Semantic Warnings and Feature Interaction in Call Processing Language on Internet Telephony

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 Technorogy, 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

The Call Processing Language (CPL in short, RFC2824) is an XML-based language, which provides a flexible mean to create end-user services in the Internet Telephony (VoIP). However, the service description by non-experts does not always guarantee reliable services, thus, could be a cause of system faults.

This paper first proposes six classes of the semantic warnings *within the CPL scripts. For a given CPL script, the semantic warnings identify the sources of ambiguities, redundancies and inconsistencies, even though the script is syntactically well-formed with respect to the Document Type Definition.*

Next, we address the problem of Feature Interaction *(FI, in short) between multiple CPL scripts, which is a kind of inconsistent conflict between services. We characterize the FIs as the semantic warnings over the multiple CPL scripts. Then, we propose a new FI detection method to combine the multiple CPL scripts and to detect script-to-script interactions. We also discuss architecture to detect achieve the run-time detection of FIs.*

1 Introduction

As the Internet is widely spread in society, various services are implemented and deployed on the Internet, such as Video on Demand, e-learning, on-line banking and Web services. Thus, the Internet services are now required to be guaranteed-quality, despite the fact that the Internet is a best-effort network.

Among various Internet services, this paper especially focuses on the *Internet telephony*[7], which is also called *Voice over IP* (VoIP, in short). The Internet telephony has been widely studied and standardized at the *protocol level* (i.e., H323[13] by ITU-T, SIP[8] by IETF). The concern is recently shifting to the *service level*; how to provide supplementary services (e.g., call forwarding, call screening, voice mail, etc.) on the Internet telephony.

One of the major issues is the *programmable service*, which allows users to define and create their own supplementary services. The *Call Processing Language* [4, 5] (CPL, in short), based on XML, is recommended as a service description language in RFC2824 of the Internet Engineering Task Force (IETF) [4]. By just putting a CPL script on a local VoIP server, a user can easily deploy a custom service. Thus, the programmable service significantly improves range of user's choice and flexibility in supplementary services on the Internet telephony. Moreover, the creation of the value-added services is also opened to third parties.

There are, however, two major drawbacks of the programmable service with respect to reliability. The first thing is reliability in each single service. The service description by non-expert users cannot always achieve high quality. Although the syntax of the CPL is defined by the Document Type Definition (DTD), compliance with the DTD is not a sufficient condition for correctness of a CPL script[5]. There are enough rooms for the non-expert users to make *semantical mistakes* in the service logic, which could lead to serious system down. As far as we know, there exists no concrete guideline on how to create reliable services with the CPL.

The second thing is reliability over multiple services. Even if each single service works correctly, combined use of multiple services results in unexpected behaviors against user's intension, due to functional conflicts between the services. This is known as *Feature Interaction* (FI, in short). The FI has been considered a major obstacle to the introduction of new features and the provision of reliable services. Therefore, much research has been conducted to tackle this problem (See survey [3] and books [11]). However, most of the research focus on FIs in the conventional telephony.

Proceedings of the 2003 Symposium on Applications and the Internet (SAINT'03)
0-7695-1872-9/03 $17.00 © 2003 IEEE

There are few research to address FIs in the Internet telephony. Lennox et al. address the problem of FIs in the Internet telephony, and present a brief categorization of the FI [4, 6]. However, no method to detect and resolve FIs has been shown yet. Smirnov [9] defined FIs as network resource conflicts in a programmable service environment. However, it is primarily aimed at FIs among network components, but not FIs among end-user services.

To cope with the above two drawbacks, this paper addresses two major issues: *semantic warnings* and *Feature Interaction* in the context of the CPL programmable service. Firstly, we propose six classes of the semantic warnings for individual CPL scripts. As seen in many programming languages, the warnings are not necessarily errors. However, they could cause ambiguity, redundancy and inconsistency, which are often the major source of errors. We believe that the proposed warnings will help users to improve the quality of the CPL scripts. Secondly, we define the FIs between multiple scripts by using the semantic warnings for the single script. The key idea is to characterize the FIs as the semantic warnings over the multiple CPL scripts, each of which is semantically valid. To achieve detection of the FIs, we propose a combine operator, which merges multiple CPL scripts into a single one. Then, we propose a new algorithm to detect FIs among all scripts involved in a call at run time.

2 Programmable services in the Internet telephony

2.1 Call processing language (CPL)

The CPL is an XML-based language to allow end users to describe and control their own *signaling services*. The signaling services involve user location, call delivery, behavior when end systems are busy and the like, and are independent of a particular end system.

Just putting a service description in the CPL (called a *CPL script*) on the local signaling server, a user can control his/her incoming and outgoing calls passing through the signaling server. The CPL is meant to be simple, extensible, easily edited, and independent of operating system or signaling protocol (e.g., H.323 or SIP). To prevent users from executing complex operations and cracking, the CPL has no variables, loops, or ability to run external programs.

First of all, we present a brief review of the CPL definition. The full specification can be found in RFC 2824 [4]. A CPL scripts is composed of mainly four types of constructors: *top-level actions*, *switches*, *location modifiers* and *signaling operations*.

Top-level actions: There are four kinds of actions 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 reusability and modularity. `ancillary` provides additional information for a CPL extension.

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 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 Describing services in CPL

Let us first describe a simple service, Originating Call Screening (OCS, in short), with the CPL. Suppose the following situation: 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.

```
<?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

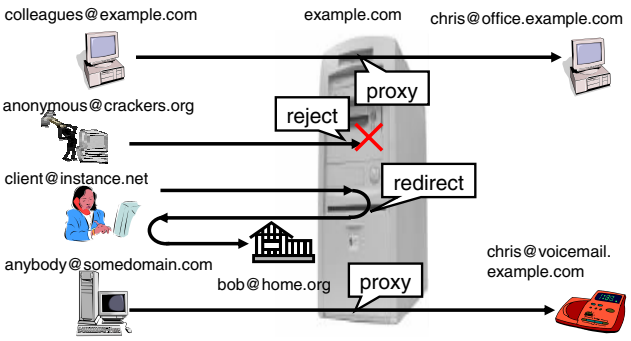

Figure 2. Requirements for CCF

In Figure 1, the first two lines are for declaration of XML and DTD (Document Type Definition). The tag `<cpl>` means the start of a body of the CPL script. The top-level action `<outgoing>` has subsequent 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 will 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 3.2).

The next example is a bit complicated, let it say Conditional Call Filtering (CCF). Suppose a user Chris (`chris@example.com`) and the following requirements, which are also depicted in Figure 2.

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

Figure 3 shows an implementation of Chris's script s_c. The portion surrounded by `<subaction> </subaction>` defines a *subaction*, which is a sub-routine called from the main-routine. `<incoming>` tag 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 host's domain matches `example.com` (`<address`

```
<?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 is permitted" />
    </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 3. A CPL script s_c of CCF

`subdomain-of= "example.com">`), then the location is set to `sip: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 Semantic warnings for single CPL script

3.1 Proposed warnings

There are many ways to make a CPL script *semantically* complex, ambiguous and inconsistent. Here we propose six classes of *semantic warnings*, which capture the source of the semantical flaws.

3.1.1 Multiple forwarding addresses (MF)

Definition: After multiple addresses are set by `<location>` tags, `<proxy>` or `<redirect>` comes.

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 confuse 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 voice mail that immediately answers the call. Then the call never reaches his handy phone.

3.1.2 Identical switches with the same parameters (IS)

Definition: After a switch tag with a parameter, the same switch with the same parameter comes.

Effects: The CPL has no variables or no loop. So, a condition evaluated in the former switch tag never changes in the latter switch tag. Hence, the conditional branch specified in the latter switch is in vain, since the condition must have been evaluated already. This would increase the ambiguity of the CPL script.

Example CPL: Figure 4 shows an example. When a call arrives the user, this script will check the originator's host name. If it matches `home.org`, the call will be proxied to `pattara@home.org`. On the other hand, the originator's host name will be checked again if it matches `home.org` or not. If yes, this script tries to proxy the call to `pattara@mobile.net`. But in fact this proxy is never executed. The second switch is redundant and meaningless.

```
<?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">
      <location url="sip:pattara@home.org">
        <proxy />
      </location>
    </address>
    <otherwise>
      <address-switch field="origin" subfield="host">
        <address subdomain-of="home.org">
          <location url="sip:pattara@mobile.net">
            <proxy />
          </location>
        </address>
        <otherwise>
          <location url="sip:pattara@office.com">
            <proxy />
          </location>
        </otherwise>
      </address-switch>
    </otherwise>
  </address-switch>
 </incoming>
</cpl>
```

Figure 4. Example CPL script of IS

3.1.3 Call rejection in all paths (CR)

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 will make this problem difficult to be found, on the contrary to user's intention.

3.1.4 Address set after address switch (AS)

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 5 shows an example. When the user make an outgoing call, this script will check 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.

```
<?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>
```

Figure 5. Example CPL script of AS

3.1.5 Unused Subactions (US)

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.

3.1.6 Overlapped Conditions in Switches (OS)

Definition: The condition is overlapped among the multiple output tags of a switch.

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. If the conditions specified in the outputs are overlapped (or identical), then the former tag is always taken. In extreme cases, the latter tag is never executed, which is a redundant description.

Example CPL: Figure 6 shows an example. When a call reaches the subscriber, this script will check the destination of the call. If the destination's address contains `pattara`, the call will be proxied to his home telephone address. However, if the destination's address is `pattaraleelaprute`, this script tries to proxy the call to his mobile phone. But in fact, if the proxy to `pattara` has already occurred, proxy to `pattaraleelaprute` will never occur.

```
<?xml version="1.0" ?>
<!DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx
                          CPL 1.0//EN" "cpl.dtd">
<cpl>
  <incoming>
    <address-switch field="destination" >
     <address contains="pattara">
       <location url="sip:pattara@home.example.com">
         <proxy />
       </location>
     </address>
     <address is="pattaraleelaprute">
       <location url=
         "sip:pattaraleelaprute@home.example.com">
         <proxy />
       </location>
     </address>
    </address-switch>
  </incoming>
</cpl>
```

Figure 6. Example CPL script of OS

Note that the above six warnings can occur even if the given CPL script is syntactically well-formed and valid (in the sense of XML).

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

3.2 Default behaviors of CPL

When an execution of a CPL in a signaling server reaches an unspecified condition or an empty signaling operation, the execution follows the *default behavior*, implicitly determined by the server's policy and/or the underlying signaling protocol (See Section 11 of [4] for more details). Here are some examples:

- If no signaling operation is reached in an outgoing action, then the call should be proxied to the destination of the call. If it is in the case of an incoming operation, the server tries to connect the call to an end device of the owner the script.
- If location modifier exists but no signaling operation is specified, the call is proxied or redirect the call to the location, based on the server's standard policy.

These default behaviors can be simulated deterministically for each signaling server. So, even if a CPL execution dares to take a default behavior due to absence of some information, we do not regard the absence as semantic warnings.

Definition (Complete Script): We say that a CPL script is *complete* iff all possible default behaviors are explicitly specified in the script.

We assume that every CPL script on a signaling server can be transformed into a completed script, using auxiliary information on the signaling server. 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, add an appropriate signaling operation to every terminating node of the script, if it is not explicitly specified.

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 7. In the following sections, we assume that all given scripts have been completed with appropriate transformation, unless especially specified.

```
<?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>
</cpl>
```

Figure 7. A complete CPL script s_a of OCS

4 Feature Interactions in CPL scripts

4.1 Characterizing Feature Interactions by semantic warnings

Even if each individual service works correctly, another problem arises when multiple scripts are executed simultaneously, which is called *Feature Interaction* (FI, in short).

The FI can occur also in the Internet telephony[4, 6]. Especially for third parties operating the Internet telephony in a business basis, it would be a serious obstacle. The FI in the Internet telephony is not necessarily the same as the one in the conventional telephony. In the programmable service framework, the users can create, add, delete and modify their custom services at any time. This means that it is impossible to enumerate all possible services. Therefore, we cannot perform FI detection and resolution by off-line analysis. Also, the services are distributed in different servers. This makes FI analysis more difficult and complex.

According to a categorization presented in [4], FIs discussed in this paper can be classified in *script-to-script interactions*. We present an example of FI below.

Interaction between OCS & CCF: Let us recall two services OCS and CCF in Section 2.2, implemented as s_a in Figure 7 and s_c in Figure 3, 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 AS (See Section 3.1.4), although it occurs within the combination of multiple scripts s_a and s_c. Based on this observation, we try to define the FIs as *semantic warnings over multiple scripts*. In order to reduce the problem of FIs into the semantic warning, we need to combine multiple scripts into a single script.

Let s and t be CPL scripts of a call originator and a call terminator, respectively. We expand the incoming actions of t (i.e. portion between `<incoming>` and `</incoming>`), into the proxy operation of s.

Let us consider again the above example. In s_a (in Figure 7), Alice's call is proxied to Chris in `<proxy />` tag. So, actions performed next are Chris's incoming actions in s_c. So, we replace the `<proxy />` in s_a with actions specified in incoming block in s_c, which yields a combined script as shown in Figure 8.

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

Figure 8. A combined CPL script of s_a and s_c

The individual scripts s_a and s_c are both semantically safe. However, the combined script causes a semantic warning AS, since address `bob@home.org` evaluated in `<address>` is set in `otherwise` block. This fact explains the FI between s_a and s_c as a semantic warning.

Now we define a combine operator and FIs for a given pair of scripts. In the following, let s and t be given complete CPL scripts [1], and let c be a given call scenario. Before the composition, we eliminate any subaction `<subaction id=foo>` in s (or t), by expanding the subaction in `<sub ref=foo>`.

Definition (Combine Operator): A *combined script* $r = s \triangleright_c t$ is defined as a CPL script r obtained as follows: substitute `<proxy>` tag (node) in s that is executed in the call scenario c with incoming actions of t.

Definition (Feature Interaction): We say that s and t *interact* with respect to a call scenario c iff both s and t are semantically safe, but $s \triangleright_c t$ is not semantically safe.

Note that the combine operator $\triangleright_c$ does not ruin the syntax well-formedness of s and t. In the DTD of the CPL, both `<proxy>` and the incoming actions are defined as nodes. Therefore, substituting `<proxy>` with the incoming operations does not break the syntax structure of the DTD. Thus, if both s and t are syntactically well-formed, then $s \triangleright_c t$ is also well-formed.

[1] If not, transform them into completed ones.

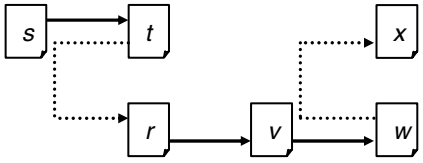

Figure 9. Multiple scripts involved in a call scenario

4.2 Feature Interaction detection

In the previous subsection, we have defined FI between a pair of scripts. However, a call scenario could involve more than two scripts in general, because of successive redirect and proxy operations.

Figure 9 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 FIs in this call scenario c, we must check the semantic warnings for the following six scripts [2]: (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) $s \triangleright_c r \triangleright_c v \triangleright_c x$.

The problem is how to compute the set of scripts to be checked, when a CPL script s of the originator and a call scenario c are given. To implement our FI detection algorithm, we first describe some definitions and assumptions.

A (complete) CPL script ends its execution always with either a signaling operation or empty tag (in this case, the default behavior connects the call to the end device). When an execution of a CPL script is terminated, the following information is supposed to be available.

Exit point: Where in the script is the execution terminated?

Next address: Where is the call directed next?

Processing type: How is the call processed (proxied, redirected, rejected or connected to end system)?

Note that values of the above items vary 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 CPL scirpt s and a a call scenario c, we define the following functions.

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

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

[2] By definition, $\triangleright_c$ is associative, that is, $(s \triangleright_c r) \triangleright_c v = s \triangleright_c (r \triangleright_c v)$

```
scripts Succ(script s, scenario c) {
    R = s;
    if (type(s, c) == 'proxy' && ¬is_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 10. Algorithm Succ(s,c) for computing a set of scripts to be checked

$is\_loop(s, c)$: returns true if s appears more than once in c. It checks if the successive signaling operations form a undesirable loop of the call legs [3].

For example, consider the CPL script s_a in Figure 7. Under a call scenario c_1 where Alice calls Bob, then $next(s_a, c_1)$ = none (since call is rejected), and $type(s_a, c_1)$ = reject. Next, if we suppose a call scenario c_2 where Alice calls Chris, then $next(s_a, c_2) = s_c$, and $type(s_a, c_2)$ = proxy.

For just simplicity, we assume that each script has at most one next script, that is, $next(s, c)$ returns exactly one script or empty. Also, we assume that the proxy and redirect operations always succeed.

Now we present an algorithm to compute a set of combined scripts that must be checked in the FI detection. Figure 10 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. The algorithm Succ first puts the given script s itself in the set R. Next, if the processing type is proxy, Succ 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 9. 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) $s \triangleright_c r \triangleright_c v \triangleright_c x$.

Finally, we are ready to present the FI detection algorithm. We assume that each individual script is semantically safe.

[3] In [5], this function is supposed to be available in underlying signaling protocols.

FI detection algorithm :

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

Output: FI occurs 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 "FI does not occur". Otherwise, return "FI occur" with the corresponding (combined) scripts.

5 Discussion

5.1 Limitation

In this paper, we have proposed six classes of the semantic warnings first, then addressed the Feature Interaction problem in CPL. However, there are possibilities that other types of semantic warnings exist. We need to investigate more case studies and some quantitative evaluation to make it clear how much FIs can be covered by the proposed six classes. Also, we have discussed the semantic warnings and FIs only in the context of the CPL programmable service in the Internet telephony. However, since the underlying protocols (H.323 and SIP) provide interfaces for the conventional telephone network, the FI problem must be considered in the integrated network as well [14], i.e., FIs between the programmable services and the ready-made services in conventional telephony. This is a very challenging issue and our future work.

5.2 Architecture for run-time FI detection

In order to perform a run-time FI detection within the context of programmable service, we would need some special architecture. A possible solution is to deploy an *FI 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 FIs 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.

Once an FI is found, some resolution schemes would be necessary. However, as mentioned in Section 4.1, it is impossible to list all possible CPL scripts, due to the nature of the programmable service. Thus, we cannot prepare resolution schemes that always work well. This is the point that the conventional run-time approaches (e.g., [10]) cannot be applied directly. As for the resolution of FIs, it would be natural to prompt users to make decision on how the call should be processed. The examination of the FI resolution schemes is also our feature research.

References

[1] C. Cooper, "The Perl extension module XML::Parser", http://wwwx.netheaven.com/coopercc/xmlparser/intro.html

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

[3] 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.

[4] 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

[5] 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

[6] 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.

[7] 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.

[8] 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

[9] 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.

[10] 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.

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

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

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

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