Evaluating Semantic Warnings in VoIP Programmable Services with Open Source Environment

Pattara Leelaprute[1], Masahide Nakamura[2], Ken-ichi Matsumoto[2] and Tohru Kikuno[1]
[1]Graduate School of Information Science and Technology, Osaka University, Japan
{pattara, kikuno}@ist.osaka-u.ac.jp
[2]Graduate School of Information Science, Nara Institute of Science and Technology, Japan
{masa-n, matumoto}@is.aist-nara.ac.jp

Abstract

The programmable service for Internet telephony (VoIP) allows end-users or third parties to define their own customized services. However, it imposes a serious drawback that service description created by end-users is likely to contain problems that are semantically ambiguous or inconsistent. To cope with this problem, we have so far proposed semantic warnings, which are the guidelines to guarantee the semantic correctness for the CPL (Call Processing Language) programmable service environment.

In this paper, we evaluate the proposed semantic warnings with practical VoIP system, VOCAL (Vovida Open Communication Application Library). In the experiment, the proposed warnings revealed a semantic redundancy in a ready-made feature of VOCAL. It is also shown that customized features containing the semantic warnings often led VOCAL to problematic situations. Thus, the proposed warnings can help feature provisioning system to detect semantic flaws in programmable service environment.

1. Introduction

Internet telephony (VoIP) has been extensively studied at the network protocol level. Based on the standard protocols (i.e., SIP [4] and H.323 [8]), many companies have already released the commercial services. Internet telephony has the great advantage of low cost, since various tasks, which had been processed by expensive switching hardware and dedicated lines so far, can be performed by software and the Internet. In fact, Internet telephony is about to win the position of traditional POTS (Plain Ordinary Telephone Service).

As the quality of the network level improves, the concern is shifting to the *service level*, that is, how to provide value-added features in Internet telephony. For this, there are two complementary approaches. The first one is to use features in the traditional IN/PSTN networks from IP networks[9]. Although this is quite challenging, it is beyond this paper.

Another approach, which is interesting for us here, is the *programmable service*[3]. It allows end-users or third parties to define and create their own features. The *Call Processing Language* [2] (CPL, in short), based on XML, is recommended as a service description language in RFC2824 of the Internet Engineering Task Force (IETF). By just putting a CPL script on a local server, a user can easily deploy a customized service. Thus, the programmable service significantly improves the range of user's choice and flexibility in supplementary services.

However, there is a major drawback. Since not all users are experts in telephony features, the service description is very likely to contain ambiguity and inconsistency. Although the syntax of CPL is defined by the Document Type Definition (DTD), DTD cannot guarantee the semantic correctness of a CPL script.

In order to cope with this problem, we have previously proposed a notion of *semantic warnings* [5][6][7] for the CPL programmable service environment. Focusing on the structure of CPL and semantic aspects of telephony features, we have identified eight classes of warnings. The warnings are supposed to be a certain guideline by which the users can improve the semantic correctness for their own CPL script. However, we have not yet evaluated the proposed warnings with practical VoIP systems.

The goal of this research is to examine the applicability of the semantic warnings with the practical VoIP systems. For this, we chose an open-source VoIP system, VOCAL (Vovida Open Communication Application Library)[10], as the target test-bed. We applied the semantic warnings to the ready-made features of VOCAL. Also, we wrote customized CPL scripts with the semantic warnings, and had them run on the VOCAL system.

In the result of the experiment, the proposed warnings reveals a semantic redundancy in a ready-made feature of VOCAL. It is also shown that customized features containing the semantic warnings often lead VOCAL to problematic situations. Thus, the proposed warnings can help feature provisioning system to detect semantic flaws in CPL programmable service environment.

The rest of the paper is organized as follows: In Section 2, we review CPL briefly with an example. Section 3 summarizes the proposed semantic warning. Section 4 conducts an experimental evaluation with VOCAL. Finally, we conclude this paper with future work in Section 5.

2. CPL programmable services in VoIP

2.1. Call Processing Language (CPL)

We review definition of the CPL briefly. The full specification can be found in [2][3]. 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. For the outgoing call processing, the location set 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. Example of CPL programmable services

Let us consider the following requirements. 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`.

```
<?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 1. Example of a CPL script

Figure 1 shows an implementation of Chris's script.

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. 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 in CPL

As seen in the previous example, the CPL provides a flexible means of service creation to the end-users. However, naive CPL description causes semantic inconsistency of the service logic. To detect the source of such semantic flaws in individual CPL scripts, we have proposed the concept of *semantic warnings* in our previous research. Note that we use the term *warnings* instead of *errors*, since such flaws are not necessarily errors depending on the intention of the users.

Focusing on constraints of CPL and semantic aspects of telephony features, we have identified eight types of warnings so far. Due to limited pages, we present their definitions and effects only. The example scripts are found in [5][6][7] or subsection 4.4.

Multiple forwarding addresses (MFAD):

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

Effects: By design, 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 where 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 cell phone phone and voicemail that immediately answers the call. Then the call never reaches his cell phone.

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 the server's overhead for parsing the CPL script.

Call rejection in all execution (CRAE):

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

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

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 should have been filtered.

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 that 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.

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 the complexity of the logic.

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.

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 complementary 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.

The above eight warnings can occur even if a given CPL script is syntactically well-formed and valid against DTD. These semantic warnings in a single script can be detected by a simple static (thus, off-line) analysis.

We say that a CPL script is *semantically safe* iff the script is free from the semantic warnings. For example, the CPL script in Figure 1 is semantically safe, since it contains none of the above warnings.

4. Evaluation with VOCAL Internet telephony system

The main goal of this paper is to examine the applicability of the proposed semantic warnings to practical settings. In this section, we conduct an experimental evaluation with one of the practical VoIP systems, VOCAL (Vovida Open Communication Application Library)[10]. The reason why we chose VOCAL is that: (a) it supports the CPL programmable services, and (b) since the application is open-source, many feedbacks and comments are opened for public. Also availability of the source codes makes our experiment efficient.

4.1. VOCAL System

Overview The VOCAL system, developed by an open-source community `vovida.org`, is a collection of server applications that provides VoIP telephony services.

VOCAL contains a SIP stack as its standard protocol. It works as a SIP proxy, which can communicate with a variety of phone appliances, including SIP phones and SIP User Agent (UA) software applications.

At the service level, VOCAL supports CPL-based feature provisioning. Upon each call setup for a user, the *CPL feature server* tells the *redirect server* how the call should be processed, based on the CPL script of the user.

Various other functions, such as translation of SIP-H.323 messages and marshaling to analog phones, are also supported by VOCAL. However, these issues are beyond this paper, since our concern here is VOCAL as the CPL programmable service environment.

CPL features in VOCAL *Features* [1] in VOCAL system are the enhanced functions of the phone system that enable users to do more than simply make and receive phone calls. VOCAL adopts CPL for feature description. The features are controlled by the CPL feature server. In order for users to run own features, VOCAL provides two options.

The first way is to use VOCAL's *ready-made features*, which are originally implemented in the VOCAL system. Using a built-in interface, called *feature provisioning GUI*, users can activate/deactivate the features, and configure the feature setting. Based on the feature configuration, the GUI automatically generates the corresponding CPL scripts. Among various ready-made features in VOCAL, we focus the following five *core* features in the experiment.

Call Blocking(CB): Figure 12 shows the script of CB. CB prevents the user from establishing connections to specified parties such as, 1-900 numbers or 976 numbers.

Calling Party Identity Blocking (CIB): Figure 13 shows the script of CIB. CIB allows a user to control whether or not his/her name and number are delivered.

Call Forward All Calls (CFA): Figure 14 shows the script of CFA. CFA allows a user to re-route all calls to a specified alternative number.

Call Forward No Answer or Busy (CFB) Figure 15 shows the script of CFB. CFB allows a user to specify where a busy or an unanswered call should be re-routed.

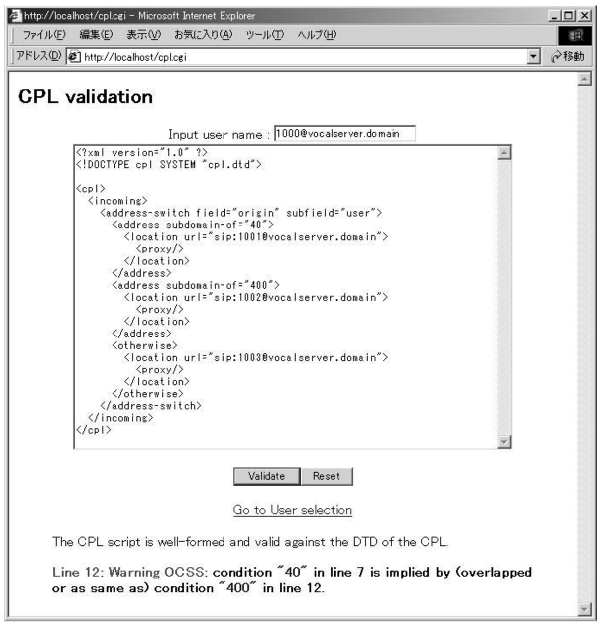
Call Screening (CS) Figure 16 shows the script of CS. CS prevents incoming calls from specified parties to establish connections with the user.

Another option is to write *customized features* from scratch. For this, there is no specific GUI available. The user just carefully writes a CPL script in a designated directory. Then, restarting the feature server makes the new script effective.

4.2. Experiment environment

We have installed VOCAL-1.4.0 on a Linux server (Vine Linux 2.6). As a SIP client (user agent), we have chosen Mi-

1 The terms *features* and (*supplementary*) *services* are often used interchangeably in telecom domain.


http://localhost/cpl.cgi - Microsoft Internet Explorer
ファイル(F) 編集(E) 表示(V) お気に入り(A) ツール(T) ヘルプ(H)
アドレス(D) http://localhost/cpl.cgi
移動
Validate
Reset
Go to User selection

CPL validation

Input user name : 1000@vocalserver.domain

```
<?xml version="1.0" ?>
<!DOCTYPE cpl SYSTEM "cpl.dtd">

<cpl>
  <incoming>
    <address-switch field="origin" subfield="user">
      <address subdomain-of="40">
        <location url="sip:1001@vocalserver.domain">
          <proxy/>
        </location>
      </address>
      <address subdomain-of="400">
        <location url="sip:1002@vocalserver.domain">
          <proxy/>
        </location>
      </address>
      <otherwise>
        <location url="sip:1003@vocalserver.domain">
          <proxy/>
        </location>
      </otherwise>
    </address-switch>
  </incoming>
</cpl>
```

The CPL script is well-formed and valid against the DTD of the CPL.

Line 12: Warning OCSS: condition "40" in line 7 is implied by (overlapped or as same as) condition "400" in line 12.

Figure 2. Screenshots of CPL checker

crosoft Windows Messenger 4.7 running on Windows XP, and used Voice Chat for voice communication. Thus, our test-bed consists of one Linux server (with VOCAL) and several Windows clients (with MS Messenger).

Deployment of the ready-made features is performed by the provisioning GUI with Web Interface on the clients. For creation of the customized feature, we edited the CPL script directly on the server.

We have also developed a tool, called the *CPL checker* to detect the semantic warnings. The CPL checker 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 2 shows a screenshot, where semantic warning OCSS is detected.

4.3. Semantic warnings in ready-made features

First, we apply the warnings to the five ready-made features of VOCAL (See Section 4.1) . Specifically, for each of the ready-made features, we check if the CPL script automatically generated from the provisioning GUI is semantically safe.

Among the five ready-made features with various configurations, we have identified a semantically redundant case in CS by semantic warning OCSS. CS allows a user to configure multiple screening addresses. If the user sets two screening address where the second address is implied by the first one, the second one is always ignored.

Figure 3 shows an example script generated from the GUI. In this script, the user specifies two screening numbers 40 and 400. All calls from addresses containing these numbers are screened (rejected). However, since the address containing 400 also contains 40, the second condition with 400 is always ignored. This case is not necessarily an error, but just a redundancy. However, if the user wants to make a different reject action for the number 400, this redundancy possibly violates user's intension.

For the other four ready-made features, no semantic warnings was found. Thus, it can be said that the ready-made features are relatively safe. In fact, the current set of the ready-made features is not very complicated. However, in future extension, there is enough room for introducing more semantic problems. In the next subsection, we explore this possibility by writing new customized features.

```
<?xml version="1.0" ?>
<!DOCTYPE cpl SYSTEM "cpl.dtd">
<cpl>
<subaction id="rejectcall">
  <reject reason="feature activated"
   status="reject"></reject>
</subaction>
<incoming>
  <address-switch field="origin" subfield="user">
    <address subdomain-of="40">
      <sub ref="rejectcall"></sub>
    </address>
    <address subdomain-of="400">
      <sub ref="rejectcall"></sub>
    </address>
    <address subdomain-of=";">
      <sub ref="rejectcall"></sub>
    </address>
    <otherwise>
      <lookup clear="yes" source="registration"
       timeout="2">
        <success>
          <proxy ordering="first-only"></proxy>
        </success>
        <notfound>
          <sub ref="rejectcall"></sub>
        </notfound>
      </lookup>
    </otherwise>
  </address-switch>
</incoming>
</cpl>
```

Figure 3. Script of CS containing OCSS

4.4. Semantic warnings in customized features

Our interest here is to observe how VOCAL behaves for the customized features containing the semantic warnings. For this purpose, we wrote new customized features in CPL. In each script, we intentionally put a semantic warning.

In the following CPL scripts, we use a four-digit number to represent a user's name (This is due to VOCAL's convention). Also, `vocalserver.domain` denotes our Linux server.

(A) Multiple forwarding address (MFAD) Figure 4 shows the script that contains MFAD. This script is activated when an incoming call arrives at the user. The user sets the forwarding address to three destinations, "1001", "1002" and "7000" simultaneously. Users "1001" and “1002” are not subscribed to any feature. However, “7000” activates the voicemail service and configures it to answer the phone immediately.

Execution result: When this user has an incoming call, the call is proxied to “1001”, “1002” and “7000” simultaneously. However, since “7000” configures voicemail to answer the call immediately, the call never reaches “1001” and “1002”.

Discussion: In the result, we see unreachable terminal (“1001” and “1002”) when MFAD exists in the scripts. In this script, the problem occurs because of the simultaneous ordering of `<proxy>`. In fact, `<proxy>` can include *ordering parameter*—parallel, sequential or first-only to specify the ordering of `<proxy>` (See [3] for more details). However, when the user does not describe this parameter explicitly, *parallel* which is the default action is used. Sometimes, this action brings violated result to the user who does not notice the default action of `<proxy>`. One of the solution for this is to recommend the user to describe parameter of `<proxy>`, explicitly. For `<redirect>`, however, no reasonable solution is available. Since `<redirect>` has no parameter to specify the ordering of redirection, we should recommend the user not to use cascading `<location>`’s with `<redirect>`.

(B) Unused subactions (USUB) Figure 5 shows the script that contains USUB.

Execution result & Discussion: No problem occurs in this script, but a subaction `reject call` that was declared in the subaction part is not used in the body of the script. So, the unused subaction `reject call` is redundant and should be removed.

(C) Call rejection in all execution (CRAE) Figure 6 shows the script that contains CRAE.

Execution result & Discussion: No problem occurs in this script, but by this script, any incoming call is rejected, no matter who the originator is. All actions and evaluated conditions are meaningless after all.

(D) Address set after address switch (ASAS) Figure 7 shows the script that contains ASAS. When the user makes an outgoing call, this script checks the destination of the call. The call should be rejected if the destination address is `5000`, according to the condition specified in `<address>`. However, in the `otherwise` block, the call is proxied to `5000` who should have been rejected.

Execution result: When the user makes a call to `5000`, the call will be rejected. But when the user makes a call to other users, the call will be automatically proxied to `5000` who should have been rejected.

Discussion: It is seen that an inconsistent destination occurs when ASAS exists in the script. The solution for this problem is to recommend user to remove one of the action of addresses that is contradictory to each other. (In this case, `5000` in `<address>` or `5000` in `otherwise` block.)

(E) Overlapped conditions in single switch (OCSS) Figure 8 shows the script that contains OCSS. In this script, the user intends to proxy the incoming call from the address that begins with “40” and “400” to `1001` and `1002` respectively. And then proxy the other incoming calls to `1003`.

Execution result: All of the incoming calls comes from the address that begin with both “40” and “400” (e.g., `4011` or `4001`) are proxied to `1001`. And no call is proxied to `1002`.

Discussion: The problem in this script is due to the order of two implied (overlapped) conditions of *“address begins with 40”* and *“address begins with 400”* of `<address-switch>`. Because CPL evaluates the conditions of switch in the order the tags are represented, the first tag to match is taken. When the condition of *“address begins with 40”*($cond_{40}$) is implied by the condition of *“address begins with 400”*($cond_{400}$), whenever $cond_{400}$ becomes true, $cond_{40}$ is true. Hence, if $cond_{40}$ comes before $cond_{400}$, when the former condition is true, the output of the former condition is always taken and the latter condition is never executed. Actually we can say that unreachable terminal contains in the latter condition. This problem can be solved by recommending users to re-arrange the order of $cond_{40}$ and $cond_{400}$.

(F) Identical actions in single switch (IASS) Figure 9 shows the script that contains IASS.

Execution result & Discussion: This script has a `<address-switch>` which specifies a conditional branch depending on the address of the incoming call. However, the same action of *proxy to* `1001` occurs independently in all conditional branches. So, we can say that this `<address-switch>` is completely meaningless and redundant, and should be removed. In this case, we can just only describe *proxy to* `1001`, instead of describing the content of this switch.

(G) Overlapped conditions in nested switches (OCNS) Figure 10 shows the script that contains OCNS. In this script, if the incoming call comes from the address that begins with “400”, the second address switch evaluates the address again. If the address begins with “40”, the call is assumed to be proxied to `1001`. In the case that the call does not come from the address that begins with “400”, it will be proxied to `1002`. But when OCNS exists in the script, some of these requirements are not satisfied.

Execution result: The incoming call from the address that begins with "400" is proxied to `1001` but the one coming from the address that begins with "40" is not proxied to `1001` but proxied to `1002` instead.

Discussion: The problem in this script is due to the two overlapped conditions of *"address begins with 400" (let it be* $cond_{400}$*)* and *"address begins with 40" (*$cond_{40}$*)* in two `address-switch`. Note that $cond_{400}$ implies $cond_{40}$. Therefore, whenever $cond_{400}$ becomes true, $cond_{40}$ is true. If $cond_{400}$ comes before $cond_{40}$ in nested switch, when $cond_{400}$ is true, $cond_{40}$ is always true. So, the block of $cond_{40}$ is unconditionally executed. Also, in case of $cond_{40}$ having an otherwise block, that block cannot be executed. As a result, the switch of $cond_{40}$ is completely redundant and should be removed. This problem can be solved by recommending user to re-arrange the order of $cond_{400}$ and $cond_{40}$, or remove the switch of $cond_{40}$.

(H) Incompatible conditions in nested switches (ICNS) Figure 11 shows the script that contains ICNS. In this script, if the incoming call comes from the address that begins with "400", the second address switch evaluates the address again. If the address begins with "200", the call is assumed to be proxied to `5000`, otherwise it will be proxied to `1001`. In case that the call is not coming from the address that begins with "400", it will be proxied to `1002`. But when ICNS exists in the script, some of these requirements are not satisfied.

Execution result: The incoming call from the address that begins with "400" is proxied to `1001`. The one coming from the address that begins with "200" is proxied to `1002` instead of `5000` as it is supposed to be.

Discussion: The problem in this script is due to the two *Mutual exclusive* conditions of *"address begins with 400"*(say it $cond_{400}$) and *"address begins with 200"*($cond_{200}$) in two `address-switch`'s. $cond_{400}$ comes before $cond_{200}$ in the nested switch. Hence, whenever $cond_{400}$ is true, $cond_{200}$ will never be executed. Therefore, we can say that $cond_{200}$ is unreachable, completely redundant and should be removed. This problem can be solved by recommending user to avoid describing nested switch with *Mutual exclusive condition.*

As seen in the above experiment, the CPL scripts containing the semantic warnings allow VOCAL to perform unintentional behaviors. Although some of the observed results are not directly connected to the errors, these can often be potential sources of faults. Also, semantic ambiguities and inconsistency characterized by the semantic warnings decreases maintainability of features as well as interoperability with other features.

5. Conclusion

In this paper, we have evaluated the semantic warnings for the CPL programmable service environment with the practical VoIP system, VOCAL. Through the experiment, it is shown that the proposed warnings can be used to detect semantic problems in service logic even in practical settings.

5.1. Discussion

For the ready-made features, we have found that a CPL script of CS (Call Screening) generated from the provisioning GUI can contain semantically redundant portion. For other ready-made features, no warning is detected. We think that this is because the set of ready-made features currently available is not very complex. In other words, the current service provisioning GUI supports only a very limited class of features.

As for the customized features that contain semantic warnings, we have observed unintentional behaviors in VOCAL as expected in the eight types of the semantic warnings. This fact implies that we need careful consideration in the semantic aspect of own scripts when developing more sophisticated features.

In the context of programmable services, feature developers are users (or third parties). Therefore, not all users are experts in telephony feature logic. Thus, the feature provisioning system must possess a mechanism to validate semantics of the logic. If the validation is not passed, suggestion and recommendation to resolve the problem should be given to the users. The proposed warnings are expected to help to implement such a back-end of the provisioning system.

5.2. Future work

In our previous research [5][6], we addressed *feature interaction problem*[1], which is known as a functional conflict among features. We characterized feature interactions as semantic warnings over multiple scripts, and proposed a detection method. We are currently conducting evaluation of this method with VOCAL.

It is also essential in our future work to clarify a certain class of the semantic flaws characterized by the proposed warnings. It is currently difficult to prove all the semantic flaws in CPL scripts can be exhausted by the proposed eight warnings, as the general programming languages cannot do so. Identifying such a class makes applicability of the semantic warnings clearer, and this is our challenging goal.

```
<?xml version="1.0" ?>
<!DOCTYPE cpl SYSTEM "cpl.dtd">
<cpl>
  <incoming>
    <location url="sip:1001@vocalserver.domain">
    <location url="sip:1002@vocalserver.domain">
    <location url="sip:7000@vocalserver.domain">
      <proxy>
      </proxy>
    </location>
    </location>
    </location>
  </incoming>
</cpl>
```

Figure 4. Example of MFAD

```
<?xml version="1.0" ?>
<!DOCTYPE cpl SYSTEM "cpl.dtd">
<cpl>
  <subaction id="rejectcall">
    <reject reason="feature activated"
     status="reject"></reject>
  </subaction>
  <incoming>
    <location url="sip:1001@vocalserver.domain">
      <proxy/>
    </location>
  </incoming>
</cpl>
```

Figure 5. Example of USUB

```
<?xml version="1.0" ?>
<!DOCTYPE cpl SYSTEM "cpl.dtd">
<cpl>
  <incoming>
    <address-switch field="origin" subfield="user">
      <address subdomain-of="1">
        <reject status="reject" reason=
         "I don't accept call from subdomain of 1"/>
      </address>
      <address subdomain-of="2">
        <reject status="reject" reason=
         "I don't accept call from subdomain of 2"/>
      </address>
      </address>
      <otherwise>
        <reject status="reject" reason=
         "I don't accept call from anyone"/>
      </otherwise>
    </address-switch>
  </incoming>
</cpl>
```

Figure 6. Example of CRAE

```
<?xml version="1.0" ?>
<!DOCTYPE cpl SYSTEM "cpl.dtd">
<cpl>
  <outgoing>
    <address-switch field="original-destination"
     subfield="user">
      <address subdomain-of="5000">
        <reject status="reject"
         reason="I don't call 5000"></reject>
      </address>
      <otherwise>
        <location url="sip:5000@vocalserver.domain">
          <proxy/>
        </location>
      </otherwise>
    </address-switch>
  </outgoing>
</cpl>
```

Figure 7. Example of ASAS

```
<?xml version="1.0" ?>
<!DOCTYPE cpl SYSTEM "cpl.dtd">
<cpl>
  <incoming>
    <address-switch field="origin" subfield="user">
      <address subdomain-of="40">
        <location url="sip:1001@vocalserver.domain">
          <proxy/>
        </location>
      </address>
      <address subdomain-of="400">
        <location url="sip:1002@vocalserver.domain">
          <proxy/>
        </location>
      </address>
      <otherwise>
        <location url="sip:1003@vocalserver.domain">
          <proxy/>
        </location>
      </otherwise>
    </address-switch>
  </incoming>
</cpl>
```

Figure 8. Example of OCSS

```
<?xml version="1.0" ?>
<!DOCTYPE cpl SYSTEM "cpl.dtd">
<cpl>
<incoming>
  <address-switch field="origin" subfield="user">
    <address subdomain-of="30">
      <location url="sip:1001@vocalserver.domain">
        <proxy/>
      </location>
    </address>
    <address subdomain-of="40">
      <location url="sip:1001@vocalserver.domain">
        <proxy/>
      </location>
    </address>
    <otherwise>
      <location url="sip:1001@vocalserver.domain">
        <proxy/>
      </location>
    </otherwise>
  </address-switch>
</incoming>
</cpl>
```

Figure 9. Example of IASS

```
<?xml version="1.0" ?>
<!DOCTYPE cpl SYSTEM "cpl.dtd">
<cpl>
  <incoming>
    <address-switch field="origin" subfield="user">
      <address subdomain-of="400">
        <address-switch field="origin" subfield="user">
          <address subdomain-of="40">
            <location url=
              "sip:1001@vocalserver.domain">
              <proxy/>
            </location>
          </address>
          <otherwise>
            <reject status="reject"></reject>
          </otherwise>
        </address-switch>
      </address>
      <otherwise>
        <location url="sip:1002@vocalserver.domain">
          <proxy/>
        </location>
      </otherwise>
    </address-switch>
  </incoming>
</cpl>
```

Figure 10. Example of OCNS

```
<?xml version="1.0" ?>
<!DOCTYPE cpl SYSTEM "cpl.dtd">
<cpl>
  <incoming>
    <address-switch field="origin" subfield="user">
      <address subdomain-of="400">
        <address-switch field="origin"
         subfield="user">
          <address subdomain-of="200">
            <location url=
             "sip:5000@vocalserver.domain">
              <proxy/>
            </location>
          </address>
          <otherwise>
           <location url=
             "sip:1001@vocalserver.domain">
              <proxy/>
            </location>
          </otherwise>
        </address-switch>
      </address>
      <otherwise>
        <location url=
         "sip:1002@vocalserver.domain">
          <proxy/>
        </location>
      </otherwise>
    </address-switch>
  </incoming>
</cpl>
```

Figure 11. Example of ICNS

```
<?xml version="1.0" ?>
<!DOCTYPE cpl SYSTEM "cpl.dtd">
<cpl>
<subaction id="rejectcall">
  <reject reason="feature activated"
   status="reject"></reject>
</subaction>
<outgoing>
  <address-switch field="original-destination"
   subfield="user">
    <address subdomain-of="1900">
      <sub ref="rejectcall"></sub>
    </address>
    <address subdomain-of=";">
      <sub ref="rejectcall"></sub>
    </address>
    <otherwise>
      <lookup clear="yes" source="registration"
      timeout="2">
        <success>
          <proxy ordering="first-only"></proxy>
        </success>
        <notfound>
          <sub ref="rejectcall"></sub>
        </notfound>
      </lookup>
    </otherwise>
  </address-switch>
</outgoing>
</cpl>
```

Figure 12. Script of CB

```
<?xml version="1.0" ?>
<!DOCTYPE cpl SYSTEM "cpl.dtd">
<cpl>
  <outgoing>
    <lookup clear="yes" source="registration">
      <success>
        <remove-location
          location="complete_calleridblock">
          <proxy ordering="first-only"/>
        </remove-location>
      </success>
        <notfound>
          <reject status="reject"/>
        </notfound>
    </lookup>
  </outgoing>
</cpl>
```

Figure 13. Script of CIB

```
<?xml version="1.0" ?>
<!DOCTYPE cpl SYSTEM "cpl.dtd">
<cpl>
  <incoming>
    <location clear="yes" url="sip:1900">
      <redirect></redirect>
    </location>
  </incoming>
</cpl>
```

Figure 14. Script of CFA

```
<?xml version="1.0" ?>
<!DOCTYPE cpl SYSTEM "cpl.dtd">
<cpl>
<incoming>
  <lookup clear="yes" source="sip:3000;user=phone">
    <success>
      <proxy ordering="first-only" timeout="15">
        <busy>
          <location clear="yes"
             url="sip:1900;user=phone">
            <redirect></redirect>
          </location>
        </busy>
        <noanswer>
          <location clear="yes"
             url="sip:1900;user=phone">
            <redirect></redirect>
          </location>
        </noanswer>
        <failure>
          <location clear="yes" url="sip:1900">
            <redirect></redirect>
          </location>
        </failure>
      </proxy>
    </success>
    <notfound>
      <reject status="reject"></reject>
    </notfound>
  </lookup>
</incoming>
</cpl>
```

Figure 15. Script of CFB

```
<?xml version="1.0" ?>
<!DOCTYPE cpl SYSTEM "cpl.dtd">
<cpl>
<subaction id="rejectcall">
  <reject reason="feature activated"
     status="reject"></reject>
</subaction>
<incoming>
  <address-switch field="origin" subfield="user">
    <address subdomain-of="4000">
      <sub ref="rejectcall"></sub>
    </address>
    <address subdomain-of=";">
      <sub ref="rejectcall"></sub>
    </address>
    <otherwise>
      <lookup clear="yes" source="
         registration" timeout="2">
        <success>
          <proxy ordering="first-only"></proxy>
        </success>
        <notfound>
          <sub ref="rejectcall"></sub>
        </notfound>
      </lookup>
    </otherwise>
  </address-switch>
</incoming>
</cpl>
```

Figure 16. Script of CS

References

[1] 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.

[2] 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

[3] 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

[4] 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

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

[6] M. Nakamura, P. Leelaprute, K. Matsumoto, T. Kikuno, “Detecting script-to-script interactions in call processing language”, Proc. of Seventh Int’l. Workshop on Feature Interactions in Telecommunication Networks and Distributed Systems (FIW’03), pp.215-230, Jul. 2003.

[7] P. Leelaprute, M. Nakamura and T. Kikuno, “Characterizing semantic warnings of service description in Call Processing Language on Internet telephony”, International Technical Conference On Circuits/Systems, Computers and Communications (ITC-CSCC2002), Vol. 1, pp. 556-559, Jul. 2002.

[8] ITU-T Recommendation H.323, “Packet-Based Multimedia Communications Systems”, February 1998.

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

[10] “VOCAL: The Vovida Open Communication Application Library”, http://www.vovida.org/