Feature Interactions in Telecommunications and Software Systems VII

Edited by

Daniel Amyot

School of Information Technology and Engineering, University of Ottawa, Ottawa, Canada

and

Luigi Logrippo

Département d'informatique et d'ingénierie, Université du Québec en Outaouais, Gatineau, Canada

Amsterdam • Berlin • Oxford • Tokyo • Washington, DC

Feature Interactions in Telecommunications and Software Systems VII

Edited by

Daniel Amyot

School of Information Technology and Engineering, University of Ottawa, Ottawa, Canada

and

Luigi Logrippo

Département d'informatique et d'ingénierie, Université du Québec en Outaouais, Gatineau, Canada

Amsterdam • Berlin • Oxford • Tokyo • Washington, DC

Feature Interactions in Telecommunications and Software Systems VII
D. Amyot and L. Logrippo (Eds.)
IOS Press, 2003

Detecting Script-to-Script Interactions in Call Processing Language

Masahide NAKAMURA[1], Pattara LEELAPRUTE[2], Ken-ichi MATSUMOTO[1], and Tohru KIKUNO[2]

[1] *Graduate School of Information Science, Nara Institute of Science and Technology, Japan*
{masa-n, matumoto}@is.aist-nara.ac.jp
[2] *Graduate School of Information Science and Technology, Osaka University, Japan*
{pattara, kikuno}@ist.osaka-u.ac.jp

Abstract. This paper addresses a problem to detect feature interactions in a CPL (Call Processing Language) programmable service environment on Internet telephony. In the CPL environment, the previous works cannot be directly applied, because of new complications introduced:(a) features created by non-experts and (b) distributed feature provision. To cope with the problem (a), we propose eight types of *semantic warnings* which guarantee some aspects of semantic correctness in each individual CPL script. Then, as for (b), we present an alternative definition of feature interactions, and propose a method to implement run-time feature interaction detection. The key idea is to define feature interactions as the semantic warnings over multiple CPL scripts, each of which is semantically safe. We also demonstrate tools, called CPL checker and FI simulator, to help users to construct reliable CPL scripts.

1 Introduction

Internet telephony [8] is expected to enable a new generation of telecommunication services. It facilitates integration with other Internet services, which allows rich and sophisticated telephony services. Internet telephony has been widely studied at the *network protocol level* (i.e., SIP [9] and H.323 [14]). Some companies have already started commercial services, and its protocol stacks are also released by open source communities (e.g., VOCAL [16]).

The concern is now shifting to the *service level*, that is, how to provide value-added features in Internet telephony. There are two complementary approaches to achieving the feature provision.

The first one is based on *network convergence*, which integrates IP and the traditional IN/PSTN networks [13]. The idea is to activate the IN services from Internet telephony through APIs (e.g., JAIN API [15]). Many telecom industries are conducting research and development for it, in order to make full reuse of their legacy services. Though this approach is quite challenging, we do not discuss it in this paper.

Another approach, which is interesting for us here, is *programmable service* [6] [10]. The service and feature creation is opened to end users and third parties. The service definitions can be deployed in the local (and distributed) servers over the Internet. The users can create, delete and modify their own services at any time.

As in the IN/PSTN networks, feature interactions occur as well in Internet telephony. The previous research works might be helpful to understand a part of the interactions. However, feature interactions in Internet telephony are a more serious problem than the conventional ones, because of various new complications introduced [1][7]. Especially in the context of the programmable service, the following are essential issues that make the problem more difficult.

(a) Features created by non-expert users: In conventional telephony, quality of individual features is guaranteed by the telecom companies, and end users just *subscribe* to the ready-made features. On the other hand, programmable services allow end users or third parties to freely create and define their own custom-made features. Most of the end users are not as expert as telecom engineers. Each user is very likely to create a feature, without the greatest care of logical consistency and correctness in the (single) feature, much still less feature interactions with others.

(b) Distributed feature provision: The created features can be easily installed in local signaling servers. The features are completely distributed over the Internet and there is no centralized feature server. This fact means that it is impossible to enumerate all possible features. Thus, we cannot conduct off-line feature interaction detection, nor prepare resolution schemes in advance, such as feature priorities.

In this paper, we tackle the problem of feature interaction in Internet telephony with *Call Processing Language* (CPL, in short) [5][6]. The CPL is an XML-based language for the programmable service in Internet telephony, and is proposed as RFC2824 in IETF. The CPL is gaining popularity, and major VoIP systems (e.g., [16][17]) adopt it as feature description language. The goal of this paper is to establish a definition and a detection method of feature interactions within the CPL programmable service environment. To achieve this, we propose two new methods corresponding to the above problems (a) and (b).

Firstly, we propose *semantic warnings* for the CPL scripts to address the problem (a) feature created by non-experts. In a CPL environment, each user describes his/her own feature in a (single) *CPL script* at a time. The syntax of the CPL is strictly defined by DTD (Document Type Definition). However, compliance with the DTD is not a sufficient condition for correctness of a CPL script. As far as we know, there exist no guidelines for users to assure semantical correctness of individual CPL scripts. The proposed warnings are not necessarily errors, but they identify the source of ambiguity, redundancy and inconsistency for a given CPL script.

Secondly, to address the problem (b) distributed feature provision, we propose an alternative definition of feature interaction, and its detection method in the CPL environment. In [5][6], a brief categorization of feature interactions in the CPL environment is presented [1]. However, no concrete method to detect feature interactions in the CPL environment has been proposed yet.

In general, feature interactions can be defined (informally) as violation of user's requirement that is caused by combination of multiple features. Here, a CPL script described by a user can be considered as an exact requirement of the user. So, the violation of the requirement occurs when the script is not executed as described, under the influence of CPL scripts

[1] According to the categorization in [5], feature interactions discussed in this paper are *script-script* and/or *server-to-server* interactions

by other users within the call. Note that the violation can be observed only at run time, and can never be predicted by off-line analysis. Thus, the new definition of feature interactions must be dependent on a call scenario at run time, which is significantly different from definitions in the literatures [4] [12].

Our key idea is to define feature interactions (i.e., the violation) as the semantic warnings over multiple CPL scripts, each of which is semantically valid. For this, we propose a combine operator and some new notions for CPL scripts (i.e., complete, safe). Then, we present a procedure to implement run-time feature interaction detection. We also present tools to detect the semantic warnings in a single CPL script, and to feature interactions over multiple scripts.

The rest of the paper is organized as follows. Section 2 describes a brief review of CPL. In Section 3, we propose the semantic warnings for a single CPL script. Then, Section 4 presents definition and detection method of feature interactions among multiple CPL scripts. Section 5 describes the tool support for the proposed method. Finally we conclude the paper with discussion and future work in Section 6.

2 Call Processing Language (CPL)

2.1 Overview

Internet telephony basically consists of two types of components: end systems and signaling servers. The CPL is meant to describe *network-based features* which process calls on the signaling servers in a network. Terminal-based features, like camp-on, call waiting and voicemail, that heavily depend on end-system states and devices should be implemented on the end systems, and thus are out of scope of the CPL.

First of all, we review the CPL definition briefly. The full specification can be found in [5][6]. A CPL script is composed of mainly four types of constructors: *top-level actions, switches, location modifiers* and *signaling operations.*

Top-level actions: Top-level actions are firstly invoked when a CPL script is executed: `outgoing` (or `incoming`) specifies a tree of actions taken on the user's outgoing call (or incoming call, respectively). `subaction` describes a sub routine to increase re-usability and modularity.

Switches: Switches represent conditional branches in CPL scripts. Depending on types of conditions specified, there are five types: `address-switch`, `string-switch`, `language-switch`, `time-switch` and `priority-switch`.

Location modifiers: The CPL has an abstract model, called *location set,* for locations to which a call is to be directed. The set of the locations is stored as an implicit global variable during call processing action by the CPL. For the outgoing call processing, the location is initialized to the destination address of the call. For the incoming call processing, the location set is initialized to the empty set. During the execution, the location set can be modified by three types of modifiers: `location` adds an explicit location to the current location set; `lookup` obtains locations from outside; `remove-location` removes some locations from the current location set.

Signaling operations: Signaling operations trigger signaling events in the underlying signaling protocol for the current location set. There are three operations: `proxy` forwards

```
<?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>
    </address-switch>
  </outgoing>
</cpl>
```

Figure 1: A CPL script s_a of OCS

```
<?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>

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

Figure 2: A CPL script s_c of DCF

the call to the location set currently specified; `redirect` prompts the calling party to make another call to the current location set, then terminates the call processing; `reject` causes the server to reject the call attempt and then terminates the call processing.

2.2 *Feature examples*

We start with a simple feature, namely Originating Call Screening (OCS, in short). Suppose the following requirement: Alice (`alice@instance.net`) wants to block any outgoing calls to Bob (`bob@home.org`) from her end system. Figure 1 shows an implementation of Alice's script s_a. In Figure 1, the first three lines represent declaration of XML and DTD. The tag `<cpl>` means the start of a body of the CPL script. The top-level action `<outgoing>` describes actions activated when Alice makes a call. Next, `<address-switch>` specifies a conditional branch. In this example, the condition is extracted from the destination address of the call (`field="destination"`). If the destination address matches `bob@home.org` (`<address is="bob@home.org">`), the call is rejected (`<reject status... />`). If it does not match, the call is be proxied to the destination address (This is done by *default behavior* of the CPL, although the proxy operation is not explicitly specified. See Section 4.2.1).

The next example is a bit complicated. A user Chris (`chris@example.com`) wants to:

- receive calls from domain `example.com` at office `chris@office.example.com`.
- reject any call from malicious crackers belonging to `crackers.org`.
- redirect any call from clients within `instance.net` to Bob's home at `bob@home.org`.
- proxy any other calls to his voicemail at `chris@voicemail.example.com`.

Figure 2 shows an implementation of Chris's script s_c. Let us call this feature Domain Call Filtering (DCF). The portion surrounded by `<subaction> </subaction>` defines a subaction called from the main-routine. `<incoming>` specifies actions activated when Chris receives an incoming call.

Next, in <address-switch>, a condition for the switch is extracted from the host address of the caller (field="origin" subfield="host"). If the domain matches example.com (<address subdomain-of="example.com">), then the location is set to chris@office.example.com, and the call is proxied to his office (<proxy />). If the domain matches crackers.org, the call is rejected by <reject />. Else if the domain matches instance.net, the location is set to bob@home.org. Then, the call is redirected to Bob and the caller places a new call to Bob. Otherwise, the subaction voicemail is called. In the subaction voicemail, the location is set to the voicemail at chris@voicemail.example.com, and then the call is proxied there.

3 CPL semantic warnings

The definition of CPL with DTD [5][6] guarantees the syntactical structure of CPL, but does not cover any semantical aspects of each individual script. Hence, there is enough room, especially for non-expert users, to make semantical flaws in the script. The proposed warnings aim to help each user find such semantical flaws in his/her script. Focusing on constraints of CPL and semantic aspects of telephony features, we have identified eight kinds of warnings so far.

3.1 Multiple forwarding addresses (MFAD)

Definition: The execution reaches <proxy> or <redirect> while multiple addresses are contained in the location set.

Effects: The CPL allows calls to be proxied (or redirected) to multiple address locations by cascading <location> tags. However, if the call is redirected to multiple locations, then the caller would be confused to which address the next call should be placed. Or, if the call is proxied, a race condition might occur depending on the configuration of the proxied end systems. As a typical example, if a user simultaneously sets the forwarding address to his handy phone and voicemail that immediately answers the call. Then the call never reaches his handy phone.

Example CPL: Figure 3(a) shows an example. The user is setting the forwarding address to his handy phone pattara@mobile.example.com and voicemail pattara@voicemail.example.com, simultaneously. If the user configures the voicemail to immediately answer the call, then no call reaches the mobile phone.

3.2 Unused subactions (USUB)

Definition: Subaction <subaction id= "foo" > exists, but <subaction ref= "foo" > does not.

Effects: The subaction is defined but not used. The defined subaction is completely redundant, and should be removed to decrease server's overhead for parsing the CPL script.

Example CPL: Figure 3(b) shows an example. In this script, a subaction mobile that is declared in the subsection part is not used in the body of the script. So, the unused subaction mobile is redundant and should be removed.

3.3 Call rejection in all paths (CRAP)

Definition: All execution paths terminate at `<reject>`.

Effects: No matter which path is selected, the call is rejected. No call processing is performed, and all executed actions and evaluated conditions are nullified. This is not a problem only when the user wants to reject all calls explicitly. However, complex conditional branches and deeply nested tags make this problem difficult to be found, on the contrary to the user's intention.

Example CPL: Figure 3(c) shows an example. By this script, any incoming call is rejected, no matter who the originator is. All actions and evaluated conditions are meaningless after all.

3.4 Address set after address switch (ASAS)

Definition: When `<address>` and `<otherwise>` tags are specified as outputs of `<address-switch>`, the same address evaluated in the `<address>` is set in the `<otherwise>` block.

Effects: The `<otherwise>` block is executed when the current address does not match the one specified in `<address>`. If the address is set as a new current address in `<otherwise>` block, then a violation of the conditional branch might occur. A typical example is that, after screening a specific address by `<address-switch>`, the call is proxied to the address, although any call to the address must have been filtered.

Example CPL: Figure 3(d) shows an example. When the user make an outgoing call, this script checks the destination of the call. The call should be rejected if the destination address is `pattara@example.com`, according to the condition specified in `<address>`. However, in the `<otherwise>` block, the call is proxied to `pattara@example. com`, which must have been rejected.

3.5 Overlapped conditions in single switch (OCSS)

Definition: Let A be a switch, and let $cond_{A1}$ and $cond_{A2}$ (arranged in this order) be conditions specified as output tags of A. Then, $cond_{A1}$ is implied by $cond_{A2}$.

Effects: According to the CPL specification, if there exist multiple output tags for a switch, then the condition is evaluated in the order the tags are presented, and the first tag to match is taken. By the above definition, whenever $cond_{A2}$ becomes true, $cond_{A1}$ is true. So, the former tag is always taken and the latter tag is never executed, which is a redundant description.

Example CPL: Figure 3(e) shows an example. This script is supposed to do a typical call processing in a support center. Calls for general help (with a subject containing `help`) are meant to be redirected to `general-support`. Emergency calls (with a subject matching `emergency help`) are to be proxied to an attendant `staff`. However, in fact, all calls are redirected to `general-support`, and no emergency call reaches to the attendant. Since `"help"` is a substring of `"emergency help"`, two conditions are overlapped.

3.6 Identical actions in single switch (IASS)

Definition: The same actions are specified for all conditions of a switch.

Effects: No matter which condition holds, the same action is executed. Therefore, the conditional branch specified in the switch is meaningless. In such case, this switch should be eliminated to reduce complexity of the logic.

Example CPL: Figure 3(f) shows an example. This script has a `language-switch` to check the language preference of the call. This switch specifies a conditional branch depending on whether the preference is Japanese (`jp`) or not. However, the same action `<sub ref="voicemail">` occurs independently of the language preference. So, the switch is completely meaningless.

3.7 Overlapped conditions in nested switches (OCNS)

Definition: Let A and B be switches of the same type, and let $cond_A$ and $cond_B$ be the conditions of A and B, respectively. Then, [A is nested in B's condition block] and [$cond_B$ implies $cond_A$].

Effects: This warning is derived by the fact that CPL has no variable assignment. So, any condition that is evaluated to be true (or false) remains true (or false, respectively) during the execution. Assume that $cond_B$ implies $cond_A$. B's condition block, in which A is specified, is executed only when $cond_B$ is true. So, by the assumption, $cond_A$ always becomes true when evaluated. Thus, A's condition block is unconditionally executed. Also, if A has an otherwise block, then the block cannot be executed. As a result, the switch A is completely redundant and should be removed.

Example CPL: Figure 3(g) shows an example. When an incoming call arrives, the script first checks a domain of the caller's host. If the domain matches `home.org`, then the second switch does the same checking again. However, since the condition for the second switch is the same as the first one, which have already been shown to be true, it is redundant description. Also, `<reject />` in `<otherwise>` is unreachable.

3.8 Incompatible conditions in nested switches (ICNS)

Definition: Let A and B be switches of the same type, and let $cond_A$ and $cond_B$ be the conditions of A and B, respectively. Then,

(α) [A is nested in B's condition block] and [$cond_A$ and $cond_B$ are mutually exclusive], or

(β) [A is nested in B's otherwise block] and [$cond_A$ implies $cond_B$].

Effects: Let us consider (α) first. B's condition block, in which A is specified, is executed only when $cond_B$ is true. However, $cond_A$ and $cond_B$ are exclusive, so $cond_A$ cannot be true at this time. Therefore, A's condition block is unexecutable. (β) is the complemental case of (α). B's otherwise block is executed only when $cond_B$ is false. Now that $cond_A$ must be false, which is implied by $\neg cond_B$. Consequently, A's condition block is unexecutable also.

```
<?xml version="1.0" ?>
<!DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx
                      CPL 1.0//EN" "cpl.dtd">
<cpl>
 <incoming>
   <location
      url="sip:pattara@mobile.example.com">
   <location
      url="sip:pattara@voicemail.example.com">
     <proxy />
   </location>
   </location>
 </incoming>
</cpl>
```

(a) Example of MFAD

```
<?xml version="1.0" ?>
<!DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx
                      CPL 1.0//EN" "cpl.dtd">
<cpl>
  <subaction id="mobile">
    <location url="sip:jones@mobile.example.com" >
      <proxy />
    </location>
  </subaction>
  <incoming>
    <location url="sip:jones@example.com">
      <proxy />
    </location>
  </incoming>
</cpl>
```

(b) Example of USUB

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

(c) Example of CRAP

```
<?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:pattara@example.com">
        <reject status="reject"
            reason="I don't call Pattara" />
      </address>
      <otherwise>
        <location url="sip:pattara@example.com">
            <proxy/>
        </location>
      </otherwise>
    </address-switch>
  </outgoing>
</cpl>
```

(d) Example of ASAS

```
<?xml version="1.0" ?>
<!DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx
                      CPL 1.0//EN" "cpl.dtd">
<cpl>
  <incoming>
    <string-switch field="subject">
     <string contains="help">
      <location url=
         "sip:general-support@example.com">
       <redirect />
      </location>
     </string>
     <string is="emergency help">
      <location url="sip:staff@example.com">
       <proxy />
      </location>
     </string>
    </string-switch>
  </incoming>
</cpl>
```

(e) Example of OCSS

```
<?xml version="1.0" ?>
<!DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx
                      CPL 1.0//EN" "cpl.dtd">
<cpl>
  <subaction id="voicemail">
    <location url="sip:nakamura@voicemail.org" >
      <proxy />
    </location>
  </subaction>
  <incoming>
    <language-switch>
      <language matches="jp">
        <sub ref="voicemail" />
      </language>
      <otherwise>
        <sub ref="voicemail" />
      </otherwise>
    </language-switch>
  </incoming>
</cpl>
```

(f)Example of IASS

```
<?xml version="1.0" ?>
<!DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx
                      CPL 1.0//EN" "cpl.dtd">
<cpl>
 <incoming>
  <address-switch field="origin"
                       subfield="host">
    <address subdomain-of="home.org">
     <address-switch field="origin"
                       subfield="host">
      <address subdomain-of="home.org">
        <location url="sip:pattara@mobile.net">
         <proxy />
        </location>
      </address>
      <otherwise>
        <reject status="reject"
          reason="Some reason" />
      </otherwise>
     </address-switch>
    </address>
  </address-switch>
 </incoming>
</cpl>
```

(g) Example of OCNS

```
<?xml version="1.0" ?>
<!DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx
                      CPL 1.0//EN" "cpl.dtd">
<cpl>
 <incoming>
  <address-switch field="origin"
                       subfield="host">
    <address subdomain-of="home.org">
     <address-switch field="origin"
                       subfield="host">
      <address subdomain-of="example.com">
        <location url="sip:pattara@mobile.net">
         <proxy />
        </location>
      </address>
     </address-switch>
    </address>
  </address-switch>
 </incoming>
</cpl>
```

(h)Example of ICNS

Figure 3: Example CPL scripts

Example CPL: Figure 3(h) shows an example for the above (α). When an incoming call arrives, the script first checks a domain of the caller's host. If the domain matches `home.org`, the second address switch evaluates the domain again. If the domain matches `example.com`, the call is proxied to `pattara@mobile.net`. However, this proxy operation never occurs, since conditions for the two switches are mutually exclusive. That is, it is impossible that the domain matches both `home.org` and `example.com`, simultaneously.

The above eight warnings can occur even if a given CPL script is syntactically well-formed and valid in the sense of XML. Note that the semantic warnings in a single script can be detected by a simple *static* (thus, off-line) analysis.

Definition (Semantically Safe): We say that a CPL script is *semantically safe* iff the script is free from the semantic warnings.

4 Feature interaction detection in CPL scripts

4.1 Key idea

Even if each user creates a safe script by means of the proposed semantic warnings, feature interactions may occur when multiple scripts are executed simultaneously. In the CPL environment, each user cannot have more than one script at a time. Hence, interactions between features allocated in the same user (e.g, CW v.s. TWC) cannot occur [7]. Instead, interactions may occur between scripts owned by different users.

Interaction between OCS & DCF: Let us recall two features OCS and DCF in Section 2.2, implemented as s_a in Figure 1 and s_c in Figure 2, respectively. Now, consider a call scenario where Alice (`alice@instance.net`) calls Chris (`chris@example.com`). First, Alice's script s_a is executed. Since Chris is not screened in s_a, the call is proxied to Chris. Next, Chris's script s_c is executed. Since Alice belongs to a domain `instance.net`, the call is redirected to Bob (`bob@home.org`). As a result, Alice makes a call to Bob, although this call must have been blocked in s_a. Thus we can say that s_a and s_c *interact*.

The situation in the above example is quite similar to the semantic warning ASAS (See Section 3.4), although it occurs within the combination of multiple scripts s_a and s_c. The key idea of our approach is to define feature interactions as *the semantic warnings over multiple CPL scripts*.

4.2 Preliminaries

Before formalizing feature interactions in the CPL environment, we define some new notions with respect to CPL scripts.

4.2.1 Complete CPL scripts

When an execution of a CPL in a signaling server reaches an unspecified condition or an empty signaling operation, the execution follows the *default behaviors* (See Section 11 of [5] for more details). Here are some examples:

```
<?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>

  </incoming>
</cpl>
```

Figure 4: A complete CPL scripts_a of OCS

D1: In an outgoing action, if there is no location modifier and no signaling operation is reached, then proxy to the destination of the call.

D2: In an incoming action, if there is no location modifier and no signaling operation is reached, then treat as if there is no CPL script (i.e., the server tries to connect the call to an end system of the owner of the script).

D3: If location modifier exists but no signaling operation is specified, proxy or redirect to the location, based on the server's standard policy.

These default behaviors are usually taken *implicitly* from user's point of view, based on the server's policy and/or the underlying protocol [2], and may sometimes contradict to the user's intension. Hopefully, the implicitness caused by the default behaviors should be eliminated from every script. For this purpose, we define a new class of CPL scripts:

Definition (Complete Script): We say that a CPL script is *complete* iff no default behavior is taken in any possible execution path.

The default behaviors must be simulated deterministically by using auxiliary information on the signaling server. Hence, we assume that every CPL script on a signaling server can be transformed into a completed script without changing logics of the original script. The followings are guidelines to achieve the transformation.

(a) Make all conditional branches complementary. For instance, `<otherwise>` block must be added to every switch, if it is not present.

(b) Based on the server's standard policy, specify an appropriate signaling operation in every terminating node (i.e., leaf of XML's tree structure) that has location modifier.

(c) Add empty `<incoming>` or `<outgoing>` blocks if either of them is not present.

As an example, consider again the CPL script in Figure 1. This script is not complete, since there is no action specified when the destination address is not `bob@home.org`. Based on the default behavior and the guidelines above, the script can be transformed into a complete one as shown in Figure 4.

[2]For example, the VOCAL system [16] adopts redirect for the above D3.

4.2.2 Successor functions

Suppose that we have a complete CPL script s, and that we want to examine feature interactions between s and other related scripts. Then, we need to know at least which script should be executed after s is terminated. Since s is complete, the execution of s must exit on an empty tag or a certain signaling operation (proxy, redirect or reject), with a location set containing the next address(es) the call is directed to.

Note that the above information dynamically varies depending on given call scenarios. More specifically, we assume that the following functions are available at run time for a given CPL script s and a call scenario c.

Definition (Functions): For a complete CPL script s and a call scenario c, we define the following functions.

$exit(s, c)$: returns a pointer to a signaling operation in s executed at the end under c.

$next(s, c)$: returns the next CPL script executed following s under c, obtained based on the next address.

$type(s, c)$: returns a type of the signaling operation: *proxy*, *redirect*, *reject* or *end* (for empty signaling operation).

For example, consider again the example in Section 2.2 and the scripts s_a in Figure 4 and s_c in Figure 2. Table 1 summarizes values of the functions, with respect to two instances c_1 and c_2 of call scenarios.

Table 1: Example of successor functions for s_a

Call scenarios	$exit(s_a, c_i)$	$next(s_a, c_i)$	$type(s_a, c_i)$
c_1 (Alice calls Bob)	`<reject .../> line 8-9`	*none*	reject
c_2 (Alice calls Chris)	`<proxy .../> line 12`	s_c	proxy

4.3 Feature interactions among two scripts

Firstly, let us consider two complete scripts s and t only. In order to define feature interactions between s and t, we need to capture a combined behavior of s and t. For this purpose, we propose a *combine operator*.

Intuitively, the combine operator merges two scripts s and t such that t *is executed after* s. This partial order is defined only when the call is *proxied* from s to t. In the case that s *redirects* a call to t, the call is once reverted to the caller, and s terminates. Then, a new call is originated from the caller to t without passing through s. Note that the combine operator depends on a given call scenario, because t depends on the scenario.

Definition (Combine operator): Let c be a given call scenario, and let s and t be complete scripts such that $type(s, c) = proxy$ and $next(s, c) = t$. Then, a *combined script* $r = s \triangleright_c t$ is a CPL script obtained from s and t by the following procedures:

Step1: If any subaction (let it be `<subaction id="foo">`) is defined in s (or t), eliminate it by replacing `<sub ref="foo" />` with the body of the subaction.

```
<?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 5: A combined CPL script $s_a \triangleright_{c_2} s_c$

Step2: Let $In(t)$ be the body of incoming action of t (i.e., the portion surrounded by `<incoming>` $\cdots$ `</incoming>`). In s, replace `<proxy />` pointed by $exit(s, c)$ with `<remove-location>` $In(t)$ `</remove-location>`. Let the resulting script be r.

The combine operator $\triangleright_c$ makes a chain between s and t, by merging the `<proxy>` operation executed at the end of s, with the `<incoming>` action of t executed next. The `<remove-location>` inserted in Step 2, is to simulate that the location set is initialized to empty when the incoming action occurs (see Section 2.1). Note that the combine operation does not ruin the syntax structure, since both `<remove-location>` $In(t)$ `</remove-location>` and `<proxy />` are defined as *nodes* in the DTD of CPL. So, if both s and t are syntactically valid, then $s \triangleright_c t$ is also valid.

Now we are ready to define feature interactions among two scripts.

Definition (Feature interaction among two scripts): Let s and t be given complete scripts, and let c be a given call scenario. Then, we say that s *interacts* with t with respect to c, iff both s and t are semantically safe, but $s \triangleright_c t$ is not safe.

Let us consider two scripts s_a and s_c in Figures 4 and 2, respectively. Also, take a scenario c_2 in Table 1, where Alice calls Chris. Figure 5 shows a combined script $s_a \triangleright_{c_2} s_c$. Now, both s_a and s_c are semantically safe, but $s_a \triangleright_{c_2} s_c$ is not safe. It contains a semantic warning ASAS, since address `bob@home.org` evaluated in `<address>` is set in `<otherwise>` block. This is just the interaction explained in Section 4.1.

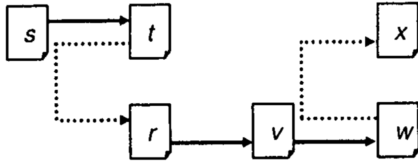


Figure 6: Multiple scripts involved in a call scenario

```
scripts Succ(script s, scenario c) {
        R = s;
        if (type(s, c)) == 'proxy') {
                check_loop(next(s, c), c);
                foreach t ∈ Succ(next(s, c), c)
                        R = R ∪ (s ▷c t);
        } else if (type(s, c) == 'redirect') {
                R = R ∪ Succ(next(s, c), c);
        }
        return R;
}
```

Figure 7: Algorithm Succ(s,c) for computing a set of scripts to be checked

4.4 *Feature interaction detection*

In the previous subsection, we have defined feature interactions between two scripts. However, a call could involve more than two scripts in general, because of successive redirect and proxy operations. So, the definition of feature interactions is generalized as follows:

Definition (Feature interactions): Let s_0 be a given script of the call originator, and let c be a given call scenario. Also, let s_1, s_2, $\cdots$, s_n be scripts, where s_i proxies the call to s_{i+1} under a call scenario c. Then, we say that *feature interactions occur with respect to* s_0 *and* c, iff all of $s_i(0 \leq i \leq n)$ are semantically safe, but there exists some $k(1 \leq k \leq n)$ such that $s_0 \triangleright_c s_1 \triangleright_c \cdots \triangleright_c s_k$ is not semantically safe.

Figure 6 shows an example of a call scenario where multiple scripts are successively executed. In the figure, a box represents a CPL script. A solid arrow represents a proxy operation between scripts, while a dotted arrow describes a redirect operation. To identify feature interactions in this call scenario c, we must check the semantic warnings for the following six scripts (1) s, (2) $s \triangleright_c t$, (3) $s \triangleright_c r$, (4) $s \triangleright_c r \triangleright_c v$, (5) $s \triangleright_c r \triangleright_c v \triangleright_c w$ and (6) $s \triangleright_c r \triangleright_c v \triangleright_c x$.

We present an algorithm to compute a set of combined scripts that must be checked in feature interaction detection. Figure 7 shows a C-like pseudo code to compute the set R of the scripts for a given originating script s and a call scenario c. In the algorithm, we define a procedure `check_loop(t,c)`. This abstracted procedure checks if script `t` forms a *forwarding loop* in the call scenario `c`, by using a loop detection mechanism in the underlying protocol [5]. If a loop is detected, the procedure terminates the algorithm with some error reports.

The algorithm Succ first puts the given script s itself in the set R. Next, if the processing type is proxy, Succ first checks a forwarding loop by `check_loop`. If no loop is detected, it combines s with its successive scripts, which are recursively computed by setting the proxied script as the initial script, and put them in R. If the processing type is redirect, Succ recursively obtains a set of scripts starting with the redirected script, and then puts them in R. Finally, Succ returns the set R. For example, consider again a call scenario c in Figure 6. Succ(s, c) computes the six combined scripts: (1) s, (2) $s \triangleright_c t$, (3) $s \triangleright_c r$, (4) $s \triangleright_c r \triangleright_c v$, (5) $s \triangleright_c r \triangleright_c v \triangleright_c w$ and (6) $s \triangleright_c r \triangleright_c v \triangleright_c x$.

We are ready to present a feature interaction detection algorithm. We assume that each individual script is semantically safe.

Feature interaction detection algorithm :

Input: A CPL script s of a call originator, and a call scenario c.

Output: Feature interactions occur or not.

Procedure: Compute Succ(s,c), and check if each script in Succ(s,c) is semantically safe. If all of the scripts are semantically safe, return "feature interaction does not occur". Otherwise, return "feature interaction occurs" with the corresponding (combined) scripts.

5 Tool support

We are currently implementing a set of tools for the proposed framework. Here we introduce two of them: CPL checker and FI simulator.

CPL checker: For a given CPL script, CPL checker detects the proposed semantic warnings. It also performs syntax checking to validate the conformance to the XML syntax and the DTD of CPL. Thus, it can be used for debugging CPL scripts as well. Figure 8(a) shows a screenshot, where semantic warning OCSS is detected. Every validated script can be registered in the system with the (virtual) owner's address of the script. The registered scripts can be used by FI simulator to perform off-line simulation.

FI simulator: This tool simulates execution of CPL scripts registered through CPL checker. Then, for a given call scenario, it tries to identify feature interactions by combining appropriate scripts. A user of the tool firstly chooses some of the registered scripts, then configures a call scenario [3]. Finally, the tool computes a combined script by algorithm Succ, and detects feature interactions as semantic warnings. Figure 8(b) shows a screenshot, where interaction ASAS presented in Section 4.1 is detected. The call scenario and simulated call processing are also seen in the figure.

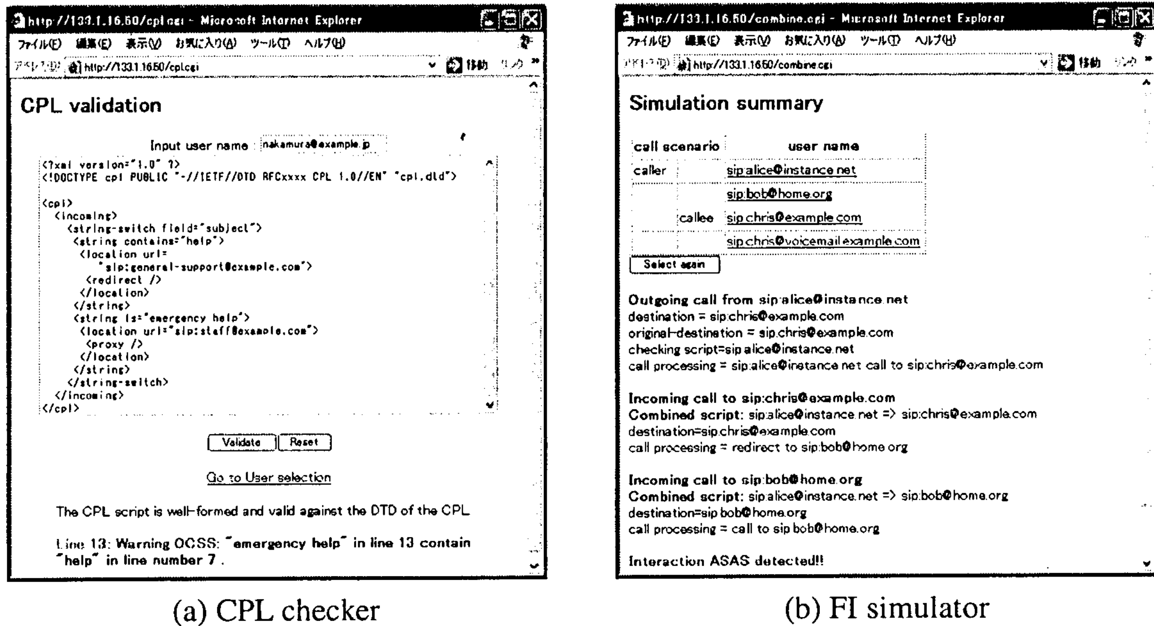
The tools are implemented as a collection of CGI programs written in Perl open-source modules [2][3]. Since all operations to the tools are performed through a simple Web interface, a user can easily conducts validation and simulation of his/her CPL script. A prototype of the tools can be freely used at http://www-kiku.ics.es.osaka-u.ac.jp/~pattara/CPL/, though it is still experimental.

We are also planning to develop modules and libraries that can be used for on-line feature interaction detection (See Section 6.2).

6 Discussion

In this paper, we have presented two major issues of the CPL programmable service in Internet telephony: semantic warnings and feature interactions among CPL scripts. Finally, we summarize some important issues to be discussed further.

[3]Currently, a call scenario involves selection of caller and callee only.



(a) CPL checker (b) FI simulator

Figure 8: Screenshots of the developed tools (prototype)

6.1 *Other semantic warnings*

We have proposed eight classes of the semantic warnings in this paper. However, there could exist other types of semantic warnings. We need to investigate more case studies and some quantitative evaluation to make it clear how much feature interactions can be covered by the proposed eight classes.

Also, we have discussed the semantic warnings and feature interaction in the context of the CPL programmable service only. However, feature interactions can occur between programmable services and the conventional telephony services provided by network convergence framework [15]. This is a very challenging issue and our future work.

6.2 *Architecture for run-time detection*

In order to perform a run-time detection of feature interaction in the CPL environment, we would need a special architecture to compute $Succ(s, c)$ and detect semantic warnings. A possible solution is to deploy an *Feature Interaction server* in the global network. Upon every call setup, signaling servers involving the call upload the relevant CPL scripts to the FI server. Then, the FI server performs appropriate combine operations and then detects feature interactions in the call. The overhead of the script uploading can be reduced if users voluntarily registers their own scripts in a *global service repository* of the FI server beforehand. To implement the architecture, we have to, of course, tackle related issues such as security, privacy and authentication.

6.3 *Resolution of feature interactions*

In the conventional telephony network, once an feature interaction is detected, some resolution schemes, such as feature priorities, are applied. However, in the CPL environment, it is impossible to prepare in advance appropriate resolution schemes. This is the point that the conventional run-time approaches (e.g., [11]) cannot be applied directly.

The Internet basically adopts "*use at your own risk*" policy. So, it would be natural to prompt users to make decision on how the call should be processed by themselves. However, if the programmable service environment is operated on the commercial basis, a certain guideline for users to resolve feature interactions must be prepared. The examination of the resolution schemes is also our feature research.

Acknowledgment

This research was partially supported by: the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Young Scientists (B) (No.13780234, 2002), and Grant-in-Aid for 21st century COE Research (NAIST-IS — Ubiquitous Networked Media Computing, 2003).

References

[1] L. Blair, J. Pang, "Feature Interactions - Life Beyond Traditional Telephony", Proc. of Sixth Int'l. Workshop on Feature Interactions in Telecommunication Networks and Distributed Systems (FIW'00), pp.83-93, May. 2000.

[2] C. Cooper, "XML::Parser - A perl module for parsing XML documents", http://search.cpan.org/author /COOPERCL/XML-Parser-2.31/Parser.pm

[3] E. Derksen, "Overview of libxml-enno", http://www.socsci.umn.edu/ssrf/doc/xml/enno-xml-docs/ users.erols.com/enno/xml/index.html

[4] D. Keck and P. Kuehn, "The feature interaction problem in telecommunications systems: A survey," *IEEE Trans. on Software Engineering*, Vol.24, No.10, pp.779-796, 1998.

[5] J. Lennox and H. Schulzrinne, "Call processing language framework and requirements," Request for Comments 2824, Internet Engineering Task Force,May 2000, http://www.ietf.org/rfc/rfc2824.txt?number=2824

[6] J. Lennox and H. Schulzrinne, "CPL:A Language for User Control of Internet Telephony Service", Internet Engineering Task Force, Jan 2002, http://www.ietf.org/internet-drafts/draft-ietf-iptel-cpl-06.txt

[7] J. Lennox and H. Schulzrinne, "Feature Interaction in Internet Telephony", Proc. of Sixth Int'l. Workshop on Feature Interactions in Telecommunication Networks and Distributed Systems (FIW'00), pp.38-50, May. 2000.

[8] H. Schulzrinne and J. Rosenberg, "Internet Telephony: Architecture and protocols - an IETF perspective," Computer Networks and ISDN Systems, vol.31, pp.237-255, Feb 1999.

[9] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg, "SIP:session initiation protocol", Request for Comments 2543, Internet Engineering Task Force, Feb 2002, http://www.ietf.org/internet-drafts/draft-ietf-sip-rfc2543bis-09.txt

[10] M. Smirnov, "Programming Middle Boxes with Group Event Notification Protocol", Proceedings of the Seventh IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS2002), pp. 198-205, Jan 2002.

[11] S. Tsang and E. H. Magill, "Learning to Detect and Avoid Run-Time Feature Interactions in the Intelligent Network", IEEE Transactions on Software Engineering, Volume 24, Number 10, Oct 1998.

[12] "Feature Interaction in Telecommunications", Vol. I-VI, IOS Press (1992-2000)

[13] ITU-T Recommendations Q.1200 Series: Intelligent Network Capability Set 1, ITU-T (1990)

[14] ITU-T Recommendation H.323, "Packet-Based Multimedia Communications Systems", February 1998.

[15] JAIN initiative, "The JAIN^{TM} APIs: Integrated Network APIs for the Java Platform", http://java.sun.com/products/jain/

[16] "VOCAL: The Vovida Open Communication Application Library", http://www.vovida.org/

[17] NetCentrex^{TM}, "Application Execution and Service Creation Environment", http://www.netcentrex.net/ products/application_server.shtml