original invitation.

We have shown that a simple and effective negotiation extension to SIP could resolve complex multiple-users-multiple-components feature interactions such as OCS and CFB. We have taken feature priority into account for our feature predicate scheme and have shown that simple comparison of feature priority values appear to be sufficient for resolving feature interactions. It is conceivable that more complex applications of feature priority may offer better resolution results. However, this is outside the scope of our study.

7 New Feature Interactions in SIP

Lennox introduced two feature interaction categories for IP telephony services called cooperative and adversarial interactions [9]. They are incidentally similar to, but not the same as, the concepts called cooperative and interfering interactions defined by Gorse [13]. In the following subsections, the SIP FI's described by Lennox are associated to our proposed taxonomy described in Section 3. Corresponding preventive measures will also be presented. Furthermore, we have used our intuitions to discover some new cooperative and adversarial feature interactions that are unique to SIP. We will show how they could be associated to our proposed taxonomy.

7.1 Cooperative Feature Interactions

Cooperative FI's are multi-component interactions (SUMC or MUMC) where all components share a common goal, but have different and uncoordinated ways of achieving it [9]. Specific examples would be livelock, deadlock, unfairness, and unexpected non-deterministic multi-component interactions. The fact that features contend for a single resource or for each other's resources in order to establish call connections, resulting in violations of safeness and liveness properties, is a form of cooperative interactions. Let us examine the following SIP feature interaction proposed by Lennox.

7.1.1 Request Forking (RF) and Auto-Answer (voicemail)

Request Forking (RF) and Auto-answer (voicemail) is a potential cooperative FI. Request forking is unique to SIP; it allows a SIP proxy server to attempt to locate a user by forwarding a request to multiple destinations in parallel [9]. The first destination to accept the request will be connected, and the call attempts to the others will be cancelled. The Auto-Answer (AA) or Call Forward to Voice Mail is designed to accept all incoming calls while the user is unavailable. Both features are intended to ensure that the user does not miss the call. However, since AA

would answer the call request almost immediately, RF would always forward the call request to the same AA destination. Therefore, the user who may be closer to the other destinations would never be able to answer the call, and hence the intention of RF is ignored. To resolve this violation of fairness, one may introduce a delay timer to AA (e.g. answer after 4 rings) such that all destinations would be given a fair chance to answer the call.

7.2 Adversarial Feature Interactions

Adversarial FI's, by contrast, are multi-component interactions (SUMC, MUMC) where all components disagree about something to be done with the call (e.g. violation of feature assumptions) [9]. They are a form of incoherent interactions and are difficult to detect. The following new interactions are specific to SIP and have not been mentioned in [9].

7.2.1 Timed ACD and Timed Terminating Call Screening

Timed ACD and Timed Terminating Call Screening (TCS) is a potential adversarial FI. With IP telephony, service designers can rapidly create innovative features; for example, time or calendar-based forwarding and screening features. In this example, TCS restricts incoming calls that are originating from certain callers at the destination. ACD may be programmed to distribute different kinds of incoming calls to different sets of destinations depending on the time of the day (e.g. calls from A or B to destinations S or T in the daytime, and calls from C or D to destinations X or Y in the nighttime). However, destinations S and T are programmed to screen calls from A or B in daytime and X and Y to screen calls from C and D at nighttime. Although ACD is intended to select the best route for incoming calls, TCS at various destinations is programmed to screen calls from these destinations at given time periods. As a result, their policies contradict with each other and no calls between these callers and callees can ever be completed in the conflicting time period. When time is involved in incoherent interactions, the solution to such conflicting policies is not trivial. This interaction is particularly relevant to user call policies specified in CPL [46]. With CPL, the user may indicate the action to be performed (e.g. forwarding an 'INVITE" request to another proxy) based on

the header information of the incoming or outgoing messages. In addition to the header field values such as the addressing fields, a CPL script that specifies a user's call policy may perform actions based on time. Obviously, interactions (e.g. time based interactions) between CPL scripts belonging to different users are even more difficult to detect and resolve.

7.2.2 Call Screening and Register

Call Screening and Register is another form of multi-component interaction unique to SIP; it involves dynamic address changes in SIP. A user agent or proxy may add, delete, or modify the current contact address of a user stored in the registrar. The address can be changed at any time by sending a "Register" request message to the registrar. Since a user cannot possibly stay current with the latest address change of another user, the call screening features would be rendered useless. It is a form of incoherent interaction. This interaction involves a debate of privacy for the call screener and the caller. These security issues are beyond the scope of this thesis.

7.2.3 Dynamic Addressing and User Mobility and Anonymity

Dynamic Addressing and User Mobility and Anonymity is another adversarial FI that centers on the controversial issue on the balance between the lack of address scarcity in the Internet [1] and the correct programming of features that depend on reliable addresses. As it becomes the norm that a user owns more than one intelligent wireless and/or wired personal communication device, user mobility is a serious issue in designing reliable services. Furthermore, anonymous email accounts and email spamming are big problems in the Internet. Could anonymous accounts and telemarketing spam calls emerge in the SIP world when SIP becomes popular? If SIP addresses were just as easy to obtain as email addresses, the incoherent feature interactions between features like call screening and call forwarding would be further complicated. For example, if a user, called Jane, programs her phone to screen all incoming calls coming from the address "sip:joe@msn.com", Joe might easily obtain an anonymous SIP address from another SIP service provider and by-pass Jane's call screening.

Similarly, if a proxy is programmed to forward all incoming calls to an anonymous SIP account, the true identity of the receiver would be unknown. We believe SIP features that rely on "accurate" addresses would need to be more sophisticated than those POTS features.

8 Experience with SDL and Telelogic Tau and possible enhancements

In the course of modeling SIP services, we discovered not only the advantages but also some shortcomings of using SDL and CASE tools such as Telelogic Tau to model an IETF application protocol such as SIP. SDL is a good language to model SIP because SIP and its services can be easily expressed as interactions between extended communicating finite state machines. We can simulate a distributed SIP system, a transition at a time, using Tau. Other advantages of using Tau include 1) the ability to express test scenarios as MSCs and to automatically verify these MSCs against the model, 2) the ability to verify that the model does not violate certain properties (e.g. livelock) that are written in the form of Observer Process Assertions.

However, we find that the lack of SDL packages describing certain abstract data types in Tau presents a problem for modeling SIP. Unlike bit-oriented protocols such as Ethernet and Token Ring, SIP is an ASCII-based, attribute-rich signaling protocol with mandatory, optional and variable size parameters. We have to model SIP messages as SDL signals; and we cannot easily insert, remove, search, and modify values from the optional and/or variable size SIP header fields. Although it is not impossible to model variable header fields with fixed size arrays in Tau, it is a very inconvenient task. Many programming languages include the language feature of pointer (C++ [32]) and/or reference (Java [31], C++) which allows the programmer to build his/her customized data structures. However, a pointer is a dangerous programming feature that is normally not included in specification languages. The SDL language could be extended with additional built-in ADTs. We believe that the support for certain common abstract data types, such as *linked list* and *hash table*, should be made mandatory in SDL. This would be sufficient to enhance our modeling experience. A *linked list* can be used to describe a variable–length parameter list whereas a *hash table* can be used to store all the key-value pairs in an SDP body. These two abstract data types eliminate the need to re-index the content. Although SDL has the *Vector* type which is basically a subtype of *Array*, this *Vector* type does not have the properties of a Java *Vector*; it does not have insert, remove, and update operators.

Secondly, we believe that the SDL language should be extended with string processing facilities. In programming languages such as Java and C++, the language definition comes with string processing libraries. For example, the *int indexOf(String substring)* operator of the *String*

class in Java returns the index position of the first occurrence of the *substring* in that *string* object instance [16]. Similar operations could be added to the SDL language. Furthermore, regular expression operators may be added to the SDL language. These string processing operations would facilitate the checking and comparisons of SIP header values. Without these operations, it is difficult to develop a complete model of an IETF application protocol like SIP.

Furthermore, we find that the Tau validation tool needs to offer more flexible verification features. For example, the ability to incorporate model checking of the SDL system using temporal logic formula would be a bonus. We may use temporal logic formula to verify more complex distributed system properties such as live-locks. In addition, we could not easily specify actors in our use case scenarios; we group all the actors together as the environment. We could create additional actor/user processes to simulate actors. However, if we were to show the interactions between actors in our use case scenarios, the message sequence charts would also display the exchange of internal SIP messages between user agent and proxy processes. We could probably develop an actor-layer to partition user signals from internal signals but it would not be a trivial task. Ideally, Tau could offer different environment components for different channels to a system. This way, we would not need to develop our actor-layer in the model.

Last but not least, we find that the SDL editor lacks GUI context sensitive features, such as allowing a designer to select one of the existing state names when he/she creates a new state label, that are normally found in other CASE tools such as Microsoft Visual Studio [33]. Cutting and pasting objects between pages is also not always allowed; these some the minor improvements that can be made to the tools.

9 Conclusion and Future Work

With all the contributions we have made in our work, we have shown that we have successfully come up with methods for formally specifying SIP services with fewer feature interactions. Although RFC 2543 [4] gives a detailed protocol specification for SIP and the textual service description and call flow diagrams described in [6,7] offer us some clues on SIP services, they do not represent the formal service specification for SIP. As the IETF Session Initiation Protocol (SIP) is becoming one of the most important Internet telephony signaling protocols, we believe a formal service specification for SIP is much needed by the IP Telephony community. Also, we have explained the importance of reducing (better yet, eliminating) feature interactions in our service specifications. Our methods for detecting and preventing (resolving) feature interactions have also proved to be successful in achieving our objective. In this chapter, we will conclude with a list of contributions of this thesis. We will describe briefly each of these contributions. Finally, we will discuss the future work items that could be important to the IP Telephony, formal methods, and feature interaction communities.

9.1 List of Contributions

The main contributions of this thesis are the following:

1. **A formal model of SIP (IP Telephony) and its services in the form of an SDL specification:** The SDL model covers some of the important characteristics of SIP, such as caller-id, command sequence and, to a certain extent, most of the mandatory addressing header fields. It includes the basic call model and many additional features as described in (see Section 3). Although we have not mapped our service specifications to other Voice over IP (VoIP) signaling protocols like H.323, we believe the fact that we have a protocol-independent "Abstract User interface" to describe our use case scenarios would facilitate porting our work to other VoIP protocols in the future.

2. **An "Abstract User" interface for IP Telephony:** As part of the main contribution to formal modeling of SIP services, we have explained that the "Abstract User interface" which we developed has allowed us to develop a more user-centric and precise formal service specification for SIP. We believe SIP or any IETF application protocols should be specified from a user-centric perspective. Currently, some of the IETF RFCs and drafts for the application protocols are not written with use case s in mind. Many researchers have found IETF documents difficult to read. In addition, we have shown that our SDL framework allows us to reuse and to add SIP services to the core protocol quite easily.
3. **Discussion of the benefits and shortcomings of using SDL and related tools:** We have discussed the benefits and shortcomings of using a modeling and verification tool like Telelogic Tau (see Section 8). Our experience with verifying and validating our SDL model is also described. In addition, we have discussed the advantages and shortcomings of using a formal language such as SDL to model an IETF application signaling protocol like SIP. We have shown that modeling a complex IETF protocol like SIP using CASE tools such as Telelogic Tau SDL & TTCN Suite is feasible. Moreover, we have proposed potential SDL and Tau enhancements to make these tools more developer-friendly. Although Tau has a reliable editor and simulation engine, we have run into many difficulties with Tau's Validator (the validation engine). The validation engine is the tool that we used to perform various state space explorations for verification and validation, and also verification of Observer Process Assertions. It is the basis of our methods for detecting feature interactions. However, it frequently crashes with General Application Errors, which are unrecoverable in the Windows environment. We are not certain of the root cause of this problem but we suspect that the validation engine may not be able to handle very big state spaces or certain complex data structures. Hopefully, these issues would be addressed in future versions of the tool.
4. **Methods for preventing (resolving) feature interactions:** We have described methods for preventing (resolving) traditional feature interactions, and potentially new feature interactions, in SIP (see Section 6). We have proposed a feature negotiation framework that was inspired by the caller preference extension to SIP [36] and the negotiating agent approach [20]. By inserting and updating feature parameters in the

message headers of request and response messages throughout a session, we were able to resolve feature interactions independent of the number of users and components (nodes). Furthermore, we have shown the framework can resolve the five proposed types of feature interactions, namely livelocking, deadlocking, incoherent, unfair, and unexpected non-deterministic feature interactions. However, we have only implemented OCS and CFB with the simple proposed scheme using our feature negotiation approach in our SDL model. Implementing the full framework for all other services may be part of the future work.

5. **Methods for detecting feature interactions:** We have discussed how to detect feature interactions with the Tau tool (see Section 5). We have tried to characterize feature interactions using MSCs and Observer Process Assertions. However, we discovered that the use of Observer Process Assertions is the only viable approach in Tau for detecting feature interactions. We were able to discover livelocking, deadlocking, and incoherent feature interactions among CFB, OCS, TCS, CW using Observer Process Assertions.
6. **An extension to the traditional feature interaction taxonomy:** We have described an extension to the traditional feature interaction taxonomy (see Section 3). We have shown that our extended taxonomy allows us to associate the causes of feature interactions to our methods for detecting and resolving (preventing) feature interactions. In addition, the extended taxonomy (classification system) helped us to discover some new feature interactions (see Section 7).
7. **Description of new SIP feature interactions:** Last but not least, we have identified some new SIP feature interactions (see Section 7), such as *Request Forking (RF) and Auto-answer (voicemail)*, *Timed ACD and Timed Terminating Call Screening (TCS)*, *Call Screening and Register*, and *Dynamic Addressing and User Mobility and Anonymity*. Although we were not able to come up with resolution (preventive) schemes for all these new SIP feature interactions, we could associate some of them to our extended taxonomy: *Request Forking (RF) and Auto-answer (voicemail)* to unfair interactions, *Timed ACD and Timed Terminating Call Screening (TCS)* to incoherent interactions, and *Call Screening and Register* to incoherent interactions.

9.2 Future Work

Although we have made a number of significant contributions to three research areas: the service specification of SIP (IP Telephony), the application of formal modeling using SDL, and the pragmatic application of feature interactions, we have also discovered that there are many interesting issues to be explored. First of all, an important future work item would be to modify our model (see Section 4) to support the new SIP standard which is specified under [29]. We intended to start the model with the latest standard but unfortunately the new RFC came to our attention only in the later stage of our project. However, an IETF standard is always an evolving standard; a new RFC could obsolete another RFC in a couple years. It is a never-ending catch-up game. Since most of the current reference implementations are based on RFC 2543 [4], we believe our model is very much useful to many researchers and industrial partners. Furthermore, there are many similarities between RFC 2543 [4] and RFC 3261 [29]. Our model could easily be modified to address the new standard. However, it is beyond the scope of this thesis.

We have used MSCs to characterize user interaction scenarios and used these scenarios to verify the SDL model of SIP (see Section 5). However, MSCs have limitations in terms of expressing quantification of instances and their behaviors. Live Sequence Charts (LSC) [30], which have not been discussed much in this thesis, appear to be a promising extension to MSCs in this context. We believe describing feature interactions in the form of LSC could be an interesting approach to detecting feature interactions.

We have also used the observer processes provided by Tau to detect FI's. This seems to be a practical semi-automated approach, particularly to detect incoherent interactions and to verify liveness properties. It would be interesting to characterize other feature interactions as assertions. In addition, we could not generalize our Observer Process Assertions to automate our detection process because the syntax of Tau's Observer Process Assertions is too restrictive; Tau should allow the assertion writer to quantify over SDL instances in their assertions. This way, the assertions for certain feature interactions could be reused for other features. Thus, automating the feature interaction detection process may become possible.

We have shown that our feature negotiation framework is useful for resolving feature interactions (see Section 6). The framework appears to resolve various types of feature interactions in theory, and we have simulated the scheme for the feature interaction case of

CFB and OCS in our model. However, it would be interesting to implement our feature negotiation approach as a generalized framework in our SDL model, and better yet, in a target executable. We could then execute our target executable against other SIP implementations and verify whether our resolution scheme would work with other vendor's solutions. In addition, we believe our feature negotiation scheme is compatible with the IETF caller preference [36] draft, we believe it would be interesting either to incorporate our ideas into these drafts or to propose another IETF draft so that the SIP community can be benefited from our feature interaction work.

Furthermore, our extension to the classical feature interaction classification has shown the importance of categorizing feature interactions for preventing (resolving) feature interactions (see Section 3). However, we have not explored potentially useful properties of the feature interaction tree (FIT); properties that would help us to deduce new feature interactions. This is a task that involves the evolution of the "FIT" from a visual notation for merely categorizing feature interactions to a formalism for specifying feature interactions. It is a big step forward and is beyond the scope of the thesis.

Finally, we believe it is also interesting to use the Target Expert tool included in Tau to generate target executables in the future. We could then use the executable to verify whether our implementation would interoperate with other SIP implementations. Then, we could compare the performance of our model against those of other vendors. This would not only give us the confidence in our model but also promote our work to other researchers and SIP developers.

References

[1] K. Chan, and G. v. Bochmann, "Methods for Designing IP Telephony Services with Fewer Feature Interactions", Feature Interactions in Telecommunications and Software Systems VII, IOS Press, accepted and to be published, June 2003.

[2] K. Chan, and G. v. Bochmann, "Modeling IETF Session Initiation Protocol and its services in SDL", In Proceeding of Eleventh SDL Forum, LNCS, Springer-Verlag Heidelberg, accepted and to be published, Stuttgart, Germany, July 1-4, 2003.

[3] E.J.Cameron, N.D.Griffeth, Y.-J. Lin, M. E. Nilson, W. K. Shure, and H. Velthuijsen, "A feature interaction benchmark for IN and beyond," Feature Interactions in Telecommunications Systems, IOS Press, pp. 1-23, 1994.

[4] M. Handley, H. Shulzrinne, E. Schooler, and J. Rosenberg, "SIP: Session Initiation Protocol", Request For Comments (Proposed Standard) 2543, Internet Engineering Task Force, Mar. 1999.

[5] M. Handley, and v. Jacobson, "SDP: Session Description Protocol", Request For Comments (Proposed Standard) 2327, Internet Engineering Task Force, April 1998.

[6] A. Johnston, R. Sparks, C. Cunningham, S. Donovan, and K. Summers, "SIP Service Examples", draft-ietf-sipping-service-examples-04.txt, Internet Draft, Internet Engineering Task Force, Expires Aug 2 2003, Work in progress.

[7] A. Johnston, S. Donovan, R. Sparks, C. Cunningham, and K. Summers, "Session Initiation Protocol Basic Call Flow Examples", draft-ietf-sipping-basic-call-flows-01.txt, Internet Draft, Internet Engineering Task Force, Expires April 2003.Work in progress.

[8] J. Lennox, H. Schulzrinne, and T. Porta, "Implementing Intelligent Network Services with Session Initiation Services", http://www.cs.columbia.edu/~lennox/cucs-002-99.pdf, accessed on October 7, 2002

[9] J. Lennox, and H. Schulzrinne, "Feature Interaction in Internet Telephony", Sixth Feature Interaction Workshop, IOS Press, May. 2000.

[10] International Telecommunication Union, "ITU-T Recommendation Z.100: Specification and Description Language (SDL)", ITU-T, Geneva, Switzerland, 1999.

[11] International Telecommunication Union, "ITU-T Recommendation Z.120: Message Sequence Chart (MSC)", ITU-T, Geneva, Switzerland, 1999.

[12] Telelogic Inc., "Telelogic Tau SDL & TTCN Suite", version 4.3 and 4.4, http://www.telelogic.com, accessed on Dec 20, 2002.

[13] N. Gorse, "The Feature Interaction Problem: Automatic Filtering of Incoherences & Generation of Validation Test Suites at the Design Stage" (Master Thesis), University of Ottawa, Ottawa, Ontario, Canada, 2001.

[14] Holzmann, G.J. (1988) 'An improved protocol reachability analysis technique', Software Practice and Experience, Vol. 18, No. 2, pp. 137-161.

[15] Ø. Haugen, "MSC Methodology", http://www.informatics.sintef.no/projects/sisu/sluttrapp/publicen.htm, accessed on Dec. 23, 2002, SISU, DES 94, Oslo, Norway, 1994.

[16] J. Ellsberger, D. Hogrefe, and A. Sarma, "SDL - Formal Object-oriented language for Communication Systems", Prentice Hall Europe, ISBN 0-13-621384-7, 1997.

[17] J. Bowen, "The World Wide Web Virtual Library: Formal Methods", http://www.afm.sbu.ac.uk/, Centre for Applied Formal Methods, SCISM, South Bank University, London, UK, accessed on November 2, 2002

[18] M. Amer, A. Karmouch, T. Gray, and S. Mankovskii, "Feature-Interaction Resolution Using Fuzzy Policies", Feature Interactions in Telecommunications and Software Systems VI, pp. 94-111, IOS Press, 2000.

[19] M. Calder, M. Kolberg, E. Magill, and S.Reiff-Marganiec, "Feature Interaction: A Critical Review and Considered Forecast", http://www.dcs.gla.ac.uk/~muffy/papers/calder-kolberg-magill-reiff.pdf, accessed on February 06, 2002.

[20] N.D. Griffeth and H. Velthuijsen, "The Negotiating Agents Approach to Runtime Feature Interaction Resolution", in: Feature Interactions in Telecommunications Systems, L.G. Bouma and H. Velthuijsen (eds.), IOS Press, Amsterdam, pp. 217-235, 1994.

[21] Pansy Au, and Joanne Atlee, "Evaluation of a State-Based Model of Feature Interactions", Fourth International Workshop on Feature Interactions in Telecommunication and Software Systems, pp. 153-167, June 1997.

[22] L. Blair, and J. Pang, "Feature Interactions – Life Beyond Traditional Telephony", Feature Interactions in Telecommunications and Software Systems VI, IOS Press, pp.83-93, 2000.

[23] Amy P. Felty and Kedar S. Namjoshi, "Feature Specification and Automatic Conflict Detection", In M. Calder and E. Magill, editors, *Feature Interactions in Telecommunications and Software Systems VI*, IOS Press, May 2000

[24] J. Kamoun and L. Logrippo, "Goal-Oriented Feature Interaction Detection in the Intelligent Network Model", School of Information Technology and Engineering, University of Ottawa, Ottawa, Ontario, Canada, 1998.

[25] R. Pressman, "Software Engineering A practitioner's approach", 3rd edition, pp.173-661, ISBN 0-07-112779-8, Computer Science Series, McGraw-Hill International Editions, McGraw-Hill Inc., U.S.A, 1992.

[26] Cinderella Inc., "Cinderella SDL", version 1.3, http://www.cinderella.com, accessed on January 14, 2002.

[27] Object Management Group Inc., "Unified Modeling Language", version 1.5, http://www.omg.org/uml/, accessed on March 1, 2003.

[28] ITU, "Packet based multimedia communication systems", Recommendation H.323, Geneva, Switzerland, Feb. 1998.

[29] J. Rosenberg, H. Shulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: Session Initiation Protocol", Request For Comments (Standards Track) 3261, Internet Engineering Task Force, June 2002.

[30] W. Damm, and D. Harel, "LSCs: Breathing Life into Message Sequence Charts*",

Formal Methods in System Design, Kluwer Academic Publishers, pp. 19,45-80, 2001.

[31] Sun Microsystems Inc., "The Source for JavaTM Technology", version 1.x, http://java.sun.com, accessed on Feb 10, 2003.

[32] B. Stroustrup, "Stroustrup: C++", http://www.research.att.com/~bs/C++.html, accessed on Feb 10, 2003.

[33] Microsoft Inc., "Visual Studio Home Page", http://msdn.microsoft.com/vstudio, accessed on Feb 10, 2003.

[34] R. Fielding, J. Gettys, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", Request For Comments (Standards Track) 2616, Internet Engineering Task Force, June 1999.

[35] A. Tanenbaum, "Computer Networks", Third Edition, Prentice Hall Inc., New Jersey, U.S.A., pp. 496-498, 1996.

[36] J. Rosenberg, H, Schulzrinne, and P.Kyzivat, "Caller Preferences and Callee Capabilities for the Session Initiation Protocol (SIP)", draft-ietf-sip-callerprefs-08.txt, Internet Draft, Internet Engineering Task Force, Expires Aug 2003, Work in progress.

[37] J. Rosenberg, and H, Schulzrinne, "An Offer/Answer Model with the Session Description Protocol (SDP)", Request For Comments (Standards Track) 3264, Internet Engineering Task Force, June 2002.

[38] G. Holzmann, "Design and Validation of Computer Protocols", Lucent Technologies, Bell Laboratories, Prentice Hall Inc., ISBN 0-13-539925-4, New Jersey, U.S.A., pp. 90-351, 1991.

[39] D. Amyot, "Use Requirements Notation", http://www.usecasemaps.org/urn, accessed on March 2,2003.

[40] D. Amyot, "Use Case Maps Web Page", http://www.usecasemaps.org, accessed on March 2,2003.

[41] University of Toronto, "Goal-Oriented Requirement Language", http://www.cs.toronto.edu/km/GRL, accessed on March 2, 2003.

[42] G. v. Bochmann, "Course Notes ELG 7186G – Formal Methods for the Development of Real Time System Applications ", http://beethoven.site.uottawa.ca/CourseModules/IntroRTSystems&FormalMethods/Informal&FormalMethods.pdf, accessed on March 27, 2003.

[43] K. Chan, "Ken Chan University of Ottawa Thesis Page", http://beethoven.site.uottawa.ca/DSRG/PublicDocuments/REPORTS-THESES/Thesis-Chan/, accessed on March 31, 2003.

[44] H. Schulzrinne, and J. Rosenberg, "Internet Telephony: architecture and protocols – an IETF perspective", Computer Networks, Vol. 31, Issue 3, pp. 237-255, Feb. 1999.

[45] ISO, IS 8807, Information Processing Systems – Open Systems Interconnect – LOTOS: A Formal Description Technique Based on Temporal Ordering of Observational Behaviors, May 1989 (E. Brinksma, editor).

[46] J. Lennox, and H. Schulzrinne, "Call processing language framework and

requirements," RFC 2824, Internet Engineering Task Force, May 2000.

[47] G. Klyne, "A Syntax for describing media feature sets", RFC 2533, Internet Engineering Task Force, Mar. 1999.

[48] G. Klyne, "Protocol-independent content negotiation framework," RFC 2703, Internet Engineering Task Force, Sept. 1999.

[49] S. Mauw, and M. A. Reniers, "High-level message sequence charts," In Proceedings of the Eighth SDL Forum (SDL'97), pp.291-306, 1997.

[50] Sun Microsystems Inc., "Java APIs for Integrated Networks", http://java.sun.com/products/jain/index.html, accessed on March 12,2003.

[51] J.Glasmann, W.Kellerer, and H. Müller, "Service Architectures in H.323 and SIP – A Comparison", http://www.h323forum.org/papers/Service_Architecures_SIP-H323.pdf, accessed on April 2, 2003.