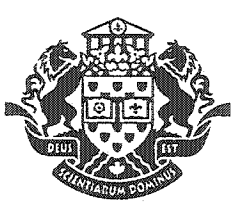

Université d'Ottawa · University of Ottawa

Université d'Ottawa · University of Ottawa

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Yiqun XU

AUTEUR DE LA THÈSE - AUTHOR OF THESIS

Master of Computer Science

GRADE - DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT - FACULTY, SCHOOL, DEPARTMENT

TITRE DE LA THÈSE - TITLE OF THE THESIS

Detecting Feature Interactions and Feature Inconsistencies in CPL

L. Logrippo

DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

CO-DIRECTEUR DE LA THÈSE - THESIS CO-SUPERVISOR

EXAMINATEURS DE LA THÈSE - THESIS EXAMINERS

B. Pagurek

A. Williams

J.-M. De Koninck, Ph.D.

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

SIGNATURE

DEAN OF THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Detecting Feature Interactions and Feature Inconsistencies in CPL

Yiqun Xu

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

Master of Computer Science

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

University of Ottawa
Ottawa, Ontario, Canada
September 2003

Your file Votre référence
ISBN: 0-612-90360-5
Our file Notre référence
ISBN: 0-612-90360-5

Abstract

Internet Telephony grants users the power of developing their own telephony services, and the Call Processing Language (CPL) has been designed to to fulfil this target. However, this objective confronts a major obstacle known as the feature interaction problem, which describes the situation that one feature or service is violated by another in overall system behaviour.

This thesis addresses a Feature Interaction detection approach for CPL. Starting with a review of the issue of Feature Interaction, we extend the traditional Feature Interaction definition to Intention Interaction and Policy Interaction in Internet Telephony. Existing related work is discussed as well.

We also give an overview of Internet Telephony, and analyse the structure of CPL. A logic-based language, the Simple Formal Specification Language (SFSL) is introduced to express formally the intention of CPL scripts. Method of translating CPL into SFSL is presented as well. Based on the SFSL specifications, we propose detection rules to identify feature interactions in CPL, locally and pair-wise.

An automatic detection tool applying the detection rules is implemented in SWI-Prolog. Finally, in order to validate the correctness of the detection rules, we prove the logical incoherencies behind these rules using Predicate Logic.

Keywords: Feature Interaction, Detection, Logic, Prolog, CPL, Internet Telephony, Formal Method, Software Engineering

Acknowledgement

I would like to express my deepest gratitude to my supervisor, Professor Luigi Logrippo, for his guidance, encouragement and support during this research. He not only has given me his great insights and valuable suggestions for this work, but also set up a model of how to conduct academic research with his dedication to his work and students. I also wish to thank Prof. Amy Felty and Prof. Daniel Amyot, for their insightful suggestions and the innovative discussions during this research. The knowledge I acquired from one of Prof. Felty's wonderful courses helped me a lot in Formal Method and Logic.

I wish to thank the current and former members of the University of Ottawa LOTOS group for their support and friendship, especially Jacques Sincennes, Nicolas Gorse, Ruoshan Guan, Dongmei Jiang, Romelia Plesa and Waël Hassan. Particularly, I would like to thank Jacques Sincennes who gave me precious technical support and helped me greatly in Prolog programming and CPL structure analysis.

I thank the Communications and Information Technology Ontario (CITO) and the Natural Science and Engineering Research Council (NSERC) for their financial support.

Finally, I would like to express my eternal gratitude to my parents, Baojun Xu and Zhimin Dou, for their endless love and encouragement. I would like to dedicate this thesis to my wife, Wei Dong, who always shared my challenges and achievements during my Master studies.

Table of Contents

Chapter 1 Introduction

1.1 Motivation

With the coming Internet age, Internet Telephony [20] is the subject of intense research by telecom operating companies, telecom device producers and consumer groups. It promises sophisticated telephony-like services over the Internet with lower prices and more flexibility. However, the methods of deploying services on the Internet Telephony are far from mature. On one hand, the Internet platform and new signaling systems enable more opportunities for new services and new features; on the other hand, the Internet Telephony network is more distributed and less controlled. One of the most serious problems caused by immaturely deployed services is Feature Interaction, which describes the situation that one feature or service is violated by another in overall system behavior. Such interactions were possible in traditional telephony but the risk increases significantly in Internet Telephony, where users are offered more power such as they can program their own features. A typical example of feature interactions in Internet Telephony is the case where a user programs a service that all the incoming calls to him should be forwarded to his colleague when he is in the meeting room (the system could know his location from his end-phone registration information); meanwhile, he sets another feature that the calls from his lawyer should be forwarded to his personal voice mail when he is not in his office. Suppose he is attending a meeting in the meeting room (not in his office) when his lawyer calls him, what should the system do, forward this call to his colleague or to his voice mail?

Researchers are offering many approaches for creating and managing services on Internet Telephony. CPL, the Call Processing Language [28], is one of them. CPL is XML [4] based, fairly safe and signalling independent. Also, it is easy to implement and currently is the easiest and most powerful tool for the deployment of Internet Telephony services. CPL prevents some types of feature interaction problems by setting feature priorities within a CPL script; however, it still cannot guarantee that features do not conflict with each other or with subscribers' intention. Things could be even worse with the spreading of CPL implementations.

Hence, it is necessary to develop a new method to detect Feature Interactions in CPL scripts before they are activated. In this thesis, we propose FI detection rules and implement an automatic filter tool to reduce potential Feature Interactions in CPL both locally and pair-wise. We believe that this effort has the potential to significantly improve the deployment of services in CPL.

1.2 Internet Telephony

Internet Telephony, which is also called voice-over-IP or IP telephony, has been designed to provide telephony services over the Internet. It claims to be able to offer not only traditional voice services, but also many new ones such as email and integration of voice, multimedia and data (see [41], [26] for more information about the deployment of features in Internet Telephony). Moreover, reducing cost and ease of deploying new services are also motivations of the Internet Telephony.

Among all the challenges in the Internet Telephony, the key one is the establishment and control of real-time sessions. Figure 1.1 shows the stack of protocols in the Internet Telephony, among which, Session Initiation Protocol (SIP) [18], H.323 [24] [16] and Real Time Streaming Protocol (RTSP) [37] have the function of signalling protocols; RTSP is responsible for controlling multimedia streams while H.323 and SIP play similar roles on initialising and managing sessions or calls [36].

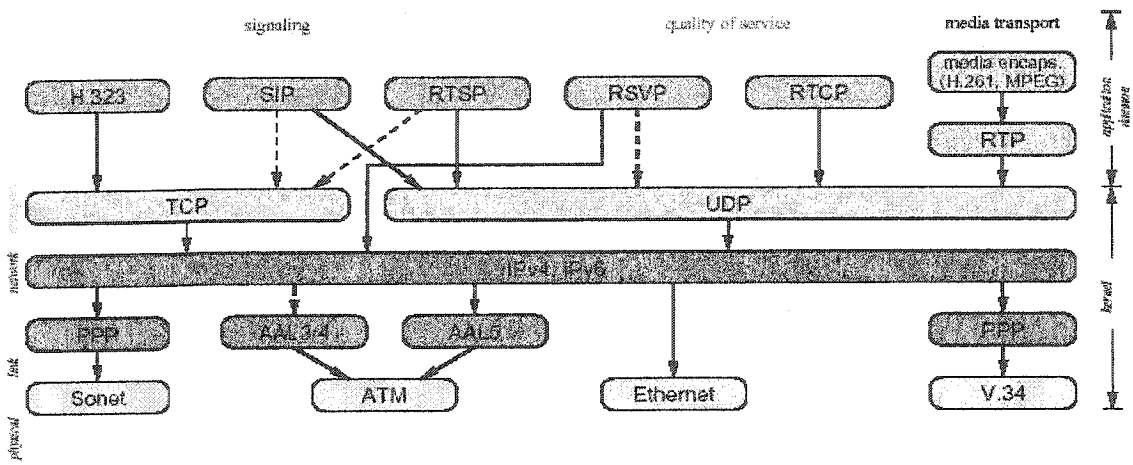

Figure 1.1 Internet Telephony Protocol Stack [36]

Differences between the Internet Telephony and the traditional circuit-switched telephony lie on signalling protocols but also on Telephony Features, on which we mainly concentrate. Besides making new features possible, the Internet Telephony also influences the deployment and maintenance of features. As we discussed in Section 1.1, the location of features can be more distributed and end users are granted more power to create and maintain their own features. Although the idea of separating the service from the server was firstly proposed by the Intelligent Network (IN) (see [21], [23]), only the appearance of the Internet Telephony gives not-expert users real power of programming their personal phone features. The Call Processing Language (CPL) has been designed for this task.

1.3 Overview of the Call Processing Language

The Call Processing Language (CPL) is an XML based language that can be used to describe and control Internet Telephony services. It is designed to be implementable on either network servers or user agent servers [28]. In order to avoid potential serious errors in highly distributed systems such as the Internet, CPL was designed in a way that it is Not-Turing complete; therefore, loops and recursions do not exist in CPL scripts.

From the view of CPL, the Internet Telephony network generally consists of two types of components: end systems or signalling servers. "End systems are devices which originate and /or receive signalling information and media. These include simple and complex telephone devices. Signalling servers are devices which relay or control signalling information. In SIP, they are proxy servers, redirect servers, or registrars; in H.323, they are gatekeepers" [29]. Users' features that are developed by CPL scripts may be located on signalling servers.

Figure 1.2 depicts how these components work together in a CPL call process:

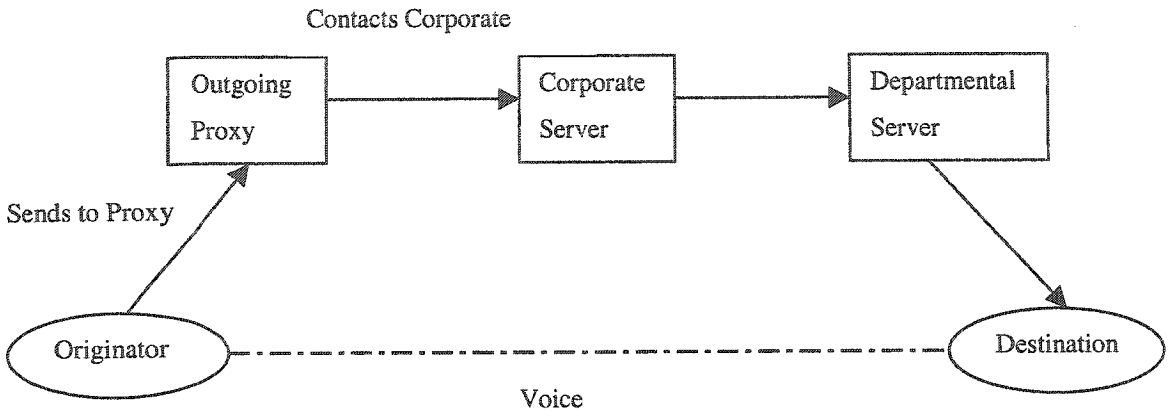

Figure 1.2 A CPL call set-up process (originally from [29])

In this figure, the originator could have outgoing features in the "outgoing proxy", and the destination could have incoming features on both the corporate server and the department server [29].

Figure 1.3 demonstrates a graphical representation of a CPL action:

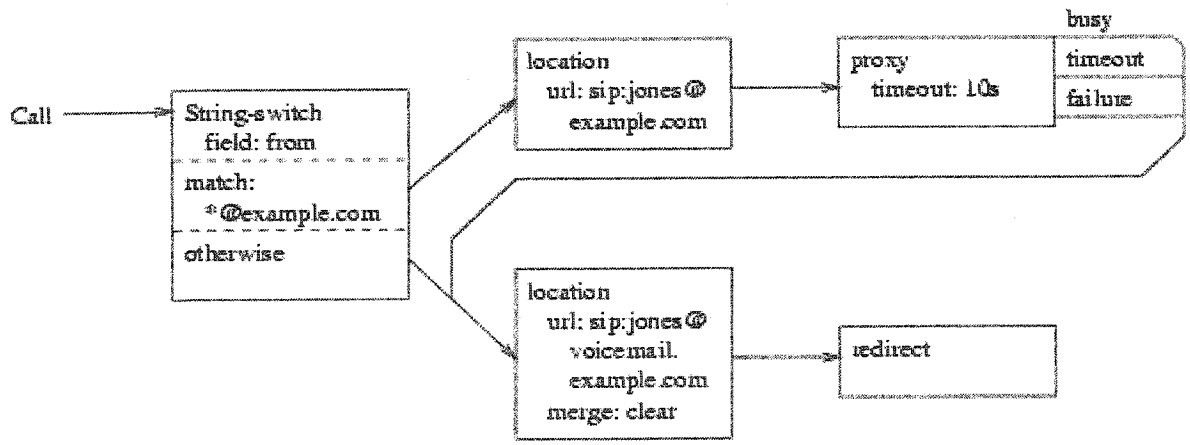

Figure 1.3 Sample CPL Action: Graphical Version [28]

In Figure 1.3, if the incoming call is from "example.com", this call will be forwarded to jones@example.com; otherwise or if the forwarding fails, the incoming call will be redirected to a voicemail, jones@voicemail.example.com. We will discuss the structure of CPL further and give examples of CPL scripts in Chapter 3.

1.4 Goal of this thesis

This thesis seeks to provide a pramatic method to detect potential Feature Interactions in CPL scripts, which are seen as logical inconsistencies in single and pair-wise scripts. As discussed in Section 8.2, our method consists of the following steps:

- CPL scripts are translated into a logic-based language
- Feature Interaction detection rules are identified and justified on the basis of logical proofs
- An automatic detection tool to perform Feature Interaction detection based on these rules has been developed.

Compared to other related work such as [31], which describes a method and a tool to check semantic corrections of CPL scripts (see Section 2.2.3 and 8.3.1), our work is more concerned with logically unsatisfiable situations; as a consequence, our method detects several interactions that are not addressed in [31].

1.5 Organisation of this thesis

This thesis consists of one chapter of introduction, six main chapters and a conclusion. A brief description of each chapter is presented below:

Chapter 1 Introduction

Chapter 1 presents the introduction of this thesis. It gives an overview of the architecture of Internet Telephony; introduces CPL and its network components.

Chapter 2 Feature Interactions and related approaches

Chapter 2 studies the issue of Feature Interactions. Starting with the definition of Feature Interaction, chapter 2 discusses the categories and origins of FIs. Related existing approaches are presented as well.

Chapter 3 Abstracting Logic-based Specifications from CPL Scripts

Chapter 3 analyses the structure of CPL scripts and proposes a logic-based language SFSL. The method of translating CPL scripts into SFSL, examples of specifying features in CPL, and examples of translation are presented too.

Chapter 4 Detecting Local Inconsistency in a Single CPL Script

Chapter 4 presents origins and categories of local inconsistency in single CPL scripts, methods of detecting these inconsistencies are proposed as well.

Chapter 5 Identification of Feature Interactions in pairs of CPL scripts

Chapter 5 studies the types of Feature Interactions in pairs of CPL scripts. We provide two basic principles and five concrete rules to detect potential Feature Interactions between two CPL scripts.

Chapter 6 Logic Proofs of Detection Rules

Chapter 6 proves the logical incoherence behind the Feature Interactions detection rules proposed in Chapter 4 and Chapter 5 using Predicate Logic. It also can be considered as a step towards validating these rules.

Chapter 7 Implementation of Automatic detection of FIs in CPL

Chapter 7 introduces an automatic detection tool. We choose Swi-prolog as the implementation language and present the method of building a translator from CPL to SFSL and designing a filter to detect FIs.

Chapter 8 Conclusions and Future work

Chapter 8 concludes this thesis. It reviews the contributions and discusses the possible directions for future research.

Chapter 2 Feature Interactions and Related Issues

This chapter gives an overview of the Feature Interaction problem and various approaches to solve it.

2.1 What is Feature Interaction

The phenomenon of Feature Interaction was originally identified in telephony service management and has attracted the attention of researchers since the emergence of Intelligent Networks [21]; it also exists in other domains such as computer-aided design [35].

2.1.1 What is a Feature

A feature is often considered as an incremental functionality to the core part of a telephony system. However, the concept of feature is not limited to the scope of telephony systems and can be extended to software systems as components of additional functionalities.

As we might see here, features play very similar roles as services. According to the ITU, a service is offered by an administration to its customers in order to satisfy a specific Telecommunication requirement [22] while a feature is the smallest part of a service that can be perceived by the service user [21]. The common understanding of the relationship between feature and service is that services consist of features [1]. The Internet Telephony provides users more freedom and power to create their personalised services and the distinction between these services and traditional features is not significant. In this thesis, both "feature" and "service" refer to the additional functionality that users intend to add; more details will be discussed in Chapter 5. As well, the concept of service interaction [27] and intention interactions (see Chapter 5) become undistinguishable with the concept of feature interaction.

2.1.2 Origins and Categories of Feature Interactions

Features are usually developed and tested in isolation or in a specific environment; when several features are combined together, a feature or features may modify or influence another feature in defining overall system behaviour or phenomenon, which is called Feature Interaction [40]. Feature Interactions are understood to be all interactions that interfere with the desired operation of features and that occur between a feature and its environment, including other features or other instances of the same feature [7]. Feature interaction is necessary and inevitable in a feature-oriented specification, because so little can be accomplished by features that are completely independent [40].

Research in this area divides Feature Interactions roughly into two types, "co-operative" interactions and "interfering" interactions [17], or "good" interactions and "bad" interactions [40]. A "good" or "co-operative" interaction describes a situation when features interact together desirably without causing problems while a "bad" or "interfering" one harms the system and results in undesired behaviour from the user's or the system's point of view [17].

A different classification divides Feature Interactions into SUSC (Single-User-Single-Component), SUMC (Single-User-Multiple-Component), MUSC (Multiple-User-Single-Component), MUMC (Multiple-User-Multiple-Component) and CUSY (CUstomer-SYstem) [7].

In terms of Feature Interactions in CPL, although a real user may have more than one email address in different organisations and more than one CPL script located in different servers, only one CPL script will be active at one time; therefore, the SUMC category of feature interactions is excluded in our work. The CUSY feature interaction refers to interactions between a customer feature and any system feature for operations, administrative services, or maintenance [7]; since the analysis of CUSY interactions would require taking into consideration elements that are outside of CPL, this type of interactions will not be studied in this thesis.

Therefore, with consideration of the distributed character of the deployment of CPL scripts, Feature Interactions in CPL are mostly confined in SUSC, MUSC and MUMC, more details will be discussed in Chapter 4 and Chapter 5.

2.2 Existing Approaches

After years of exploration (Feature Interaction Workshop, see [39] for the most recent one), it is believed that it is not feasible to resolve all possible feature interactions at any single stage of a feature lifecycle or with any single technique [25].

Generally, the FI problems can be approached from three different angles: detection, avoidance, and resolution [8]. Another point is that there are three major research trends: software engineering approaches, formal methods, and online technique [6]. Software engineering and formal method approaches are also sometimes called off-line techniques. Off-line means that the approach is applied during design-time of features, in contrast to on-line approaches that are applied while the features are actually running [6].

Since our work mostly concentrates on Feature Interactions detection, this section only reviews work closely related to ours, which applies formal methods to detect FIs.

2.2.1 Linear Temporal Logic & Model Checking

Linear Temporal Logic (abbreviated as LTL) "define(s) sets of infinite sequences; hence, the logic is particularly well suited to describe time dependent properties of concurrent, reactive systems such as telephony and other network protocols" [13]. Model Checking [9] "is a method for formally verifying systems using temporal logic. The idea of temporal logic is that a formula is not statically true or false in a model, as it is in propositional and predicate logic. Instead, the models of temporal logic contain several states and a formula can be true in some states and false in others. Thus, the static notion of truth is replaced by a dynamic one, in which the formulas may change their truth values as the system evolves from state to state" [19].

A. Felty and K. S. Namjoshi proposed a method of automatically detecting FIs at the specification stage after specifying features in LTL. It is based on a logical view of FI that two features conflict essentially if their specifications are mutually inconsistent under axioms about the underlying system behaviour [13]. Features are specified in linear temporal logic and an existing model checking tool is applied to detect the inconsistency.

Our approach is similar in the sense that we abstracted features into logic-based specifications although we chose Predicate Logic. We adopted the same idea that Feature Interactions correspond to logical inconsistency among features. Moreover, we applied this idea to validate the correctness of our detection rules.

2.2.2 Feature Interaction Analysis Tool (FIAT)

Nicolas Gorse developed a method of automating the detection of incoherences among telephony features on the basis of logical descriptions of these features. He defined a formal notation for describing features with information from requirements and also provided detection rules based on few principles. Through this, incoherences corresponding to potential pair-wise feature interactions at the requirements stage [17] can be identified, which will help designers refine the requirements and produce a better specification [17]. An automatic tool for feature interaction detection, called FIAT, was implemented too.

In general, Gorse decomposed a feature into four properties: pre-conditions, triggering events, results and constraints, and considered feature interactions as specific incoherences between their set of properties. These incoherences are identified by detection rules. For instance, one detection rule is that a user X could not be busy and idle at the same time: contradiction_pair(busy(X), idle(X)).

Although Gorse's work aims at detecting FIs in traditional Telecom systems and some of his detection rules do not work in the area of CPL, it inspired our research in many ways. For instance, during our process of deriving the detection rules, we adopted the same method of identifying contradicting results first, then backtracking and collecting the

possible preconditions. Another important influence is the way of implementing detection rules -- we chose the same programming language, SWI-Prolog.

2.2.3 Detecting Script-to-Script Interactions in CPL

The motivation of M. Nakamura, P. Leelaprute, K. Matsumoto and T. Kikuno's work is very much the same as ours. Their approach first defines eight types of semantic warnings which may occur in individual CPL scripts, then extends these warning to multiple CPL scripts by combining these scripts together. The key idea is to define feature interactions as the semantic warnings over multiple CPL scripts [31] while each of the CPL scripts is individually semantically safe.

Corresponding tools were also implemented such as a CPL checker and a FI simulator; the former is for detecting the proposed semantics warnings and the latter is for simulating the execution of CPL scripts.

We will discuss their work further in the conclusion chapter where we will make a detailed comparison between their approach and ours.

2.2.4 Modelling functionality as connection equations

Many approaches for detecting FIs are based on modelling the functionality of features. M. Kolberg and E. Magill proposed to model the service functionality as connection equations and then apply detection rules to pairs of service specifications to find interaction prone call scenarios. This service functionality model concentrates on call control aspects, more specifically, the originally intended connection and the finally established connection after triggering the service [27]. "This model assumes call control services to be extensions of the basic call model". "That leads to the definition of service interactions as problems between different extensions (services) to the basic call model. As a consequence, only the pure service functionality is modelled, i.e. without the basic call. In addition, six detection rules were given straightforwardly. These rules are tested by careful selection of the case study services covering both sides of the call and span

across all major call control functions" [27]; however, correctness and coverage of these rules need to be discussed further.

2.3 In Summary

In this chapter, we have presented our understanding of feature and feature interaction, and discussed the origin and classification of feature interactions as well. Moreover, we briefly reviewed several methods of detecting feature interactions using formal methods, which are closely related to our work.

Chapter 3 Abstracting Logic-based Specifications from CPL Scripts

This chapter addresses the method of deriving logical specifications from CPL scripts. With a description of CPL architecture, the Simple Formal Specifying Language (SFSL), which is well suited for expressing the essential information in CPL scripts, is introduced. In addition, we address how to translate CPL scripts into SFSL specifications. At the end of this chapter, examples of several typical telecom features in CPL are presented, as well as their SFSL translation.

3.1 Structure of CPL Scripts

"A CPL script runs in a signalling server, and controls that system's proxy, redirect, or rejection actions for the set-up of a particular call. It does not attempt to co-ordinate the behaviour of multiple signalling servers, or to describe features on a 'Global Functional Plane' as in the Intelligent Network architecture" [28].

3.1.1 Syntax of CPL: Overview

A CPL script consists of ancillary information, subactions, and top-level actions [28]. Figure 3.1 shows the syntax of top-level CPL tags:

```
Tag: "cpl"
  Parameters: None
  Sub-tags: "ancillary"
            "subaction"
            "outgoing"  Top-level action to take on this user's
                  outgoing calls
            "incoming"  Top-level action to take on this user's
                  incoming calls
```

Tag: "ancillary"

Parameters: None

Subtags: None

Tag: "subaction"

Subtags: Any node

Parameters: "id" Name of this subaction

Pseudo-node: "sub"

Outputs: None in XML tree

Parameters: "ref" Name of subaction to execute

Figure 3.1 Syntax of top-level CPL tag (originally from [28])

Ancillary tags have not been defined in CPL so far. Subaction is defined for "script re-use and modularity" [28], and an example of subaction will be given in section 3.4.5. There are only two types of top-level actions, and these are "incoming" and "outgoing". We will see in section 3.2.2 what the lower level actions can be.

Both "top-level actions and sub-actions consist of a tree of nodes and outputs, which are both described by XML tags". "There are four categories of CPL nodes: switches, which represent choices a CPL script can make; location modifiers, which add or remove locations from the location set; signalling operations, which cause signalling events in the underlying protocol; and non-signalling operations, which trigger behaviour which does not effect the underlying protocol" [28].

Figure 3.2 presents a common structure of CPL script:

```
<cpl>
    <incoming>
        .....
        <address-switch....>
```

```
            ......
                <reject.../>
            </address-switch>
        </incoming>
        <outgoing>
            ......
            <time-switch....>
                ......
                    <proxy/>
            </time-switch>
        </outgoing>
    </cpl>
```

Figure 3.2 An example of CPL script's structure

A CPL script can be considered as a decision-tree, where a set of conditions constitutes a branch and a single action constitutes a leaf at the end of each branch as shown in Figure 3.3:

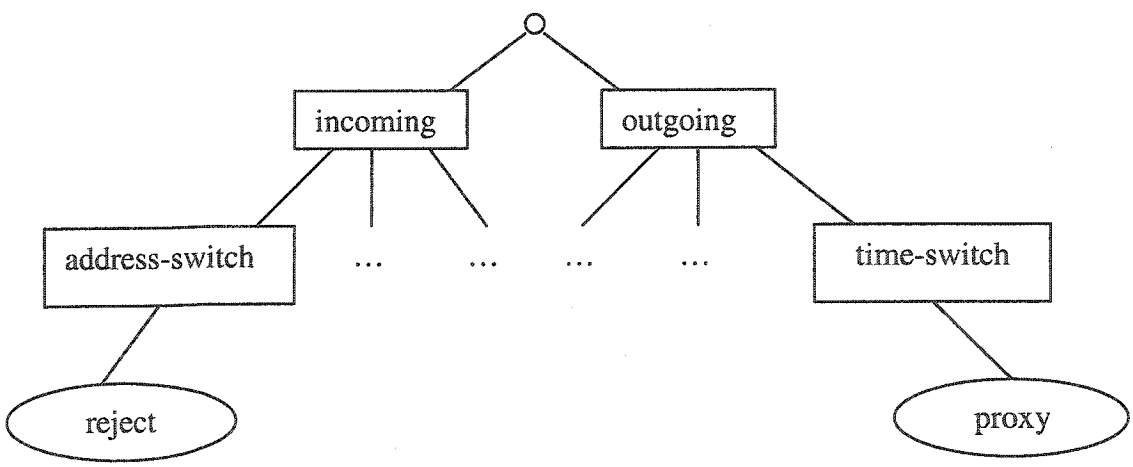

Figure 3.3 Decision-tree of CPL scripts in Figure 3.2

In Figure 3.3, ellipses represent actions (leaves) and rectangles stand for conditions (part of branches). "reject" and "proxy" are the only two leaves shown in this tree, whose

corresponding branches are "incoming→address-switch" and "outgoing→time-switch" respectively.

Note that although CPL only has two top-level conditions ("incoming" and "outgoing") and the switch conditions such as address-switch and time-switch only determine "match" or "not match", this does not mean that a decision tree has to be a binary tree since more than two switch conditions may be present at the same level after incoming or outgoing (as shown in Figure 3.3) and the outcome of some actions (they also belong to conditions in CPL, see Table 3.1) may have more than two possibilities.

A very simple CPL script is shown in Figure 3.4:

```
<?xml version="1.0" ?>
<!DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx CPL 1.0//EN" "cpl.dtd">
<cpl>
    <incoming>
        <address-switch field="origin" subfield="user">
            <address is="anonymous">
                <reject status="reject" reason="I don't accept anonymous calls" />
            </address>
        </address-switch>
    </incoming>
</cpl>
```

Figure 3.4 Example script: incoming call screening [28]

In Figure 3.4, the first two lines indicate the corresponding XML and RFC version, which is additional information and won't influence the call process. "address-switch" (see section 3.2.2.2) is used to filter incoming calls that are from anonymous originators, and these calls will be rejected by the action "reject" (see section 3.2.2.2).

3.1.2 Condition and Action

From the example in Figure 3.2, we may conclude that **Condition** and **Action** are the two types of lexicalities in CPL: **Conditions** are used to determine the executing path whereas **Actions** indicate what will be executed or the results that the path will lead to. Among those actions, "incoming" and "outgoing" are two top-level ones that make up branches. Both of them are optional, which means a CPL script does not have to possess both "incoming" and "outgoing" policies if unnecessary.

In Figure 3.2 "reject" and "proxy" are the only two actions. They will be executed only if their corresponding sets of conditions are true. In this case, we say that the action is **executable** and that the leaf is **reachable**. For action "reject", the corresponding set of conditions (its precondition) is that there is an incoming call and the address attribute of this call matches the strategy set by the address-switch. For action "proxy", the set of conditions is that there is an outgoing call and the time attribute of this call matches the strategy set by the time-switch.

3.1.3 Policy and Intention

Just as an Intelligent Network Conceptual Model is described in four planes [21], we may consider a CPL script at different levels. In a general view, an entire CPL script represents the overall **policy** of a user, which contains two sub-policies: incoming and outgoing. On the other hand, from the point of view of function, a CPL script implements the user's concrete **intentions** that indicate what will be done under different situations. Therefore, with respect to the structure of CPL scripts, we say that an entire CPL script stands for a user's whole **policy**, which can be seen as a decision tree as well; meanwhile, we say that an action and its corresponding conditions, which is a leaf and its associated branch in a CPL script, represents one **intention** of a user.

Just as one tree may contain several branches, one policy may include one or more intentions. For the example in Figure 3.2, the incoming policy contains one intention that indicates that all the incoming calls will be rejected if its address meets the condition

denoted by the address-switch; the outgoing policy contains one intention as well, which indicates that all the outgoing calls will be transferred during the period of time defined by the time-switch.

3.1.4 Intention: specifying Features in the Internet Telephony

As mentioned in Chapter 2, a feature is a functionality offered by a system and has the purpose of fulfilling certain user intentions in the context of a call [17]. Internet telephony can provide more sophisticated telephony-like services and features than traditional telephony. As well, it offers users the opportunity of programming and deploying their own services, which authorises users to design their personal telecom services and dramatically reduces the differences between telecom features and personal intentions. Therefore, one of our main assumptions is that users' intentions specify features in Internet telephony. This is usually true at the level of service. For lower level designs that focus on functionality, other approaches may be more appropriate.

For instance, one traditional feature Call Forward on Busy can be specified in terms of user intentions in this way: If I am busy, then transfer all the incoming calls to a colleague. Feature Outgoing Call Screening can be specified as: If a user tries to call that number from my place, then block this call.

3.2 Simple Formal Specifying Language (SFSL): A logic-based language for abstracting CPL scripts

CPL is designed to be easily used and understood; however, it also has some disadvantages such as redundancy, or ambiguities that may lead to logical incoherence within a single CPL script. Moreover, CPL fails to offer a mechanism to prevent potential Feature Interactions in pairs of CPL scripts. Chapter 4 will demonstrate the first type of problems and possible solutions; the second ones will be discussed in Chapter 5.

Our approach is based on logic analysis, for which a logic-based specification of CPL is more appropriate than the XML-based. This target is achieved by defining the Simple

Formal Specification Language (SFSL) that is suitable for representing CPL scripts and developing translation rules from CPL to SFSL. The current section defines the syntax and semantics of SFSL and the following one, section 3.3, addresses the method of translating CPL scripts into SFSL specifications.

3.2.1 Defining the syntax of SFSL

Like many other languages, SFSL is "defined by means of a formal syntactic description and an informal semantic description" [34] as well.

In brief, the syntax of SFSL is defined to describe CPL scripts in the format of:

Condition1 $\wedge$ condition2 $\wedge$ ... $\rightarrow$ action.

On the left of symbol "$\rightarrow$" is the set of enabling conditions while on the right is the result of implication.

The syntax of SFSL is defined in BNF (Backus-Naur Form) as follows:

<program>	::= <predict>
<predict>	::= <conditions>"→"<action>
	| <predict>; <conditions>"→"<action>
<conditions>	::= <incoming(<variable>, <variable>) > {"/\" <_condition>}
	| <outgoing(<variable>, <variable>)> {"/\" <_condition}>
<_condition>	::= <condition>
	| ¬<condition>
<condition>	::= <address-switch(<variable> <operation> <variable>)
	| <time-switch("tzid="<variable>, "tzurl="<variable>
	{, "dtstart="<time>, "duration="<variable>})>
	| <language-switch("language="<variable>
	| <string-switch(<variable> <operation> <variable>)>
	| <priority-switch("priority" <operation> <variable>)>
	| outcome(<action>, <variable>)
	| lookup("source="<variable> {, "timeout="<variable>,
	"use="<variable>, "ignore="<variable>, "clear="<variable>}) "="
	<variable> <action>::= redirect(<addressset>, <addressset>)
	| proxy(<addressset>, <addressset>{, <time>, <variable>, <variable>})
	| reject(<addressset>, <addressset>)
<operation>	::= = | ⊇ | > | < | ⊂
<time>	::= <digit> {<letter> | <digit>}
<addressset>	::= <variable>
	| "{" <variable>{, <variable>} "}"
<variable>	::= <letter> {<letter> | <digit>}
<digit>	::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter>	::= a | b | c |...| x | y | z | A | B | C|...| X | Y | Z | . | @

Table 3.1 Syntax of SFSL

Note that this syntax is simplified. For instance, although we define that the second parameter of address-switch is <operation>, not all kinds of operations defined above are

suitable. In practice, address-switch only accommodates "=", " ⊇" and "⊂". More details are discussed in the following section 3.2.2.

3.2.2 Defining the Semantics of SFSL

3.2.2.1 Overview

The semantics of SFSL are derived from CPL, but we make them more formal and precise by adding parameters and removing ancillary information. For instance, we use proxy(x, y) to represent the action "proxy" whose format in CPL is <proxy/>. Here two parameters are added, the first one, "x", stands for the originator of the call and the second one, "y", represents the destination. In SFSL, we use single lower-case letters such as "x", "y" to represent unbound variables and strings starting with capital letters to specify constants such as "Alice@uottawa.ca".

As shown in Table 3.1, <conditions> consists of two types of labels (incoming and outgoing), five types of switches (address, time, language, string and priority) and the outcomes of actions. Although there are six types of actions in CPL, we only need to take into account the three signalling ones since only signalling actions can influence real-time call processes. These signalling actions are proxy, reject and redirect.

Both conditions and actions may contain parameters to represent originators and destinations in call processes. In this case, we stipulate the order of these two parameters that is always caller first and callee second. For instance, reject (x, y) represents that the call from x to y is rejected where x is the caller and y is the callee.

3.2.2.2 Labels, Switches, and Signalling Actions

Specific formal notation for SFSL include:

- 2 types of labels:

incoming(x, A): indicates that this is the incoming policy for user A and it influences all the incoming calls from x to A, where "x" means any possible user.

outgoing(A, x): indicates that this is the outgoing policy for user A and it influences all the outgoing calls from A to x, where "x" means any possible user.

- 5 kinds of switches:

Note that switches in CPL define conditions that have to be satisfied for certain actions to be executed.

address-switch:

address-switch(user.field.subfield = "Z") : indicates that the switch type is address-switch; "user" represents the host address which contains "field" and "subfield"; "field" and "subfield" represent two attributes attached to address-switch. In CPL, the possible values for "field" are "origin", "original-destination" and "destination", and those for "subfield" are "address-type", "user", "display", "host", "port" and "tel"; "=" represents "is", which is a possible operator in CPL and means exact match with the operand "Z". The other two possible operators are "contains" and "subdomain-of", we use "⊇" and "⊃" in SFSL to represent them respectively. As an example, the following CPL script segment for "sip:Alice@uottawa.ca"

```
.....
<outgoing>
        <address-switch field="destination" subfield="tel">
                <address subdomain-of="1866">
                        .....
                </address>
        </address-switch>
</outgoing>
.....
```

is translated into SFSL as:

outgoing("sip:Alice@uottawa.ca", x) /\ address-switch(x.destination.tel ⊃ "1866")

where x stands for the destination of outgoing calls which could be any user.

string-switch:

string-switch(user.field = "Z"): indicates that the switch type is string-switch; "user" represents the host address containing "field"; "field" is the attribute attached to string-switch and in CPL it can have four possible values: "object", "user-agent", "organization" and "display"; "=" represents the operator "is", which means an exact match to the operand "Z". The other operators for string-switch in CPL is "contains", represented by "⊇" in SFSL. For instance, the following segment of CPL script, which belongs to "sip:Alice@uottawa.ca"

```
.....
<incoming>
        <string-switch field="organization">
                <string is="uottawa.ca">
                        .....
                </string>
        </string-switch>
</incoming>
.....
```

is represented in SFSL as

incoming(x, "sip:Alice@uottawa.ca") /\ string-switch(x.organization="uottawa.ca")

where x stands for the originator of all incoming calls which could be any user.

time-switch:

time-switch(tzid="...", tzurl="...", dtstart="20020812T090000", duration="PT8H", freq="weekly"): indicates that the switch type is time switch, the most complicated switch in terms of format. "tzid" represents the time zone identifier, "tzurl" represents the time zone URL, these two parameters are mandatory. There are also 17

optional parameters and here we list three of them: "distart" indicates the start interval, "duration" sets the length of the interval and "freq" means the frequency of recurrence. As an example, the following CPL script segment

```
.....
<time-switch tzid="EST" tzurl="http://zones.america.est">
        <time dtstart="20020812T090000" duration="PT8H">
                .....
        </time>
</time-switch>
.....
```

is specified in SFSL as:

```
time-switch(tzid="EST", tzurl="http://zones.america.est", dtstart="20020812T090000",
duration="PT8H").
```

language-switch:

language-switch(language="es"): indicates that the switch type is language switch. We use "=" in SFSL to represent the operator "match", the only operator that language-switch has in CPL. "es" in this example is the operand of "match" and it means that the language type is English.

priority-switch:

priority-switch(priority = "urgent"): indicates that the switch type is priority switch. We use "=" in SFSL to represent the operator "equal" in CPL, which is one of the three possible operators in priority-switch, the other two are "less" and "greater", represented by "<" and ">" in SFSL respectively. "urgent" in this example is the operand of "equal" and other possible operands could be "emergency", "normal" and "non-urgent".

Note that in CPL two special switch outcomes apply to every switch type [28]. One is "not-present" which describes the case where "the variable the switch was to match was not present in the original call setup request". We discard this output because without the necessary variables attached to switches logic incoherence or Feature Interactions cannot exist either. The other switch output is "otherwise" which "MUST be the last output specified if it is present, matches if no other condition matched" [28]. SFSL treats "otherwise" as "¬(switch-condition)" where "switch-condition" is determined by its attached switches. For instance, the associated conditions for "action1" in the following CPL script segment

```
.....
<outgoing>
        <address-switch field="destination" subfield="tel">
                <address subdomain-of="1866">
                        .....
                </address>
                <otherwise>
                        <action1>
                </otherwise>
        </address-switch>
</outgoing>
.....
```

is translated in SFSL as:

outgoing("sip:Alice@uottawa.ca", x) ∧ ¬address-switch(x.destination.tel ⊃ "1866")

where x stands for the destination of outgoing calls which could be any user.

- 3 types of signalling actions:

proxy(x, destination-address, timeout, recurse, ordering): indicates that the call from x will be forwarded to the user represented by destination-address; as well, x means the

caller could be any user. "timeout" sets the available time; recurse can be set as "yes" or "no" to "specify whether the server should automatically attempt to place further call attempts to telephony addresses in redirection responses that were returned from the initial server" [28]. And, the value of "ordering" can be "parallel" or "sequential". The first two parameters (x and destination-address) are required and the last three (timeout, recurse, ordering) are optional.

reject(caller, callee): indicates that the specific call from caller to callee should be rejected.

redirect(x, destination-address): indicates that the call from user x is redirected to destination-address.

"redirect" causes the server to direct the calling party to attempt to place its call to the currently specified set of locations [28] and "proxy" causes the triggering call to be forwarded on to the currently specified set of locations [28]. Obviously, if "redirect" is used, the calling party will transfer his call by himself while if "proxy" is used, the called party will do the transfer and the calling party won't even know that his call is going to be transferred.

- 2 particular switches:

lookup(source="...", timeout="...", use="... ", ignore="...", clear="...") ="success" indicates that the switch type is lookup; "source" is the only mandatory parameter which represents the source to be checked; we use "timeout" to set the trying time before giving up; "use" and "ignore" represent caller preferences fields to use and to ignore respectively; "clear" represents whether to clear the location set before adding the new values; "success" is one of three possible outputs of lookup, the other two are "notfound" and "failure". As an example, the following CPL script segment

```
.....
<lookup source="http://groups.yahoo.com/login?user=Richard" timeout="9">
```

```
        <success>
                .....
        </success>
</lookup>
.....
```

can be translated in SFSL as:

```
lookup(source=http://groups.yahoo.com/login?user=Richard, timeout="9")="success"
```

outcome(proxy(x, y), "busy"): indicates that the outcome of action proxy(x, y) is busy. As discussed above, proxy(x, y) specifies the action of transferring the call from originator x to destination y; "busy" is one possible output, others could be "noanswer", "timeout", "redirection" or "failure".

3.3 Method of translating CPL scripts into SFSL

Section 3.1.3 states that CPL intentions have the format of condition1 $\wedge$ condition2 $\wedge$ ... $\rightarrow$ action, which is what SFSL is designed to support. Hence, translating a CPL script into SFSL involves splitting a decision-tree into branches as shown in Figure 3.4, since a CPL script may contain one or more intentions and each intention can be considered as a branch in the CPL tree.

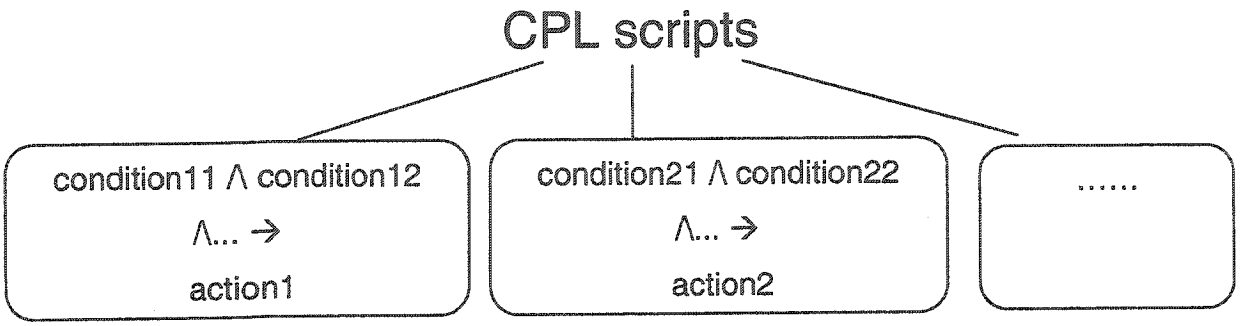

Figure 3.4 CPL scripts can be divided into several independent branches

From Figure 3.4, we see that each intention contains one and only one action, and conditions do not exist independently; on the contrary, they are always attached to actions. Consequently, the first step of translating CPL into SFSL is to recognise actions, which are leaves in a decision tree.

3.3.1 Identifying and translating actions

As mentioned in 3.2.2.1, there are six types of actions in CPL but we only take into account the three signalling ones. Since the three non-signalling actions do not influence the call processes, they and the attached conditions are ignored during the translation process.

For signalling action "proxy", there is always a URL location present before "proxy" to indicate the destination of this action (Otherwise the destination is the CPL script subscriber himself). In this case, we simply use proxy(x, url-location) to represent the following three sentences in CPL, where "x" denotes all the incoming calls which could be from any user and "url-location" denotes the forwarding destination:

```
<location url="...">
    <proxy />
</location>
```

If there are operational parameters such as "ordering" associated with proxy, they are always present right behind proxy and are easy to identify.

For signaling action "reject", if it is attached to "address-switch" and "address", then we can use reject(x, owner) or reject(owner, x) to replace the "reject" sentence in CPL where "owner" stands for the subscriber of the current CPL script. Since the semantic of reject is reject(caller, callee), if it is for incoming policy, then we use reject(x, owner); if it is for outgoing policy, then we use reject(owner, x). "x" is the attempting user whose attributes satisfy the conditions in "address-switch" and "address".

For instance, assume that the following CPL script is located in Alice's incoming policy:

```
<address-switch field="..." subfield="...">
  <address is="...">
    <reject status="..." reason="..." />
  </address>
</address-switch>
```

We can use reject(x, Alice) to represent the sentence <reject status="..." reason="..." />, where x represents any user whose address attributes match the address conditions; the ancillary information such as status and reason is ignored in order to simplify the target specification.

For action "redirect", there should also be a url location before the "redirect" in the CPL script to indicate the destination of the redirection. Similarly to action "proxy", we can use redirect(x, url-location) to represent these three sentences in CPL:

```
<location url="...">
        <redirect />
</location>
```

CPL also has a specific notation "subaction" that is defined for script re-use and modularity and can be considered as an action as well. It acts like a sub-function, which consists of conditions and actions also. The notation of subaction can be replaced with its source code while the functionality remains the same. Therefore, by convention, we

assume that all subactions are replaced before CPL scripts are translated. An example with subaction is discussed in Section 3.4.5.

3.3.2 Translating associated conditions

The second step of translating CPL to SFSL is to select all the associated conditions of the identified action and concatenate them in a conjunctive formula. This requires tracing from the position of that action back to the decision-tree root to include all preconditions. These preconditions may consist of seven types of switches and two types of labels, as described in section 3.2.2.2.

Address-switches are always followed by the content of "subfield", address. Thus, the following CPL sentences can be specified in the form of: address-switch(user.field.subfield operator operand):

```
<address-switch field="..." subfield="...">
        <address is="...">
                .....
        </address-switch>
    </address is="...">
```

Here, operator should be "=" to represent "is" in the address sentence, and the operand is the quoted content after "is=". The "user", which presents the host address of "field" and "subfield", can be represented by variable x in case the user is determined only at running time.

For the other six types of switches and the two labels, since their structures are quite simple, we can translate them directly with the same semantics. Therefore we will not discuss them in detail.

3.4 Examples of CPL scripts and their Translation

After the introduction of SFSL and of its translating method, this section presents examples of how to develop typical telecom features in CPL and how to translate these features into SFSL as well.

3.4.1 Outgoing Call Screening in CPL and its Translation

The first example presents the case where the CPL script of Alice@uottawa.ca behaves like OCS with Carl@phone.example.com in the screening list:

```
<?xml version="1.0" ?>
 <!DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx CPL 1.0//EN" "cpl.dtd">

 <cpl>
  <outgoing>
   <address-switch field="original-destination" subfield="user">
    <address is=" sip:Carl@phone.example.com ">
     <reject status="reject"
          reason="Not allowed to make a call to Carl." />
    </address>
   </address-switch>
  </outgoing>
 </cpl>
```

For translation, we first identify the action, which is "reject" in this case, and then combine it with its enable conditions. Finally, the CPL script can be translated as:

```
Outgoing("sip:Alice@uottawa.ca", x)
/\ address-switch(x.original-destination.user= "sip:Carl@phone.example.com")
 →
reject("sip:Alice@uottawa.ca", x)
```

3.4.2 Call Forward Always in CPL and its Translation

The second example describes a situation where the CPL script of Bob@uottawa.ca behaves like CFA, with all the incoming calls to be forwarded to Carl@phone.example.com:

```
<?xml version="1.0" ?>
  <!DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx CPL 1.0//EN" "cpl.dtd">

  <cpl>
   <incoming>
    <location url="sip:Carl@phone.example.com">
     <proxy />
    </location>
   </incoming>
  </cpl>
```

For translation, we first identify the action, which is "proxy" in this case, and then combine it with its enable conditions. Finally, the CPL script can be translated as:

Incoming(x, "sip:Bob@uottawa.ca") → proxy (x, "sip:Carl@phone.example.com")

3.4.3 Incoming Call Screening in CPL and its translation

The third example describe a case where the CPL script of Alice@uottawa.ca behaves like ICS with Carl@phone.example.com in the screening list:

```
<?xml version="1.0" ?>
  <!DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx CPL 1.0//EN" "cpl.dtd">

  <cpl>
   <incoming>
    <address-switch field="origin" subfield="user">
```

```
    <address is=" sip:Carl@phone.example.com ">
      <reject status="reject"
           reason=" I do not accept Carl's call." />
    </address>
   </address-switch>
  </incoming>
 </cpl>
```

For translation, we first identify the action, which is "reject" in this case, and then combine it with its enable conditions. Finally, the CPL script can be translated as:

```
Incoming(x, "sip:Alice@uottawa.ca")
/\ address-switch(x.origin.user = "sip:Carl@phone.example.com")
→
 reject(x, "sip:Alice@uottawa.ca")
```

3.4.4 Call Forward on Busy in CPL and its Translation

The fourth example specifies the case where the CPL script of Alice@uottawa.ca behaves like CFBL. If she is busy, all the incoming calls will be forwarded to Carl@phone.example.com:

```
<?xml version="1.0" ?>
 <!DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx CPL 1.0//EN" "cpl.dtd">

 <cpl>
  <incoming>
   <proxy>
     <busy>
       <location url="sip:Carl@phone.example.com">
         <proxy />
       </location>
```

```
        </busy>
      </proxy>
    </incoming>
  </cpl>
```

For translation, we first identify the action, which is "proxy" in this case, and then combine it with its enable conditions. Finally, the CPL script can be translated as:

```
Incoming(x, "sip:Alice@uottawa.ca")
/\ outcome(proxy(x, "sip:Alice@uottawa.ca"), busy)
→
proxy(x, "sip:Carl@phone.example.com")
```

3.4.5 Subaction of Voicemail in CPL and its Translation

The fifth example shows how to use CPL to fulfil a voicemail as well as how to translate a CPL script with subactions. Suppose that the CPL script of Bob@site.uottawa.ca behaves like voicemail:

```
<?xml version="1.0" ?>
  <!DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx CPL 1.0//EN" "cpl.dtd">

  <cpl>
    <subaction id="voicemail">
      <location url="sip:Bob@voicemail.example.com">
        <redirect />
      </location>
    </subaction>

    <incoming>
      <sub ref="voicemail" />
    </incoming>
```

```
</cpl>
```

For translation, we first replace the sentence of <sub ref="voicemail"> with the source code inside subaction:

```
<incoming>
        <location url="sip:Bob@voicemail.example.com">
            <redirect />
    </location>
  </incoming>
```

Then pursue a normal translation: identify the action first, which is "redirect" in this case, and then combine it with its enable conditions. Finally, the CPL script can be translated as:

```
Incoming(x, "sip:Bob@site.uottawa.ca")
→
redirect(x, "sip:Bob@voicemail.example.com" )
```

3.4.6 Call Forking Outgoing in CPL and its Translation

The sixth example describes a policy of Call Forking Outgoing which indicates that the outgoing calls will be forwarded to several users simultaneously. Suppose that the CPL script of Alice@uottawa.ca behaves like Call Forking Outgoing:

```
<?xml version="1.0" ?>
  <!DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx CPL 1.0//EN" "cpl.dtd">

<cpl>
  <outgoing>
    <location url="sip:Bob@phone.example.com, sip:Bob@site.uottawa.ca">
      <proxy ordering="parallel">
    </location>
  </outgoing>
```

```
</cpl>
```

For translation, we first identify the action, which is "proxy" in this case, and then combine it with its preconditions. Finally, the CPL script can be translated as:

```
Outgoing("sip:Alice@uottawa.ca", x)
→
proxy({"sip:Bob@phone.example.com, sip:Bob@site.uottawa.ca"}, x, parallel)
```

Note that in this example, outgoing calls will be forwarded to two destinations (Bob@phone.example.com and Bob@site.uottawa.ca) in parallel.

3.5 In summary

This chapter proposes an analytic approach to abstract formal specifications in CPL scripts, which provides the possibility of detecting feature interactions and other logical incoherences in CPL automatically and efficiently as will be shown in the following chapters.

Section 3.1 presents the structure of CPL scripts while section 3.2 introduces a new language SFSL with its syntax and semantics. The method of translating CPL scripts to SFSL is proposed in Section 3.3. In Section 3.4, several traditional features are deployed in CPL and then translated into SFSL. These examples are simplified in the sense that each user's policy contains one intention only. In practice, the translation method also works in the case where one user's policy contains several intentions, as explained in Section 3.3.

We also discuss the relationship between policy and intention in section 3.1.2. Some details such as the relationship among policy, intention and feature will be discussed in the next chapter.

Chapter 4 Detecting Local Inconsistency in Single CPL Scripts

As discussed in Section 3.2, the Call Processing Language (CPL) is subject to several intrinsic weaknesses such as redundancy and ambiguity. This chapter addresses the possible effects of these disadvantages on a single CPL script. We name this problem Local Inconsistency. Possible solutions to this issue are presented as well.

4.1 Categories and Origin of Local Inconsistency in the Context of Single CPL Scripts

Chapter 3 explained the structure of CPL scripts and the relationship between features and user intentions (see Section 3.1.4). A CPL script is considered as a decision tree consisting of branches while each branch represents a feature or a user intention, which is an optional unit or increment of functionality [40]. These branches are independent in terms of format since each of them is complete and could be separated from the whole tree without violating others. However, like other software systems, this format independence does not prevent Feature Interactions, the logical conflicts among features. The characteristics of CPL removes some types of FIs caused by ambiguity such as the case where two features are triggered by the same event, but introduces a new one (see Section 4.3).

In addition, logical incoherence inside a feature also needs to be taken into account and we use the word "Feature Inconsistency" to specify this situation.

Since a CPL script contains one or more features (see Section 3.1.2 and 3.1.3), both Feature Interaction and Feature Inconsistency could exist in the context of a single CPL script. We will discuss them respectively in the following two sections.

4.2 Feature Inconsistency in CPL

Feature Inconsistency describes logical problems within a feature. Considering that features in CPL have the format of (condition1 /\ condition2 /\ ... → action), logical inconsistency only exists in the set of conditions (condition1 /\ condition2 /\ .../\ conditionN). This fact confines the scope of Feature Inconsistency and divides inconsistencies into two main types: Unexecutable Actions and Redundant Conditions.

4.2.1 Unexecutable Actions and Corresponding Solutions

In contrast to executable actions (See Section 3.1.1), the case of unexecutable actions describes the situation where an action cannot be executed because its associated conditions (preconditions) can never be satisfied. This can also be considered as the consequence of invalid paths in CPL scripts. Therefore, detecting unexecutable actions can be done by detecting contradictions in CPL branches. We divide the contradictions in one branch mainly into two types: direct contradictions and indirect contradictions:

Direct contradiction describes the situation that some conditions (normally two) in one path contradict directly. An example is

incoming(x, "sip:Alice@site.uottawa.ca")
/\ ¬address-switch(x.origin.user= "sip:Carl@phone.example.com")
/\... /\
address-switch(x.origin.user = "sip:Carl@phone.example.com")
→
reject(x, "sip:Alice@site.uottawa.ca").

The action "reject(x, 'sip:Alice@site.uottawa.ca')" will never be executed since no incoming calls could satisfy ¬address-switch(x.origin.user = "sip:Carl@phone.example.com") and address-switch(x.origin.user = "sip:Carl@phone.example.com") together.

Note that this type of Feature Inconsistency usually hides in a path containing "otherwise" since it is the only origin of negation in CPL.

To detect this type of Feature Inconsistency is quite simple: translate CPL scripts into SFSL specifications, and then check each feature's preconditions. If the Boolean value of a set of conditions (preconditions) is "False", then their action must be an unexecutable one.

Indirect contradiction describes the case where an action's preconditions conflict with system axioms, "which describe properties that should be true of any reasonable system implementations" [12]. For example, suppose that a feature is defined as follows:

....outcome(proxy(x, "sip:Carl@site.uottawa.ca"), "busy")
/\ outcome(proxy(x, "sip:Carl@site.uottawa.ca"), "noanswer")
/\ ....
→
redirect(x, "sip:Bob@uottawa.ca")

We can see that the action "redirect(x, 'sip:Bob@uottawa.ca')" can never be executed since a user cannot be at the status of "busy" and "noanswer" simultaneously.

To find this type of Feature Inconsistency, we may take two steps:

First of all, enumerate all system axioms that come from the assumptions and requirements of telecom systems. For example, ¬(outcome(proxy(x, y), "busy") /\ outcome(proxy(x, y), "noanswer")), where "x", "y" could be any user, is a system axiom that explains the fact that a user could not be busy and noanswer at the same time.

Secondly, combine each feature in SFSL specifications with these system axioms to check whether there is a contradiction. If yes, there must exist an unexecutable action.

Notice that the checking process is NP-complete and takes exponential computing time [11] [15]. A NP-complete problem is intractable so far where NP stands for Nondeterministic Polynomial [33]. For instance, if we have N proposition variables (say cnd1, cnd2,..., cndn), in order to check the consistence of this set of propositions cnd1 $\wedge$ cnd2 $\wedge$...$\wedge$ cndn (the same as detecting direct contradictions), we have to compare each pair of these propositions which takes 2^N time. For indirect contradictions detection, the computation is much more complex since permutation and combination is inevitable. Therefore, our proposed solution will encounter difficulties in dealing with large numbers of propositional variables like other existing solutions. It is unlikely, however, that such large numbers will be encountered in practice.

The detection methods discussed in this section were not implemented and are left for future work.

4.2.2 Redundant Conditions and Corresponding Solutions

We have redundant conditions in CPL if some conditions in a branch's preconditions are repeated. This is not good in terms of CPL scripts performance; however, compared to unexecutable actions, this does not generate serious consequences.

The logical connection in the set of conditions (preconditions) is quite simple -- individual conditions are linked together by connector "$\wedge$". Therefore, similar to unexecutable actions, redundant conditions mainly include two categories as well: conditions repeated directly or indirectly.

An example for the first type is:

incoming(x, "sip:Alice@site.uottawa.ca")
$\wedge$ address-switch(x.origin.user = "sip:Carl@uottawa.ca")
$\wedge$ ...
$\wedge$ address-switch(x.origin.user= "sip:Carl@uottawa.ca")
$\rightarrow$
proxy(x, "sip:Bob@uottawa.ca").

As we can see, the precondition unnecessarily contains two 'address-switch(x.origin.user= "sip:Carl@uottawa.ca")'.

An example for the second type is:

incoming(x, "sip:Alice@site.uottawa.ca")
/\ address-switch(x.origin.user= "sip:Carl@uottawa.ca")
/\ string-switch(x.organization="uottawa.ca")
→
proxy(x, "sip:Bob@uottawa.ca").

If x.origin.user in address-switch is "sip:Carl@uottawa.ca", the x.organization in string-switch must be "uottawa.ca".

Direct conditions repeat can be detected easily after CPL scripts are translated into SFSL specifications; for indirect condition repeat, theoretically, we can enumerate all possible conditions implied by existing ones and then detect redundancy. However, in practice, to enumerate all implied conditions is unfeasible. For instance, x.origin.user = "sip:Carl@site.uottawa.ca" also implies that x.origin.host = "site.uottawa.ca", x.origin.host ⊃ "uottawa.ca", x.origin.user ⊇ "Carl", x.origin.user ⊇ "Ca", x.origin.user ⊇ "uottawa", etc. This type of redundancy requires more study.

Another type of redundancy is discussed in [31]. It describes the case that

condition1 /\ condition2 → action1 and
condition1 /\ ¬condition2 → action1

are redundant since these two features reduce to one

condition1 → action1.

This redundancy is more like a "feature redundancy" than a "condition redundancy". Rule #L1 is used to identify this type of Feature Redundancy which might occur between two features in one CPL script.

Rule #L1

Feature Redundancy is present if the following characteristics hold:

(1) User A has two features that have the same action.

(2) A condition in the first feature's preconditions is the opposite of a condition in the second feature's preconditions; and the rest of these two preconditions are the same.

Formally:

(1) conditions1 /\ condition2 → action1

(2) conditions1 /\ ¬condition2→ action1

Notes:

1. action1 could be one of any possible CPL actions.
2. conditions1 is a set of enable-conditions, which cannot contain contradictions; condition2 is an individual condition.

Of the detection methods discussed in this section, only #L1 was implemented.

4.3 Feature Interaction in a Single CPL Script: Feature Shadowing

Concerning Feature Interactions within a CPL script, because the order of execution of features is already determined by CPL's mechanism, many traditional Feature Interaction problems are avoided. For instance, a subscriber in traditional telecom systems might unthinkingly request both "Call Forward Always" and "Call Forward on Busy", which leads to uncertainty under the condition of line busy. However, this FI does not occur in the context of CPL since the CPL script is only able to "trigger one action in response to the condition 'a call arrives while the line is busy' "[28], and only the feature that is encountered first will be executed undoubtedly. This predetermined execution order,

unfortunately, may cause another problem such as Feature Shadowing that indicates that features with lower-priority may never be executed.

For instance, if Alice@uottawa.ca has two features in the order as follows:

incoming(x, "sip:Alice@site.uottawa.ca")
/\ address-switch(x.origin.user = "sip:Carl@uottawa.ca")
→
proxy(x, "sip:Bob@uottawa.ca");

incoming(x, "sip:Alice@site.uottawa.ca")
/\ address-switch(x.origin.user = "sip:Carl@uottawa.ca")
/\ time-switch(x.tstart = "8:30 am", x.dtend = "5:00 pm")
→
reject(x, "sip:Alice@uottawa.ca").

We see that the second feature will never be executed because if its preconditions are satisfied, the first feature's preconditions are satisfied too which happens to have the priority. Rule #L2 provides an approach to identify this type of Feature Interactions which might occur inside one CPL script.

Rule #L2

Feature Shadowing is present if the following characteristics hold:

(1) User A has at least two features.

(2) The precondition of the first feature is implied by that of the second feature.

Formally:

(1) incoming (x, A) /\ conditions1 → action1; incoming(x, A) /\ conditions2 → action2

or

outgoing(A, x) /\ conditions1 → action1; outgoing(A, x) /\ conditions2 →action2

(2) conditions2 → conditions1

Notes:

1. action1 and action2 mean any possible CPL actions.
2. Conditions1 and conditions2 are sets of enable-conditions, which cannot contain contradictions.

Another similar problem is semi-Feature Shadowing, which indicates that an overriding feature may influence others. For instance, suppose that Alice (Alice@uottawa.ca) does not want to take any calls from people whose name contains "Carl" and she gives this feature the highest priority; also suppose that calls from "ibm.com" are so important to Alice that she wants all these calls handled by Bob in case she is absent, as shown below:

incoming(x, "sip:Alice@site.uottawa.ca)
∧ address-switch(x.origin.user ⊇ "Carl")
→
reject(x, "sip:Alice@uottawa.ca:);

incoming(x, "sip:Alice@site.uottawa.ca")
∧ ¬address-switch(x.origin.user ⊇ "Carl")
∧ address-switch(x.original.host= "ibm.com")
∧ outcome(proxy(x, "sip:Alice@site.uottawa.ca"), "noanswer")
→
proxy(x, "sip:Bob@uottawa.ca")

However, if somebody named "Carl" in IBM.com (say "Carl@ibm.com") calls Alice@site.uottawa.ca when Alice is absent, then this call will be rejected rather than taken by Bob. We cannot say that there is a logical error but it will be helpful to remind Alice the drawbacks of setting that overriding feature. This phenomenon will be discussed further in our future work.

Rule #L2 was implemented in our tool.

4.4 In Summary

A single CPL Script may be free of syntax errors but surely may contain logical problems that can violate the subscribers' original intentions. We call this Local Inconsistency. According to the mechanism of CPL, the consequences of invalid paths, redundant conditions and Shadowed features may not be fatal, but they are definitely not what subscribers want. Beside that, potential bugs may hide inside invalid paths or redundant conditions.

Therefore, detecting these syntactically correct but logically false situations is very helpful in terms of improving the quality of CPL scripts.

Chapter 5 Identification of Feature Interactions in pairs of CPL Scripts

In Chapter 4 we saw that the possibilities of Feature Interactions are limited in single CPL scripts. The current chapter, after introduction of essential background on CPL, discusses Feature Interactions in pairs of CPL Scripts and presents identifying rules together with a description of the origin of these rules.

On the basis of two base rules (see Section 5.1), five concrete rules (rule D1-D4 and rule I1) are introduced (see section 5.3). The two base rules are used to prove the correctness of these concrete rules (see Chapter 6). Rules D1-D4 and I1 are programmed in Prolog and used to check the occurrence of the situations they describe in the SFSL formulas that were obtained from the CPL scripts.

5.1 General rules of Feature Interaction and Intention Contradiction between two users

Since our intentions specify features in Internet telephony (see 3.1.3), Feature Interactions and Intention Contradictions describe the same situation where one feature or intention is violated by another in overall system behavior. Furthermore, the reason behind these problems is the same, which is, those intentions or features cannot be satisfied at the same time, because logically, their conjunction is unsatisfiable [12]. As a consequence, traditional Feature Interactions are Intention Contradictions in our framework.

On the basis of this analysis, we can transplant some general rules that are established to detect Feature Interactions in traditional telecom system into Internet Telephony. Obviously, these general rules are valid beyond specific platforms:

Basic Rule #1: A feature which results in rejecting a call from user A to user B conflicts with a feature which results in forwarding a call from user A to user B. This rule can be specified in SFSL as:

conditions1 → reject(A, B)

contradicts

conditions2 → proxy(A, B)

Notes:

1. These two features may belong to one or two users, but we only consider the situation of two users in this chapter.
2. Conditions1 and conditions2 must be such that they can be true together.

Basic Rule #2: Endless forwarding loops among users conflict with system axioms in any telephony system. Since in this thesis we confine our research to two users, this rule can be simplified by stating that forwarding loops existing between two users conflict with system axioms. Again, with SFSL specifications, this rule can be specified as:

A: conditions1→proxy(x, B) and B: conditions2→proxy(x, A) contradicts system axioms.

Notes:

1. "A:" means that the following features belong to user A, and "B:" means that the following features belong to user B.
2. In principle, unlike the case of Basic Rule #1, the two conditions need not to be true together, however, this fact could lead to complicated consequences, and so we simplify the discussion by assuming that conditions1 can be enabled by action proxy(x, A) while proxy(x, B) can enable conditions2.
3. "x" represents the originator of the incoming call, which could be any user.

Beside situations covered by these two basic rules, there are some kinds of Feature Interactions that are difficult to determine and define. For instance, suppose that a user invokes a parallel call to another user (this feature is named Call Forking Outgoing), which probably indicates that the caller wants to talk to the callee immediately regardless

of location. We can guess how eager the caller is but how can automatic agents know this? Certainly, these types of contradiction need more study.

5.2 Feature Interactions in Pairs of CPL Scripts

The characteristics of CPL influence Feature Interactions in pairs of CPL scripts in many ways.

First of all, CPL has no state and does not support the signaling of busy tone, which implies that some traditional features will not be feasible in CPL, such as Call Waiting, 3-Way Call and Call Conference. Because of the same reason, several types of feature interactions in traditional telephony, for example, Call Waiting vs. 3-way Call, are not possible with CPL either.

Secondly, the CPL scripts of different users may be located in the same server if these users belong to the same organization, or in different ones. More complicated, one user may have several email addresses from several organizations, such as yahoo.com and hotmail.com, which indicates that the CPL scripts attached with these email addresses are located in different servers although they belong to the same real user. This distributed deployment of CPL increases the risk of occurrence of Feature Interactions significantly.

In this chapter, our discussion focuses on interactions occurring in pairs of CPL scripts, particularly, between two users' CPL scripts.

5.2.1 How Interactions Occur Between Two Different Users' CPL Scripts

Before going further, it is necessary to point out some potential constraints for Feature Interactions between two users. First of all, two users between whom Feature Interaction occurs must be involved in one call rather than two separate ones. This does not mean that user A has to call a specific user B directly, but user A does need to get user B involved somehow; secondly, there must be a contradiction: at least one user's policy or a

system axiom cannot be satisfied. An example of the first type of contradiction is that user A wants to forward user B's call to user C while user B does not want to talk to user C and blocks all calls to user C. In this case, either user A's or user B's policy cannot be satisfied. The second type of contradictions does not occur often; one example is the case where two users forward incoming calls to each other, which certainly violates the system axiom.

As explained in section 3.1, a CPL script includes two distinct policies, outgoing and incoming, dealing with outgoing calls and incoming calls respectively. Considering this context and based on the above analysis, we can say that there are only three types of possible interactions in terms of CPL and policies, which are:

Interactions between one user's outgoing policy and another user's incoming policy or
Interactions between one user's incoming policy and another user's incoming policy or
Interactions between one user's outgoing policy and another user's outgoing policy

Again, in these possibilities, interactions not only mean two users' policies contradicting directly but also two users' policies conflicting with system axioms.

As shown in Figure 5.1, the first case specifies the situation where user B's outgoing policy conflicts with user A's incoming policy; it occurs when user B calls user A directly or user B's call to somebody else is forwarded to user A. The example of Outgoing Call Screening vs. Call Forwarding illustrates this case.

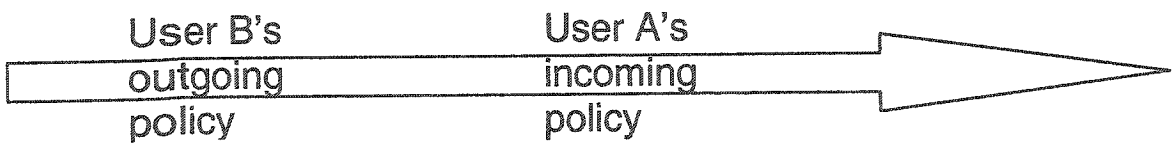

Figure 5.1 B's outgoing policy and A's incoming policy are involved in one call

The second case demonstrates the situation where there are interactions between two users' incoming policies as shown in Figure 5.2. It can be caused by the fact that one user's incoming policy results in invoking another user's incoming policy such as in the case of Forwarding Loop.

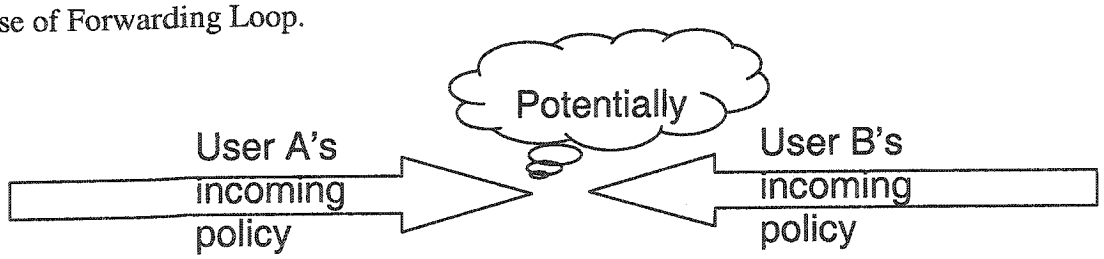

Figure 5.2 Interactions between the incoming policies of A and B

The third case would be possible only if there could be two originators for a call, but this is impossible in current or predictable telecom systems. Therefore, we exclude this possibility.

Note that the second and third cases seem similar, however they lead to different conclusions because one call can have several destinations (some of which could be intermediate) but only one originator.

5.3 Rules of Detecting Feature Interactions in pairs of CPL scripts

Translating CPL scripts into logic notation provides us with an opportunity of applying basic principles to derive specific rules for detecting Feature Interactions in pairs of CPL scripts. And, according to what was said in section 5.2, these rules are confined to two users, particularly to one user's incoming policy vs. the other user's outgoing policy or one user's incoming policy vs. the other user's incoming policy.

In this section, detection rules will be proposed separately in terms of contradicting directly or indirectly, which is originally from the definition of two users' policies contradicting directly and two users' policies conflicting with system axioms as discussed in 5.2.

5.3.1 Direct Contradiction Rules

Four concrete rules for detecting direct contradictions in CPL scripts of two different users have been identified. Three of them are based on Basic Rule#1 and the other one is to deal with the situation where the CPL scripts of two users contradict directly but implicitly.

Rule #D1

Rule #D1 identifies potential interactions between features subscribed by two different users, which addresses the case of interaction between Outgoing Call Screening and Call Forwarding. This type of FI is present if the following characteristics hold:

(1) User A's outgoing policy has a reachable action reject(A, C) where A represents the owner of this outgoing policy and C represents the blocked destination user. In other words, at least one action of reject(A, C) can be executed in user A's outgoing policy.

(2) User B's incoming policy has a reachable action proxy(x, C) where x means the forwarding destination could be any user and C represents the same user

C as in (1). In other words, at least one proxy(x, C) action can be executed in user B's incoming policy.

(3) A, B are different users

Formally:

(1) outgoing(A, C) /\ conditions1 → reject(A, C)

(2) incoming(x, B) /\ conditions2 → proxy(x, C)

(3) A!= B

Notes:

1. A, B, C and x are variables, but A, B and C are global for all formulas in this rule while x's range is limited to formula (2).
2. Conditions1 and conditions2 are other possible enable-conditions, which could be empty but cannot be contradictory.
3. This interaction will take place when A calls B or A's call to other subscriber originally is forwarded to B.
4. Attempts to reduce the number of involved users to less than three cannot lead to interaction. Assume user A's outgoing policy is outgoing(A, B) /\ conditions1 → reject(A, B) and user B's incoming policy is (incoming(x, B) /\ conditions2 → proxy(x, B)). No interaction is going to occur since the attempt of A calling B will be stopped by A's outgoing policy and B's incoming policy will not even be invoked.

As an example, suppose that the CPL script for Alice@uottawa.ca rejects all calls to Carl@uottawa.ca, which can be specified as:

outgoing("sip:Alice@uottawa.ca", x)

/\ address-switch(x.original-destination.user="sip:Carl@uottawa.ca")

→

reject("sip:Alice@uottawa.ca", x)

As we can see, only Carl@uottawa.ca can make the enable conditions true and the consequence of reject("sip:Alice@uottawa.ca", x) hold. Hence, the above formula is equivalent to the following one by replacing the variable x with the constant "sip:Carl@uottawa.ca":

outgoing("sip:Alice@uottawa.ca", "sip:Carl@uottawa.ca")
/\ address-switch(("sip:Carl@uottawa.ca").original-
destination.user="sip:Carl@uottawa.ca")
→
reject("sip:Alice@uottawa.ca", "sip:Carl@uottawa.ca")

Meanwhile, Bob@uottawa.ca's CPL script forwards all the incoming calls to Carl@uottawa.ca, which can be specified as:

incoming(x, "sip:Bob@uottawa.ca")
→
proxy (x, "sip:Carl@uottawa.ca")

Therefore, Rule #D1 is satisfied since:

(1) outgoing("sip:Alice@uottawa.ca", "sip:Carl@uottawa.ca")
/\ address-switch(("sip:Carl@uottawa.ca).original-
destination.user="sip:Carl@uottawa.ca")
→
reject("sip:Alice@uottawa.ca", "sip:Carl@uottawa.ca")

(2) incoming(x, "sip:Bob@uottawa.ca") → proxy(x, "sip:Carl@uottawa.ca")
(3) Alice@uottawa.ca != Bob@uottawa.ca

This example describes a situation where Alice does not want to talk to Carl whereas Bob forwards all the incoming calls to Carl. When Alice calls Bob or Alice's call is forwarded to Bob, both Alice's outgoing policy and Bob's incoming policy are invoked and both actions reject("sip:Alice@uottawa.ca", "sip:Carl@uottawa.ca") and proxy("sip:Alice@uottawa.ca", "sip:Carl@uottawa.ca") are executed. However, according to our Basic Rule#1, we know that these two contradicting intentions cannot be satisfied at the same time. Therefore, Feature Interaction occurs.

Rule #D2

Rule #D2 identifies another type of potential interaction which also might occur between CPL scripts of two users because of direct intention contradiction. It addresses the case of interaction between incoming Call Screening and Call Forwarding. This type of FI is present if the following characteristics hold:

(1) User A's incoming policy has a reachable action reject(C, A) where A represents the owner of this incoming policy and C represents the user whose call to A is forbidden. In other words, at least one action of reject(C, A) can be executed in user A's incoming policy.

(2) User B's incoming policy has a reachable action proxy(x, A) where x represents the caller which could be any user and A represents the forwarding destination that is the same user A as in (1). In other words, at least one action of proxy(x, A) can be executed in user B's incoming policy.

(3) A, B are different users.

Formally:

(1) incoming(C, A) /\ conditions1 → reject(C, A)

(2) incoming(x, B) /\ conditions2 → proxy(x, A)

(3) A!=B

Notes:

1. A, B, C and x are all variables, but A, B and C are global for all formulas in this rule while x's range is limited to formula (2).
2. Conditions1 and conditions2 are sets of other possible enable-conditions, which could be empty but cannot be contradictory.
3. This interaction will occur when C calls B or C's call is forwarded to B.

As an example, suppose that the CPL script of Alice@uottawa.ca rejects all calls from Carl@uottawa.ca, which can be specified as:

incoming(x, "sip:Alice@uottawa.ca")

/\ address-switch(x.origin.user= "sip:Carl@uottawa.ca")

→

reject(x, "sip:Alice@uottawa.ca")

Because of the same reason explained in rule #D1, this is equivalent to the following formula by replacing the variable x with the constant "sip:Carl@uottawa.ca":

incoming("sip:Carl@uottawa.ca", "sip:Alice@uottawa.ca")
/\ address-switch(("sip:Carl@uottawa.ca").origin.user="sip:Carl@uottawa.ca")
→
reject("sip:Carl@uottawa.ca", "sip:Alice@uottawa.ca")

And the CPL scripts of Bob@uottawa.ca forwards all the incoming calls to Alice@uottawa.ca when he is busy, which can be specified as:

incoming(x, "sip:Bob@uottawa.ca")
/\ outcome(proxy(x, "sip:Bob@uottawa.ca"), busy)
→
proxy (x, "sip:Alice@uottawa.ca")

They satisfy Rule #D2 since:

(1) incoming("sip:Carl@uottawa.ca" , "sip:Alice@uottawa.ca")
/\ address-switch(("sip:Carl@uottawa.ca").origin.user="sip:Carl@uottawa.ca")
→
reject("sip:Carl@uottawa.ca", "sip:Alice@uottawa.ca")

(2) incoming(x, "sip:Bob@uottawa.ca")
/\ outcome(proxy(x, "sip:Bob@uottawa.ca"), busy)
→
proxy ("sip:Carl@uottawa.ca", "sip:Alice@uottawa.ca")

(3) Alice@uottawa.ca != Bob@uottawa.ca

This example shows such a scenario where Alice refuses to take any call originally from Carl whereas Bob forwards all the incoming calls to Alice in case he is busy. When Carl calls Bob or Carl's call is forwarded to Bob, and Bob is busy, both incoming policies of

Bob and Alice are invoked and both actions of reject("sip:Carl@uottawa.ca", "sip:Alice@uottawa.ca") and proxy("sip:Carl@uottawa.ca", "sip:Alice@uottawa.ca") should be executed. However, according to Basic Rule#1, we know that these two contradicting intentions cannot be satisfied at the same time. Therefore, Feature Interaction occurs.

Rule #D3

Rule #D3 identifies another type of potential interaction which also might occur between two CPL scripts of two users because of direct intention contradiction. It also addresses the case of interaction between incoming Call Screening and Call Forwarding. This type of FI is present if the following characteristics hold:

(1) User A's incoming policy has a reachable action reject(B, A) where A represents the owner of this incoming policy and B represents the user whose call to A is forbidden. In other words, at least one action of reject(B, A) can be executed in user A's incoming policy.

(2) User B's outgoing policy has a reachable action proxy(B, A) where B represents the owner of this outgoing policy as well as the originator of calls and A represents the forwarding destination, both A and B are the same users as in (1). In other words, at least one action of proxy(B, A) can be executed in user B's outgoing policy.

(3) A, B are different users.

Formally:

(1) incoming(B, A) $\wedge$ conditions1 $\rightarrow$ reject(B, A)

(2) outgoing(B, x) $\wedge$ conditions2 $\rightarrow$ proxy(B, A)

(3) A != B

Notes:

1. A, B and x are all variables, but A and B are global for all formulas in this rule while x's range is limited to formula (2).
2. Conditions1 and conditions2 are sets of other possible enable-conditions, which could be empty but cannot be contradictory.

3. This interaction will occur only when B calls out.

As an example, suppose that the CPL script of Alice@uottawa.ca rejects all calls from Bob@uottawa.ca, which can be specified as:

incoming(x, "sip:Alice@uottawa.ca")
/\ address-switch(x.origin.user= "sip:Bob@uottawa.ca")
→
reject(x, "sip:Alice@uottawa.ca")

Because of the same reason we presented in rule #D1, this is equivalent to the following formula by replacing the variable x with the constant "sip:Bob@uottawa.ca":

incoming("sip:Bob@uottawa.ca", "sip:Alice@uottawa.ca")
/\ address-switch(("sip:Bob@uottawa.ca").origin.user="sip:Bob@uottawa.ca")
→
reject("sip:Bob@uottawa.ca", "sip:Alice@uottawa.ca")

Suppose now that Bob's CPL scripts sets up a speed dial number "12" for help desk. When Bob needs technical support between 8:30am-5 pm, he simply dials "12" and his call will be automatically forwarded to the appropriate person, Alice@uottawa.ca. This CPL scripts can be specified as:

outgoing("sip:Bob@uottawa.ca", x)
/\ address-switch(address-switch(x.destination.tel= "12"))
/\ time-switch(x.tstart="8:30 am", x.dtend="5:00 pm")
→
proxy ("sip:Bob@uottawa.ca", "sip:Alice@uottawa.ca")

They satisfy Rule #D3 since:

(1) incoming("sip:Bob@uottawa.ca", "sip:Alice@uottawa.ca")
/\ address-switch(("sip:Bob@uottawa.ca").origin.user="sip:Bob@uottawa.ca")
→
reject("sip:Bob@uottawa.ca", "sip:Alice@uottawa.ca")

(2) outgoing("sip:Bob@uottawa.ca", x)

/\ address-switch(x.destination.tel= "12")

/\ time-switch(x.tstart="8:30 am", x.dtend="5:00 pm")

→

proxy ("sip:Bob@uottawa.ca", "sip:Alice@uottawa.ca")

(3) Alice@uottawa.ca != Bob@uottawa.ca

This example shows such a scenario where both incoming policies of Bob and Alice can be invoked and both actions of reject("sip:Bob@uottawa.ca", "sip:Alice@uottawa.ca") and proxy("sip:Bob@uottawa.ca", "sip:Alice@uottawa.ca") can be executed. However, according to Basic Rule#1, we know that these two contradicting intentions cannot be satisfied at the same time. Therefore, Feature Interaction occurs.

Rule #D4

Rule #D4 identifies one type of potential interaction which might occur between CPL scripts of two users because of direct but implicit intention contradiction. This interaction is not related to either of the two Basic Rules in Section 5.1. It addresses the case of interaction between Call Forward and Call Forking Outgoing (See definition in Section 5.1.2). This type of FI is present if the following characteristics hold:

(1) User A's incoming or outgoing policy has a reachable action proxy(x, *C*, parallel) where x represents the originator of this incoming call which could be any user; *C* represents the set of forwarding destinations and "parallel" means the destination set will be tried simultaneously. In other words, at least one action of proxy(x, *C*, parallel) can be executed in user A's either incoming or outgoing policy.

(2) User B's incoming policy has a reachable action proxy(x, y) where x represents the originator of incoming calls which could be any user and y represents the forwarding destination which could be any user too.

(3) A, B are different users.

(4) C is a set of mail addresses and B belongs to C

Formally:

(1) incoming(x, A) /\ conditions1→proxy(x, C, parallel)

or outgoing(A, x) /\ conditions1→proxy(x, C, parallel)

(2) incoming(x, B) /\ conditions2 → proxy(x, y)

(3) A!=B

(4) B ∈ C

Notes:

1. A, B, C and x, y are all variables, but A, B and C are global for all formulas in this rule while the ranges of x and y are limited to formula (2).
2. Conditions1 and conditions2 are sets of other possible enable-conditions, which could be empty but cannot be contradictory.
3. This interaction will occur when A calls B.

As an example, suppose that Alice@uottawa.ca forwards all the incoming calls to Bob and she really wants these calls to reach Bob as soon as possible. Therefore, Alice@uottawa.ca forwards all the incoming calls to the two addresses that she has for Bob in parallel, which can be specified as:

incoming(x, "sip:Alice@uottawa.ca")

→

proxy(x, {"sip:Bob@uottawa.ca, sip:Bob@site.uottawa.ca"}, parallel)

At the same time, the CPL script of Bob@uottawa.ca redirects all the incoming calls to voicemail after 5 pm and before 8:30 am, which can be specified as:

incoming(x, "sip:Bob@uottawa.ca")

/\ time-switch(x.tstart="5:00 pm", x.dtend="8:30 am")

→

proxy(x, "sip:Bob@voicemail.example.com")

This satisfies Rule #D4 since:

(1) incoming(x, "sip:Alice@uottawa.ca")

→

proxy(x, {"sip:Bob@uottawa.ca, sip:Bob@site.uottawa.ca"}, parallel)

(2) incoming(x, "sip:Bob@uottawa.ca")

∧ time-switch(x.tstart="5:00 pm", x.dtend="8:30 am")

→

proxy(x, "sip:Bob@voicemail.example.com")

(3) Alice@uottawa.ca != Bob@uottawa.ca

(4) Bob@site.uottawa.ca ∈ {Bob@uottawa.ca, Bob@site.uottawa.ca}

This example shows such a scenario where Alice wants to assure that all the calls to her will reach Bob immediately. When somebody calls her after 5 p.m. and before 8:30 a.m., according to Alice's incoming policy, this call will immediately be forwarded to Bob@site.uottawa.ca and Bob@uottawa.ca in parallel. However, the incoming policy of Bob@uottawa.ca is going to transfer this call to a voicemail automatically. It is very possible that the voicemail will take over the incoming call before Bob@site.uottawa.ca (where Bob really is) can be reached it in the other place. Therefore, Alice's intention is violated and Feature Interaction occurs.

5.3.2 Indirect Contradiction Rules

Indirect Contradiction Rules are developed from Basic Rule#2. They focus on detecting Feature Interactions in pairs of CPL scripts such that two users' intentions do not contradict directly but these two intentions together could violate system axioms. We were able to identify only one such rule leading to detection of loops of call forwards.

Rule #I1

Rule #I1 identifies potential interactions which might occur between two users' incoming policies. It describes a situation of endless loop when A's action enables B's conditions

and B's action enables A's conditions too. This type of interaction is present if the following characteristics hold:

(1) User A's incoming policy has a reachable action proxy(x, B) where x represents the caller which could be any user and B represents the forwarding destination who is the same user as in (2). In other words, at least one action of proxy(x, B) can be executed in user A's incoming policy.

(2) User B's incoming policy has a reachable action proxy(x, A) where x represents the originator of this call which could be any user and A represents the forwarding destination who is the same users as in (1). In other words, at least one action of proxy(x, A) can be executed in user B's incoming policy.

(3) A, B are different users.

Formally:

(1) incoming(x, A) $\wedge$ conditions1 $\rightarrow$ proxy(x, B)

(2) incoming(x, B) $\wedge$ conditions2 $\rightarrow$ proxy(x, A)

(3) A!=B

Notes:

1. A, B and x are all variables, but A and B are global for all formulas in this rule while x's range is limited to the formula to which it belongs.
2. Conditions1 and conditions2 are other possible enable-conditions, which could be empty. If they are not empty, conditions1 must be enabled by the action "proxy(x, A)" in formula (1) and conditions2 must be enabled by the action "proxy(x, B)" in formula (2) because of the reason we explained in Section 6.1, Basic Rule #2.
3. This interaction will occur when either A or B is called.

As an example, suppose that the CPL script of Alice@uottawa.ca forwards all the incoming calls to Bob, which can be specified as:

incoming(x, "sip:Alice@uottawa.ca") $\rightarrow$ proxy(x, "sip:Bob@uottawa.ca")

And the CPL script of Bob@uottawa.ca forwards all the incoming calls to Alice, which can be specified as:

incoming(x, "sip:Bob@uottawa.ca") $\rightarrow$ proxy(x, "sip:Alice@uottawa.ca")

They satisfy Rule #I1 since:

(1) incoming(x, "sip:Alice@uottawa.ca") → proxy(x, "sip:Bob@uottawa.ca")

(2) incoming(x, "sip:Bob@uottawa.ca") → proxy(x, "sip:Alice@uottawa.ca")

(3) Alice@uottawa.ca != Bob@uottawa.ca

This example demonstrates such a scenario where Alice and Bob forward all the incoming calls to each other possibly because of unawareness of the other's policy. Assume that Alice is called by another user who could even be Bob, Alice's incoming policy will be invoked which forwards this incoming call to Bob. Meanwhile, according to Bob's incoming policy, the same call will be forwarded back to Alice-a forwarding loop is generated. Based on Basic Rule#2, we know these two contradicting intentions cannot be satisfied at the same time. Therefore, Feature Interaction occurs.

Some telephony systems are trying to solve the feature interactions of forwarding loop by limiting the time the system can be engaged in call forwarding. However, amounts of time and system resources would be consumed before the system notices the loop and ends it. Compared with this solution, our approach of detecting feature interactions before running (either at design stage or implementation stage) could benefit future telephony systems.

5.4 In Summary

In this chapter, we have discussed two Basic Principles for identifying Feature Interactions that are very general and we have characterised FIs in pairs of CPL scripts. Based on this analysis, five concrete rules have been presented for detecting potential Feature Interactions in pairs of CPL scripts. As we discussed in 5.1.2 and 5.2.1, these five rules cover all known Feature Interactions and derive from the two Basic Principles in the context of pairs of CPL scripts between two different users. These rules will be formally justified in chapter 6.

Since these five rules are formally defined, such detection can be automated as we shall discuss in chapter 7.

Chapter 6 Logic Proofs of Detection Rules

Chapter 4 and 5 present several rules for detecting Feature Interactions in CPL locally and pair-wise. In this chapter, we will prove the logical contradictions behind these rules using Predicate Logic.

6.1 Predicate Logic

Predicate logic is also called first-order logic. It is designed to deal with the logical aspects of natural and artificial languages such as *exists...*, *all...*, *among...*, and *only...*[19], in addition to *not, and, or* and *if... then* which can be represented by propositional logic (see [19]). Following are some of the Natural deduction rules for Predicate Logic that are used in our proof process:

	Introduction	Elimination
$\wedge$	$\dfrac{\Phi \quad \Psi}{\Phi \wedge \Psi}\ \wedge i$	$\dfrac{\Phi \wedge \Psi}{\Phi}\ \wedge e1$ $\qquad$ $\dfrac{\Phi \wedge \Psi}{\Psi}\ \wedge e2$
$\vee$	$\dfrac{\Phi}{\Phi \vee \Psi}\ \vee i1$ $\qquad$ $\dfrac{\Psi}{\Phi \vee \Psi}\ \vee i2$	$\dfrac{\Phi \vee \Psi \quad \boxed{\begin{array}{c}\Phi \\ \dots \\ X\end{array}} \quad \boxed{\begin{array}{c}\Psi \\ \dots \\ X\end{array}}}{X}\ \vee e$
$\rightarrow$	$\dfrac{\boxed{\begin{array}{c}\Phi \\ \dots \\ \Psi\end{array}}}{\Phi \rightarrow \Psi}\ \rightarrow i$	$\dfrac{\Phi \quad \Phi \rightarrow \Psi}{\Psi}\ \rightarrow e$
$\neg$	$\dfrac{\boxed{\begin{array}{c}\Phi \\ \dots \\ \bot\end{array}}}{}\ \neg i$	$\dfrac{\Phi \quad \neg \Phi}{\bot}\ \neg e$

	$\neg \Phi$	
$\forall$		$\frac{\forall x\, \Phi}{\Phi\,[t/x]}\ \forall x_c$

Table 6.1 Part of Natural deduction rules for Predicate Logic (Originally from [19])

6.2 Prove the Incoherence behind FI Detection Rules

As discussed in Section 4.1 and Section 5.1, Feature Interactions present logically unsatisfiable situations. So according to our definition, logical incoherence is the premise of Feature Interaction. If there is no logical incoherence, Feature Interaction does not exist either. Therefore, the proof of the logical incoherence behind our detection rules can be considered as a step in validating the correctness of the rules.

Our proof method is originally from [12].

6.2.1 Proofs for Local FI Detection Rules

Since Rule #L1 presents an obvious logical equivalence (conditions1 $\wedge$ condition2 $\rightarrow$ action1; conditions1 $\wedge \neg$condition2$\rightarrow$ action1 are equal to conditions1$\rightarrow$ action1), we only give the proof for Rule#L2.

For Rule #L2:

(1) incoming (x, A) $\wedge$ conditions1 $\rightarrow$ action1; incoming(x, A) $\wedge$ conditions2 $\rightarrow$ action2

or

outgoing(A, x) $\wedge$ conditions1 $\rightarrow$ action1; outgoing(A, x) $\wedge$ conditions2 $\rightarrow$action2

(2) conditions2 $\rightarrow$ conditions1

Formula(1) has two possibilities, so, in order to simplify the proof, we randomly choose the first one: incoming (x, A) $\wedge$ conditions1 $\rightarrow$ action1; incoming(x, A) $\wedge$ conditions2 $\rightarrow$ action2. And because the unbound variables x in formula(1) represent one of any possible users, Rule #L2 can be specified in Predicate Logic as follows:

(1) $\forall x$(incoming (x, A) $\wedge$ conditions1 $\rightarrow$ action1;
incoming(x, A) $\wedge$ conditions2 $\rightarrow$ action2)
(2) conditions2 $\rightarrow$ conditions1

The mechanism of CPL implies that only one action is executed at one time, and that if more than one action's enable conditions are satisfied, the action that comes first in the script will be executed. This characteristic could be specified as (action1$\rightarrow\neg$action2) in our current case; and the fact that the second feature's enable condition could hold ($\forall x$(incoming(x, A) $\wedge$ conditions2)) needs to be combined with the above formulas as well.

Therefore, our task is to prove that there is no model where the six formulas above are satisfiable as shown below:

$\forall x$(incoming (x, A) $\wedge$ conditions1 $\rightarrow$ action1),
$\forall x$(incoming(x, A) $\wedge$ conditions2 $\rightarrow$ action2),
action1$\rightarrow\neg$action2,
$\forall x$(incoming(x, A) $\wedge$ conditions2),
conditions2$\rightarrow$ conditions1
$\Rightarrow \perp$

Proof:

1	$\forall x$(incoming(x, A) $\wedge$ conditions1 $\rightarrow$ action1)	premise
2	$\forall x$(incoming(x, A) $\wedge$ conditions2 $\rightarrow$ action2)	premise
3	$\forall x$(incoming(x, A) $\wedge$ conditions2)	premise
4	incoming(C, A) $\wedge$ conditions1 $\rightarrow$ action1	$\forall x$ e 1
5	incoming(C, A) $\wedge$ conditions2 $\rightarrow$ action2	$\forall x$ e 2
6	incoming(C, A) $\wedge$ conditions2	$\forall x$ e 3
7	action2	$\rightarrow$e 5, 6
8	conditions2	$\wedge$e2 6
9	incoming(C, A)	$\wedge$e1 6

10 conditions2→ conditions1	premise
11 conditions1	→e 8, 10
12 incoming(C, A) Λ conditions1	Λi 9, 11
13 action1	→e 4, 12
14 action1→¬action2	premise
15 ¬action2	→e 13, 14
16 ⊥ ¬e 7, 15	

6.2.2 Proofs of FI Detection Rules for pair-wise CPL scripts

We will show the contradictions behind rules #D, #D4, #I1.

For Rule #D1:

(1) outgoing(A, C) Λ conditions1 → reject(A, C)

(2) incoming(x, B) Λ conditions2 → proxy(x, C)

(3) A!= B

As discussed in section 6.3.1, conditions1 and conditions2 may be empty. In order to simplify the proof, we assume that they are absent; and the unbound variable x in formula(2) represents one of any possible users, so that Rule #D1 can be specified in Predicate Logic as follows:

(1) outgoing(A, C) → reject(A, C)

(2) ∀x(incoming(x, B) → proxy(x, C))

(3) A!= B

Also because of the fact that the two features above (formula(1) and formula(2)) can occur simultaneously, the combination of their enable conditions (outgoing(A, C) Λ incoming(A, B)), which means they can be true together, should be added too. In addition, a system axiom ¬(proxy(A, C) Λ reject(A, C)) (from Basic Rule #1), which should be supported by all known telecom systems, also needs to be combined with the above feature specifications.

Therefore, according to [12], our task is to prove that there is no model where all these five formulas above are satisfiable as shown below:

Outgoing(A, C) → reject(A, C),

∀x(incoming(x, B) → proxy(x, C)),

A != B,

Outgoing(A, C) ∧ incoming(A, B),

¬(proxy(A, C) ∧ reject(A, C))

⇒ ⊥

Proof:

1 outgoing(A, C) → reject(A, C)		premise
2 ∀x(incoming(x, B) → proxy(x, C))		premise
3 incoming(A, B) → proxy(A, C)		∀x e 2
4 outgoing(A, C) ∧ Incoming(A, B)		premise
5 ¬(proxy(A, C) ∧ reject(A, C))		premise
6 outgoing(A, C)	∧e1 4	
7 incoming(A, B)	∧e2 4	
8 reject(A, C)	→e 1, 6	
9 proxy(A, C)	→e 3, 7	
10 proxy(A, C) ∧ reject(A, C)	∧i 8, 9	
11 ⊥	¬e 5, 10	

The proofs of Rule #D2 and Rule #D3 are similar to that of Rule #D1, so we will not go through them.

For Rule #D4:

(1) incoming(x, A) ∧ conditions1→proxy(x, *C*, parallel)

or

outgoing(A, x) ∧ conditions1→proxy(x, *C*, parallel)

(2) incoming(x, B) ∧ conditions2 → proxy(x, y)

(3) A!=B

(4) B ∈ *C*

Formula(1) has two possibilities, in order to simplify the proof, we randomly choose the second one: outgoing(A, x) ∧ conditions1→proxy(x, *C*, parallel). And because of the same reason explained in the proof of Rule#D1, we assume that conditions1 and conditions2 are absent. Also because unbound variables x and y in formula(2) represent two of any possible users, Rule #D4 can be specified in Predicate Logic as follows:

(1) ∀x(outgoing(A, x) → proxy(x, *C*, parallel))

(2) ∀x∀y(incoming(x, B) → proxy(x, y))

(3) A!= B

(4) B ∈ *C*

Suppose that *C*, a set of email addresses, only contains two addresses, namely B and D, then proxy(x, *C*, parallel) is equal to proxy(x, B) ∧ proxy (x, D); according to Call Forking Outgoing (Formula(1)), the execution of the second feature(formula(2)) will deactivate the other parallel call request from A to D (∀x∀y(B: proxy(x, y)→ A : ¬proxy(x, D)); and we also admit the fact that both enable conditions in formula(1) and formula(2) can hold simultaneously (∀x(outgoing(A, x) ∧ incoming(x, B))).

Therefore, our task is to prove that there is no model where the six formulas above are satisfiable as shown below:

∀x(outgoing(A, x) → proxy(x, B) ∧ proxy(x, D)),
∀x∀y(incoming(x, B) → B: proxy(x, y)),
A!= B,
∀x(outgoing(A, x) ∧ incoming(x, B)),
∀x∀y(B: proxy(x, y)→ A : ¬proxy(x, D))
⇒ ⊥

Proof:

1	∀x(outgoing(A, x) → proxy(x, B) ∧ proxy(x, D)) premise

2 ∀x∀y(incoming(x, B) → B: proxy(x, y))	premise
3 incoming(C, B) → B: proxy(C, E)	∀x∀y e 2
4 ∀x(outgoing(A, x) ∧ incoming(x, B))	premise
5 outgoing(A, C) ∧ incoming(C, B)	∀x e 4
6 outgoing(A, C) → A: proxy(C, B) ∧ proxy(C, D)	∀x e 1
7 outgoing(A, C)	∧e1 5
8 incoming(C, B)	∧e2 5
9 B: proxy(C, E)	→e 3, 8
10 A: proxy(C, B) ∧ proxy(C, D)	→e 6, 7
11 A: proxy(C, D)	∧e2 10
12 ∀x∀y(B: proxy(x, y)→ A : ¬proxy(x, D))	premise
13 B: proxy(C, E) → A : ¬proxy(C, D)	∀x∀y e 12
14 A: ¬proxy(C, D)	→e 9, 13
15 ⊥	¬e 11, 14

For Rule #I1:

(1) incoming(x, A) ∧ conditions1 → proxy(x, B)

(2) incoming(x, B) ∧ conditions2 → proxy(x, A)

(3) A!=B

Because of the same reason explained in the proof of Rule#D1, we assume that conditions1 and conditions2 are absent; also the unbound variable x in formula(1) and formula(2) represents one of any possible user, so that Rule #I1 can be specified in Predicate Logic as follows:

(1) ∀x(incoming(x, A) → proxy(x, B))

(2) ∀x(incoming(x, B) → proxy(x, A))

(3) A!= B

Because these two features (formula(1) and formula(2)) may run on telecom systems simultaneously and are supposed to be handled successfully together in this case, the fact that both enable conditions can hold (∀x(incoming(x, A) ∧ incoming(x, B))) needs to be

added, so does one requirement of telecom systems that the forwarding loop is not allowed ($\neg(\forall x(B: proxy(x, A) \wedge A: proxy(x, B))$, from Basic Rule #2), where "A:" and "B:" represent the owners of the two actions.

Therefore, our task is to prove that there is no model where these six formulas are satisfiable as shown below:

$\forall x(incoming(x, A) \rightarrow proxy(x, B))$,

$\forall x(incoming(x, B) \rightarrow proxy(x, A))$,

$A != B$,

$\forall x(incoming(x, A) \wedge incoming(x, B))$,

$\neg(\forall x(B: proxy(x, A) \wedge A: proxy(x, B))$

$\Rightarrow \perp$

Proof:

1	$\forall x(incoming(x, A) \rightarrow proxy(x, B))$	premise
2	$\forall x(incoming(x, B) \rightarrow proxy(x, A))$	premise
3	$incoming(C, A) \rightarrow proxy(C, B)$	$\forall x$ e 1
4	$incoming(C, B) \rightarrow proxy(C, A)$	$\forall x$ e 2
5	$\forall x(incoming(x, A) \wedge incoming(x, B))$	premise
6	$incoming(C, A) \wedge incoming(C, B)$	$\forall x$ e 5
7	$outgoing(A, C) \wedge incoming(A, B)$	premise
8	$\neg(\forall x(B: proxy(x, A) \wedge A: proxy(x, B)))$	premise
9	$incoming(C, A)$	$\wedge$e1 6
10	$incoming(C, B)$	$\wedge$e2 6
11	$proxy(C, A)$	$\rightarrow$e 4, 7
12	$proxy(C, B)$	$\rightarrow$e 3, 6
13	$B: proxy(C, A) \wedge A: proxy(C, B)$	$\wedge$i 11, 12
14	$\neg(B: proxy(C, A) \wedge A: proxy(C, B))$	$\forall x$ e 8
15	$\perp$	$\neg$e 13, 14

Another possible approach for proving the Logical Incoherence behind these rules is to apply Linear Temporal Logic and discuss the sequence of events (see [12]). Since the rules of SFSL are much easier to specify in Predicate Logic, Predicate Logic is more appropriate in our case.

6.3 In Summary

The current chapter has used Predicate Logic to prove the logical contradiction behind some of the Feature Interactions detection rules proposed in Chapter 4 and Chapter 5. It also can be considered as a step for validating these rules.

Note that in some cases (especially in the case of rule #I1), we deal with sequential behaviours; so temporal logic concepts might have been found more satisfactory. However, temporal logic specifications and proofs would have been much more complicated.

Chapter 7 Implementation of Automatic Detection of Feature Interactions in CPL

Based on the rules presented in Chapter 4 and Chapter 5, a tool for automatic Feature Interactions detection was developed. Given CPL scripts of different users, the tool translates CPL scripts into SFSL Specifications first, and then applies detection rules to these SFSL specifications to detect all Feature Interactions that have been identified by us. At the end, corresponding reports are generated as well.

7.1 Overview

In general, our tool separates the whole process into three steps:

- Translate CPL scripts into SFSL as shown in Figure 7.1.

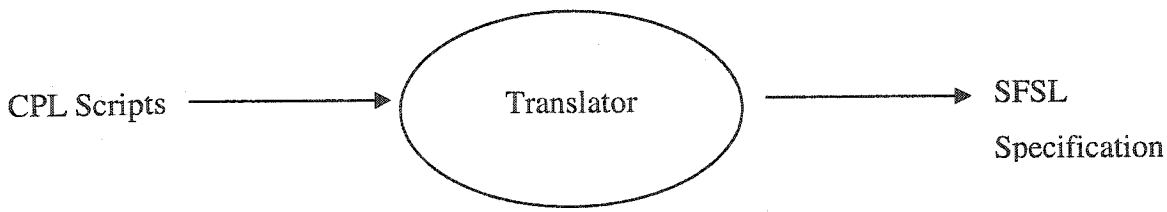

Figure 7.1 Translate CPL Scripts into SFSL Specifications

- Apply local FI detection rules to each user's policy as shown in Figure 7.2.

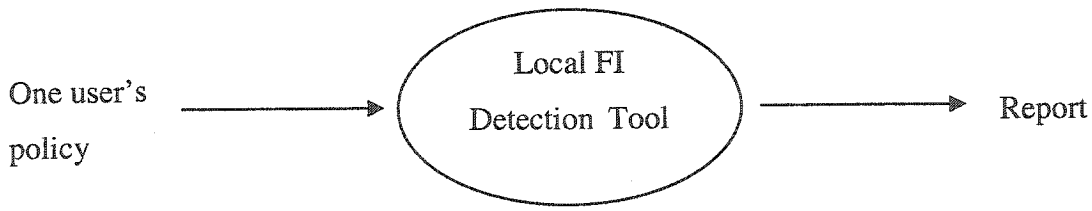

Figure 7.2 Detect Local FIs in CPL

- Apply FI detection rules for pair-wise CPL scripts as shown in Figure 7.3.

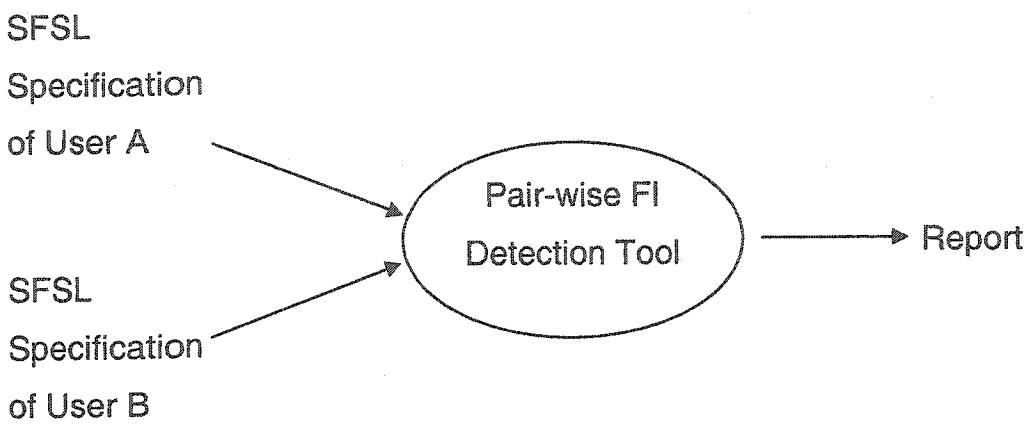

Figure 7.3 Apply FI detection rules for pair-wise CPL scripts

These three steps can be considered as three functions of a tool; and such a tool can be automated by the use of a logic programming language.

7.2 Implementing SFSL Specifications and Detection Rules in Prolog

Our approach chooses Prolog (see [14], [10]) as the implementation language. Prolog was invented by Alain Colmerauer at the Univ. of Marseilles in 1973 and is most suitable for symbolic, non-numeric computation. It is especially well suited for solving problems that involve objects and relations between objects [3].

Among different versions of Prolog, we choose SWI-Prolog, which is "a Prolog implementation based on a subset of the WAM (Warren Abstract Machine)" and "was developed as an *open* Prolog environment" [38].

7.2.1 Representing SFSL Specifications in Prolog

Section 4.2.2 shows that a CPL script can be divided into several independent branches leading to different actions. It has also been seen that one SFSL specification contains only one action. Prolog can represent a SFSL specification that stands for an intention in CPL as follows:

cpl_policy(user, implies(conj(cnds1, cnds2), action)),

where cnds1 and cnds2 represent enable conditions; conj(cnds1, cnds2) stands for the conjunction of cnds1 and cnds2 "cnds1 /\ cnds2"; implies(conj(cnds1, cnds2), action) represents "cnds1 /\ cnds2 → action"; and "user" stands for the subscriber of this CPL intention.

Note that cnds1 is only composed of either "outgoing" or "incoming" whereas cnds2 may be empty or contain one or a set of enable conditions.

As an example, suppose that Alice@uottawa.ca has a feature of Incoming Call Screening that can be specified in SFSL as:

incoming(x, "sip:alice@uottawa.ca")
/\ address-switch(x.origin.user= "sip:carl@uottawa.ca")
→
reject(x, "sip:alice@uottawa.ca").

This can be represented in Prolog as:

```
cpl_policy('sip:alice@uottawa.ca',
        implies(conj(incoming(X, 'sip:alice@uottawa.ca'),
                        address(is('sip:carl@uottawa.ca'))]),
                [reject(status(reject)), reject(reason('I do not accept carl's call.'))])).
```

Note that although we do not specify the action "reject" in the way of "reject(caller, callee)", the keyword of "incoming" already indicates the callee is "sip:alice@uottawa.ca" and the caller is X, which represents any user. Note also that, although SFSL does not include indication of reasons, we have included these in our Prolog program.

7.2.2 Representing Detection Rules in Prolog

Detection Rules given in Chapter 4 and Chapter 5 are implemented in Prolog in predicate formulae called li_check and fi_check respectively, whose formats are as follows:

li_check(R, Formulas1, Formulas2) and fi_check(R, User1, User2, Formulas1, Formulas2),

where R stands for the Rule's name; Formulas1 and Formulas2 represent a pair of SFSL specifications of intentions, which are either from the same user (for li_check) or from two different users (for fi_check).

For instance, Rule #D1 can be specified as:

```
fi_check(d1, U1, U2, implies(Cdns1,[reject(status(reject)), _]),
implies(Cdns2,[proxy([location(C)])])):-
        mem1(outgoing(U1,X), Cdns1),
        mem1([address(is(C))], Cdns1),
        mem1(incoming(X,U2), Cdns2), !.

fi_check(d1, U1, U2, implies(Cdns1,[reject(status(reject)), _]),
implies(Cdns2,[proxy([location(C)])])):-
        mem1(outgoing(U1,X), Cdns1),
        mem1([address(contains(Tem))], Cdns1),
        sub_string(C, _Start, _Length, _After, Tem),
        mem1(incoming(X,U2), Cdns2), !.

fi_check(d1, U1, U2, implies(Cdns1,[reject(status(reject)), _]),
implies(Cdns2,[proxy([location(C)])])):-
        mem1(outgoing(U1, X), Cdns1),
        mem1([address(subdomain-of(Tem))], Cdns1),
        sub_string(C, _Start, _Length, _After, Tem),
        mem1(incoming(X,U2), Cdns2), !.
```

where mem1(argu1, argu2) represents that argu1 is a part of argu2 and is specified in Prolog as:

```
        mem1(A, A) :- !.
```

```
mem1(A, conj(A,X)) :- !.
mem1(A, conj(B,X)) :-
        !, mem1(A,X);
```

, and implies(argu1, argu2) represents that argu1 implies argu2 (argu1→argu2).

Prolog has a mechanism that allows the self-binding of variables to satisfy predicates. The most difficult part is to determine whether some unbound variables located in different formulas need to be the same or not, independently of their names. As an example, let us consider two users' intentions as follows, where the first is for Alice, the second is for Bob:

```
cpl_policy(alice,
        implies(conj(incoming(X, alice), []),
                [proxy([location('sip:carl@pager.ottawahospital.com')])])).

cpl_policy(bob,
        implies(conj(outgoing(bob, X),
                        [address(is('sip:carl@pager.ottawahospital.com'))]),
                [reject(status(reject)), reject(reason('I don\'t accept carl\'s calls'))])).
```

Rule #D1 will be satisfied since Prolog tries to bind variables as follows:

implies(conj(incoming('sip:carl@pager.ottawahospital.com', alice), []), [proxy([location('sip:carl@pager.ottawahospital.com')])]). (1)

implies(conj(outgoing(bob, 'sip:carl@pager.ottawahospital.com'), [address(is('sip:carl@pager.ottawahospital.com'))]), [reject(status(reject)), reject(reason('I don\'t accept carl\'s calls'))]) (2)

We see that the variable "X" in Alice's outgoing policy is replaced by the constant 'sip:carl@pager.ottawahospital.com' because of the reasons we explained in section 5.3.1. On the other hand, the variable "X" in predicate (2) (it needs not to be the same "X" as in (1)) is replaced by the same constant too.

7.3 Development of the Translator

As discussed in Section 3.3, the first step of the translation from CPL scripts to SFSL specifications is to split the CPL tree into independent branches. However, in practice, there is another solution that creates individual CPL features as individual trees rather than bundling them together into a single CPL decision tree [1], by which the difficulty of setting up features ordering at the very beginning is avoided and priorities among features are still available in an explicit way [1]. Our development of the translator starts from this point.

Practically, the translation can be done by using a Prolog package SGML, which allows us to parse XML data into a Prolog data structure directly. Accompanied with DTD (Document Type Definition) files, CPL scripts can be loaded and separated into Prolog data structures with the predicate:

load_structure(File, ListOfContent, [dialect(xml)])

or

load_xml_file(File, -ListOfContent).

Figure 7.4 shows a simple CPL script and its corresponding Prolog translations:

```
<?xml version="1.0" ?>
 <!DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx CPL 1.0//EN" "cpl.dtd">

 <cpl>
  <incoming>
     <location url="sip:carl@phone.example.com">
        <proxy />
     </location>
  </incoming>
 </cpl>
```

```
?- load_xml_file('test.cpl', Term),
[element(
        cpl,
        [],
        [element(
                incoming,
                [],
                [element(
                        location,
                        [url='sip:carl@phone.example.com', clear=no],
                        [element(
                                proxy,
                                [recurse=no, ordering=parallel],
                                [])
                        ])
                ])
        ])
]
```

Figure 7.4 From CPL scripts to Prolog data structures

The function load_xml_file(File, -ListOfContent) parses the source file *File* and returns a corresponding Prolog structure *ListOfContent*. This structure is composed of a list of atoms and/or element/3 terms. The format of element/3 is element(Name, ListOfAttributes, ListOfContent) where "Name" is the name of the element, "ListOfAttributes" is a list of Name=Value pairs of attributes and "ListOfContent" defines the content for the element.

After a script is parsed, all the recognised elements and attributes are printed out by the function listed below:

```
printMe(feature(Name, _Level,_Cdn,Trig,Res),[]) :-
```

```
        write('cpl_policy('), write(Name), write(','), write('implies('), print(Trig),
write(','), print(Res),
        write(')).\n'), !.
```

For instance, the above CPL script is translated as:

```
    cpl_policy(luigi,implies(conj(incoming(_G474, luigi), []),
[proxy([location('sip:carl@phone.example.com')])])).
```

7.4 Detecting FIs with the Filter

After CPL scripts are translated into separate intentions in Prolog, Local FI detection rules will be applied to detect local Feature Interactions first, followed by FI detection rules for pair-wise CPL scripts to check each pair of intentions that belong to different owners. For instance, given two CPL scripts of user A and user B, where each of them has two separate features, altogether 4 (2*2) rounds of detection are executed. Let us consider two users' features as follows:

Features for alice@uottawa.ca:

```
    cpl_policy('sip:alice@uottawa.ca',
        implies(conj(outgoing('sip:alice@uottawa.ca', _G412),
                [address('subdomain-of('1700'))]),
            [proxy([location('sip:bob@uottawa.ca')], ordering(parallel))])).

    cpl_policy('sip:alice@uottawa.ca',
        implies(conj(incoming(_G398, 'sip:alice@uottawa.ca'), []),
            [proxy([location('sip:carl@pager.ottawahospital.com')])])).
```

Features for bob@uottawa.ca:

```
    cpl_policy('sip:bob@uottawa.ca',
        implies(conj(incoming(_G411, 'sip:bob@uottawa.ca'),
                [address(is('sip:alice@uottawa.ca'))]),
            [reject(status(reject)), reject(reason('I do not accept Alice\'s call.'))])).
```

```
cpl_policy('sip:bob@uottawa.ca',
    implies(conj(outgoing('sip:bob@uottawa.ca', _G412),
                [address(is('sip:carl@pager.ottawahospital.com'))]),
            [reject(status(reject)), reject(reason('I am not allowed to call Carl'))])).
```

The pair-wise detection process is started by running the following predicate:

?- checking.

And corresponding reports will be generated if Feature Interaction is detected. The report is composed of two parts: detailed explanation of the Feature Interaction and the two contradicting features. The detailed explanation contains the name of the rules that are applied, and a brief description of the Feature Interaction type. For this example, the corresponding report is:

```
%-----------------------------------------------------------------------------------------
% ****** Interaction detected by Rule D3 -> %  The first user reject calls from the
second user
%  while the second user will forward outgoing calls to the first user
%     + The first user's policy
implies(conj(incoming(_G248, sip:bob@uottawa.ca),
[address(is(sip:alice@uottawa.ca))]), [reject(status(reject)), reject(reason(I do not accept
Alice's call.))])
%     + The second user's policy
implies(conj(outgoing(sip:alice@uottawa.ca, _G248), [address(subdomain-of(1700))]),
[proxy([location(sip:bob@uottawa.ca)], ordering(parallel))])


%****** Interaction detected by Rule D1 ->
%  The first user is forbidden to call somebody
%  while the second user will forward calls to the forbidden user
%     + The first user's policy
```

```
implies(conj(outgoing(sip:bob@uottawa.ca, _G249),
[address(is(sip:carl@pager.ottawahospital.com))]), [reject(status(reject)), reject(reason(I
am not allowed to call Carl))])
%     + The second user's policy
implies(conj(incoming(_G249, sip:alice@uottawa.ca), []),
[proxy([location(sip:carl@pager.ottawahospital.com)], ordering(parallel))])
%----------------------------------------------------------------------------------------
```

7.5 In Summary

The current chapter introduced the implementation of an automatic detection tool. We chose Swi-prolog as the implementation language and explained how to translate original CPL scripts into SFSL specifications in Prolog and how to specify Detection Rules in Prolog predicates; in addition, methods of building a translator and designing a filter are discussed.

Our program still has difficulty in dealing with the situation of "otherwise" which would be part of our future work. The complete Prolog code of this tool is listed in Appendix A and B. Appendix C gives a list of policies in SFSL and associated feature interaction detection results.

Chapter 8 Conclusions and Future Work

This Chapter reviews the contributions of our work and opens some discussions of future research arising from this thesis.

8.1 Thesis Review

Our research aims at offering a method of detecting Feature Interactions in the Call Processing Language (CPL). Starting with a study of the definition and origin of Feature Interactions, we discussed some existing approaches for detecting Feature Interactions.

After an introduction to Feature Interactions, we present an overview of Internet Telephony and CPL, on which our research focuses. In chapter 3, we analysed the structure of CPL and introduced the logic-based Simple Formal Specification Language (SFSL). A method for translating CPL scripts into SFSL specifications was also proposed. This translation is helpful not only for deriving FI detection rules, but also for developing the corresponding detection tool.

Chapter 4 discussed Local Inconsistencies in single CPL scripts, which may include Feature Interactions and Feature inconsistency. Together with the analysis of origins and categories of Local Inconsistencies, detection methods were proposed as well.

Chapter 5 introduced two basic detection rules that are valid beyond specific environments and platforms. Based on these two rules and the analysis of CPL structures in chapter 3, five concrete Feature Interaction Detection rules were proposed, four of them concern direct logical conflicts between two users' CPL scripts and the last one is about indirect conflicts.

In chapter 6, we applied Predicate Logic to prove the logical incoherence behind the detection rules, which can be considered as a step in the validation of these rules. Chapter 7 described the design of an automatic detection tool using Swi-Prolog. This tool is composed of two functions: one is the translator that translates original CPL scripts into

SFSL specifications, the other is the filter that applies the detection rules presented in chapter 4 and chapter 5.

8.2 Contributions

This thesis makes three major contributions: abstracting CPL scripts, proposing Feature Interactions detection rules in CPL and developing an automatic FI detection tool.

8.2.1 Abstracting CPL Scripts

Translating CPL scripts into logic-based SFSL is considered as a necessary step of overcoming the inborn drawbacks of CPL; additionally, it provides a way to verify the correctness of CPL scripts -- some Local Inconsistency could be detected during the translation. Chapter 3 presents the method of translating original CPL scripts into SFSL specifications, which mainly contains two parts: the syntax of SFSL and the translation method.

One of the chief achievements in our approach is dividing a CPL script into independent intentions (or features) regarding individual actions. It links features to user intentions and transforms the Feature Interactions between CPL scripts into contradictions between independent intentions.

There may exist other solutions to abstract CPL scripts such as representing CPL using BDDs (Binary Decision Diagrams, see [2]). BDD is a most important application in the areas of digital system design, verification, and testing [5] and its structure is also based on decision-trees; it would be straightforward for BDDs to represent CPL scripts if both conditions and actions in CPL were considered as propositions. Thus, local inconsistencies within a CPL script could be detected by modelling CPL scripts in BDD; for detecting interactions in pairs of CPL scripts, merging the corresponding two BDDs would be inevitable. We did not use BDDs in this project because of the complicated encodings that would be required to translate CPL statements in BDD format. However,

BDDs are associated with extremely efficient rectification tools. In any case, the use of BDD for this purpose is left for future study.

8.2.2 Proposing Feature Interactions Detection Rules

Chapter 4 proposed local FI detection rules and Chapter 5 proposed pair-wise ones. Chapter 5 also discussed two basic FI detection principles. Five concrete pair-wise FI detection rules are developed from these two basic principles in the context of CPL and are classified into two categories regarding their usage -- they are used to detect either direct or indirect contradictions between different CPL scripts.

Moreover, we proved the logical incoherence behind these rules using Predicate Logic, which is also considered as a step towards validating the correctness of these rules.

8.2.3 Developing an Automatic Detection Tool

We have developed an automatic Feature Interaction Detection tool using Swi-Prolog. This tool is able to translate original CPL scripts into SFSL specifications and then apply the rules presented in Chapter 4 and Chapter 5 to detect potential Feature Interactions; corresponding reports will be given when rules are matched and potential Feature Interactions are found.

This automatic tool provides us an opportunity of dealing with FIs among CPL scripts of large size in an efficient way; it is also helpful in terms of improving the quality of CPL scripts since Local Inconsistency will also be detected.

8.3 Comparison with Related Approaches

8.3.1 The work of Nakamura et al.

Nakamura et al. proposed another approach for detecting script-to-script interactions in two papers [31] [32]. We became aware of this method only when our work was well-advanced. Some last-minute modification of our method to cover some of Nakamura's local inconsistencies were done, however, in other cases, we saw no reason for changing

our method, for the reason discussed below. Nakamura's approach addresses possible semantic warnings in individual CPL scripts, and then expands the analysis method to pairs of scripts based on "defining feature interactions as the semantic warnings over multiple CPL scripts" [31]. For instance, one type of semantics warnings, Address Set After Address Switch (ASAS), considers the case that a script sets an Outgoing Call Screening (OCS) to a destination but afterwards forwards a call to the same destination (although in another situation) as a semantic error. This analysis warning is also expanded to the scope of pair-wise features that belong to different users to detect Feature Interactions by combining two CPL scripts into one. One example where one user's OCS feature conflicts with another user's DCF (Domain Call Filtering) feature is shown below [31]:

```
<?xml version="1.0" ?>
<!DOCTYPE cpl PUBLIC "-//IETF//DTD
RFCxxxx CPL 1.0//EN" "cpl.dtd">
<cpl>
 <outgoing>
   <address-switch field="destination" >
     <address is="sip:bob@home.org">
        <reject status="reject"
          reason="No call to Bob is
                permitted" />
     </address>
     <otherwise>
        <proxy />
     </otherwise>
   </address-switch>
 </outgoing>

 <incoming>
```

```
<?xml version="1.0" ?>
<!DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx
CPL 1.0//EN" "cpl.dtd">
<cpl>
   <subaction id="voicemail">
     <location url="sip:chris@voicemail.example.com">
       <proxy />
     </location>
   </subaction>
<incoming>
   <address-switch field="origin" subfield="host">
     <address subdomain-of="example.com">
       <location url="sip:chris@office.example.com">
         <proxy />
       </location>
     </address>
     <address subdomain-of="crackers.org">
        <reject status="reject"
          reason="No call from this domain allowed" />
     </address>
     <address subdomain-of="instance.net">
       <location url="sip:bob@home.org">
          <redirect />
       </location>
     </address>
     <otherwise>
        <sub ref="voicemail" />
     </otherwise>
```

```
</incoming>
</cpl>
```

Figure 8.1 A feature of Outgoing Call Screening (OCS) for alice@instnace.net

```
    </address-switch>
</incoming>
</cpl>
```

Figure 8.2 A Feature of Domain Call Filtering (DCF) for chris@example.com

We classify this interaction as ASAS after combining these two scripts (see the combining method in [31] and [32]):

```
<?xml version="1.0" ?>
<!DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx CPL 1.0//EN" "cpl.dtd">
<cpl>
 <outgoing>
   <address-switch field="destination" >
     <address is="sip:bob@home.org">
       <reject status="reject"
         reason="No call to Bob is permitted" />
     </address>
     <otherwise>
       <remove-location>
       <address-switch field="origin" subfield="host">
         <address subdomain-of="example.com">
          <location url="sip:chris@office.example.com">
              <proxy />
          </location>
         </address>
         <address subdomain-of="crackers.org">
            <reject status="reject"
                reason="No call from this domain is permitted" />
         </address>
         <address subdomain-of="instance.net">
           <location url="sip:bob@home.org">
             <redirect />
           </location>
         </address>
         <otherwise>
           <location url="sip:chris@voicemail.example.com">
             <proxy />
           </location>
         </otherwise>
       </address-switch>
       </remove-location>
```

```
    </otherwise>
  </address-switch>
</outgoing>
<incoming>
</incoming>
</cpl>
```

Figure 8.3 A combined script from Figure 8.1 and 8.2

In general, Nakamura's research concentrates on semantic ambiguities while ours centres around logical inconsistencies. Nakamura listed eight types of semantic warnings that are possible in individual CPL scripts. We analyse these semantic warnings and compare them with the interactions detected by our rules as follows:

- Multiple Forwarding Addresses (MFAD): More than one <location> tags are set before an action <proxy> or <redirect>. We did not take this into consideration since it is more a compilation-time warning than a logical inconsistency.
- Unused SUBactions (USUB): A subaction is defined but not used. Again, this is more a compilation-time warning than a logical inconsistency.
- Call Rejection in All Paths (CRAP): All execution paths terminate at <reject>. This is not necessarily a semantic error since users may block all calls in some cases, in addition, it is certainly not a logical inconsistency.
- Address Set after Address Switch (ASAS): As mentioned above, this scenario describes a conflict between two local intentions: the first intention blocks calls to a particular destination but the second one forwards a call to the same destination. If both intentions are located in the same incoming policy, it cannot be said that there is a conflict: we could block all incoming calls from user A while forwarding other calls to A. In fact, this conflict only occurs in a very special case where outgoing calls to A are blocked while other outgoing calls are forwarded to A. Our current local inconsistency detection rules do not cover this special case; however, it will be well handled if our basic rule #1 is extended to the case of single scripts.

- Overlapped Conditions in Single Switch (OCSS): This describes the scenario of Feature Shadowing which can be detected by our rule #L2 (see Section 4.3).
- Identical Actions in Single Switch (IASS): it describes the situation of Feature Redundancy which can be detected by our rule #L1 (see Section 4.2.2).
- Overlapped Conditions in Nested Switches (OCNS): This describes the same situation of redundant conditions as we discussed in Section 4.2.2; however, as our example in Section 4.2.2 shows, indirect condition redundancy also could occur between two different switches, which is not recognised by the definition of OCNS.
- Incompatible Conditions in Nested Switches (ICNS): it describes the same situation of unexecutable actions that is discussed in Section 4.2.1.

The above comparison shows the overlap of the two approaches in the area of single CPL scripts. Pair-wise Feature Interactions detected by our rules d1, d2, d3 and i1 are also covered by ASAS and MFAD after the combination of two scripts. However, the interaction between Outgoing Call Forking and Call Forwarding which is detected by our rule d4 is not covered in their work.

8.3.2 The work of Amyot et al.

Amyot et al. proposed an approach to solve interactive conflicts for personalized services in Internet Telephony [1], which is part of a personalized services management architecture. This architecture includes creation of policies for personalized communication services, validation of services, and conflict handling. CPL was chosen as the possible service creation tool, and a translator from CPL to FIAT (see Section 2.2.2) was developed so that potential conflicts can be detected by applying FIAT.

This work covers the complete process of a service lifecycle. It solves most local inconsistencies that we discussed in chapter 4, but pair-wise conflicts, which are the most important part of our work, are not considered.

8.4 Applicability

As we discussed in Section 1.1, our FIs detection method has the potential to significantly improve the quality of the deployment of services that are specified in CPL. Moreover, it could play a key role in personalized services management in Internet Telephony. This section will discuss where and how to apply our method.

PSTN consists of two types of switches: public switches and PBXes (Private Branch Exchange). Unlike PSTN, Internet Telephony is not based on a hierarchical network architecture, and thus it has no centralised services centre. As mentioned in Section 1.3, Users' CPL scripts may be located in signalling servers. On one hand, from the point view of network components, a signalling server acts like a PBX in PSTN (there is no public switch in Internet Telephony); on the other hand, from the point view of feature management, signalling servers are "most similar in functionality to service control or switching points in the circuit-switched network" (SCP and SSP) [30]. Either way, CPL scripts residing in different signalling servers are highly distributed and have no knowledge of each other.

Hence, feature interaction detection could be performed in two stages: offline and online. Offline detection means performing FI detection at the time when features are uploaded to the signalling server but not activated yet. It can be implemented to detect conflicts between features located in the same signalling server (these features could belong to the same user or to different ones). In practice, we may set up such a configuration that signalling servers should run our detection rules after a new feature is uploaded but before it is activated. This will help the user recognise both local Feature Interactions in one user's CPL scripts and pair-wise ones inside one domain.

For potential interactions between two features that are located in different signalling servers, we propose online detection, which means performing FI detection at run time. For instance, suppose that alice@hotmal.com tries to call bob@yahoo.com. It is neither possible nor necessary to scan this pair of CPL scripts beforehand, in consideration of the amount of domains and email addresses on Internet. Therefore, run-time detection

becomes the only feasible solution when users' scripts reside in different domains. Our detection rules would are useful in online detection; however, since performing FI detection requires accessing CPL scripts in different servers, a new concern arises: who should perform this detection, the caller (which is in hotmail.com in this example) or the callee (which is in yahoo.com in this example)? Few people want their calling policies open to the public. For instance, alice@hotmail.com may block calls from bob@yahoo.com, but very possibly, Alice does not want other people, especially Bob, to know this. Thus, in order to perform a "safe" detection, a trusted third party is needed. When a call is initiated, the pre-authorised third party will access the CPL scripts of the caller and callee, and then apply our detection rules to check potential FIs. In the case where a call is forwarded from the original destination to a second one, the detection process will be executed twice, first considering the two scripts of the caller and original callee, then considering those of the caller and the second callee. This complex procedure could be justified for networks where the greatest dependability is required.

8.5 Future Work

Although we have made a number of significant achievements as mentioned in the section of contributions, many improvements are still possible. This section discusses further research directions.

8.5.1 Multi-way Feature Interactions Detection

Our detection rules only handle local and pair-wise Feature Interactions. However, some types of feature interactions only occur when three or more features are combined together although no FI exists between any pair of them. A well known example is a forwarding loop among three users—user A forwards a call to user B, user B forwards it to user C and C forwards it back to A. Obviously, identifying three-way or even n-way FIs is a necessary requirement in practical systems.

It appears that multi-way interactions could be found by generalising the method proposed in this thesis; however, more study is necessary to tackle this problem.

8.5.2 Solutions

Solutions for Feature Interactions are expected after interactions are detected. As we mentioned earlier, Feature Interactions represent logical inconsistencies: for local logical inconsistencies within a CPL script, we need to offer choices to users to "correct" the interactions, in this case, we refer readers to [1] although future work is necessary to adapt their method to ours; for pair-wise interactions detected at run time, negotiation may be necessary. The negotiation between two users could be done automatically by user agents with pre-set preferences and priorities. If negotiation cannot be done because of priority conflicts, we need to find a compromise between the two possibilities of inviting users to make decisions or simply terminating the call. Either way, a global services management model with a trusted third party requires further research.

REFERENCES

[1] D. Amyot, T. Gray, R. Liscano, L. Logrippo, J. Sincennes. "Interactive Conflict Resolution for Personalized Services". Draft, University of Ottawa, 2003.

[2] H.R. Andersen. "An introduction to Binary Decision Diagrams". Lecture Notes, http://www.cs.auc.dk/~kgl/VERIFICATION99/mm4.html, Oct 1997.

[3] I. Bratko. "Prolog Programming for Artificial Intelligence". Addison-Wesley Pub Co, 3rd edition, 2001.

[4] M. Bryan, "Introduction to XML", http://www.personal.u-net.com/~sgml/xmlintro.htm

[5] R.E. Bryant. "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams". ACM Computing Surveys, Vol. 24, No.3 (Sep, 1992), 293-318.

[6] M. Calder, E. Magill, M. Kolberg, S.Reiff-Marganiec. "Feature Interaction: A Critical Review and Considered Forecast". Computer Networks, Volume 41/1, 2003.

[7] E. Cameron, N.D Griffeth, Y.J Lin, M.E Nilson, W.K Schnure. "A Feature Interaction Benchmark for IN and Beyond". IEEE Communications Magazine, Vol. 31, no. 3, 64-69, 1993.

[8] J. Cameron and H. Velthuijsen, "Feature Interactions in Telecommunications Systems". IEEE Communications Magazine, Vol. 31, no. 8, 18-23, August 1993.

[9] E. Clarke, O. Grumberg and D. Long. "Model checking and abstraction". ACM Transactions on Programming Languages and Systems, 16(5):1512-1542, September 1994.

[10] W.F. Clocksin, C.S. Mellish. "Programming in Prolog". Fourth Edition, Springer-Verlag 1994.

[11] S.A. Cook. "The complexity of theorem proving procedures". Proceedings of the Third Annual ACM Symposium on the Theory of Computing, 151-158, ACM, New York, 1971.

[12] A. Felty. "Temporal Logic Theorem Proving and its Application to the Feature Interaction Problem". In E. Giunchiglia and F. Massacci, editors, Issues in the Design and Experimental Evaluation of Systems for Modal and Temporal Logics, Technical Report DII 14/01, University of Siena, Jun 2001.

[13] A. Felty, K.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, 2000. 179-192.

[14] J.R. Fisher, "Prolog Tutorial", http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html.

[15] M.R. Garey, D.S. Johnson. "Computers and Intractability: A Guide to the Theory of NP-Completenes". W. H. Freeman, New York, 1983.

[16] J. Glasmann, W. Kellerer and H. Müller. "Service Architectures in H.323 and SIP–A Comparison", http://www.h323forum.org/papers/Service_Architecures_SIPH323.Pdf.

[17] Nicolas Gorse. "The Feature Interaction Problem: Automatic Filtering of Incoherences & Generation of Validation Test Suites at the Design Stage". Master Thesis of Computer Science, University of Ottawa, 2000.

[18] M. Handley, H. Schulzrinne, and E. Schooler. "SIP: Session initiation protocol", Request for Comments 3261, Internet Engineering Task Force, Jun 2002.

[19] M.R.A. Huth, M.D. Ryan. "Logic in Computer Science: Modelling and reasoning about systems". Cambridge University Press, 2001.

[20] International Engineering Consortium. "Definition and Overview". http://www.iec.org/online/tutorials/int_tele.

[21] International Telecommunication Union, "Principles of Intelligent Architecture", Recommendation Q.1201, Telecommunication Standardization of ITU, Geneva, 1992.

[22] International Telecommunication Union, "Vocabulary of Terms for ISDNs", Recommendation I.112, Telecommunication Standardization of ITU, Geneva, 1989.

[23] International Telecommunications Union, "General recommendations on telephone switching and signalling intelligent network: Introduction to intelligent network capability set 1", Recommendation Q.1211, Telecommunication Standardization of ITU, Switzerland, Mar. 1993.

[24] ITU-T, Recommendation H.323, "Packet-based multimedia communications systems", http://www.itu.int/itudoc/itu-t/rec/h/s_h323_e_55647.html, Jul 2003.

[25] B. Kelly, M. Crowther and J. King. "Feature Interaction Detection Using SDL Models". In the IEEE Global Telecommunications Conference GLOBECOM'94, pp1857-1861, 1994.

[26] G. Klyne, "A Syntax for describing media feature sets", Request For Comments 2533, Internet Engineering Task Force, Mar. 1999.

[27] M. Kolberg, E. H. Magill. "A pragmatic approach to service interaction filtering between call control services". Computer Networks 38 (2002) 591-602.

[28] J. Lennox, H. Schulzrinne. "CPL: A Language for User Control of Internet Telephony Services". draft-ietf-iptel-cpl-06, Internet Draft, Internet Engineering Task Force, Jan 2002.

[29] J. Lennox, H. Schulzrinne. "Call Processing Language Framework and Requirements". Request for Comments 2824, Internet Engineering Task Force, May 2000.

[30] J. Lennox, H. Schulzrinne. "Feature Interaction in Internet Telephony". Proceedings of the Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems, May 2000.

[31] M. Nakamura, P. Leelaprute, K. Matsumoto and T. Kikuno. "Detecting Script-to Script Interactions in Call Processing Language". In D. Amyot and L. Logrippo, editors, Feature Interactions in Telecommunications and Software Systems VII, IOS Press, 2003.

[32] M. Nakamura, P. Leelaprute, K. Matsumoto and T. Kikuno. "Semantic Warnings and Feature Interaction in Call Processing Language on Internet Telephony". In IEEE 2003 International Symposium on Applications and the Internet (SAINT2003), 283-290, Jan. 2003.

[33] R.E. Neapolitan, and K. Namipour. "Foundations of Algorithms". D.C.Heath and Company, Lexington, MA, 1996.

[34] F.G. Pagan. "Formal Specification of Programming Languages: A Panoramic Primer". Prentice Hall, 1981.

[35] D.B Peng and C.F Chang. "Resolving feature interactions in 3d part editing". Computer Aided Design, 29(10): 687-699, Elsevier Science, 1997.

[36] H. Schulzrinne, J. Rosenberg. "Internet Telephony: Architecture and Protocols-an IETF Perspective". Computer Networks, Vol. 31, Issue 3, 237-255, Feb. 1999.

[37] H. Schulzrinne. "A comprehensive multimedia control architecture for the Internet". International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV), (St. Louis, Missouri), May 1997.

[38] J. Wielemaker. http://www.swi-prolog.org. SWI-Prolog.

[39] The seventh International Workshop on Feature Interactions in Telecommunication and Software Systems. http://www.site.uottawa.ca/fiw03.

[40] P. Zave. "FAQ Sheet on Feature Interaction", http://www.research.att.com/~pamela/faq.html.

[41] D.Y. Zhang. "The Deployment of Features in Internet Telephony". Master thesis of Applied Science in EE, Carleton University, 2002.

ACRONYMS

3WC	Three Way Calling
AS	Address Set After Address Switch
CCF	Conditional Call Forward
CFA	Call Forward Always
CFB	Call Forward on Busy
CFO	Call Forking Outgoing
CPL	Call Processing Language
CW	Call Waiting
FE	Functional Entity
FI	Feature Interaction
FIAT	Feature Interaction Analysis Tool
FM	Formal Methods
IASS	Identical Actions in Single Switch
ICS	Incoming Call Screening
IETF	Internet Engineering Task Force
IN	Intelligent Network
INAP	Intelligent Network Application Protocol
IP	Intelligent Peripheral
ITU	International Telecommunications Union
LTL	Linear Temporal Logic
MF	Multiple Forwarding Address
NE	Network Entity
OCS	Outgoing Call Screening
OCSS	Overlapped Conditions in Single Switch
PBX	Private Branch Exchange
PSTN	Public Switched Telephone Network
RTSP	Real Time Streaming Protocol
SCP	Service Control Point
SFSL	Simple Formal Specification Language

SIP	Session Initiation Protocol
SSF	Service Switching Function
SSP	Service Switching Point
XML	eXtensible Markup Language

APPENDIX A: Prolog Code of Translator

```
#!/usr/local/bin/pl

/*************************************************
************************
 * CPL2SFSL
 *
 * This modules combines the individual CPL specifications
generated for
 * each policy and produces a single FIAT description file
(suitable for
 * feature interaction analysis by FIAT).

 * Originally from Jacques Sincennes, and modified by
Yiqun Xu, July 7, 2003

*/

/*************************************************
************************
 * Additionnal Modules
*/
%:- [time].                              % predicates for
"time" operations (e.g. dates)

/*************************************************
************************
 * Declarations of Dynamic Predicates
*/
:- dynamic
        afeature/5,                      % to rebuild
"otherwise" statements
        dirname/1,                       % directory of
CPL specifications (hardcoded, testing)
        filename/1,                              %
CPL specification file (hardcoded, testing)
        featureNumber/1,      % numbering of multi-
definitions features
        %mytype/253,                     % indicates the
type of "sub-branch" (i.e. incoming or outgoing)
        error/4.                         % to catch
parser warnings & errors

/*************************************************
************************
 * Main predicate
*/
%cpl2fiat :-                                  % modified on
Jul 7, 2003
cpl2sfsl :-
  tell(features),                % output to file 'features'
  doall,                                      % proceed
  told,                                       % close output
file
  halt.                                       % terminate

/*************************************************
************************
 * Sample CPL files in the "Tests" sub-directory (for testing
purposes)
*/

%dirname('C:\yixu\prolog').
dirname('C:\\yixu\\prolog\\cpl2sfsl_sample').
%filename('Tests/CF-redirect-default-tom.cpl').

/*************************************************
************************
 * Loading of an XML (CPL Specification) file
*/

loadCplSpecification(FN, Term) :-
        new_dtd(cpl,DTD),
        load_dtd(DTD, 'C:\\yixu\\prolog\\cpl.dtd'),
        load_structure(FN, Term, [ dialect(xml),
dtd(DTD), file(errorFile) ]),
        free_dtd(DTD),
        !,
        (error(Dialect, File, Line, Message) ->
                writef( 'SGML2PL(%w): %w:%w:
%w\n',[Dialect, File, Line, Message] ),
                retract(error(_,_,_,_)),
                fail
        | otherwise ->
                true
        ).

/*************************************************
************************
```

```
* specname/1
*
* Providing a directory name (path), specname/1 returns
(one at a time)
* the name of a CPL file contained therein.
* Mostly for testing purposes, a set of filenames (filename/1)
and
* specification directories (dirname/1) may be hardcoded.
*/

% Normal use:  Path of CPL directory provided as argument
specname(Name) :-
        % *VERY CRUDE* analysis of command line
(ouch)
        ( unix(argv([_,_,_,Path]))          % interpreted
        | unix(argv([_,_,_,_,Path]))        % compiled
        | unix(argv([_MessDosPath,Path]))           %
**WARNING** for MESSDOS
        ), Path \= '--', !,
        ( exists_directory(Path) ->
                specdir(Path, CPL_list),
                member(Name,CPL_list)
        | exists_file(Path) ->
                Name = Path
        | otherwise ->
                write('What is that argument: '),
write(Path), write_ln('?'), halt
        ).

% For testing purposes:  hardcoded CPL specification
filename
% (in case no Path was given as argument)
specname(Name) :-
        filename(Name).

% For testing purposes:  hardcoded CPL specification
directory
% (No Path was given as argument.  Complementary to
filename/1.)
specname(Name) :-
        specdir(CPL_list),
        member(Name,CPL_list).

specdir(CPL_list) :-
        dirname(DN),
        concat_atom([DN,'*.cpl'],'/',Pattern),
        expand_file_name(Pattern,CPL_list), !.

specdir(DN, CPL_list) :-
        concat_atom([DN,'*.cpl'],'/',Pattern),
        expand_file_name(Pattern,CPL_list), !.

% Strip sequence number and extension
% CPL filenames encode feature policy as well as priority
descramble_filename(FN,FeaturesName) :-
        file_base_name(FN,FileName),
        file_name_extension(NameNoExt,cpl,FileName),
        (
file_name_extension(NameNoExtNoSeqNo,SeqNoAtom,Na
meNoExt),
         SeqNoAtom \= '',
         atom_to_term(SeqNoAtom,SeqNo,_bindings),
         number(SeqNo) ->
                FeaturesName1 =
NameNoExtNoSeqNo
        | otherwise ->
                FeaturesName1 = NameNoExt
        ),
        (
atom_chars(FeaturesName1,[N1,N2,'_'|RestName]),
         atom_to_term(N1,TN1,_), number(TN1),
         atom_to_term(N2,TN2,_), number(TN2) ->
                atom_chars(FeaturesName, RestName)
        | otherwise ->
                FeaturesName = FeaturesName1
        ), !.

rebuild_otherwise(FN,Trigs) :-
        ( negatePreviousTriggers(FN,Trigs) -> true
        | otherwise -> Trigs = []
        ), !.

negatePreviousTriggers(NameNoExtNoSeqNo,TrigsOut) :-
        bagof(Trig,
N^afeature(NameNoExtNoSeqNo,N,_PreCdns,Trig,_res),
Trigs),
        notsy(Trigs,TrigsOut), !.

notsy([],[]) :- !.
notsy([A|B],New) :-
        notsy2(A,First),
        notsy(B,Rest),
        merge_set(First,Rest,New), !.
```

```
notsy2([],[]) :- !.
notsy2([A|B],[A|Rest]) :-
        notsy2(B,Rest), !.

doall :-
        initFeatNo,
        specname(FN),
        loadCplSpecification(FN, Struct),
        %parseElement_cpl(Struct, [], Cdn, [], Trig, [],
Res),
        getFeatNo(No),
        descramble_filename(FN,FN2),
        rebuild_otherwise(FN2,NotTrigPrevious),
        parseElement_cpl(FN2, Struct, [], Cdn, [], Trig,
[], Res),% by yiqun
        printMe(FN2, No, Cdn, Trig, Res,
NotTrigPrevious),
%       ( member(time(TimeSpec),Cdn) ->
%               findPeriod(TimeSpec,Period),
write_ln(period(Period))
%       | otherwise ->
%               true
%       ),
%       ( member(time(TimeSpec),Trig) ->
%               findPeriod(TimeSpec,Period),
write_ln(period(Period))
%       | otherwise ->
%               true
%       ),
%       ( member(time(TimeSpec),Res) ->
%               findPeriod(TimeSpec,Period),
write_ln(period(Period))
%       | otherwise ->
%               true
%       ),
        fail.

doall :-
        nl,
        %write_ln(':- [time].'),nl,
        %write_ln('contradiction_pair(time(T1),time(T2))
:-\n   notOverlap(T1,T2).'), nl,
        %write_ln('contradiction_pair(proxy(A),proxy(B)
) :-\n   A \\= B.'), nl,
        %write_ln('contradiction_pair(proxy(_A),reject(_
B)) :-\n   true.'), nl,
        %write_ln('notOverlap(T1,T2) :-\n   (
overlap(T1,T2), !, fail\n   | !, true).'),
        nl.


% Initialize featureNumber/1 (to 0)
initFeatNo :-
        ( retract(featureNumber(_)) | true ),
        assert(featureNumber(0)), !.

% Obtain a featureNumber (then increment featureNumber
for next call)
getFeatNo(N2) :-
        featureNumber(N),
        ( retract(featureNumber(_)) | true ),
        N2 is N + 1,
        assert(featureNumber(N2)),
        !.

/*************************************************
**********************
 *      NB: 'encoding="US-ASCII"' gives an error (using
ISO-8859-1 instead)

*************************************************
*********************/

/*************************************************
**********************
 * cpl

  <!-- The top-level element of the script. -->

  <!ELEMENT cpl  (
%Ancillary;,%Subactions;,%TopLevelActions; ) >

*************************************************
*********************/

parseElement_cpl(FN2, [element('cpl', _Attr, List)|_], CdnIn,
Cdn, TrigIn, Trig, ResIn, Res) :-
        parseEntity_Ancilliary(List, List2, _CdnIn2,
_Cdn2, _TrigIn2, _Trig2, _ResIn2, _Res2),
        parseEntity_Subactions(List2, List3, _CdnIn3,
_Cdn3, _TrigIn3, _Trig3, _ResIn3, _Res3),
```

```
        parseEntity_TopLevelActions(FN2, List3, CdnIn,
Cdn, TrigIn, Trig, ResIn, Res).

% NB: ***NOT QUITE STANDARD*** JS
parseElement_cpl(FN2, [_|Tail], CdnIn, Cdn, TrigIn, Trig,
ResIn, Res) :-
        parseElement_cpl(FN2, Tail, CdnIn, Cdn, TrigIn,
Trig, ResIn, Res).

/*************************************************
**********************
 * Ancillary data

  <!ENTITY % Ancillary 'ancillary?' >

  <!ELEMENT ancillary EMPTY >

*************************************************
*********************/

% ancillary present:
parseEntity_Ancilliary([element('ancillary', [], [])|Rest], Rest,
_PIn, _P, _TIn, _T, _RIn, _R) :- !.

% ancillary absent:
parseEntity_Ancilliary(List, List, _CdnIn, _Cdn, _TrigIn,
_Trig, _ResIn, _Res) :- !.

/*************************************************
**********************
 * Subactions

  <!ENTITY % Subactions 'subaction*' >

  <!ELEMENT subaction ( %Node; )>
  <!ATTLIST subaction
    id          ID      #REQUIRED
  >

*************************************************
*********************/

% subaction present:
parseEntity_Subactions([element(subaction, [id=ID],
Node)|Tail], Rest, P1, P2, T1, T2, R1, R2) :-
        parseEntity_Node(Node, [], Cdn, [], Trig, [], Res),
        assert(sub(ID, Cdn, Trig, Res)), !,
        parseEntity_Subactions(Tail, Rest, P1, P2, T1, T2,
R1, R2).

% subaction absent (or end of list):
parseEntity_Subactions(List, List, Cdn, Cdn, Trig, Trig, Res,
Res) :- !.

/*************************************************
**********************
 * Top-level Actions

  <!ENTITY % TopLevelActions 'outgoing?,incoming?' >

  <!ELEMENT outgoing ( %Node; )>

  <!ELEMENT incoming ( %Node; )>
*************************************************
*********************/

% neither outgoing nor incoming are present:
parseEntity_TopLevelActions(_FN2, [], Cdn, Cdn, Trig,
Trig, Res, Res) :- !.

% process outgoing:

parseEntity_TopLevelActions(FN2, [element('outgoing', [],
List)|_Tail], _noPIn, Cdn, _noTIn, conj(outgoing(FN2, _X),
Trig), _noRIn, Res) :-
        %retract(mytype(_)),
assert(mytype(outgoing,FN2)),
        parseEntity_Node(List, [], Cdn, [], Trig, [], Res).

% outgoing was processed above; now get ready to process
'incoming':
parseEntity_TopLevelActions(FN2, [element('outgoing',
_Arg, _List)|Tail], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        %retract(mytype(_)),
assert(mytype(out,h,m,s,mm)),
        !, parseEntity_TopLevelActions2(FN2, Tail,
CdnIn, Cdn, TrigIn, Trig, ResIn, Res).

% there was no outgoing; get ready to process 'incoming':
parseEntity_TopLevelActions(FN2, List, CdnIn, Cdn, TrigIn,
Trig, ResIn, Res) :-
        !, parseEntity_TopLevelActions2(FN2, List,
CdnIn, Cdn, TrigIn, Trig, ResIn, Res).
```

```
% process incoming:

parseEntity_TopLevelActions2(FN2, [element('incoming', [],
List)|[]], _noPIn, Cdn, _noTIn, conj(incoming(_X,
FN2),Trig), _noRIn, Res) :-
        %retract(mytype(_)),
assert(mytype(incoming,FN2)),
        parseEntity_Node(List, [], Cdn, [], Trig, [], Res).

/*************************************************
**********************
 * Nodes

  <!-- Nodes are one of the above four categories, or a
subaction.
     This entity (macro) describes the contents of an output.
     Note that a node can be empty, implying default action.
-->
  <!ENTITY % Node
'(%Location;|%Switch;|%SignallingAction;|%OtherAction;|
%Sub;)?' >
*************************************************
**********************/

parseEntity_Node([], Cdn, Cdn, Trig, Trig, Res, Res) :- !.

% CAUTION:  SAMPLE ONLY; FUNCTIONNALITY
NOT GARANTEED :-)
%parseEntity_Node([First|Rest], Cdn, Cdn, Trig, Trig, Res,
Res) :-
%       parseEntity_Node(First, Cdn, Cdn, Trig, Trig,
Res, Res) :-
%       parseEntity_Node(Rest, Cdn, Cdn, Trig, Trig,
Res, Res) :-

parseEntity_Node(Struct, CdnIn, Cdn, TrigIn, Trig, ResIn,
Res) :-
        ( parseEntity_Switch(Struct, CdnIn, Cdn, TrigIn,
Trig, ResIn, Res)
        | parseEntity_Location(Struct, CdnIn, Cdn,
TrigIn, Trig, ResIn, Res)
        | parseEntity_SignallingAction(Struct, CdnIn,
Cdn, TrigIn, Trig, ResIn, Res)
        | parseEntity_OtherAction(Struct, CdnIn, Cdn,
TrigIn, Trig, ResIn, Res)
        | parseEntity_Sub(Struct, CdnIn, Cdn, TrigIn,
Trig, ResIn, Res)
        ).

/*************************************************
**********************
 * Switch Nodes
  <!-- Switches: choices a CPL script can make. -->

  <!ENTITY % Switch 'address-switch|string-switch|time-
switch|priority-switch' >

*************************************************
**********************/

%parseEntity_Switch([element('address-switch', _noAttr,
Sub)|[]], CdnIn, Cdn, TrigIn, (address-
switch(X._noAttr.Trig), ResIn, Res) :-
%       parseElement_addressSwitch(Sub, CdnIn, Cdn,
TrigIn, Trig, ResIn, Res).

% by Yiqun

parseEntity_Switch([element('address-switch', _noAttr,
Sub)|[]], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        parseElement_addressSwitch(Sub, CdnIn, Cdn,
TrigIn, Trig, ResIn, Res).

parseEntity_Switch([element('string-switch', _noAttr,
Sub)|[]], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        parseElement_stringSwitch(Sub, CdnIn, Cdn,
TrigIn, Trig, ResIn, Res).

parseEntity_Switch([element('time-switch', _noAttr, Sub)|[]],
CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        parseElement_timeSwitch(Sub, CdnIn, Cdn,
TrigIn, Trig, ResIn, Res).

parseEntity_Switch([element('priority-switch', _noAttr,
Sub)|[]], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        parseElement_prioritySwitch(Sub, CdnIn, Cdn,
TrigIn, Trig, ResIn, Res).

/*************************************************
**********************
 * all switches can have an otherwise (as the last 'case')

  <!-- All switches can have an 'otherwise' output. -->
  <!ELEMENT otherwise ( %Node; ) >
```

```
**********************************************
**********************/

parseElement_otherwise(element(otherwise, [], List), CdnIn,
Cdn, TrigIn, Trig, ResIn, otherwise(Res)) :-
        parseEntity_Node(List, CdnIn, Cdn, TrigIn, Trig,
ResIn, Res).

/**********************************************
**********************
 * all switches can have a not-present (just one, but
anywhere!!!)

  <!-- All switches can have a 'not-present' output. -->
  <!ELEMENT not-present ( %Node; ) >

**********************************************
**********************/

parseElement_notPresent(element('not-present', [], List),
CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        parseEntity_Node(List, CdnIn, Cdn, TrigIn, Trig,
ResIn, Res).

/**********************************************
**********************
  <!-- Address-switch makes choices based on addresses. -->
  <!ELEMENT address-switch ( (address|not-present)+,
otherwise? ) >
  <!-- <not-present> must appear at most once -->
  <!ATTLIST address-switch
    field       CDATA   #REQUIRED
    subfield    CDATA   #IMPLIED
  >

**********************************************
**********************/

parseElement_addressSwitch([element(address, Attr,
Node)|Tail], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        ( parseElement_address(element(address, Attr,
Node), CdnIn, Cdn, TrigIn, Trig, ResIn, Res)
        | !, parseElement_addressSwitch1(Tail, CdnIn,
Cdn, TrigIn, Trig, ResIn, Res)
        ).

parseElement_addressSwitch([element('not-present', [],
Node)|Tail], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        ( parseElement_notPresent(element('not-present',
[], Node), CdnIn, Cdn, TrigIn, Trig, ResIn, Res)
        | !, parseElement_addressSwitch2(Tail, CdnIn,
Cdn, TrigIn, Trig, ResIn, Res)
        ).

parseElement_addressSwitch1([element(address, Attr,
Node)|Tail], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        ( parseElement_address(element(address, Attr,
Node), CdnIn, Cdn, TrigIn, Trig, ResIn, Res)
        | !, parseElement_addressSwitch1(Tail, CdnIn,
Cdn, TrigIn, Trig, ResIn, Res)
        ).
parseElement_addressSwitch1([element('not-present', [],
Node)|Tail], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        ( parseElement_notPresent(element('not-present',
[], Node), CdnIn, Cdn, TrigIn, Trig, ResIn, Res)
        | !, parseElement_addressSwitch2(Tail, CdnIn,
Cdn, TrigIn, Trig, ResIn, Res)
        ).
parseElement_addressSwitch1([element('otherwise', Attr,
Node)|_Tail], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        parseElement_otherwise(element('otherwise',
Attr, Node), CdnIn, Cdn, TrigIn, Trig, ResIn, Res), !.

parseElement_addressSwitch2([element('address', Attr,
Node)|Tail], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        ( parseElement_address(element('address', Attr,
Node), CdnIn, Cdn, TrigIn, Trig, ResIn, Res)
        | !, parseElement_addressSwitch2(Tail, CdnIn,
Cdn, TrigIn, Trig, ResIn, Res)
        ).
parseElement_addressSwitch2([element('otherwise', Attr,
Node)|_Tail], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        parseElement_otherwise(element('otherwise',
Attr, Node), CdnIn, Cdn, TrigIn, Trig, ResIn, Res), !.

/**********************************************
**********************
  <!ELEMENT address ( %Node; ) >

  <!ATTLIST address
    is          CDATA   #IMPLIED
    contains    CDATA   #IMPLIED
    subdomain-of CDATA  #IMPLIED
```

```
  > <!-- Exactly one of these three attributes must appear -->
**************************************************
**********************/

parseElement_address(element(address, Attr, Node), CdnIn,
Cdn, TrigIn, Trig, ResIn, Res) :-
        procAttr(address, Attr, TrigIn, Address),
        parseEntity_Node(Node, CdnIn, Cdn, Address,
Trig, ResIn, Res).

/**************************************************
**********************
  <!-- String-switch makes choices based on strings. -->

  <!ELEMENT string-switch ( (string|not-present)+,
otherwise? ) >
  <!-- <not-present> must appear at most once -->
  <!ATTLIST string-switch
    field       CDATA   #REQUIRED
  >

**************************************************
**********************/

parseElement_stringSwitch([element(string, Attr,
Node)|Tail], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        ( parseElement_string(element(string, Attr,
Node), CdnIn, Cdn, TrigIn, Trig, ResIn, Res)
        | !, parseElement_stringSwitch1(Tail, CdnIn, Cdn,
TrigIn, Trig, ResIn, Res)
        ).
parseElement_stringSwitch([element('not-present', [],
Node)|Tail], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        ( parseElement_notPresent(element('not-present',
[], Node), CdnIn, Cdn, TrigIn, Trig, ResIn, Res)
        | !, parseElement_stringSwitch2(Tail, CdnIn, Cdn,
TrigIn, Trig, ResIn, Res)
        ).

parseElement_stringSwitch1([element(string, Attr,
Node)|Tail], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        ( parseElement_string(element(string, Attr,
Node), CdnIn, Cdn, TrigIn, Trig, ResIn, Res)
        | !, parseElement_stringSwitch1(Tail, CdnIn, Cdn,
TrigIn, Trig, ResIn, Res)
        ).
parseElement_stringSwitch1([element('not-present', [],
Node)|Tail], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        ( parseElement_notPresent(element('not-present',
[], Node), CdnIn, Cdn, TrigIn, Trig, ResIn, Res)
        | !, parseElement_stringSwitch2(Tail, CdnIn, Cdn,
TrigIn, Trig, ResIn, Res)
        ).
parseElement_stringSwitch1([element('otherwise', Attr,
Node)|_Tail], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        parseElement_otherwise(element('otherwise',
Attr, Node), CdnIn, Cdn, TrigIn, Trig, ResIn, Res), !.

parseElement_stringSwitch2([element('string', Attr,
Node)|Tail], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        ( parseElement_string(element('string', Attr,
Node), CdnIn, Cdn, TrigIn, Trig, ResIn, Res)
        | !, parseElement_stringSwitch2(Tail, CdnIn, Cdn,
TrigIn, Trig, ResIn, Res)
        ).
parseElement_stringSwitch2([element('otherwise', Attr,
Node)|_Tail], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        parseElement_otherwise(element('otherwise',
Attr, Node), CdnIn, Cdn, TrigIn, Trig, ResIn, Res), !.

/**************************************************
**********************
  <!ELEMENT string ( %Node; ) >
  <!ATTLIST string
    is          CDATA   #IMPLIED
    contains    CDATA   #IMPLIED
  > <!-- Exactly one of these two attributes must appear -->

**************************************************
**********************/

parseElement_string(element(string, Attr, Node), CdnIn,
Cdn, TrigIn, Trig, ResIn, Res) :-
        procAttr(string, Attr, TrigIn, String),
        parseEntity_Node(Node, CdnIn, Cdn, String,
Trig, ResIn, Res).

/**************************************************
**********************
  <!-- Time-switch makes choices based on the current time.
-->
```

```
  <!ELEMENT time-switch ( (time|not-present)+, otherwise?
) >
  <!ATTLIST time-switch
    tzid        CDATA    #IMPLIED
    tzurl       CDATA    #IMPLIED
  >
************************************************
**********************/

parseElement_timeSwitch([Time|_Tail], CdnIn, Cdn, TrigIn,
Trig, ResIn, Res) :-
            parseElement_time(Time, CdnIn, Cdn, TrigIn,
Trig, ResIn, Res).
parseElement_timeSwitch([_Time|Tail], CdnIn, Cdn, TrigIn,
Trig, ResIn, Res) :-
            !, parseElement_timeSwitch(Tail, CdnIn, Cdn,
TrigIn, Trig, ResIn, Res).
parseElement_timeSwitch([element('not-present', [],
Node)|Tail], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
            ( parseElement_notPresent(element('not-present',
[], Node), CdnIn, Cdn, TrigIn, Trig, ResIn, Res)
            | !, parseElement_timeSwitch2(Tail, CdnIn, Cdn,
TrigIn, Trig, ResIn, Res)
            ).
parseElement_timeSwitch(Otherwise, CdnIn, Cdn, TrigIn,
Trig, ResIn, Res) :-
            parseElement_otherwise(Otherwise, CdnIn, Cdn,
TrigIn, Trig, ResIn, Res), !.
parseElement_timeSwitch2([Time|_Tail], CdnIn, Cdn,
TrigIn, Trig, ResIn, Res) :-
            parseElement_time(Time, CdnIn, Cdn, TrigIn,
Trig, ResIn, Res).
parseElement_timeSwitch2([_Time|Tail], CdnIn, Cdn,
TrigIn, Trig, ResIn, Res) :-
            !, parseElement_timeSwitch2(Tail, CdnIn, Cdn,
TrigIn, Trig, ResIn, Res).
parseElement_timeSwitch2(Otherwise, CdnIn, Cdn, TrigIn,
Trig, ResIn, Res) :-
            parseElement_otherwise(Otherwise, CdnIn, Cdn,
TrigIn, Trig, ResIn, Res), !.

/************************************************
**********************
  <!ELEMENT time ( %Node; ) >

  <!-- Exactly one of the two attributes "dtend" and
"duration" must occur. -->

  <!-- The value of "freq" is (daily|weekly|monthly|yearly).
It is
       case-insensitive, so it is not given as a DTD switch. --
>
  <!-- None of the attributes following freq are meaningful
unless freq appears. -->
  <!-- The value of "wkst" is (MO|TU|WE|TH|FR|SA|SU).  It
is
       case-insensitive, so it is not given as a DTD switch. --
>
  <!ATTLIST time
    dtstart     CDATA #REQUIRED        % date|date-time
    dtend       CDATA #IMPLIED                 %
date|date-time
    duration    CDATA #IMPLIED                 %
duration
    freq        CDATA #IMPLIED                 %
(daily|weekly|monthly|yearly)
    until       CDATA #IMPLIED                 %
date|date-time to cease; forever if absent
    interval    CDATA "1"                      %
every <freq> (e.g. every month, if monthly)
    byday       CDATA #IMPLIED                 %
comma-separated list of weekdays


                                               %
e.g. +2MO -> second Monday, when monthly
    bymonthday  CDATA #IMPLIED                 %
comma-separated list of days of the month (-1-31)


                                               %
e.g. -10 -> from 10 til end of month
    byyearday   CDATA #IMPLIED                 %
comma-separated list of days of the year (-1-366)


                                               %
e.g. -306 -> from 306 til end of year
    byweekno    CDATA #IMPLIED                 %
comma-separated list of weeks of the year (-1-53)
    bymonth     CDATA #IMPLIED                 %
comma-separated list of months of the year (1-12)
    wkst        CDATA "MO"                     %
the day the work week starts
  >
************************************************
**********************/
```

```
parseElement_time(element(time, Attr, Node), CdnIn, Cdn2,
TrigIn, Trig, ResIn, Res) :-
%%      procAttr(time, Attr, CdnIn, Time),
%% TEST js
        procAttrTime(Attr, Time2),                  %
assert(time(Time2)),
        parseEntity_Node(Node, [time(Time2)|CdnIn],
Cdn2, TrigIn, Trig, ResIn, Res).

procAttrTime([], []) :- !.

procAttrTime([AttrName=Value|Tail], [Attr|Rest]) :-
        ( AttrName == 'duration' ->
                durationParse(Value,Duration),
                Attr = duration(Duration)
        | AttrName == 'byday' ->
                splitDays(Value,ByDays),
                Attr = byday(ByDays)
        | AttrName == 'dtstart' ->
                date(Value,DateTimeSpec),
                Attr = dtstart(DateTimeSpec)
        | AttrName == 'dtend' ->
                date(Value,DateTimeSpec),
                Attr = dtend(DateTimeSpec)
        | AttrName == 'until' ->
                date(Value,DateTimeSpec),
                Attr = until(DateTimeSpec)
        | otherwise ->
                Attr =.. [AttrName, Value]
        ),
        procAttrTime(Tail, Rest), !.

/************************************************
*****************/
/*
 * durationParse
 *
 * input: duration string (e.g. 'PT24H' )
 * output: 4-elements list (e.g. [Days, Hours, Minutes,
Seconds] )
 * NB: *MUST* succeed; failure indicates error in format
*/

/************************************************
*********
 * iCalendar Grammar according to RFC-2445
 *
 * dur-value  = (["+"] / "-") "P" dur-DTW
 * dur-DTW    = (dur-date / dur-time / dur-week)
 * dur-date   = dur-day [dur-time]
 * dur-time   = "T" (dur-hour / dur-minute / dur-second)
 * dur-week   = 1*DIGIT "W"
 * dur-hour   = 1*DIGIT "H" [dur-minute]
 * dur-minute = 1*DIGIT "M" [dur-second]
 * dur-second = 1*DIGIT "S"
 * dur-day    = 1*DIGIT "D"

************************************************
********/

durationParse(DurationString, Duration) :-
        atom_chars(DurationString, DurationTerm),
%       ( mytype(incoming) ->
%               do this
%       | mytype(outgoing) ->
%               do that
%       | otherwise ->
%               major bug in code
%       )

        durationParseValue(DurationTerm, Rest, [0, 0, 0,
0], Duration),
        (nonvar(Rest), Rest \= [] ->
                atom_chars(R, Rest),
                write_ln(['ERROR: duration format ',
DurationString, ' could not parse: ', R]),
                !, fail
        | otherwise ->
                true
        ), !.

durationParse(A, _) :-
        write_ln(['ERROR: duration format: ', A]), !, fail.

/*
 * NB:  this could accept more than one '+', '-', or even both...
*/
durationParseValue(['+'|Tail], Rest, Duration1, Duration2) :-
        durationParseValue(Tail, Rest, Duration1,
Duration2).          /* IGNORE POSITIVE SIGN */

durationParseValue(['-'|Tail], Rest, Duration1, Duration2) :-
        durationParseValue(Tail, Rest, Duration1,
Duration2).          /* IGNORE NEGATIVE VALUES */
```

```
durationParseValue(['P'|Tail], Rest, Duration1, Duration2) :-
        durationParseDTW(Tail, Rest, Duration1,
Duration2).          /* IGNORE "P" CHARACTER */

/*
 *
*/
durationParseDTW(List, Rest, Duration1, Duration2) :-
        ( durationParseDate(List, Rest, Duration1,
Duration2)
        | durationParseTime(List, Rest, Duration1,
Duration2)
        | durationParseWeek(List, Rest, Duration1,
Duration2)
        ).

/*
 * NB:  only WEEK or DAYS can be used so input not
considered
*/
durationParseWeek(List, Rest, [_, H, M, S], [Days, H, M, S])
:-
        parseDurationDigits(List, Weeks, 'W', Rest),
        Days is Weeks * 7.

/*
 *
*/
durationParseDate(List, Rest2, Duration1, Duration2) :-
        durationParseDay(List, Rest, Duration1,
DurationDay),
        ( durationParseTime(Rest, Rest2, DurationDay,
Duration2) ->
                true
        | otherwise ->
                Duration2 = DurationDay,
                Rest2 = Rest
        ).

durationParseDay(List, Rest, [_, H, M, S], [Days, H, M, S]) :-
        parseDurationDigits(List, Days, 'D', Rest).

/*
 *
*/
durationParseTime(['T'|Tail], Rest, [D1, H1, M1, S1], [D2,
H2, M2, S2]) :-
        ( durationParseHours(Tail, Rest, [D1, H1, M1,
S1], [D2, H2, M2, S2])
        | durationParseMinutes(Tail, Rest, [D1, H1, M1,
S1], [D2, H2, M2, S2])
        | durationParseSeconds(Tail, Rest, [D1, H1, M1,
S1], [D2, H2, M2, S2])
        ).

durationParseHours(List, Rest2, [D, _, M1, S1], [D, Hours,
M2, S2]) :-
        parseDurationDigits(List, Hours, 'H', Rest),
        ( durationParseMinutes(Rest, Rest2, [D, _, M1,
S1], [D, _, M2, S2]) ->
                true
        | otherwise ->
                M2 = M1,
                S2 = S1,
                Rest2 = Rest
        ).

durationParseMinutes(List, Rest2, [D, H, _, S1], [D, H, Mins,
S2]) :-
        parseDurationDigits(List, Mins, 'M', Rest),
        ( durationParseSeconds(Rest, Rest2, [_, _, _, S1],
[_, _, _, S2]) ->
                true
        | otherwise ->
                S2 = S1,
                Rest2 = Rest
        ).

durationParseSeconds(List, Rest, [D, H, M, _], [D, H, M,
Secs]) :-
        parseDurationDigits(List, Secs, 'S', Rest).


/*
 * generic predicate to parse " 1*DIGITS <LETTER> "
 * input: string to parse and "letter"
 * output: number (1*DIGITS) and remaining string (after
<LETTER>)
*/

parseDurationDigits(List, Digits, Letter, Rest) :-
```

```
        parseDurationDigits1(List, DigitsList, Letter,
Rest),
        atom_chars(DigitsChars, DigitsList),
        atom_to_term(DigitsChars, Digits, _Binding), !.

parseDurationDigits1([D|Tail], [D|Rest], Letter, More) :-
        member(D, ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']),
        parseDurationDigits2(Tail, Rest, Letter, More).

parseDurationDigits2([Letter|Tail], [], Letter, Tail) :- ! .

parseDurationDigits2([D|Tail], [D|Rest], Letter, More) :-
        member(D, ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']),
        parseDurationDigits2(Tail, Rest, Letter, More).

/*
 * parsing 'MO,TU' as ['MO','TU']
*/
splitDays(DaysListAtoms, DaysList) :-
        atom_chars(DaysListAtoms, AtomList),
        splitDays2(AtomList, DaysList).

splitDays2([], []).

splitDays2([A, B, ','|R], [AB|R2]):-
        atom_chars(AB, [A, B]), !,
        splitDays2(R, R2).

splitDays2([A, B], [AB]):-
        atom_chars(AB, [A, B]), !.

/*
 * usage:
 *
 *                  date('20010118T230000', d(Y, M, D,
H, MIN, S)),
*/

date(DateString, d(Year, Month, Day, Hour, Min, Sec)) :-
        atom_chars(DateString, [Y1, Y2, Y3, Y4, M1,
M2, D1, D2, 'T', H1, H2, Min1, Min2, S1, S2]),
        term_to_atom(Year , [Y1, Y2, Y3, Y4]),
        term_to_atom(Month , [M1, M2]),
        term_to_atom(Day , [D1, D2]),
        term_to_atom(Hour , [H1, H2]),
        term_to_atom(Min , [Min1, Min2]),
        term_to_atom(Sec , [S1, S2]), !.

/*
 * WARNING:  treating dates ending with 'Z' as the 'regular'
ones
*/
date(DateString, d(Year, Month, Day, Hour, Min, Sec)) :-
        atom_chars(DateString, [Y1, Y2, Y3, Y4, M1,
M2, D1, D2, 'T', H1, H2, Min1, Min2, S1, S2,'Z']),
        term_to_atom(Year , [Y1, Y2, Y3, Y4]),
        term_to_atom(Month , [M1, M2]),
        term_to_atom(Day , [D1, D2]),
        term_to_atom(Hour , [H1, H2]),
        term_to_atom(Min , [Min1, Min2]),
        term_to_atom(Sec , [S1, S2]), !.

/*
 * WARNING:  treating dates with LOCAL ZONE ID as the
'regular' ones
*/
date(DateString, d(Year, Month, Day, Hour, Min, Sec)) :-
        atom_chars(DateString, DateChars),
        afterColumn(DateChars, [Y1, Y2, Y3, Y4, M1,
M2, D1, D2, 'T', H1, H2, Min1, Min2, S1, S2]),
        term_to_atom(Year , [Y1, Y2, Y3, Y4]),
        term_to_atom(Month , [M1, M2]),
        term_to_atom(Day , [D1, D2]),
        term_to_atom(Hour , [H1, H2]),
        term_to_atom(Min , [Min1, Min2]),
        term_to_atom(Sec , [S1, S2]), !.

afterColumn([':'|Rest],Rest) :- !, true.
afterColumn([_Char|Rest],AfterCol) :-
        !, afterColumn(Rest,AfterCol).
afterColumn([],_) :- !, fail.

/***********************************************
**********************
  <!-- Priority-switch makes choices based on message
priority. -->

  <!ELEMENT priority-switch ( (priority|not-present)+,
otherwise? ) >
  <!-- <not-present> must appear at most once -->

  <!ENTITY % PriorityVal '(emergency|urgent|normal|non-
urgent)' >
```

```
**********/

parseElement_prioritySwitch([element(priority, Attr,
Node)|Tail], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        ( parseElement_priority(element(priority, Attr,
Node), CdnIn, Cdn, TrigIn, Trig, ResIn, Res)
        | !, parseElement_prioritySwitch1(Tail, CdnIn,
Cdn, TrigIn, Trig, ResIn, Res)
        ).
parseElement_prioritySwitch([element('not-present', [],
Node)|Tail], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        ( parseElement_notPresent(element('not-present',
[], Node), CdnIn, Cdn, TrigIn, Trig, ResIn, Res)
        | !, parseElement_prioritySwitch2(Tail, CdnIn,
Cdn, TrigIn, Trig, ResIn, Res)
        ).

parseElement_prioritySwitch1([element(priority, Attr,
Node)|Tail], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        ( parseElement_priority(element(priority, Attr,
Node), CdnIn, Cdn, TrigIn, Trig, ResIn, Res)
        | !, parseElement_prioritySwitch1(Tail, CdnIn,
Cdn, TrigIn, Trig, ResIn, Res)
        ).
parseElement_prioritySwitch1([element('not-present', [],
Node)|Tail], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        ( parseElement_notPresent(element('not-present',
[], Node), CdnIn, Cdn, TrigIn, Trig, ResIn, Res)
        | !, parseElement_prioritySwitch2(Tail, CdnIn,
Cdn, TrigIn, Trig, ResIn, Res)
        ).
parseElement_prioritySwitch1([element('otherwise', Attr,
Node)|_Tail], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        parseElement_otherwise(element('otherwise',
Attr, Node), CdnIn, Cdn, TrigIn, Trig, ResIn, Res), !.

parseElement_prioritySwitch2([element('priority', Attr,
Node)|Tail], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        ( parseElement_priority(element('priority', Attr,
Node), CdnIn, Cdn, TrigIn, Trig, ResIn, Res)
        | !, parseElement_prioritySwitch2(Tail, CdnIn,
Cdn, TrigIn, Trig, ResIn, Res)
        ).
parseElement_prioritySwitch2([element('otherwise', Attr,
Node)|_Tail], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        parseElement_otherwise(element('otherwise',
Attr, Node), CdnIn, Cdn, TrigIn, Trig, ResIn, Res), !.

/**********
**********

  <!ELEMENT priority ( %Node; ) >

  <!-- Exactly one of these three attributes must appear -->
  <!ATTLIST priority
    less        %PriorityVal;  #IMPLIED
    greater     %PriorityVal;  #IMPLIED
    equal       CDATA          #IMPLIED
  >

**********
**********/

parseElement_priority(element(priority, Attr, Node), CdnIn,
Cdn, TrigIn, Trig, ResIn, Res) :-
        procAttr(priority, Attr, TrigIn, Priority),
        parseEntity_Node(Node, CdnIn, Cdn, Priority,
Trig, ResIn, Res).


/**********
**********
 * Location Nodes

  <!ENTITY % Location 'location|lookup|remove-location' >

**********
**********/

parseEntity_Location([element(location, Attr, Sub)|[]],
CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
        parseElement_location(element(location, Attr,
Sub), CdnIn, Cdn, TrigIn, Trig, ResIn, Res).

parseEntity_Location([element(lookup, Attr, Sub)|[]], CdnIn,
Cdn, TrigIn, Trig, ResIn, Res) :-
        parseElement_lookup(element(lookup, Attr, Sub),
CdnIn, Cdn, TrigIn, Trig, ResIn, Res).

parseEntity_Location([element('remove-location', Attr,
Sub)|[]], CdnIn, Cdn, TrigIn, Trig, ResIn, Res) :-
```

```
        parseElement_removeLocation(element('remove-
location', Attr, Sub), CdnIn, Cdn, TrigIn, Trig, ResIn, Res).

parseEntity_Location([element('location', Attr1,
_)|[element('location', Attr2, _)|_]], _, _, _, _, _, _) :-
        write('ERROR: found two alternative
LOCATION elements (with '),
        write(Attr1),
        write(' and '),
        write(Attr2),
        write(')'), nl,
        abort.

/*************************************************
**********************
  <!-- Locations: ways to specify the location a subsequent
action
     (proxy, redirect) will attempt to contact. -->

  <!ENTITY % Clear  'clear (yes|no) "no"' >

  <!ELEMENT location ( %Node; ) >
  <!ATTLIST location
    url        CDATA   #REQUIRED
    priority   CDATA   #IMPLIED
    %Clear;
  >

*************************************************
*********************/

/*
 * WARNING:  only considering "url" attribute.
*/

parseElement_location(element(location, Attr, Node), CdnIn,
Cdn2, TrigIn, Trig, ResIn, Res) :-
        procAttr(location, Attr, [], Location),
        ( member(location(url(Loca)), Location) ->
                Location2 = [location(Loca)|ResIn]
        | otherwise ->
                Location2 = ResIn
        ),
        parseEntity_Node(Node, CdnIn, Cdn2, TrigIn,
Trig, Location2, Res).

/*************************************************
**********************
  <!ELEMENT lookup ( success,notfound?,failure? ) >
  <!ATTLIST lookup
    source      CDATA   #REQUIRED
    timeout     CDATA   "30"
    use         CDATA   #IMPLIED
    ignore      CDATA   #IMPLIED
    %Clear;
  >


*************************************************
*********************/

/*
 * WARNING:  only considering "source" attribute.
*/

parseElement_lookup(element(lookup, Attr, Node), CdnIn,
Cdn2, TrigIn, Trig, ResIn, Res) :-
        parseElement_lookup1(Node, CdnIn, Cdn2,
TrigIn, Trig, [], ResLook),
        procAttr(lookup, Attr, [], Source),
        ( member(lookup(source(Src)), Source) ->
                Res = [lookup(Src)|[ResLook|ResIn]]
        | otherwise ->
                Res =
[lookup(nosource)|[ResLook|ResIn]]
        ).

/*************************************************
**********************
  <!ELEMENT success  ( %Node; ) >

*************************************************
*********************/
parseElement_lookup1([element('success',_, Node)|_], Pre,
Pre2, TrigIn, Trig, ResIn, success(Res)) :-
        parseEntity_Node(Node, Pre, Pre2, TrigIn, Trig,
ResIn, Res).
parseElement_lookup1([element('success', _, _)|Rest], Pre,
Pre2, TrigIn, Trig, ResIn, Res) :-
        !, parseElement_lookup2(Rest, Pre, Pre2, TrigIn,
Trig, ResIn, Res).
parseElement_lookup1(Proxies, Pre, Pre2, TrigIn, Trig,
ResIn, Res) :-
```

```
        !, parseElement_lookup2(Proxies, Pre, Pre2,
TrigIn, Trig, ResIn, Res).

/*************************************************
**********************
  <!ELEMENT notfound ( %Node; ) >

*************************************************
*********************/
parseElement_lookup2([element('notfound',_, Node)|_], Pre,
Pre2, TrigIn, Trig, ResIn, notfound(Res)) :-
        parseEntity_Node(Node, Pre, Pre2, TrigIn, Trig,
ResIn, Res).
parseElement_lookup2([element('notfound', _, _)|Rest], Pre,
Pre2, TrigIn, Trig, ResIn, Res) :-
        !, parseElement_lookup3(Rest, Pre, Pre2, TrigIn,
Trig, ResIn, Res).
parseElement_lookup2(Proxies, Pre, Pre2, TrigIn, Trig,
ResIn, Res) :-
        !, parseElement_lookup3(Proxies, Pre, Pre2,
TrigIn, Trig, ResIn, Res).

/*************************************************
**********************
  <!ELEMENT failure ( %Node; ) >

*************************************************
*********************/
parseElement_lookup3([element('failure', _, Node)|[]], Pre,
Pre2, TrigIn, Trig, ResIn, failure(Res)) :-
        parseEntity_Node(Node, Pre, Pre2, TrigIn, Trig,
ResIn, Res).

parseElement_lookup3([], Pre, Pre, Trig, Trig, Res, Res) :- !.

/*************************************************
**********************
  <!ELEMENT remove-location ( %Node; ) >
  <!ATTLIST remove-location
    param       CDATA   #IMPLIED
    value       CDATA   #IMPLIED
    location    CDATA   #IMPLIED
  >

*************************************************
*********************/

parseElement_removeLocation(element('remove-location',
Attr, Node), CdnIn, Cdn2, TrigIn, Trig, ResIn, Res) :-
        procAttr(remloc, Attr, ResIn, RemLoc),
        parseEntity_Node(Node, CdnIn, Cdn2, TrigIn,
Trig, RemLoc, Res).


/*************************************************
**********************
  <!-- Signalling Actions: call-signalling actions the script
can take. -->

  <!ENTITY % SignallingAction 'proxy|redirect|reject' >

*************************************************
*********************/

parseEntity_SignallingAction([element(proxy, Attr, Sub)|[]],
CdnIn, Cdn, TrigIn, Trig, ResIn, [Res]) :-
        parseElement_proxy(element(proxy, Attr, Sub),
CdnIn, Cdn, TrigIn, Trig, ResIn, Res).

parseEntity_SignallingAction([element(redirect, Attr,
Sub)|[]], CdnIn, Cdn, TrigIn, Trig, ResIn, [Res]) :-
        parseElement_redirect(element(redirect, Attr,
Sub), CdnIn, Cdn, TrigIn, Trig, ResIn, Res).

parseEntity_SignallingAction([element(reject, Attr, [])|[]],
Cdn, Cdn, Trig, Trig, ResIn, Res) :-
        ( nonvar(Attr), Attr \= [] ->
                procAttr(reject, Attr, ResIn, Res)
        | otherwise ->
                Res = [reject(ResIn)]
        ).

/*************************************************
**********************
 * NB: does this really impose ordering on the children?? JS
 *                  I.e. is <noanswer> allowed to appear
before <busy>?

  <!ELEMENT proxy (
busy?,noanswer?,redirection?,failure?,default? ) >
```

```
  <!-- The default value of timeout is "20" if the <noanswer>
output exists. -->
  <!ATTLIST proxy
    timeout     CDATA    #IMPLIED
    recurse     (yes|no) "yes"
    ordering    CDATA    "parallel"
  >

**********************************************
**********************/
/*
 * WARNING:  NOT TREATING ANY ATTRIBUTE OF
PROXY
*/

parseElement_proxy(element(proxy, Attr, []), _CdnIn,
_Cdn2, TrigIn, TrigIn, ResIn, Res) :-
            procAttr(proxy, Attr, [], Order),
            member(proxy(ordering(Ord)), Order),
            Res=proxy(ResIn, ordering(Ord)).

parseElement_proxy(element(proxy, Attr,
[FirstNode|RestNode]), CdnIn, Cdn2, TrigIn, Trig, ResIn,
Res) :-
            (member(FirstNode, [element(busy, [], _Rest),

            element(noanswer, [], _Rest),

            element(redirection, [], _Rest),
                                    element(failure,
[], _Rest),
                                    element(default,
[], _Rest)]) ->

            parseElement_proxy1([FirstNode|RestNode],
CdnIn, Cdn2, TrigIn, Trig, ResIn, Res)
            |otherwise ->
                  procAttr(proxy, Attr, [], Order),
                  member(proxy(ordering(Ord)), Order),
                  Res=proxy(ResIn, ordering(Ord)),
                  Trig == TrigIn
            ).


/**********************************************
**********************
  <!ELEMENT busy ( %Node; ) >

**********************************************
**********************/
parseElement_proxy1([element(busy, Attr, Node)|_], Pre,
Pre2, TrigIn, Trig, ResIn, Res) :- % by Yiqun
            procAttr(busy, Attr, ResIn, Busy),
            Trig2=busy(Busy),

            parseEntity_Node(Node, Pre, Pre2,
[Trig2|TrigIn], Trig, [], Res).


/**********************************************
**********************
  <!ELEMENT noanswer ( %Node; ) >

**********************************************
**********************/
parseElement_proxy1([element(noanswer, Attr, Node)|_],
Pre, Pre2, TrigIn, Trig, ResIn, Res) :-
            procAttr(noanswer, Attr, ResIn, Noanswer),
            Trig2=noanswer(Noanswer),

            parseEntity_Node(Node, Pre, Pre2,
[Trig2|TrigIn], Trig, [], Res).


/**********************************************
**********************
  <!ELEMENT redirection ( %Node; ) >

**********************************************
**********************/
parseElement_proxy1([element(redirection, Attr, Node)|_],
Pre, Pre2, TrigIn, Trig, ResIn, Res) :-
            procAttr(redirection, Attr, ResIn, Redirection),
            Trig2=redirection(Redirection),

            parseEntity_Node(Node, Pre, Pre2,
[Trig2|TrigIn], Trig, [], Res).

/**********************************************
**********************
  <!-- "failure" repeats from lookup, above. -->
```

```
**********************************************
**********/
parseElement_proxy1([element(failure, Attr, Node)|_], Pre,
Pre2, TrigIn, Trig, ResIn, Res) :-
        procAttr(failure, Attr, ResIn, Failure),
        Trig2=failure(Failure),

        parseEntity_Node(Node, Pre, Pre2,
[Trig2|TrigIn], Trig, [], Res).

/**********************************************
**********
  <!ELEMENT default ( %Node; ) >

**********************************************
**********/
parseElement_proxy1([element(default, Attr, Node)|[]], Pre,
Pre2, TrigIn, Trig, ResIn, Res) :-
        procAttr(default, Attr, ResIn, Default),
        Trig2=default(Default),

        parseEntity_Node(Node, Pre, Pre2,
[Trig2|TrigIn], Trig, [], Res).
/**********************************************
**********
  <!ELEMENT redirect EMPTY >
  <!ATTLIST redirect
    permanent    (yes|no) "no"
  >


**********************************************
**********/

parseElement_redirect(element(redirect, _Attr, _Node), Cdn,
Cdn, Trig, Trig, ResIn, redirect(ResIn)).

/**********************************************
**********
  <!-- Statuses we can return -->

  <!ELEMENT reject EMPTY >
  <!-- The value of "status" is (busy|notfound|reject|error), or
a SIP 4xx-6xx status. -->
  <!ATTLIST reject
    status      CDATA   #REQUIRED
    reason      CDATA   #IMPLIED
  >

**********************************************
**********/

/**********************************************
**********
 * Other Actions
  <!-- Non-signalling actions: actions that don't affect the call
-->

  <!ENTITY % OtherAction 'mail|log' >

**********************************************
**********/

parseEntity_OtherAction(Struct, CdnIn, Cdn, TrigIn, Trig,
ResIn, Res) :-
        ( parseElement_mail(Struct, CdnIn, Cdn, TrigIn,
Trig, ResIn, Res)
        | parseElement_log(Struct, CdnIn, Cdn, TrigIn,
Trig, ResIn, Res)
        ).

/**********************************************
**********

  <!ELEMENT mail ( %Node; ) >
  <!ATTLIST mail
    url         CDATA   #REQUIRED
  >

**********************************************
**********/

parseElement_mail(element(mail, Attr, Node), CdnIn, Cdn2,
TrigIn, Trig, ResIn, mail(Res)) :-
        procAttr(mail, Attr, CdnIn, Mail),
        parseEntity_Node(Node, Mail, Cdn2, TrigIn,
Trig, ResIn, Res).

/**********************************************
**********
  <!ELEMENT log ( %Node; ) >
  <!ATTLIST log
    name        CDATA   #IMPLIED
```

```
  comment     CDATA   #IMPLIED
 >

************************************************
*********************/

parseElement_log(element(log, Attr, Node), CdnIn, Cdn2,
TrigIn, Trig, ResIn, log(Res)) :-
        procAttr(log, Attr, CdnIn, Log),
        parseEntity_Node(Node, Log, Cdn2, TrigIn, Trig,
ResIn, Res).

/************************************************
**********************
 * Links to SubActions

  <!ENTITY % Sub 'sub' >

************************************************
*********************/

parseEntity_Sub([element(sub, Attr,
[])|[]],CdnIn,Cdn,TrigIn,Trig,ResIn,Res) :-
        parseElement_sub(element(sub, Attr,
[]),CdnIn,Cdn,TrigIn,Trig,ResIn,Res).

/************************************************
**********************
  <!-- Calls to subactions. -->

  <!ELEMENT sub EMPTY >
  <!ATTLIST sub
    ref       IDREF   #REQUIRED
  >

************************************************
*********************/

parseElement_sub(element(sub, [ref=NAME], []), CdnIn,
Cdn, TrigIn, Trig, ResIn, Res) :-
        sub(NAME, CdnSub, TrigSub, ResSub),
        merge(CdnIn, CdnSub, Cdn),
        merge(TrigIn, TrigSub, Trig),
        merge(ResIn, ResSub, Res),
        !.

/************************************************
**********************

************************************************
**********************

************************************************
**********************

************************************************
*********************/

/************************************************
**********************
 * Process list of attributes
*/
procAttr(Name, Attr, In1, In2) :-
        procAttr2(Name, Attr, In1, In2),
        true    %       write('PROCATTR:
'),write_ln(In2)
        .

procAttr2(_Name, [], In, In) :- !.
procAttr2(Name, [AttrName=Value|Tail], In, [Attr|Rest]) :-
        Attribute =.. [AttrName, Value],
        Attr =.. [Name, Attribute],
        procAttr2(Name, Tail, In, Rest), !.

/************************************************
**********************
 * Intercept messages from the SGML parser (SGML2PL):
error/4 is
 * asserted such that failure can be detected (on its presence).

************************************************
*********************/
prolog:message(sgml(Parser, File, Line, Message)) -->
  { get_sgml_parser(Parser, dialect(Dialect)),
        assert(error(Dialect, File, Line, Message))
  },
        [].
%   [ 'SGML2PL(~w): ~w:~w: ~w'-[Dialect, File, Line,
Message] ].

prolog:warning(A) -->
        { assert(strange(A))
        },
```

```
        [].

/***********************************************
**********************
 * Pretty-Printing utilities

***********************************************
*********************/

printMe(FN, No, Cdn, Trig, ResIn, NotTrig) :-
        (ResIn == [] -> Res = [defaultAction]
        | otherwise -> Res = ResIn),
        printMe(feature(FN,No,[subs(user,NameNoExt)|C
dn],Trig,Res), NotTrig),
        assert(afeature(FN,No,[subs(user,NameNoExt)|Cd
n],Trig,Res)), !.
/*
printMe(feature(Name,Level,Cdn,Trig,Res),[]) :-
        write('feature([\''), write(Name), write('\','),
write(Level), write('],\n   '),
        print(Cdn), write(',\n   '),
        print(Trig), write(',\n   '),
        print(Res),
        write(') :-\n      true.'), nl, nl, !.

printMe(feature(Name,Level,Cdn,Trig,Res),NotTrig) :-
        write('feature([\''), write(Name), write('\','),
write(Level), write('],\n   '),
        print(Cdn), write(',\n   '),
        print(Trig), write(',\n   '),
        print(Res),
        write(') :-'), nl,
        printMe2(NotTrig), !.
*/

% by Yiqun
printMe(feature(Name, _Level,_Cdn,Trig,Res),[]) :-
        write('cpl_policy('), print(Name), write(','),
write('implies('), print(Trig), write(','), print(Res),
        write(')).\n'), !.

printMe(feature(Name,_Level,_Cdn,Trig,Res),_NotTrig) :-
        write('cpl_policy('), print(Name), write(','),
write('implies('), print(Trig), write(','), print(Res),
        write(')).\n'), !.

printMe2([]) :- !.
printMe2([incoming(A)|[]]) :-
        write('   ANY \\= '), print(A), nl,
        write('   | ANY = anyUser.'), nl, !.
printMe2([incoming(A)|R]) :-
        write('   ANY \\= '), print(A), write(','), nl, !,
        printMe2(R).

printMe(FN, No, Cdn, Trig, Res) :-
        print(feature(FN,No,[subs(user,NameNoExt)|Cdn]
,Trig,Res)), nl,
        assert(afeature(FN,No,[subs(user,NameNoExt)|Cd
n],Trig,Res)),
        !.

portray(incoming([])) :-
        write('incoming(ANY)'),!.
portray(incoming(List)) :-
        write('incoming('),
        print(List),
        write(')'),!.
portray(feature(Name,Level,Cdn,Trig,Res)) :-
        write('feature([\''), write(Name), write('\','),
write(Level), write('],\n   '),
        print(Cdn), write(',\n   '),
        print(Trig), write(',\n   '),
        print(Res),
        write(') :-\n      true.'), !.

%portray(Atom) :-
%       atom(Atom),
%       write('\''),write(Atom),write('\''),!.
```

APPENDIX B: Prolog Code of Filter

```
%-------------------------------------------------------------------
----
% Module:     Filter -- Detecting Feature Interactions in
multiple CPL scripts
% Version:  1.1
% Modified:  Feb 28, 2003
% Author:     Yiqun Xu <yixu@site.uottawa.ca> orginally
from Nicolas Gorse
%-------------------------------------------------------------------
----



%**************************************%

:- dynamic done/2.

%----------* Basic predicate definitions *-------------------------
---------

% Rules list
%
rule(d1).
rule(d2).
rule(d3).
rule(d4).
rule(i1).
rule(l1).
rule(l2).


linit :-
        retractall(done(_,_)), !.

localchecking :-
        cpl_policy(User, Formula1),
        cpl_policy(User, Formula2),
        Formula1 \= Formula2,
%       \+ done(Formula1, Formula2),
%       \+ done(Formula2, Formula1),
%       assert(done(Formula1, Formula2)),
        lookup_localfi(Rule, Formula1, Formula2),
        printout(Rule, Formula1, Formula2),
        fail.

lookup_localfi(Rule, Formula1, Formula2) :-
        li_check(Rule, Formula1, Formula2).
/*
li_check(l1, implies(Cdns1, Action), implies(Cdns2, Action))
:-
        mem1(conditions1, Cdns1),
        mem1(conditions1, Cdns2),
        mem1(condition2, Cdns1),
        mem1(\condition2, Cdns2).


*/

%li_check(l2, implies(Cdns1, _Action1), implies(Cdns2,
_Action2)) :-
%       mem1(Cdns1, Cdns2).

li_check(l2,        implies(conj(_Cdn,[]),        _Action1),
implies(conj(_Cdn,_Cdns2), _Action2)) :-
        !.
li_check(l2,        implies(conj(_Cdn,Cdns1),        _Action1),
implies(conj(_Cdn,Cdns2), _Action2)) :-
        mem1(Cdns1, Cdns2).



checking :-
        cpl_policy(U1, Formula1),
        cpl_policy(U2, Formula2),
        U1 \= U2,
        lookup_interaction(Rule,U1,    U2,    Formula1,
Formula2),
        printout(Rule, Formula1, Formula2),
        fail.

lookup_interaction(Rule, U1, U2, Formula1, Formula2):-
        fi_check(Rule, U1, U2, Formula1, Formula2).



%**********************************************
**********%
```

```
%-----Rule d1, A's outgoing policy leads to reject all the calls
from A to C while B's incoming policy
%-----leads to forward (proxy) all the incoming calls to C.

fi_check(d1, U1, U2, implies(Cdns1,[reject(status(reject)),
_]), implies(Cdns2,[proxy([location(C)], _)])):-
        mem1(outgoing(U1,X), Cdns1),
        mem1([address(is(C))], Cdns1),
        mem1(incoming(X,U2), Cdns2), !.

fi_check(d1, U1, U2, implies(Cdns1,[reject(status(reject)),
_]), implies(Cdns2,[proxy([location(C)], _)])):-
        mem1(outgoing(U1,X), Cdns1),
        mem1([address(contains(Tem))], Cdns1),
        sub_string(C, _Start, _Length, _After, Tem),
        mem1(incoming(X,U2), Cdns2), !.

fi_check(d1, U1, U2, implies(Cdns1,[reject(status(reject)),
_]), implies(Cdns2,[proxy([location(C)], _)])):-
        mem1(outgoing(U1, X), Cdns1),
        mem1([address(subdomain-of(Tem))], Cdns1),
        sub_string(C, _Start, _Length, _After, Tem),
        mem1(incoming(X,U2), Cdns2), !.


%------Rule d2, A's incoming policy leads to reject all the
calls from C while B's incoming policy leads to
%------forward (proxy) all the incoming calls to A.

fi_check(d2, A, B, implies(Cdns1,[reject(status(reject)), _]),
implies(Cdns2,[proxy([location(A)], _)])):-
        mem1(incoming(X, A), Cdns1),
        mem1([address(is(C))], Cdns1),
        mem1(incoming(X,B), Cdns2), !.

fi_check(d2, A, B, implies(Cdns1,[reject(status(reject)), _]),
implies(Cdns2,[proxy([location(A)], _)])):-
        mem1(incoming(X, A), Cdns1),
        mem1([address(contains(Tem))], Cdns1),
        sub_string(C, _Start, _Length, _After, Tem),
        mem1(incoming(X,B), Cdns2), !.

fi_check(d2, A, B, implies(Cdns1,[reject(status(reject)), _]),
implies(Cdns2,[proxy([location(A)], _)])):-
        mem1(incoming(X, A), Cdns1),
        mem1([address(subdomain-of(Tem))], Cdns1),
        sub_string(C, _Start, _Length, _After, Tem),
        mem1(incoming(X,B), Cdns2), !.


%------Rule d3, A's incoming policy rejects all the calls from
B while B's incoming policy leads to
%------forward (proxy) all the outgoing calls to A.
/*
fi_check(d3, A, B, implies(Cdns1, reject(B,A)),
implies(Cdns2,proxy(B,A))):-
        mem1(incoming(X,A),Cdns1),
        mem1([address(is(B))], Cdns1),
        mem1(outgoing(B,X),Cdns2),!.
*/

fi_check(d3, A, B, implies(Cdns1, [reject(status(reject)), _]),
implies(Cdns2,[proxy([location(A)], _)])):-
        mem1(incoming(X,A),Cdns1),
        mem1([address(is(B))], Cdns1),
        mem1(outgoing(B,X),Cdns2),!.


fi_check(d3, A, B, implies(Cdns1, [reject(status(reject)), _]),
implies(Cdns2,[proxy([location(A)], _)])):-
        mem1(incoming(X,A),Cdns1),
        mem1([address(contains(Tem))], Cdns1),
        sub_string(B, _Start, _Length, _After, Tem),
        mem1(outgoing(B,X),Cdns2),!.


fi_check(d3, A, B, implies(Cdns1, [reject(status(reject)), _]),
implies(Cdns2,[proxy([location(A)], _)])):-
        mem1(incoming(X,A),Cdns1),
        mem1([address(subdomain-of(Tem))], Cdns1),
        sub_string(B, _Start, _Length, _After, Tem),
        mem1(outgoing(B,X),Cdns2),!.


%------Rule d4, A's incoming policy leads to forward (proxy)
a call to a maillist parallely while incoming policy of one
user B in this maillist
%------forward (proxy) the incoming call to other people.
```

```
fi_check(d4, A, B, implies(_Cdns1, [proxy([location(C)],
ordering(parallel))]), implies(Cdns2,[proxy([location(_)],
_)])):-
        mem1(incoming(X,B),Cdns2),
        sub_string(C, _Start1, Length1, _After1, B),
        sub_string(C, _Start2, Length2, _After2, C),
        Length1<Length2,!.


%------Rule i1, A's policy leads to forward (proxy) all the
incoming calls to B while B's incoming policy leads to
%------forward (proxy) all the incoming calls to A.

fi_check(i1, A, B, implies(Cdns1, [proxy([location(B)], _)]),
implies(Cdns2,[proxy([location(A)], _)])):-
        mem1(incoming(X,A),Cdns1),
        mem1(incoming(X,B),Cdns2),!.


%----------*****---------%

% "A∧1" ==> "mem1(A,conj(A,1))"

mem1(A, A) :- !.
mem1(A, conj(A,_X)) :- !.
mem1(A, conj(_B,X)) :-
        !, mem1(A,X).

%----------information about the incoherence found----------
---%


printout(d1, Formula1, Formula2) :-
        nl,nl,write('%****** Interaction detected by Rule
D1'), write(' -> '),nl,
        write('%    The first user is forbidden to call
somebody'), nl,
        write('%  while the second user will forward calls
to the forbidden user'), nl,
        write('%    + The first user\'s policy'), nl,
        write(Formula1), nl,
        write('%    + The second user\'s policy'), nl,
        write(Formula2), nl.


printout(d2,Formula1,Formula2) :-
        nl,nl,write('% ****** Interaction detected by
Rule D2'), write(' -> '),
        write('%    The first user reject calls from
somebody'), nl,
        write('%  while the second user will forward calls
to the first user'), nl,

        write('%    + The first user\'s policy'), nl,
        write(Formula1), nl,
        write('%    + The second user\'s policy'), nl,
        write(Formula2), nl.


printout(d3,Formula1,Formula2) :-
        nl,nl,write('% ****** Interaction detected by
Rule D3'), write(' -> '),
        write('%    The first user reject calls from the
second user'), nl,
        write('%    while the second user will forward
outgoing calls to the first user'), nl,

        write('%    + The first user\'s policy'), nl,
        write(Formula1), nl,
        write('%    + The second user\'s policy'), nl,
        write(Formula2), nl.


printout(d4,Formula1,Formula2) :-
        nl,nl,write('% ****** interaction detected by Rule
D4'), write(' -> '),
        write('%  The first user broadcasts calls to a list'),
nl,
        write('%  The second user is in the broadcasting
list and will forward the call to anybody else or autoanswer'),
nl,
        write('%    + The first user\'s policy'), nl,
        write(Formula1), nl,
        write('%    + The second user\'s policy'), nl,
        write(Formula2), nl.

printout(i1,Formula1,Formula2) :-
```

```
        nl,nl,write('% ****** interaction detected by Rule
I1'), write(' -> '),
        write('%   call forward loop between tow users'),
nl,
        write('%     + The first user\'s policy'), nl,
        write(Formula1), nl,
        write('%     + The second user\'s policy'), nl,
        write(Formula2), nl.

printout(l1,Formula1,Formula2) :-
        nl,nl,write('% ****** interaction detected by Rule
L1'), write(' -> '),
        write('%   The user\'s two features are redundant'),
nl,
        write('%     + The first feature'), nl,
        write(Formula1), nl,
        write('%     + The second feature'), nl,
        write(Formula2), nl.

printout(l2,Formula1,Formula2) :-
        nl,nl,write('% ****** interaction detected by Rule
L2'), write(' -> '),
        write('%   one feature is shadowed by another'),
nl,
        write('%     + The first feature'), nl,
        write(Formula1), nl,
        write('%     + The second feature'), nl,
        write(Formula2), nl, !.
```

APPENDIX C: Examples in SFSL and Related Detection Results

CPL policies in SFSL:

cpl_policy('sip:yiqun@uottawa.ca', implies(conj(incoming(_G406, 'sip:yiqun@uottawa.ca'), [address(subdomain-of('carl@uottawa.ca'))]),[reject(status(reject)), reject(reason('Do not take carl\'s call.'))])).

cpl_policy('sip:luigi@uottawa.ca',implies(conj(incoming(_G708, 'sip:luigi@uottawa.ca'), [address(contains('Charles Trainer'))]),[proxy([location('sip:terry@uottawa.ca')], _)])).
cpl_policy('sip:terry@uottawa.ca',implies(conj(incoming(_G709, 'sip:terry@uottawa.ca'), []),[proxy([location('sip:luigi@uottawa.ca')], _)])).

cpl_policy('sip:carl@uottawa.ca', implies(conj(outgoing('sip:carl@uottawa.ca', _G578), []), [proxy([location('sip:yiqun@uottawa.ca')], _)])).

cpl_policy('sip:carl@uottawa.ca', implies(conj(incoming(_G579, 'sip:carl@uottawa.ca'), []), [proxy([location('sip:yiqun@uottawa.ca')], _)])).

cpl_policy('sip:alice@uottawa.ca',implies(conj(outgoing('sip:alice@uottawa.ca', _G412), [address(subdomain-of('1900'))]),[reject(status(reject)), reject(reason('Not allowed to make 1-900 calls.'))])).
cpl_policy('sip:alice@uottawa.ca',implies(conj(incoming(_G398, 'sip:alice@uottawa.ca'), []),[proxy([location('sip:carl@pager.ottawahospital.com')], ordering(parallel))])).
cpl_policy('sip:bob@uottawa.ca',implies(conj(incoming(_G411, 'sip:bob@uottawa.ca'), [address(is('sip:carl@pager.ottawahospital.com'))]),[reject(status(reject)), reject(reason('I do not accept Carl\'s call.'))])).
cpl_policy('sip:bob@uottawa.ca',implies(conj(outgoing('sip:bob@uottawa.ca', _G414), [address(is('sip:carl@pager.ottawahospital.com'))]),[reject(status(reject)), reject(reason('I am not allowed to call Carl'))])).

cpl_policy('sip:daniel@uottawa.ca',implies(conj(outgoing('sip:daniel@uottawa.ca', _G399), []),[proxy([location('sip:carl@pager.ottawahospital.com, sip:carl@uottawa.ca')], ordering(parallel))])).

cpl_policy('sip:Elain@uottawa.ca',implies(conj(incoming(_G709, 'sip:Elain@uottawa.ca'),[]),
[proxy([location('sip:luigi@uottawa.ca')], _)])).
cpl_policy('sip:Elain@uottawa.ca',implies(conj(incoming(_G710, 'sip:Elain@uottawa.ca'),
[address(is('sip:carl@pager.ottawahospital.com'))]),[reject(status(reject)), reject(reason('Refuse to take calls
from Carl'))])).
cpl_policy('sip:frasier@uottawa.ca',implies(conj(incoming(_G698, 'sip:frasier@uottawa.ca'),
[]),[proxy([location('sip:bob@uottawa.ca')], ordering(parallel))])).

Related Detection Results:

% ****** Interaction detected by Rule D3 -> % The first user reject calls from the second user
% while the second user will forward outgoing calls to the first user
% + The first user's policy
implies(conj(incoming(_G254, sip:yiqun@uottawa.ca), [address(subdomain-of(carl@uottawa.ca))]),
[reject(status(reject)), reject(reason(Do not take carl's call.))])
% + The second user's policy
implies(conj(outgoing(sip:carl@uottawa.ca, _G254), []), [proxy([location(sip:yiqun@uottawa.ca)],
_G294)])

% ****** interaction detected by Rule I1 -> % call forward loop between two users
% + The first user's policy
implies(conj(incoming(_G254, sip:terry@uottawa.ca), []), [proxy([location(sip:luigi@uottawa.ca)],
_G261)])
% + The second user's policy
implies(conj(incoming(_G254, sip:luigi@uottawa.ca), [address(contains(Charles Trainer))]),
[proxy([location(sip:terry@uottawa.ca)], _G288)])

% ****** Interaction detected by Rule D2 -> % The first user rejects calls from somebody
% while the second user will forward calls to the first user
% + The first user's policy
implies(conj(incoming(_G254, sip:bob@uottawa.ca), [address(is(sip:carl@pager.ottawahospital.com))]),
[reject(status(reject)), reject(reason(I do not accept Carl's call.))])
% + The second user's policy

implies(conj(incoming(_G254, sip:frasier@uottawa.ca), []), [proxy([location(sip:bob@uottawa.ca)], ordering(parallel))])

%****** Interaction detected by Rule D1 ->
% The first user is forbidden to call somebody
% while the second user will forward calls to the forbidden user
% + The first user's policy
implies(conj(outgoing(sip:bob@uottawa.ca, _G255), [address(is(sip:carl@pager.ottawahospital.com))]), [reject(status(reject)), reject(reason(I am not allowed to call Carl))])
% + The second user's policy
implies(conj(incoming(_G255, sip:alice@uottawa.ca), []), [proxy([location(sip:carl@pager.ottawahospital.com)], ordering(parallel))])

% ****** interaction detected by Rule D4 -> % The first user broadcasts calls to a list
% The second user is in the broadcasting list and will forward the call to anybody else or autoanswer
% + The first user's policy
implies(conj(outgoing(sip:daniel@uottawa.ca, _G255), []), [proxy([location(sip:carl@pager.ottawahospital.com, sip:carl@uottawa.ca)], ordering(parallel))])
% + The second user's policy
implies(conj(incoming(_G276, sip:carl@uottawa.ca), []), [proxy([location(sip:yiqun@uottawa.ca)], _G283)])

% ****** interaction detected by Rule L2 -> % one feature is shadowed by another
% + The first feature
implies(conj(incoming(_G284, sip:Elain@uottawa.ca), []), [proxy([location(sip:luigi@uottawa.ca)], _G291)])
% + The second feature
implies(conj(incoming(_G284, sip:Elain@uottawa.ca), [address(is(sip:carl@pager.ottawahospital.com))]), [reject(status(reject)), reject(reason(Refuse to take calls from Carl))])