Available online at www.sciencedirect.com

SCIENCE DIRECT®

Computer Networks 45 (2004) 605–624

COMPUTER NETWORKS

www.elsevier.com/locate/comnet

On detecting feature interactions in the programmable service environment of Internet telephony

Masahide Nakamura [a,*], Pattara Leelaprute [b], Ken-ichi Matsumoto [a], Tohru Kikuno [b]

[a] *Graduate School of Information Science, Nara Institute of Science and Technology, 8916-5, Takayama, Ikoma, Nara 630-0101, Japan*
[b] *Graduate School of Information Science and Technology, Osaka University, 1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan*

Available online 23 March 2004

Abstract

This paper presents a new method to tackle the feature interaction problem in Internet telephony with the CPL (Call Processing Language) programmable service environment. To cope with the problems of the programmable service, we first propose a notion of semantic warnings, which are guidelines for non-experts to assure semantic correctness of individual CPL scripts. Then, we define feature interactions as semantic warnings over multiple CPL scripts. On the basis of this definition, we propose a method for detecting feature interactions. We conduct an experimental evaluation with an open-source VoIP system. The results show that the proposed method identifies a semantic redundancy in a ready-made feature and five interactions among pairwise combinations of the features. We also discuss the applicability and limitations from the viewpoint of implementation.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Internet telephony; VoIP; Call processing language; Semantic warnings; Feature interactions; XML; VOCAL

1. Introduction

Recent advances in networks enable the development of increasingly sophisticated communication services. For providers of such services, supplementary features (or simply *features*) are playing an essential role to offer attractive value-added services. For example, in the traditional *Public Switched Telephone Network* (PSTN), a number of features, such as Call Waiting and Freephone Billing, were developed within the architecture of the *Intelligent Network* (IN) [20,22] to meet evolving market needs.

An individual feature is usually specified without knowledge of other features to achieve modular design and implementation. However, as two or more features can be activated simultaneously, problems arise when some functions of one feature conflict with those of others. These conflicts are generally called *feature interactions* [3]. They cause unexpected (often undesirable) behavior for customers and/or systems, possibly leading the whole system to malfunction. Thus, feature

[*] Corresponding author. Tel.: +81-743-72-5312; fax: +81-743-72-5319.
E-mail addresses: masa-n@is.aist-nara.ac.jp (M. Nakamura), pattara@ist.osaka-u.ac.jp (P. Leelaprute), matumoto@is.aist-nara.ac.jp (K.-i. Matsumoto), kikuno@ist.osaka-u.ac.jp (T. Kikuno).

interactions can severely decrease the total quality of service.

During the last decade, a number of research efforts to tackle the feature interaction problem have been conducted (see books [19] and surveys [2,6]). Most of them deal with the classic telephony features of PSTN/IN networks. Interactions in this domain are now fairly well understood. Recently, feature interactions in new application areas are being actively discussed, including e-mail [4], building control [10] and Web services [18]. The conventional approaches can be helpful to understand some interactions in these emerging application. However, many domain-specific problems remain to be solved.

This paper focuses on *Internet telephony* (VoIP) [15] among such emerging applications. Internet telephony has been extensively studied at the *network protocol level* (i.e., SIP [13] and H.323 [21]). The concern is now shifting to the *application service level*, i.e., how to provide value-added features on Internet telephony. For this, there are two complementary approaches. The first one is to reuse the PSTN/IN features from IP networks through APIs (e.g., JAIN [23]). Although this is quite challenging, we do not discuss it in this paper.

Another approach, which is the one of interest here, is the *programmable service* [8] approach. It allows end-users or third parties to define and create their own features. The *Call Processing Language* (CPL) [7], based on XML, is recommended as a feature description language in RFC2824 of IETF. CPL is gaining popularity, and major VoIP systems (e.g., [24,25]) have adopted it as a feature description language. Just by writing feature definitions in CPL on the *local* VoIP server, users can easily deploy their own features. Also, in CPL, users can delete and modify the features at any time. Thus, the programmable service significantly improves the range of the user's choice and flexibility.

Feature interactions also occur in Internet telephony. They are becoming an increasingly pressing problem, since more and more sophisticated features are created and deployed [1,9]. Especially in the context of programmable services, the following issues are domain-specific problems to which the conventional approaches cannot be directly applied.

(P1) *Features created by non-expert users*: In programmable services, end users have to control the quality of features, although most end users do not have the expertise of telecom engineers. Therefore, users are very likely to create features without concern for logical consistency and correctness of the features, and even less so for feature interactions with others.

(P2) *Distributed feature provision*: The features defined 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 (e.g., [12,16]), nor prepare resolution schemes in advance such as feature priority [5,17].

The goal of this paper is to establish a definition and describe a detection method for feature interactions within the CPL programmable service environment. To achieve this, we propose two new methods corresponding to the above problems P1 and P2.

First, we present *semantic warnings* for CPL scripts to address problem P1. The syntax of CPL is strictly defined by a DTD (Document Type Definition). However, compliance with the DTD does not guarantee the semantic correctness of a CPL script. As far as we know, there exist no guidelines for users to assure the semantic correctness of individual CPL scripts. Focusing on the structure of CPL and semantic aspects of telephony features, we have identified eight classes of warnings. The warnings can provide certain guidelines by which the users can improve the semantic correctness of their own CPL scripts.

Next, to address problem P2, we propose a new definition of feature interaction, and of its detection method in the CPL environment. In general, feature interactions can be defined informally as violations of feature requirements (or intentions) that are caused by the combination of multiple features. In the CPL environment, the violation occurs when a CPL script is not executed as described, under the influence of other CPL scripts within the call. Note that the violation can be observed only at run time, and cannot be predicted reasonably by off-line analysis. Thus, the new definition of feature interaction must be dependent on call scenarios at run time, which is significantly different from the definitions in the literature [6,19]. Our key idea is to define feature interaction (i.e., the violation) as a semantic warning over multiple CPL scripts, each of which is semantically valid. We propose a combine operator and some new notions for CPL scripts (i.e., complete, safe).

We evaluate the proposed method through application to a practical VoIP system—*VOCAL* (Vovida Open Communication Application Library) [25]. In the experiment, the proposed semantic warnings revealed a semantic redundancy in a ready-made feature of VOCAL. Also, it was shown that CPL scripts containing warnings could lead VOCAL to problematic behaviors. Moreover, the proposed detection method has identified five feature interactions among pairwise ready-made features.

We also examine the applicability and limitations of the proposed method from the viewpoint of *implementation*. We demonstrate that the proposed method is well suited to the detection of script-to-script interactions in a single VoIP server, or server-to-server interactions within a trusted network environment.

This paper was originally published as a workshop paper in the International Workshop FIW2003 [11]. Changes were made for this version, most significantly the addition of the evaluation and implementation parts. We believe that these new results clarify the applicability and limitations of the proposed method against practical VoIP systems.

2. CPL programmable services in VoIP

2.1. Call Processing Language (CPL)

We first review the definition of CPL briefly. A *CPL script* is XML text whose syntax definition is prescribed by the CPL's DTD (see [7,8] for the *full specification*). Each user is supposed to have at most *one* CPL script at a time. [1] 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 first invoked when a CPL script is executed: `outgoing` (or `incoming`) they specify a tree of actions taken on the user's outgoing call (or incoming call, respectively). `subaction` describes a subroutine to increase re-usability and modularity.

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

Location modifiers: CPL has an abstract model, called *location set*, for locations to which a call is to be directed. The set of locations is stored as an implicit global variable during call processing. For outgoing call processing, the location is initialized to the destination address of the call. For incoming call processing, the location set is initialized to the empty set. During 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 the script; `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 features

We start with a simple feature, namely Originating Call Screening (OCS, for short). Suppose the following requirement: Alice (`alice@instance.net`) wants to block any outgoing calls from her end system to Bob (`bob@home.org`).

Fig. 1 shows an implementation of Alice's script s_a. In Fig. 1, the first three lines represent the

[1] Some practical systems (including our evaluation test-bed VOCAL) do not follow this recommendation, because of maintainability of ready-made features. We discuss this problem later in Section 5.4.

```
<?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>
```

Fig. 1. A CPL script s_a of OCS.

declaration of XML and DTD. The tag <cpl> means the start of the 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 proxied to the destination address (This is done by *default behavior* of CPL, although the proxy operation is not explicitly specified. See Section 4.2.1).

The next example is a bit more complicated. Chris (chris@example.com) wants to

- receive calls from domain example.com at office chris@office.example.com.
- reject any call from users 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.

Fig. 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 invoked from the main-routine, where the call is proxied to the voicemail of Chris. <incoming> specifies actions activated when Chris receives an incoming call, where each of the above requirements is coded as a branch (output) of <address-switch>.

```
<?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>
```

Fig. 2. A CPL script s_c of DCF.

3. Semantic warnings for CPL scripts

As described in [7], compliance with the DTD is not a sufficient condition for the correctness of a CPL script. This is because DTD is not powerful enough to express *semantic aspects* of the scripts. Therefore, there is enough room for non-expert users to create semantic flaws in the feature logic. To detect the source of such semantic flaws in individual CPL scripts, we propose the concept of *semantic warnings*. Note that we use the term *warnings* instead of *errors*, since they do not necessarily identify errors depending on the intention of the users. [2] Focusing on constraints of CPL and semantic aspects of telephony features, we have identified eight types 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.

[2] The use of the term *warning* is borrowed from those of compilers.

Effects: By design, CPL allows calls to be proxied (or redirected) to multiple locations simultaneously, by cascading `<location>` tags (i.e., a *forking proxy* [13]). However, this drastically increases the potential of race conditions among multiple end systems. A typical example is that a user simultaneously sets the forwarding address to his cell phone and voicemail that immediately answers the call. Then, the call never reaches his cell phone.

Example CPL: Fig. 3(A) shows an example. The user is setting three different addresses `1001`, `1002`, `7000`. When the user makes a call, the call is proxied simultaneously to the three destinations, among which race conditions may be expected.

3.2. Unused subactions (USUB)

Definition: A subaction (let it be `<subaction id="foo">`) exists, but it is not called `<sub ref="foo">`.

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

Example CPL: Fig. 3(B) shows an example. In this script, a subaction `rejectcall` is declared in the subaction part, but it is not used in the body of the script. Hence, `rejectcall` is redundant and should be removed.

3.3. Call rejection in all execution paths (CRAE)

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

Effects: The call is rejected no matter which path in the script is taken. No call processing is performed, and all actions executed so far are in vain. If the user wants to reject all calls explicitly, this might not be a problem. However, complex conditional branches and deeply nested tags make this problem difficult to find, and could be contrary to the user's intention.

Example CPL: Fig. 3(C) shows an example. By this script, all incoming calls are 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 `<otherwise>` block.

Effects: The `<otherwise>` block is executed when the current address does not match the one specified in `<address>`. If the same address is set as a new current address in an `<otherwise>` block, then a violation of the conditional branch might occur. A typical example is that a certain address is blocked by `<address-switch>`, however, the call is proxied to the address in `<otherwise>`.

Example CPL: Fig. 3(D) shows an example. 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 prefixed with `5000`, according to the condition specified in `<address>` (`subdomain-of` works as "prefix matching" when used with digits). However, in the `<otherwise>` block, the call is proxied to `5000@vocalserver.domain`, which must have been rejected.

3.5. Overlapping 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: CPL evaluates multiple conditions in the order in which their tags are presented, and the first tag to match is taken. By the above definition, whenever $cond_{A2}$ becomes true, $cond_{A1}$ is true. Hence, $cond_{A1}$ is always taken and $cond_{A2}$ tag is never executed, which is a redundant description.

Example CPL: Fig. 3(E) shows an example. The first condition (`subdomain-of="40"`) is implied by the second one (`subdomain-of="400"`), since `40` is a substring of `400`. As a result, the second condition block is shadowed and thus is unreachable.

```
<?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>
```

(A)

```
<?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>
```

(B)

```
<?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>
```

(C)

```
<?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>
```

(D)

```
<?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>
```

(E)

```
<?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>
```

(F)

```
<?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>
```

(G)

```
<?xml version="1.0" ?>
<!DOCTYPE cpl SYSTEM "cpl.dtd">
<cpl>
  <incoming>
    <address-switch field="origin" subfield="user">
      <address is="sip:4000@vocalserver.domain">
        <address-switch field="origin"
         subfield="user">
          <address is="sip:2000@vocalserver.domain">
            <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>
```

(H)

Fig. 3. Example CPL scripts containing semantic warnings. (A) Example of MFAD, (B) example of USUB, (C) example of CRAE, (D) example of ASAS, (E) example of OCSS, (F) example of IASS, (G) example of OCNS and (H) example of ICNS.

3.6. Identical actions in single switch (IASS)

Definition: The same action is specified in all conditions of a switch.

Effects: Whichever conditional branch is taken, 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.

Example CPL: Fig. 3(F) shows an example. This script specifies a conditional branch of the caller's address. However, whatever the address is, the call is proxied to `1001@vocalserver.domain`, and thus, the switch is not needed.

3.7. Overlapping 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 Bs condition block] and [$cond_B$ implies $cond_A$].

Effects: Since CPL has no variable assignment, any condition that is evaluated to be true (or false) remains true (or false, respectively). [3] Assume that $cond_B$ implies $cond_A$. Bs condition block, in which A is specified, is executed only when $cond_B$ is true. By the assumption, $cond_A$ always becomes true when evaluated, therefore, As condition block is unconditionally executed. If A has an otherwise block, then the block is unreachable.

Example CPL: Fig. 3(G) shows an example. When an incoming call arrives, the script first checks the address of the caller. If the address contains `400`, then the second switch checks if `40` is contained. However, since the condition for the second switch is implied by the first one, it is a redundant description. Also, `<reject/>` in `<otherwise>` is unreachable.

[3] Indeed, `<location>` virtually defines variable assignment to the location set. However, the location set is referred to by the signaling operations only, but is not evaluated explicitly in any conditions in the script.

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 Bs condition block) and ($cond_A$ and $cond_B$ are mutually exclusive), or
(β) (A is nested in Bs otherwise block) and ($cond_A$ implies $cond_B$).

Effects: Let us consider (α) first. Bs 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, As condition block is unexecutable. (β) is the complementary case of (α), where As condition block is also unreachable.

Example CPL: Fig. 3(H) shows an example for ICNS (α). When an incoming call arrives, the script first checks the caller's address. If the address matches `4000@vocalserver.domain`, the second address-switch evaluates the caller's address. If the address matches `2000@vocalserver.domain`, the call is proxied to `5000@vocalserver.domain`. However, this proxy operation never occurs, since the address cannot match both `4000` and `2000`, simultaneously.

The above eight warnings can occur even if a given CPL script is syntactically valid against the DTD. The semantic warnings in a single script can be detected by a simple *static* (thus, off-line) analysis. Theoretically speaking, it is not easy to statically determine a condition overlap or implication in general logic formulae. However, a condition in CPL is basically constructed by the fixed number of parameters and static values (quoted by "). Hence, evaluating the overlap and inclusion among these static values associated with a certain parameter practicably allows us to detect OCSS, OCNS, ICNS in a static way, as implemented in our tool CPL checker (see Section 5.2).

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

4. Feature interactions in CPL environment

4.1. Key idea

Even if each user creates a safe script, feature interactions may occur when multiple scripts are executed simultaneously. Since each user has at most one CPL script (see Section 2.1), interactions occur among different scripts owned by different users.

Interaction between OCS and DCF: Let us recall two features OCS and DCF in Section 2.2, implemented as s_a in Fig. 1 and s_c in Fig. 2, respectively. Suppose 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 can make 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 interaction in the above example is quite similar to the semantic warning ASAS. However, this situation happens within the *combination of multiple scripts*. Based on this observation, we extend the concept of semantic warnings over multiple scripts, and characterize feature interactions by the semantic warnings. Before formalizing feature interactions in the CPL environment, we define some new notions with respect to CPL scripts in the next subsection.

4.2. Preliminaries

4.2.1. Complete CPL scripts

When the execution of a CPL script reaches an unspecified condition or an empty signaling operation, it follows a *default behavior* (see Section 11 of [7] for more details). Here are some examples:

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

The default behaviors are taken *implicitly* from the viewpoint of users, based on the server's policy and the underlying protocol. [4] Hence, they may sometimes contradict a user's intention; the implicitness should be hopefully 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 (i.e., all actions taken are explicitly specified in the script).

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

(a) Make all conditional branches complementary. For instance, an `<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 that has any location modifier.
(c) Add empty `<incoming>` or `<outgoing>` blocks if either is not present.

For example, consider again the CPL script in Fig. 1. This script is not complete, since there is no

[4] For example, the VOCAL system adopts `redirect` for D3.

```
<?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>
```

Fig. 4. A complete CPL script s_a of OCS.

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 Fig. 4.

4.2.2. Successor functions

Suppose that we have a complete and safe 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 immediately after s. 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 next script varies depending on a given *call scenario*, which is dynamically characterized by a set of parameters, typically including destination address, originator address, time, caller preferences, and so on. Note also that the number of the next script is at most one, since s is safe (i.e., s must be free from MFAD). These dynamic parameters are supposed to be obtained from the call setup message issued by a user agent. 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 and safe CPL script s and a call scenario c, we define the following functions:

exit(s, c): returns the tag of a signaling operation executed at the end of s under c.

next(s, c): returns the next CPL script executed following s under c.

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

For instance, consider again the example in Section 2.2, and the scripts s_a in Fig. 4 and s_c in Fig. 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

First, let us consider two complete scripts s and t only. To define feature interactions between s and t, we need to capture a combined behavior of s and t. For this purpose, we introduce a *combine operator*. Intuitively, the combine operator merges two scripts s and t such that *t is executed after s*. This 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 reverts 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 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:

Step 1: 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>
```

Fig. 5. A combined CPL script $s_a \rhd_{c_2} s_c$.

Step 2: Let $In(t)$ be the body of the incoming action of t (i.e., the portion surrounded by `<incoming>` $\cdots$ `</incoming>`). In s, replace `<proxy/>` pointed to by $exit(s, c)$ with `<remove-location>` $In(t)$ `</remove-location>`. Let the resulting script be r.

The combine operator $\rhd_c$ 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 simulates that the location set is initialized to empty when the incoming action occurs (see Section 2.1). Note that the combine operation does not compromise the syntax structure of s and t, since both `<remove-location>` $In(t)$ `</remove-location>` and `<proxy/>` are defined as *nodes* in the DTD of CPL. Hence, if both s and t are syntactically valid, then $s \rhd_c t$ is also valid.

Based on the key idea discussed in Section 4.1, we now define feature interactions among two scripts s and t as semantic warnings over $s \rhd_c t$. Moreover, we also regard the *forwarding loop* as an interaction (i.e., s proxies the call to t, while t proxies the call to s, too). [5]

Definition (*Feature interaction among two scripts*). Let s and t be complete scripts, and let c be a call scenario. We say that s and t *interact* with respect to c, iff either of the following conditions holds: (a) both s and t are semantically safe, but $s \rhd_c t$ is not safe, or (b) $next(s, c) = t$ and $next(t, c) = s$ hold.

Let us consider two scripts s_a (in Fig. 4) and s_c (in Fig. 2). Also, take a scenario c_2 in Table 1. Fig. 5

[5] The forwarding loop can be managed by a *loop detection mechanism* in the underlying protocol [13]. However, how to process the detected loop is left to the application, thus we regard it as an interaction.

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 the `<otherwise>` block. This is just the interaction explained in Section 4.1.

4.4. Feature interaction detection

A call can involve more than two scripts, because of successive redirect and proxy operations. Hence, 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. Let $s_1, s_2, \ldots, s_n$ 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 either of the following conditions holds:

(a) all of $s_i (0 \leqslant i \leqslant n)$ are semantically safe, but there exists some k $(1 \leqslant k \leqslant n)$ such that $s_0 \triangleright_c s_1 \triangleright_c \cdots \triangleright_c s_k$ is not semantically safe, or
(b) there exists some i, j $(i < j)$ such that $s_i = s_j$.

Fig. 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. Fig. 7 shows pseudo code to

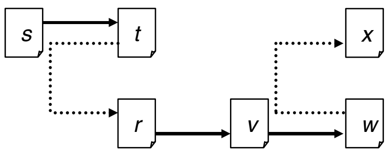


Fig. 6. Multiple scripts involved in a call scenario.

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

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

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 abstract procedure checks if script `t` forms the forwarding loop in the call scenario `c`, by using a loop detection mechanism. We assume that the loop detection is performed at run time in our implementation framework, which will be discussed later in Section 6. 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, Succ combines s with its successive scripts t that are recursively computed by setting the next script as the initial script. Then, it puts 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. For example, for the call scenario c in Fig. 6, Succ(s, c) computes the above six scripts.

Finally, we 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. Also, if Succ(s, c) stops due to a forwarding loop (detected by `check_loop`), return "loop interaction occurs".

5. Evaluation with VOCAL Internet telephony system

5.1. VOCAL VoIP System

The VOCAL system, developed by an open-source community `vovida.org` [25], is a collection of server applications that provides VoIP telephony services. The reason why we chose VOCAL as our evaluation test-bed is that: (a) it supports the CPL programmable service environment, and (b) the source code, feedback and comments are open for public use. 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 application service level, VOCAL supports CPL-based feature provisioning. Upon each call setup from a user, a *feature server* tells the *redirect server* how the call should be processed, based on the CPL script of the user.

VOCAL provides two options for a user to deploy a feature. The first option is to use the VOCAL *ready-made features*, whose default templates are already prepared by VOCAL. Using a built-in interface, called the *feature provisioning GUI*, a user can activate, deactivate and configure features. Based on the feature configuration, the GUI automatically generates the corresponding CPL scripts. In the following, we briefly explain the five *core* features. Each of the features is classified into *originating features* (CB and CIB) or *terminating features* (CFA, CFB and CS), depending on whether the feature is activated by a caller or a callee.

Call blocking (CB): CB prevents the user from establishing connections to specified parties such as 1–900 or 976 numbers.

Calling party identity blocking (CIB): CIB allows a user to control whether or not his/her name and number are delivered.

Call forward all calls (CFA): CFA allows a user to re-route all calls to a specified alternative number.

Call forward no answer or busy (CFB): CFB allows a user to specify where a busy or unanswered call should be re-routed.

Call screening (CS): CS prevents incoming calls from specified parties from establishing connections with the user.

The second option is to write a CPL script from scratch. We call these scratch-built features *customized features* to distinguish them from the ready-made features. For this, there is no specific GUI available. The user just carefully writes a CPL script in a designated directory. Then, restarting a feature server's process enables the new script.

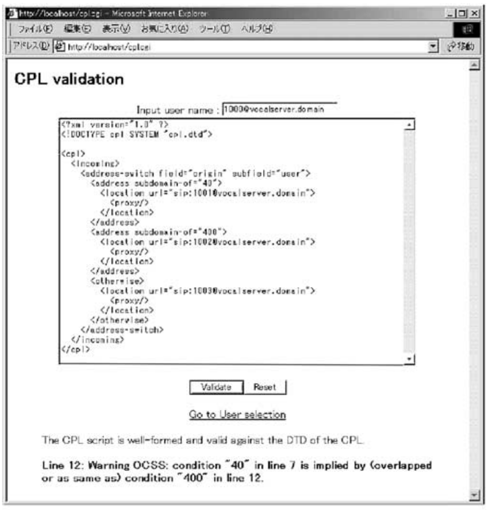
5.2. Setting up the experiment

The goal of the experiment is to validate the applicability of the proposed method to a practical situation. The experiment consists of the following two parts:

Validation of semantic warnings: We check semantic problems in each of the VOCAL ready-made features by means of the semantic warnings. Also, we validate the effect of the semantic warnings by observing how VOCAL behaves for the customized feature scripts containing the warnings.

Detection of feature interactions: We apply the proposed feature interaction detection method to the VOCAL ready-made features.

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


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

CPL validation

Input user name : 1000@vocalserver.domain

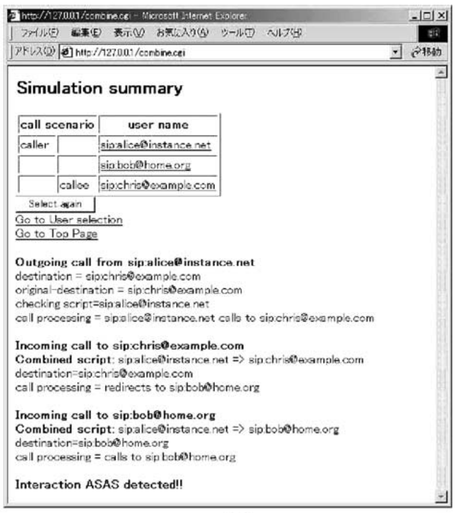
```
<?xml version="1.0" ?>
<!DOCTYPE cpl SYSTEM "cpl.dtd">

<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>
```

Validate
Reset
Go to User selection

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.

(a)


http://127.0.0.1/combine.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://127.0.0.1/combine.cgi
移動

Simulation summary

call scenario		user name
caller		sip:alice@instance.net
		sip:bob@home.org
	callee	sip:chris@example.com

Select again
Go to User selection
Go to Top Page

Outgoing call from sip:alice@instance.net
destination = sip:chris@example.com
original-destination = sip:chris@example.com
checking script=sip:alice@instance.net
call processing = sip:alice@instance.net calls to sip:chris@example.com

Incoming call to sip:chris@example.com
Combined script: sip:alice@instance.net => sip:chris@example.com
destination=sip:chris@example.com
call processing = redirects to sip:bob@home.org

Incoming call to sip:bob@home.org
Combined script: sip:alice@instance.net => sip:bob@home.org
destination=sip:bob@home.org
call processing = calls to sip:bob@home.org

Interaction ASAS detected!!

(b)

Fig. 8. Screenshots of developed tools. (a) CPL checker, (b) FI simulator.

tures is performed by the provisioning GUI with a Web Interface on the clients. For the creation of the customized features, we edited the CPL script directly on the server.

We have developed a set of tools to support the proposed method. Fig. 8(a) shows the *CPL checker*, which detects the semantic warnings for a given CPL script. The CPL checker also performs syntax checking to validate conformance to the XML syntax and to the DTD of CPL. Thus, it can be used for debugging CPL scripts as well. Fig. 8(b) shows the *FI simulator*, which simulates execution of CPL scripts. For a given CPL script and a call scenario, the FI simulator automatically combines related scripts by the Succ algorithm, then reports feature interactions if detected.

5.3. Validation of semantic warnings

5.3.1. Application to ready-made features

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

```
<?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>
```

Fig. 9. Script of CS containing OCSS.

Fig. 9 shows a CPL script of CS generated from the provisioning 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. [6] This case is not necessarily an error, but just a redundancy. However, if the user wants to specify a different rejecting action for the number 400, this redundancy may violate the user's intention.

For the other four ready-made features, no semantic warnings were found. This is implied by the fact that the current version of the GUI supports a very limited class of features programmable in CPL. However, there is the possibility of introducing additional semantic checks in future extensions, as shown in the next subsection.

5.3.2. Validation with customized features

To justify the proposed warnings, we took the CPL scripts in Fig. 3, and let VOCAL execute them as customized features. We executed each script one-by-one and observed how VOCAL behaved. As a result, the problematic effects anticipated in Section 3 were exactly reproduced for all warnings.

(A) Multiple forwarding address (MFAD) in Fig. 3(A)

This script forwards (proxies) an incoming call to three destinations `1001`, `1002` and `7000`, simultaneously. The problem is that `7000` is a number reserved by VOCAL for a voicemail service which *immediately* answers incoming calls. Therefore, the voicemail always replies, and the call never reaches 1001 or 1002. A possible solution to this problem is to recommend to the user to specify the *ordering parameter* `sequential` in `<proxy>` (see [8] for more details).

(B) Unused subactions (USUB) in Fig. 3(B)

Upon deploying a CPL feature, VOCAL compiles the CPL script into a C++ virtual state machine and loads it into memory. Hence, the unused subactions increase memory overhead on the feature server although they are never executed. They cause no harm from the feature's point of view. However, from the viewpoint of the system, they should be optimized to reduce memory usage of the server.

(C) Call rejection in all execution (CRAE) in Fig. 3(C)

By this script, VOCAL rejects all incoming calls. Whether or not the script is intentional, all actions and conditions are just wasting the resource.

(D) Address set after address switch (ASAS) in Fig. 3(D)

When the user originates a call directly to `5000`, the call is rejected by VOCAL. But when the user makes a call to other users, the call is automatically proxied to `5000` which should have been rejected. Thus, we can see that an inconsistent destination problem occurs when ASAS exists in the script.

(E) Overlapping conditions in single switch (OCSS) in Fig. 3(E)

In this script, we suppose that the user intends to forward calls (a) from numbers containing "40" to `1001`, (b) from numbers containing "400" to `1002`, and (c) from any other numbers to `1003`. However, we observed that any call incoming from numbers prefixed by "40" (e.g., `400` or `4001`) were always forwarded to `1001`, and that no call was proxied to `1002`. Thus, we have an unreachable part of the script.

(F) Identical actions in single switch (IASS) in Fig. 3(F)

This script has an `<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 the branches. Thus, the `<address-switch>` is completely meaningless and redundant, and should be removed.

(G) Overlapping conditions in nested switches (OCNS) in Fig. 3(G)

Incoming calls from addresses containing "400" are proxied to `1001`. However, it is not necessary to evaluate the second condition of `subdomain-of="40"`, since string `40` is a substring of `400`. Also, note that calls from addresses containing `40` but not `400` (e.g., 4010) are not proxied to `1001` but to `1002`. Thus, we can see that OCNS adds a very complicated logic structure to the scripts.

[6] We could not figure out what the third condition means in this automatically-generated script. It seems to be an implementation issue of the provisioning system.

(H) Incompatible conditions in nested switches (ICNS) in Fig. 3(H)
In order for VOCAL to proxy a call to 5000, the originating address must be 2000 and 4000 simultaneously. However, this is impossible. Therefore, the proxy operation to 5000 never occurs.

Thus, it was shown that the proposed semantic warnings identify the possibility that VOCAL may execute problematic behaviors. Although some of them are not directly connected to errors, they can often highlight potential sources of faults and interactions. An effective application of the semantic warnings is to implement a *semantic checker* (like our CPL checker) in a provisioning GUI or a CPL server, so that they are reported to the user upon each provisioning of a CPL script.

5.4. Detecting feature interactions among pairwise ready-made features

Next, we evaluate the proposed interaction detection method by applying it to every *pair* of the five ready-made features. We do not especially discuss customized features here, since there are a number of *safe* customized scripts and thus it is difficult to pick up specific ones without loss of generality.

As mentioned in Section 2.1, in the CPL framework each user is supposed to have at most one CPL script at a time. However, the VOCAL implementation does not follow this regulation. Specifically, VOCAL manages each ready-made feature as an independent CPL script. Therefore, when a user activates multiple ready-made features, VOCAL generates multiple CPL scripts for the user without merging them into one script. This implementation issue, specific to VOCAL, may introduce new feature interactions that are out of our scope, since a user is enabled to run multiple scripts in response to a condition (these are called *feature-to-feature interactions* [7]). However, these interactions (resulting in *non-determinism*) are thoroughly discussed in previous research (e.g., [12,16]). Hence, we do not discuss the case where a user activates multiple features.

Accordingly, combinations of the ready-made features considered in the experiment are summarized as follows. Each category of combinations has a *critical call scenario* which activates two features simultaneously.

O–T combinations: Combinations of an originating feature s_1 and a terminating feature s_2, where s_1 and s_2 are activated by different users u_1 and u_2, respectively. The critical call scenario is that u_1 calls u_2.

T–T combinations: Combinations of a terminating feature s_1 and a terminating feature s_2, where s_1 and s_2 are activated by different users u_1 and u_2, respectively. The critical call scenario is that a third party u_0 calls u_1 (or u_2).

First, we confirm that each CPL script of the ready-made features is safe by using the CPL checker. Then, for each feature combination, we execute a critical call scenario on the test-bed, and detect feature interactions with the assistance of the FI simulator. Moreover, for each execution of a scenario, we perform careful observation and interpretation manually, to find other types of interactions not covered by our method.

Table 2 shows the result. In the table, **FI** (or **FI-free**) represents the fact that a feature interaction

Table 2
Result of interaction detection for VOCAL ready-made features

	T–T combinations			O–T combinations	
	CFA	CFB	CS	CB	CIB
CFA	**FI** Loop	**FI** Loop	FI-free SW-free	**FI** ASAS	FI-free SW-free
CFB		**FI** Loop	FI-free SW-free	**FI** ASAS	FI-free SW-free
CS			FI-free SW-free	FI-free SW-free	**FI** SW-free

was observed (or not) in the combination. Each combination with **FI** may be associated with a name of a warning or a (forwarding) loop found in the combined CPL script. **SW-free** means that no semantic warning was found. Using the FI simulator and manual observation, we have identified a total of six interactions, of which five were detected by the proposed method.

FIs between CB and CFA, CB and CFB

User `1000` activates CB to block outgoing calls to user `1900`. User `1001` activates CFA to forward all incoming calls to `1900`. When `1000` calls `1900`, the call is immediately rejected by `1900` and `1000` receives a rejection message (as required). Strangely however, when `1000` makes a call to `1001`, `1000` receives no response though no connection is established. Then, `1000` keeps redialing `1001` until the call is terminated by timeout.

These interactions were detected by the semantic warning ASAS. Fig. 10 shows the combined script obtained from the two scripts of CB and CFA. The script contains ASAS since the address of `1900` evaluated in `<address>` is set in the `<otherwise>` block. This strange behavior of VOCAL would not have been observed if we had changed `1900` to any other address. From this fact, it seems that actions for rejection and forwarding to the same address `1900` were conflicting in the VOCAL implementation, which unexpectedly put VOCAL to silence. The same problem was observed for CB and CFB when the called party was busy.

```
<?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>
          <remove-location>
            <location clear="yes" url="sip:1900">
              <redirect></redirect>
            </location>
          </remove-location>
        </success>
        <notfound>
          <sub ref="rejectcall"></sub>
        </notfound>
      </lookup>
    </otherwise>
  </address-switch>
</outgoing>
</cpl>
```

Fig. 10. Combined script of CB and CFA.

FIs between CFA and CFA, CFA and CFB, CFB and CFB

User `1000` activates CFA to forward all incoming calls to `1001`. User `1001` also activates CFA and sets the forwarding address to `1000`. When a third user `2000` calls `1000`, `2000` receives a rejection message from `1000`, although `2000` never knows what the reason for the rejection is.

These interactions were detected by the forwarding loop. `1000` forwards the incoming call to `1001`, then `1001` forwards the call back to `1000`, which forms a forwarding loop. The VOCAL implementation seems to reject further forwardings when a forwarding loop is detected in the underlying protocol. However, the problem is that `2000` cannot recognize that the rejection is due to the forwarding loop. A similar forwarding loop also occurs for combinations CFA and CFB and CFB and CFB, when the callee is busy.

FI between CIB and CS

User `1000` activates CIB to block his/her caller-ID (name and address). User `1001` activates CS to screen incoming calls from `1000`. When `1000` makes a call to `1001`, `1001` accepts the call, although any call from `1000` should have been rejected by CS. The problem is that CS activated by `1001` was not able to identify the caller's address `1000`, since CIB made `1000` anonymous. This interaction could not be covered by the proposed semantic warnings, which is justified by the following reason. VOCAL uses a special parameter value `"complete_caller_idblock"` in `<remove-location>` to handle CIB. This parameter value is a *proprietary keyword* of VOCAL and is beyond the CPL specification. Therefore, it is currently out of our scope.

5.5. Discussion

The interactions detected in the experiment were due to ASAS and forwarding loops only. This is because the VOCAL ready-made features were too simple to cause more 'interesting' interactions, and is not because of limitations of the potential of the semantic warnings. We believe that for more sophisticated features, the other types of warnings would detect additional potential interactions.

For instance, let us consider the following feature interaction: Alice rejects any outgoing calls issued in the morning (using `<time-switch>`). Similarly, Bob rejects all incoming calls arriving in the afternoon. Then, Alice can never reach Bob. This problematic situation can be characterized by CRAE and surely detected by the proposed method.

Another point is that the defined semantic warnings range from rather minor to rather more significant. It is expected that more essential interactions tend to concentrate in certain types of semantic warnings. Furthermore, semantic problems that cannot be covered by the proposed semantic warnings may still exist. We consider that clarifying the coverage and optimization of the proposed warnings requires more experiments with more sophisticated features, which is left as our challenging future work.

6. Implementing run-time FI detection

6.1. FI detection server and FI detector

To implement the proposed method on practical network architectures, we need at least the following three components: a *semantic checker*, a *script repository* and a *script combination engine*. The semantic checker is a module to detect semantic warnings for a given CPL script, which is exactly like our CPL checker. A user can create or modify a script dynamically at any time. So, the script repository is needed to keep *up-to-date* CPL scripts of all users. The script combination engine implements the proposed algorithm $\mathrm{Succ}(s,c)$. Specifically, for a given pair of script s and call scenario c, the engine simulates s based on c. Then, it determines the successive scripts to be executed, and picks them up from the script repository. It also checks the existence of the forwarding loop. Finally, it derives a set of scripts combined by $\triangleright_c$, according to $\mathrm{Succ}(s,c)$. The combination engine is also implemented in our FI simulator.

Using these components, we construct an *FI detection server*, which works as follows. First, the server receives an *FI detection request* from a client program (*FI detector*, explained later), with the originating address of a user u and a call scenario c. Then, the server takes a script s of u from the script repository. Next, the script combination engine computes $\mathrm{Succ}(s,c)$ by combining scripts in the repository. For each of the combined scripts, the semantic checker detects semantic warnings. If any warning is detected, the server returns a report of the interaction to the client with an appropriate message.

As a client module, we add an *FI detector* to a VoIP server. When a call setup from a user agent u arrives at the VoIP server, the FI detector *intercepts* the call setup, and sends an FI detection request to the FI detection server, with the originating address u and a call scenario c derived from the call setup request. If the FI detector receives a report of FI from the FI detection server, it sends u the report to ask whether it should continue the call setup or not.

Note that the interaction detection can be performed locally in the FI detection server, with an assumption that all the up-to-date CPL scripts of users are available in the script repository. Hence, the key issue is how to *synchronize* the scripts in the repository with the latest ones. In general, this synchronization is a difficult problem. So, we need to identify applicable cases of the proposed implementation.

6.2. Implementation for script-to-script interactions

It is not difficult to detect feature interactions between CPL scripts in a *single* VoIP server. According to [7], these types of interactions are called *script-to-script interactions*. In this case, all users involved in a call use the same VoIP server, and all the up-to-date scripts are stored in the

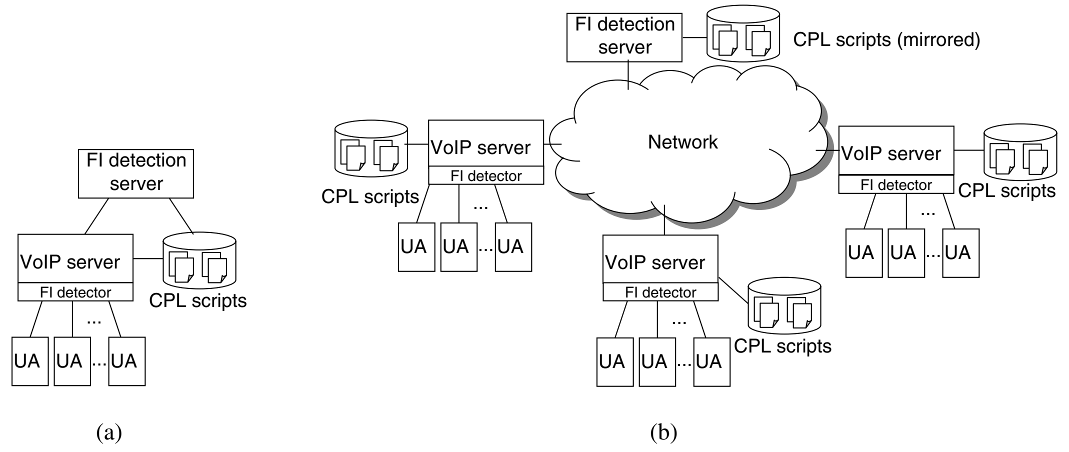


Fig. 11. Implementation architectures of the proposed method. (a) Single-server system, (b) multiple-server system.

VoIP server. Therefore, if an FI detection server can use the storage directly as the script repository, the synchronization of scripts is not especially needed.

Fig. 11(a) shows an implementation of the detection system. In the figure, all user agents (represented by UA) use a VoIP server, and the storage of the VoIP server is shared by the FI detection server. The administrator first deploys an FI detection server in the same network of the VoIP server, then connects it with the VoIP server so that the detection server can access the CPL scripts. The administrator also installs the FI detector in the VoIP server. On each call setup, the FI detector issues an FI detection request to the FI detection server. Then, the detection server performs the interaction detection by using the latest CPL scripts in storage. Thus, script-to-script interactions can be detected at run-time. Our experiment in Section 5.4 was conducted in a similar environment.

6.3. Implementation for server-to-server interactions

Feature interactions can occur between CPL scripts in different VoIP servers. These are called *server-to-server interactions* [7]. Fig. 11(b) shows a network architecture where multiple VoIP servers and an FI detection server are connected by a network. In order for the FI detection server to detect the server-to-server interactions, we need to *mirror* all the latest CPL scripts in the VoIP servers to the script repository of the FI detection server.

Basically, mirroring can be achieved by a VoIP server in such a way that every time a user creates or modifies a CPL script, the VoIP server uploads the script to the FI detection server. However, a CPL script may contain private information of a user such as forwarding addresses, screening conditions, and secret messages. Hence, the user will not expose the CPL script unless both the network and the FI detection server are completely *trusted.* Also, exposing CPL scripts to an untrusted network could leak security information of the VoIP server.

In this sense, we consider that detection of the server-to-server interactions by the proposed method is feasible only for trusted VoIP systems, where the VoIP servers and the network are securely administered by a trusted network provider (or a group of allied providers). In this case, the provider deploys an FI detection server, and installs the FI detector in all the VoIP servers. The provider also customizes the CPL provisioning system, so that the latest CPL scripts are uploaded to the detection server. Then, the detection of server-to-server interactions can be performed in the same way as that of script-to-script interactions.

If a trusted server and network are unavailable, the proposed method cannot be directly applied. A technique of *mobile cryptography* [14], which allows an untrusted remote server to execute an operation for encrypted data, could give a solution. The proposed method is also difficult to implement on VoIP systems in an administratively heterogeneous network. This is because some administrators may refuse to install the FI detector in the VoIP server, or to share the FI detection server with other providers. The problem caused by heterogeneous networks is essentially unsolvable as described in [7], and is also the limitation of the proposed method.

7. Conclusion

In this paper, we have presented a method to detect feature interactions for the CPL programmable service environment of Internet telephony. We first proposed eight types of semantic warnings to identify semantic problems in each individual CPL script. Then, by extending the warnings to multiple scripts, we proposed a definition of feature interaction, and an interaction detection method.

The proposed method has been evaluated with a practical VoIP system, VOCAL. The semantic warnings revealed a semantically redundant case in a ready-made feature. It was also shown that CPL scripts with semantic warnings can lead VOCAL to certain problematic situations. Although only a small set of features was tested, the proposed method detected five of six feature interactions among ready-made features of VOCAL. We also discussed the applicability and limitations of the proposed method from the implementation viewpoint. As a result, the proposed method is shown to be feasible for the detection of script-to-script interactions, or server-to-server interactions with a trusted VoIP system.

Finally, we summarize our future work. We are currently implementing an FI detection system, based on the design issues described in Section 6. Performance evaluation of the system is an important topic. As seen in the interaction between CIB and CS in Section 5.4, the proposed method does not cover interactions specific to a server implementation. To cover such server-specific interactions or newly discovered warnings, we are developing a framework for administrators to customize semantic warnings themselves. Although this paper only discussed the detection of interactions, it is also important to consider how to *resolve* the detected interactions. Development of an effective resolution scheme is a challenging goal.

Acknowledgements

This work is partly supported by Grant-in-Aid for Encouragement of Young Scientists (No. 15700058), from Japan Society for the Promotion of Science.

References

[1] L. Blair, J. Pang, Feature interactions—life beyond traditional telephony, Proceedings of Sixth International Workshop on Feature Interactions in Telecommunication Networks and Distributed System (FIW'00), May 2000, pp. 83–93.

[2] M. Calder, E. Magill, M. Kolberg, S. Reiff-Marganiec, Feature interaction: a critical review and considered forecast, Computer Networks 41 (1) (2003) 115–141.

[3] E.J. Cameron, N.D. Griffeth, Y.-J. Lin, M.E. Nilson, W.K. Schnure, A feature interaction benchmark for IN and beyond, in: Feature Interaction in Telecommunications System (FIW'94), May 1994, pp. 1–23.

[4] R.J. Hall, Feature interactions in electronic mail, Proceedings of Sixth International Workshop on Feature Interactions in Telecommunication Networks and Distributed Systems (FIW'00), July 2000, pp. 67–82.

[5] S. Homayoon, H. Singh, Methods of addressing the interactions of intelligent network services with embedded switch services, IEEE Communications 26 (12) (1998) 42–70.

[6] D. Keck, P. Kuehn, The feature interaction problem in telecommunications systems: a survey, IEEE Transactions on Software Engineering 24 (10) (1998) 779–796.

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

[8] J. Lennox, H. Schulzrinne, CPL: a language for user control of Internet telephony service, Internet Engineering Task Force, January 2002, Available from <http://www.ietf.org/internet-drafts/draft-ietf-iptel-cpl-06.txt>.

[9] J. Lennox, H. Schulzrinne, Feature interaction in Internet telephony, Proceedings of Sixth International Workshop

on Feature Interactions in Telecommunication Networks and Distributed Systems (FIW'00), May 2000, pp. 38–50.

[10] A. Metzger, C. Webel, Feature interaction detection in building control systems by means of a formal product model, Proceedings of Seventh International Workshop on Feature Interactions in Telecommunication Networks and Distributed Systems (FIW'03), July 2003, pp. 105–122.

[11] M. Nakamura, P. Leelaprute, K. Matsumoto, T. Kikuno, Detecting script-to-script interactions in call processing language, Proceedings of Seventh International Workshop on Feature Interactions in Telecommunication Networks and Distributed Systems (FIW'03), July 2003, pp. 215–230.

[12] M. Nakamura, Y. Kakuda, T. Kikuno, Analyzing non-determinism in telecommunication services using P-invariant of Petri net model, Proceedings of IEEE International Conference on Computer Communication (INFOCOM'97), April 1997, pp. 1253–1260.

[13] J. Rosenberg, H. Schulzrinne, G. Camarillo, E. Schooler, A. Johnston, J. Peterson, R. Sparks, M. Handley, E. Schooler, SIP:session initiation protocol, Request for Comments 3261, Internet Engineering Task Force, June 2002, Available from <http://www.ietf.org/rfc/rfc3261.txt?number=3261>.

[14] T. Sanders, C.F. Tscudin, Towards mobile cryptography, Proceedings of IEEE Symposium on Security and Privacy, May 1998, pp. 215–224.

[15] H. Schulzrinne, J. Rosenberg, Internet telephony: architecture and protocols—an IETF perspective, Computer Networks and ISDN Systems 31 (2) (1999) 237–255.

[16] B. Stepien, L. Logrippo, Feature interaction detection by using backward reasoning with LOTOS, Proceedings of International Conference of Protocol Specification, Testing and Verification (PSTV'95), 1995, pp. 71–86.

[17] S. Tsang, E.H. Magill, Learning to detect and avoid run-time feature interactions in the intelligent network, IEEE Transactions on Software Engineering 24 (10) (1998).

[18] M. Weiss, Feature interactions in web services, Proceedings of Seventh International Workshop on Feature Interactions in Telecommunication Networks and Distributed Systems (FIW'03), July 2003, pp. 149–156.

[19] Feature Interaction in Telecommunications, vol. I–VII, IOS Press, Amsterdam, 1992–2002.

[20] ITU-T, Q.1200 Series: Intelligent Network Capability Set 1, ITU-T, 1990.

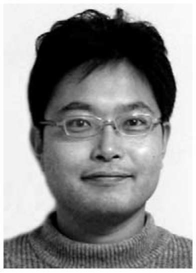
[21] ITU-T, H.323, Packet-Based Multimedia Communications Systems, February 1998.

[22] Bellcore, Advanced Intelligent Network (AIN) Release 1, Switching Systems Generic Requirements, Bellcore Technical Advisory TA-NWT-001123, 1991.

[23] JAIN initiative, The JAIN^{TM} APIs: Integrated Network APIs for the Java Platform, Available from <http://java.sun.com/products/jain/>.

[24] NetCentrex^{TM}, Application Execution and Service Creation Environment, Available from <http://www.netcentrex.net/ products/application_server.shtml>.

[25] VOCAL: The Vovida Open Communication Application Library, Available from <http://www.vovida.org/>.



Masahide Nakamura received the B.E., M.E., and Ph.D. degrees in Information and Computer Sciences from Osaka University, Japan, in 1994, 1996, 1999, respectively. From 1999 to 2000, he has been a post-doctoral fellow in SITE at University of Ottawa, Canada. He joined Cybermedia Center at Osaka University from 2000 to 2002. He is currently an assistant professor in the Graduate School of Information Science at Nara Institute of Science and Technology, Japan. His research interests include the feature interaction problem in network services, software validation and verification, and software metrics and security. He is a member of the IEEE and a member of the IEICE.

Pattara Leelaprute received the M.E. degree in Computer Engineering from Osaka University in 2003. He is currently studying towards the Ph.D. degree in the Graduate School of Engineering Science at the same university. His research interests include telecommunication services and Feature Interaction. He is a member of the IEEE and a member of the IEICE.

Ken-ichi Matsumoto received the B.E., M.E. and Ph.D. degrees in Information and Computer Sciences from Osaka University, Japan, in 1985, 1987, 1990, respectively. He is currently a professor in the Graduate School of Information Science at Nara Institute of Science and Technology, Japan. His research interests include software measurement and software user process. He is a senior member of the IEEE, and a member of the ACM, IEICE and IPSJ.

Tohru Kikuno received the B.E., M.Sc., and Ph.D. degrees in Electrical Engineering from Osaka University, Japan, in 1970, 1972, and 1975, respectively. He joined Hiroshima University from 1975 to 1987. Since 1990, he has been a Professor of the Department of Information and Computer Sciences at Osaka University. His research interests include the analysis and design of fault-tolerant systems, the quantitative evaluation of software development processes, and the design of procedures for testing communication protocols. He is a senior member of IEEE, a member of ACM, IEICE (the Institute of Electronics, Information and Communication Engineers), and a fellow of IPSJ (Information Processing Society of Japan). He received the Paper Award from IEICE in 1993. He served as a program co-chair of the 1st International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98) and the 5th International Conference on Real-Time Computing Systems and Applications (RTCSA'98). He also served as a symposium chair of the 21st Symposium on Reliable Distributed Systems (SRDS2002).