ARTICLE IN PRESS

Available online at www.sciencedirect.com

SCIENCE DIRECT®

Journal of Network and Computer Applications ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Journal of NETWORK and COMPUTER APPLICATIONS

www.elsevier.com/locate/jnca

Detecting feature interactions in CPL

Yiqun Xu[a], Luigi Logrippo[a,b,*], Jacques Sincennes[a]

[a]*School of Information Technology and Engineering (SITE), University of Ottawa, Ottawa, Ont., Canada K1N 6N5*
[b]*Department of Computer Science and Engineering, University of Quebec in the Outaouais, Gatineau, Que., Canada J8X 3X7*

Received 1 September 2004; received in revised form 3 October 2005; accepted 4 October 2005

Abstract

An approach for detecting feature interactions in IETF's Call Processing Language (CPL) scripts is presented. The approach is logic based in the sense that it uses a logic representation of CPL scripts, of requirements and of detection rules and, in several cases, specific detection rules are shown to be derived from requirements by logical deduction. The Simple Formal Specification Language (SFSL) is introduced to express the intentions of CPL scripts in logic format. A method for translating CPL into SFSL is presented. The rules address both interactions within a single script, and interactions between two scripts. An automatic feature interaction detection tool applying these rules was implemented in SWI-Prolog. The general method is not specific to CPL and could be used in other feature interaction research.

Keywords: Internet telephony; VoIP; Features; Services; Feature interaction; CPL; Call Processing Language

1. Introduction

Internet telephony (International Engineering Consortium, 2005) is the subject of intense research and development by telecom operators, telecom manufacturers and consumer groups. It promises sophisticated telephony services over the Internet with lower prices and higher flexibility (Turner et al., 2004). However, the methods of deploying

*Corresponding author. Department of Computer Science and Engineering, University of Quebec in the Outaouais, Gatineau, Que., Canada J8X 3X7. Tel.: +18195953900x1885; fax: +18197731638.
E-mail addresses: yixu@site.uottawa.ca (Y. Xu), luigi@uqo.ca (L. Logrippo), jack@site.uottawa.ca (J. Sincennes).

1084-8045/$ - see front matter
doi:10.1016/j.jnca.2005.10.001

services in Internet telephony are far from mature. On one hand, the Internet platform and new signaling systems enable more opportunities for new services and new features; on the other hand, the Internet telephony network is more distributed and less controlled. One of the problems caused by immaturely deployed services is feature interaction, which we describe as the situation where the requirements of features or services are inconsistent. Such interactions were possible in traditional telephony (and they have been extensively studied, see Amyot et al., 2005; Calder et al., 2003a; Reiff-Marganiec and Ryan, 2005; Zave, 2004, http://www.research.att.com/~pamela/faq.html) but the risk increases significantly in Internet telephony, where users may have the power to program their own features (Calder et al., 2003a; Lennox and Schulzrinne, 2000; Reiff-Marganiec and Turner, 2004; Turner et al., 2004).

A typical example of single-user feature interaction in Internet telephony is the case where a user programs a service to have all incoming calls forwarded to a colleague at her regular meeting times. She also sets another feature to forward calls from her spouse to her personal voice mail for the same meeting time. If the first feature is set with higher priority than the second one, the spouse's call will be forwarded to her colleague instead of her voicemail during the meeting period (see Section 4.1).

An example of feature interaction involving more than one user is the case where a user tries to reach another user immediately, therefore he uses call forking to broadcast three simultaneous calls to the callee's cell phone, office phone and home phone, knowing that all other calls will be aborted once one call is answered. However, the callee programs his cell phone to transfer all incoming calls to his voice mail automatically. It is likely that the voicemail will take the incoming call before the second user answers personally. Therefore, the first user's purpose is violated and feature interaction occurs (see Section 5.3).

Several approaches are being developed for creating telephony services on Internet telephony. The Call Processing Language (CPL, Lennox and Schulzrinne, 2002), is one of the best known. CPL is XML-based (Zisman, 2000), signalling independent and designed for safety. It is under study as an IETF standard, being at the Request for Comments (RFC) stage. Feature interactions are possible in CPL, and therefore methods should be developed to detect feature interactions in CPL scripts before they are activated. Related work has already been done focusing on either semantic ambiguities (Nakamura et al., 2003) or conflicts inside single CPL scripts (Amyot and Logrippo, 2003). In this paper, we propose a feature interaction detection method based on logic and we present the implementation of an automatic filtering tool to detect potential feature interactions in single and pairs of CPL scripts.

The method proposed here is a generalization, with adaptation to CPL, of ideas presented in Felty (2001), Gorse (2000) and Gorse et al. (to appear). We apply to CPL a general method of feature interaction detection. This consists of translating CPL into a logic representation. Feature interactions are either immediate logic contradictions, or contradictions with respect to basic requirements to which phone systems are expected to conform. In several cases, detection rules are given, which are justified by the fact that the truth of the condition detected by the rule, together with the basic requirements, leads to a contradiction. Because the detection rules are simple to check, the detection process is efficient.

What is an appropriate choice of requirements in the Internet telephony world? This is often touted as a 'free world' where anything will be possible and perhaps tolerated. If this is really true, then there will be no reason to look for feature interactions. If, however,

some situations are still considered to be undesirable (especially according to user expectations), then we must establish requirements and methods that can be used to find their violation.

Further details on many aspects of this work can be found in Xu (2003).

2. Overview of CPL and features in Internet telephony

2.1. The Call Processing Language

CPL can be used to describe and control Internet telephony services. It is designed to be implementable on either network servers or user agent servers (Lennox and Schulzrinne, 2002). In order to limit the potential for disastrous errors in highly distributed systems such as the Internet, CPL was designed with many limitations: for one, loops and recursion are not possible in CPL scripts. The example in Fig. 1 presents a common structure of CPL scripts:

A CPL script can be considered as a decision-tree, where a set of conditions constitutes a branch and a single action constitutes a leaf at the end of each branch as shown in Fig. 2.

In this figure, ellipses represent *actions* (leaves) and rectangles stand for *conditions* (part of branches). "reject" and "proxy" are the only two leaves shown in this tree, whose corresponding branches are "incoming→address-switch" and "outgoing→time-switch", respectively. Condition and action are the two types of lexicalities in CPL: conditions are used to determine the executing path whereas actions indicate what will be executed or the results that the path will lead to. In Fig. 1 "reject" and "proxy" are the only two actions. They will be executed only if their corresponding sets of conditions are true, in which case, we say that the action is executable and that the leaf is *reachable*. In Fig. 1, for action "reject", the corresponding set of conditions (or precondition) is that there is an incoming call and the address attribute of this call matches the criteria set by the address-switch; for

```
<cpl>
  <incoming>
    <address-switch field="origin " subfield="user">
      <address is="anonymous">
        <reject status="reject" reason="I don't accept anonymous calls." />
      </address>
    </address-switch>
  </incoming>
  <outgoing>
    <time-switch tzid="America/New_York"
     tzurl="http://zones.example.com/tz/America/New_York">
      <time dtstart="20000703T090000" duration="PT8H">
        <location url="sip:jones@voicemail.example.com">
          <proxy />
        </location>
      </time>
    </time-switch>
  </outgoing>
</cpl>
```

Fig. 1. An example of CPL script's structure.

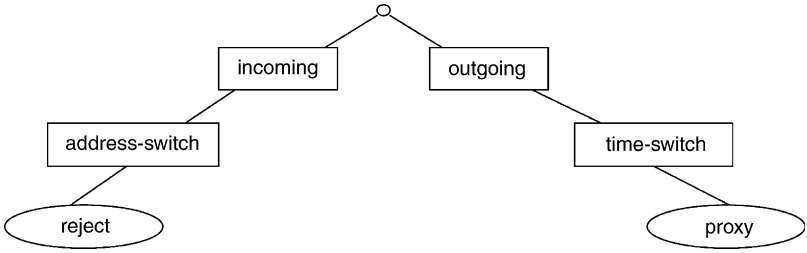

Fig. 2. Decision-tree of the CPL script in Fig. 1.

action "proxy", the set of conditions is that there is an outgoing call and the time attribute of this call matches the criteria set by the time-switch.

Although CPL only has two top-level conditions ("incoming" and "outgoing") and the switch conditions such as address-switch and time-switch can be either "matched" or "not matched", a decision tree does not have to be binary since more than two switch conditions may be present at the same level after incoming or outgoing and the outcome of some actions may have more than two possibilities.

2.2. Policy, intention, features and feature interactions in CPL

The concepts of *policy* and *intention* are key concepts in the study of features and feature interactions. *Policies* have been defined as "high-level statements to support personal, organizational, or system goals" (Reiff-Marganiec and Turner, 2004). An *intention* is one of these goals. A *feature* is a functionality offered by a system and is specified by means of policies, which can be implicit or explicit.

In our view, a CPL script represents the overall policy of a user, which contains two sub-policies: incoming and outgoing. It specifies the user's intentions for incoming and outgoing calls. Intentions will be represented by us as rules of the form *conditions*→*action*. We will take into consideration intentions that are directly derived from the CPL code, and also intentions that can be deduced from it. Note that the word *intention* has been used in a wider sense in feature interaction research, e.g. in Stepien and Logrippo (1995) intentions are not necessarily expressed in policies or features, neither directly nor indirectly, rather they are user or system requirements, more like the contradictions we will encounter later.

Just as one tree may contain several branches, one policy may include one or more intentions. For the example in Fig. 1, the incoming policy expresses one intention that indicates that all the incoming calls will be rejected if their address meets the condition denoted by the address-switch; the outgoing policy expresses one intention as well, which indicates that all the outgoing calls will be transferred during the period of time defined by the time-switch.

The well-known feature Call Forward on Busy (CFB) can be specified in terms of user intentions in this way: if I am busy, then transfer all the incoming calls to a colleague. Feature Outgoing Call Screening (OCS) can be specified as: if a user tries to connect to that number from my phone, then block the call.

For the study of feature interactions, it is useful to consider a type of constraint that has been called *incoherence* in Gorse et al. (to appear). These identify situations that are considered to be undesirable in a system, from the user's point of view or from the system's

point of view. In this work, they will be contradictions between intentions. Basic rules BR1 and BR2, introduced in Section 5.2, are examples.

In the light of these definitions, feature interactions can be defined as situations of inconsistency in the set of logical statements that includes the intentions of coexisting features and the stated contradictions. Inconsistency means that logically, the conjunction of these statements in unsatisfiable (Felty, 2001; Gorse et al., to appear). With these definitions, intention inconsistency (or contradiction) and feature interaction are essentially synonyms.

The topics of policies, intentions and features in the context of forthcoming Internet telephony services are discussed in Dini et al. (2004) and Reiff-Marganiec and Turner (2004).

We will consider feature interactions in single CPL scripts, and feature interactions in pairs of CPL scripts (roughly, this corresponds to the distinction between single user and two-user feature interactions (Cameron et al., 1993)). CPL prevents feature interactions in single scripts by setting feature priorities within CPL scripts. Simply, the script is traversed top-down and the first applicable action is executed, while actions appearing further down in the tree are ignored (except under some circumstances). For instance, a subscriber in a traditional system might unthinkingly request both "Call Forward Always" and "CFB", which leads to uncertainty under the condition of line busy. This problem does not occur in the context of CPL since the CPL script is only able to "trigger one action in response to the condition 'a call arrives while the line is busy'"(Lennox and Schulzrinne, 2000), and this will be the first action encountered. However, the user might have wanted a different behaviour as we have seen in the introduction.

3. Simple Formal Specification Language (SFSL): a logic-based language for abstracting CPL scripts

Our approach to address the feature interaction problem is based on logic analysis, for which a formal, logic-based specification of the intentions represented by CPL scripts is useful. This latter purpose is fulfilled by our SFSL.

3.1. The syntax of SFSL

Briefly, SFSL represents the intentions of CPL scripts in the following format:

$$\text{condition1} \bigwedge \text{condition2} \bigwedge \ldots \rightarrow \text{action}.$$

On the left of symbol → is the set of enabling conditions while on the right is their result. Details about the SFSL syntax can be found in Xu (2003).

3.2. Translating CPL scripts into SFSL

Translating a CPL script into SFSL involves splitting a decision tree into branches as shown in Fig. 3. A CPL script may contain one or more intentions and each intention can be considered as a branch in the CPL structure.

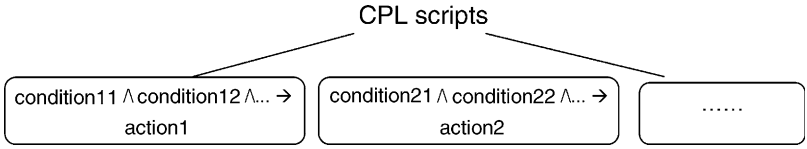

Fig. 3. CPL scripts divided into several independent branches.

From Fig. 3, we see that each intention contains one and only one action, and conditions do not exist independently; on the contrary, they are always attached to actions. Consequently, the first step of translating CPL into SFSL is to recognize actions, which are leaves in a decision tree.

3.2.1. Identifying and translating actions

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

For the signalling action "proxy", there may be a URL location present prior to "proxy" to indicate a new destination of this action; otherwise the initial destination address would be used, namely the callee's number if the proxy instruction is in the outgoing branch, or the caller's number if the proxy instruction is in the incoming branch. We use proxy(x, url-location) to represent the following CPL excerpt, where "x" denotes all the incoming calls which could be from any user and "url-location" denotes the forwarding destination:

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

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

CPL's signalling action "reject" is represented in SFSL by reject(caller, callee). If it is for incoming policy, we use reject(x, owner); if it is for outgoing policy, we use reject(owner, x), where "owner" stands for the subscriber of the current CPL script. "x" is the attempting user whose attributes satisfy the conditions in "address-switch" and "address".

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

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

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

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

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

CPL also has a specific notation "subaction" that is defined for script re-use and modularity. It acts like a sub-function, consisting of conditions and actions also. We assume that all subactions are replaced by their definition before CPL scripts are translated.

3.2.2. Translating associated conditions

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

Address-switches may be followed by the keyword "subfield" and an address. Thus, the following CPL sentences can be translated in the form of: address-switch(user.field.subfield operator address):

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

Here, the operator should be " = " to represent "is" in the address sentence. The "user", which gives the host address of "field" and "subfield", can be represented by a variable in case the user is determined only at running time. An expression such as this is a truth function, which is true if the enclosed expression is true. Examples are given below.

The structure of the other types of switches and labels is similar. We translate them with similar conventions, and the resulting functions are easy to interpret according to their intuitive meanings. Therefore we will not discuss them in detail.

Finally, semicolons are used to separate coexisting intentions.

3.3. Examples of CPL scripts and their translation to SFSL

This section presents examples of features expressed in CPL, together with their translation into SFSL.

3.3.1. Outgoing call screening in CPL and its translation to SFSL

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

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

For translation, we first identify the action, which is "reject" in this case, and then combine it with its enabling conditions. Finally, the CPL script is translated into the following intention:

$$\text{Outgoing}(\text{"sip:Alice@uottawa.ca"}, x) \wedge \text{address-switch}(x.\text{original-destination.user} = \text{"sip:Carl@phone.example.com"}) \rightarrow \text{reject}(\text{"sip:Alice@uottawa.ca"}, x)$$

3.3.2. Call forward always in CPL and its translation to SFSL

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

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

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

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

4. Identification of interactions and inconsistencies in single CPL scripts

4.1. Interactions in single CPL scripts

Single-user feature interactions (Cameron et al., 1993) appear in CPL as inconsistencies within a single CPL script. We will discuss them here to show how they can be handled by using our method. This analysis leads to rules L1, L2 and L3 which are implemented in our tool. In Section 4.3 we will identify several cases requiring further study. We use the letter L to denote 'local' inconsistencies.

As already mentioned, because the order of execution of intentions and features is determined by CPL's mechanism, many interactions are avoided, and in the case of contradiction of two intentions only the one that is encountered first in the decision tree will be executed. This predetermined execution order, however, may prevent intentions with lower-priority from ever being executed. For instance, if Alice@uottawa.ca has two intentions in the order as follows:

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

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

We see that the second intention will never be executed because if its preconditions are satisfied, the first intention's preconditions are also satisfied, and the latter has the priority. There appear to be conflicting intentions, a logical contradiction. It can be fixed by reversing the order of the intentions, and probably the user should be asked whether she wishes to do so. Rule L1 provides an approach to identify this type of interaction.

Rule **L1 (shadowing)**:
Single-user Feature Interaction is present if the following characteristics hold:

(1) User A has at least two intentions.

(2) The precondition of the first intention is implied by that of the second intention (in the order of occurrence in the script).

Formally:

(1) incoming (x, A) $\bigwedge$ conditions1 $\rightarrow$ action1; incoming(x, A) $\bigwedge$ conditions2 $\rightarrow$ action2
or
outgoing(A, x) $\bigwedge$ conditions1 $\rightarrow$ action1; outgoing(A, x) $\bigwedge$ conditions2 $\rightarrow$ action2
(2) conditions2 $\rightarrow$ conditions1

where we assume that action1 and action2 are different CPL actions, and conditions1 and conditions2 are sets of enable-conditions which can be true together.

Specialization is the converse situation where we have a high priority special intention with a low priority general intention. An example of specialization can be obtained by simply interchanging the two intentions in the example above. Specialization may be intentional, however, the user may not be aware of it and so it is a good idea to signal its presence. Our rule L2 deals with specialization and its precise enunciation will be left to the reader. Shadowing and specialization, their detection and their resolution in CPL are further discussed in Amyot et al. (to appear).

4.2. Redundancy

Another type of logical problem that can occur between intentions in a single CPL script is redundancy (Nakamura et al., 2003). It describes the case where

condition1 $\bigwedge$ condition2 $\rightarrow$ action1 and
condition1 $\bigwedge$ $\neg$condition2 $\rightarrow$ action1
are redundant since these two intentions reduce to one
condition1 $\rightarrow$ action1.

Redundancy is not a feature interaction, at least not in the usual sense (Zave, 2004, http://www.research.att.com/~pamela/faq.html) since it results in repetitions rather than conflicts. Definitely, it is not a conflict of intentions. However, redundancy demonstrates disorganized script logic and reduces performance; therefore detecting it is useful for improving the quality of CPL scripts. Rule L3 is used for this purpose.

Rule **L3 (redundancy)**
Intention redundancy is present if the following characteristics hold:

User A has two intentions that have the same action.
A part of the first intention's preconditions is the opposite of a part in the second intention's preconditions; and the rest of these two preconditions are the same.

Formally:

conditions1 $\bigwedge$ condition2 $\rightarrow$ action1
conditions1 $\bigwedge$ $\neg$condition2 $\rightarrow$ action1

4.3. Inconsistency in CPL

Logical inconsistency can exist in the set of conditions in a feature. We can identify two main types of inconsistency: unexecutable actions and redundant conditions. Again, these are not interactions according to common definitions but are cases of disorganized script logic, probably due to improperly specified intentions. Our tool does not detect these conditions at present, and tool support is left for further study. Some kind of user-provided information, as well as theorem-prover support, seems to be necessary to identify cases of implication and contradiction.

4.3.1. Unexecutable actions and corresponding solutions

The case of unexecutable actions describes the situation where an action cannot be executed because its associated conditions (preconditions) cannot be satisfied. These are unfeasible paths in CPL scripts, i.e. contradictions inside CPL intentions. We classify the contradictions in one intention into two main types: direct contradictions and indirect contradictions.

In the case of direct contradiction, some conditions in one intention contradict directly. An example is:

```
Incoming(x, ''sip:Alice@site.uottawa.ca'')
⋀ ¬address-switch(x.origin.user = ''sip:Carl@phone.example.com'')
⋀…
⋀address-switch(x.origin.user = ''sip:Carl@phone.example.com'')
→
reject(x, ''sip:Alice@site.uottawa.ca'').
```

The action ''reject'' will never be executed since no incoming calls could satisfy the two contradictory conditions.

Indirect contradiction describes the case where an action's preconditions conflict with system axioms, ''which describe properties that should be true of any reasonable system implementations'' (Calder et al., 2003a). For example, suppose that an intention is defined as follows:

```
…outcome(proxy(x, ''sip:Carl@site.uottawa.ca''), ''busy'')
⋀ outcome(proxy(x, ''sip:Carl@site.uottawa.ca''), ''noanswer'')
⋀…
→
redirect(x, ''sip:Bob@uottawa.ca'')
```

We can see that the action ''redirect(x, 'sip:Bob@uottawa.ca')'' can never be executed under the reasonable assumption that the conditions ''busy'' and ''noanswer'' cannot be true for a user simultaneously.

4.3.2. Redundant conditions and corresponding solutions

Redundant conditions characterize the case where conditions are repeated in an intention. The only consequence of this is to reduce the performance of a CPL script.

Similar to unexecutable actions, there are two types of redundant conditions: conditions repeated directly or indirectly.

Repetition of direct conditions can be detected easily after CPL scripts are translated into SFSL specifications. For indirect repetition, theoretically, we can enumerate all possible conditions implied by existing ones and then detect redundancy. However, in practice, to consider all implied conditions is time consuming. For instance, x.origin.user = "sip:Carl@site.uottawa.ca" also implies that x.origin.host = "site.uottawa.ca", x.origin.host $\supset$ "uottawa.ca", x.origin.user $\supseteq$ "Carl", x.origin.user $\supseteq$ "ca", x.origin.user $\supseteq$ "uottawa", etc. This type of redundancy requires more study.

5. Identification of feature interactions in pairs of CPL scripts

5.1. General characteristics of feature interactions in pairs of CPL scripts

The characteristics of CPL must be taken into consideration in the study of the feature interaction problem between CPL scripts.

First of all, CPL has no state and relinquishes control once the call has been established, which implies that some traditional features cannot be defined in CPL, such as Call Waiting, 3-Way Call and Conference Call. This means that several types of feature interactions in traditional telephony involving these features cannot exist in CPL.

Second, CPL scripts of different users may be located in the same server if these users belong to the same organization, or may also be in different servers. Even worse, one user may have several addresses from several organizations, such as yahoo.com and hotmail.com, therefore the CPL scripts attached to these e-mail addresses are located in different servers although they belong to the same real user. This distributed deployment of CPL increases the risk of occurrence of feature interactions significantly.

Further, two users between which feature interaction occurs must be involved in one call rather than two separate ones. This does not mean that one of the two users has called the other directly, but the two users must have become involved in one call somehow. Second, there must be a contradiction, in the sense mentioned above.

As explained in Section 2.1, a CPL script includes two distinct policies, outgoing and incoming, dealing with outgoing calls and incoming calls respectively. Considering this context and on the basis of the above analysis, one could envisage three types of interactions between policies, which are:

- interactions between one user's outgoing policy and another user's incoming policy or,
- interactions between one user's incoming policy and another user's incoming policy or,
- interactions between one user's outgoing policy and another user's outgoing policy.

Again, in all these cases, interaction can mean not only that two users' policies contradict mutually but also that they conflict with explicitly stated contradictions.

As shown in Fig. 4, the first case specifies the situation where user B's outgoing policy conflicts with user A's incoming policy; it occurs when user B calls user A directly or user B's call to somebody else is forwarded to user A. The example of OCS vs. Call Forwarding illustrates this case.

The second case demonstrates the situation where there are interactions between two users' incoming policies as shown in Fig. 5. It can be caused by the fact that one user's

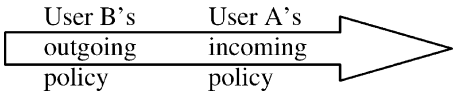

Fig. 4. Interaction between B's outgoing policy and A's incoming.

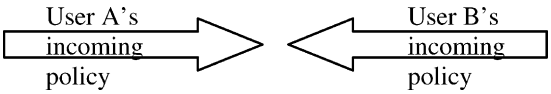

Fig. 5. Interaction between the incoming policies of A and B.

incoming policy results in invoking another user's incoming policy as it happens in the case of forwarding loop.

The third case must be excluded, because it implies that there could be two originators for a call. We believe that this is impossible in current or predictable telecom architectures.

5.2. General rules of feature interaction and contradiction, and their consequences

Our method of feature interaction detection is based on identifying certain contradictions between intentions. The following two basic rules are two contradictions that we postulate.

Basic rule BR1 (incompatibility of reject and proxy): An intention which results in rejecting a call from user A to user B conflicts with an intention which results in establishing a call from user A to user B. This rule can be specified in SFSL as:

conditions1 → reject(A, B)
  contradicts
conditions2 → proxy(A, B)

Notes:

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

We consider the coexistence of these two intentions to be pathological because a reject implies unsuccessful call establishment, while proxy implies continuation of the call establishment.

Basic Rule BR2 (forwarding loops): Endless forwarding loops among users are undesirable. Since in this paper we confine ourselves to the case of two users, this rule can be simplified by stating that the possibility of forwarding loops between two users is a contradiction. In SFSL, this rule can be specified as:

A: conditions1 → proxy(x, B) and B: conditions2 → proxy(x, A) is a contradiction.

Notes:

1. "*A*:" means that the following intention belongs to user *A*, and "*B*:" means that the following intention belongs to user *B*.
2. "*x*" represents the originator of the incoming call, which could be any user.
3. In principle, unlike the case of BR1, the two conditions need not to be true together; however, this fact could lead to timing considerations, so we simplify the discussion by assuming that conditions1 can be enabled by action proxy(*x*, *A*) while proxy(*x*, *B*) can enable conditions2.

Technically, both rules have been described as situations of deadlock, i.e. situations where the system cannot continue without violating constraints that represent reasonable expectations on its behaviour. Interestingly however, the second rule describes also a situation of livelock (unproductive infinite loop), which we have turned into a situation of deadlock by introducing a constraint on the unacceptability of such loops.

It could be possible to use basic rules BR1 and BR2 directly to detect feature interactions; however, this would involve the use of some type of constraint satisfaction engine for which in general efficiency or even termination could not be guaranteed. Rather, we will use more specific rules which can lead to efficient detection. According to what was said in Section 5.1, these rules are confined to two sets of policies, particularly to one user's incoming policy vs. the other user's outgoing policy or one user's incoming policy vs. the other user's incoming policy. The logical relation between these rules and BR1, BR2 will be discussed in Section 5.4.

We will deal separately with rules for incompatibility of reject and proxy, which derive from Basic rule BR1, and rules for forwarding loops, which derive from BR2.

5.2.1. Rules of incompatibility of reject and proxy

Three concrete rules for detecting interactions related to incompatibility of reject and proxy in CPL scripts of two different users have been identified. The letter D in these rules stands for Detection.

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

(1) User *A*'s outgoing policy has a reachable action reject(*A*, *C*) where *C* represents the blocked destination user. In other words, at least one action reject(*A*, *C*) can be executed in user *A*'s outgoing policy.
(2) User *B*'s incoming policy has a reachable action proxy(*x*, *C*) where *x* means the originator could be any user and *C* represents the same user *C* as in (1). In other words, at least one proxy(*x*, *C*) action can be executed in user B's incoming policy.
(3) *A*, *B* are different users.

Formally:

(1) outgoing(*A*, *C*) $\bigwedge$ conditions1 $\rightarrow$ reject(*A*, *C*)

(2) incoming(x, B) $\bigwedge$ conditions2 $\rightarrow$ proxy(x, C)
(3) $A \neq B$

Notes:

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

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

```
outgoing(''sip:Alice@uottawa.ca'', x)
⋀ address-switch(x.original-destination.user = ''sip:Carl@uottawa.ca'')
→
reject(''sip:Alice@uottawa.ca'', x)
```

We see that only Carl@uottawa.ca can make the enabling conditions true and if so the consequence reject(''sip:Alice@uottawa.ca'', x) holds. Hence, the above formula is equivalent to the following one by replacing the variable x with the constant ''sip:Carl@uottawa.ca'':

```
outgoing(''sip:Alice@uottawa.ca'', ''sip:Carl@uottawa.ca'')
⋀ address-switch((''sip:Carl@uottawa.ca'').original-destination.user = ''sip:Carl@uottawa.ca'')
→
reject(''sip:Alice@uottawa.ca'', ''sip:Carl@uottawa.ca'')
```

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

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

Therefore, Rule D1 is satisfied since:

(1) outgoing(''sip:Alice@uottawa.ca'', ''sip:Carl@uottawa.ca'')
⋀ address-switch((''sip:Carl@uottawa.ca).original-destination.user = ''sip:Carl@uottawa.ca'')
→
reject(''sip:Alice@uottawa.ca'', ''sip:Carl@uottawa.ca'')
(2) incoming(*x*, ''sip:Bob@uottawa.ca'') → proxy(*x*, ''sip:Carl@uottawa.ca'')
(3) Alice@uottawa.ca≠Bob@uottawa.ca

This example describes a situation where Alice does not want to talk to Carl whereas Bob forwards all the incoming calls to Carl. When Alice calls Bob or Alice's call is forwarded to Bob, both Alice's outgoing policy and Bob's incoming policy are invoked and both actions reject(''sip:Alice@uottawa.ca'', ''sip:Carl@uottawa.ca'') and proxy (''sip:Alice@uottawa.ca'', ''sip:Carl@uottawa.ca'') should be executed. However, according to our BR1, we know that these two contradicting intentions cannot be satisfied at the same time. Therefore, feature interaction occurs.

The following rules will be given in less complete form for the sake of conciseness.

5.2.1.2. Rule D2. Rule D2 identifies another type of potential interaction which also can occur between CPL scripts of two users because of direct intention contradiction. It addresses the case of interaction between incoming Call Screening and Call Forwarding. Formally:

(1) incoming(C, A) ⋀ conditions1 → reject(C, A)
(2) incoming(x, B) ⋀ conditions2 → proxy(x, A)
(3) $A \neq B$

An example is such a scenario where Alice refuses to take any call originating from Carl whereas Bob forwards all the incoming calls to Alice in case he is busy. When Carl calls Bob or Carl's call is forwarded to Bob, and Bob is busy, both incoming policies of Bob and Alice are invoked and both actions of reject(''sip:Carl@uottawa.ca'', ''sip:Alice@uottawa.ca'') and proxy(''sip:Carl@uottawa.ca'', ''sip:Alice@uottawa.ca'') should be executed. However, according to BR1, these two contradicting intentions cannot be satisfied at the same time. Therefore, we have a feature interaction.

5.2.1.3. Rule D3. Rule D3 identifies another type of potential interaction which also can occur between two CPL scripts of two users because of direct intention contradiction. It also addresses the case of interaction between incoming Call Screening and Speed Dialing.

As an example, suppose that the CPL script of Alice@uottawa.ca rejects all calls from Bob@uottawa.ca; also suppose that Bob's CPL scripts sets up a speed dial number ''12'' for help desk. When Bob needs technical support, he simply dials ''12'' and his call will be automatically forwarded to the appropriate person, which happens to be Alice@uottawa.ca. When Bob dials ''12'', both incoming policies of Bob and Alice can be invoked and both actions of reject(''sip:Bob@uottawa.ca'', ''sip:Alice@uottawa.ca'') and

proxy(“sip:Bob@uottawa.ca”, “sip:Alice@uottawa.ca”) can be executed. However, according to BR1, these two contradicting intentions cannot be satisfied at the same time. Therefore, feature interaction occurs.

5.2.2. Forwarding loops

We have identified one rule leading to the detection of loops of call forwards, which is derived from BR2.

5.2.2.1. Rule F1. Rule F1 identifies potential interactions which can occur between two users' incoming policies. It describes a situation of endless loop when A's action enables B's conditions and also B's action enables A's conditions.

Formally:

(1) $\text{incoming}(x, A) \bigwedge \text{conditions1} \rightarrow \text{proxy}(x, B)$
(2) $\text{incoming}(x, B) \bigwedge \text{conditions2} \rightarrow \text{proxy}(x, A)$
(3) $A \neq B$

As an example, suppose that the CPL script of Alice@uottawa.ca forwards all the incoming calls to Bob, but the CPL script of Bob@uottawa.ca forwards all the incoming calls to Alice. Assume that Alice is called by another user who could possibly be Bob: a forwarding loop is generated. On the basis of BR2, we know that these two intentions cannot be satisfied at the same time, because they contradict system constraints. Therefore, feature interaction occurs.

Some telecom systems deal with forwarding loops by limiting the lifetime of a message, or they check whether a message passes twice through the same node. However, time and system resources are consumed before the system notices the loop and ends it. Our method provides a way to check for such conditions before this happens.

5.3. Call forking interactions

We should mention a different type of contradiction, which is not related to the Basic Rules given in Section 5.1. It addresses the case of interaction between Call Forward (CF) and Call Forking Outgoing (CFO). CFO generates several simultaneous calls looking for a subscriber, and will abort all other calls as soon as one succeeds. We can assume that it is used when a subscriber wants to find another subscriber as quickly as possible. However, this purpose can be frustrated if one of the phones to which the call is forked answers immediately and forwards the call, possibly to an answering machine. The precise formulation of this rule is left to the reader.

5.4. Proofs of feature interaction detection rules

As mentioned, detection rules D1, D2, D3 and F1 are logically related to basic rules BR1 and BR2, in the sense that for each detection rule it is possible to show that the conjunction of the detection rule and one of the basic rules is not logically satisfiable. Thus the presence of the condition described by one of the detection rules implies violation of at least one basic rule. This is shown below for one rule. Other proofs are found in Xu (2003).

For Rule D1:

(1) outgoing(A, C) $\bigwedge$ conditions1 $\rightarrow$ reject(A, C)
(2) incoming(x, B) $\bigwedge$ conditions2 $\rightarrow$ proxy(x, C)

As discussed in 5.3.1, conditions1 and conditions2 may be empty. In order to simplify the proof, we assume that they are absent; and the unbound variable x in formula (2) represents one arbitrary user, so that Rule D1 can be specified in predicate logic as follows:

(1) outgoing(A, C) $\rightarrow$ reject(A, C)
(2) $\forall x$(incoming(x, B) $\rightarrow$ proxy(x, C))

Since we are assuming that the two features above (formula (1) and formula (2)) are enabled simultaneously, the conjunction of their enabling conditions (outgoing(A, C) $\bigwedge$ incoming(A, B)) is added to the set of assumptions. We combine this with the system constraint $\neg$(proxy(A, C) $\bigwedge$ reject(A, C)) (from BR1). According to Calder et al. (2003a) and Felty (2001), our task is to prove that there is no model where the conjunction of these five formulas is satisfiable, i.e. that $\perp$ (false) is generated. This is shown using the deduction system described in Huth and Ryan (2000):

Implication to prove:
outgoing(A, C) $\rightarrow$ reject(A, C)
$\bigwedge$ $\forall x$(incoming(x, B) $\rightarrow$ proxy(x, C))
$\bigwedge$ outgoing(A, C) $\bigwedge$ incoming(A, B)
$\bigwedge$ $\neg$(proxy(A, C) $\bigwedge$ reject(A, C))
$\Rightarrow \perp$

Proof:

1	outgoing(A, C) $\rightarrow$ reject(A, C)	premise
2	$\forall x$(incoming(x, B) $\rightarrow$ proxy(x, C))	premise
3	incoming(A, B) $\rightarrow$ proxy(A, C)	$\forall x$ e 2
4	$\neg$(proxy(A, C) $\bigwedge$ reject(A, C))	premise
5	outgoing(A, C)	premise
6	incoming(A, B)	premise
7	reject(A, C)	$\rightarrow$e 1, 5
8	proxy(A, C)	$\rightarrow$e 3, 6
9	proxy(A, C) $\bigwedge$ reject(A, C)	$\bigwedge$i 7, 8
10	$\perp$	$\neg$e 4, 9

6. Implementation and applicability

6.1. Implementation

On the basis of the rules presented so far, we have developed a tool for automatic feature interaction detection using SWI-Prolog. Given CPL scripts of different users, the tool translates CPL scripts into SFSL specifications first, and then applies the detection rules presented above to these SFSL specifications. At the end, corresponding reports are generated.

Each report is composed of two parts: detailed explanation of the feature interaction and the two inconsistent features. The detailed explanation contains the name of the rules that were applied, and a brief description of the feature interaction type. Here is an example:

```
%%-------------------------------------------------------------------
% ****** Interaction detected by Rule D3 -> % The first user rejects calls from the second user
% while the second user will forward outgoing calls to the first user
% + The first user's policy
implies(conj(incoming(_G248, sip:bob@uottawa.ca), [address(is(sip:alice@uottawa.ca))]), [reject(status(reject)), reject(reason(I do not accept Alice's call.))])
% + The second user's policy
implies(conj(outgoing(sip:alice@uottawa.ca, _G248), [address(subdomain-of(1700))]), [proxy([location(sip:bob@uottawa.ca)], ordering(parallel))])
%****** Interaction detected by Rule D1 ->
% The first user is forbidden to call somebody
% while the second user will forward calls to the forbidden user
% + The first user's policy
implies(conj(outgoing(sip:bob@uottawa.ca, _G249), [address(is(sip:carl@pager.
ottawahospital.com))]), [reject(status(reject)), reject(reason(I am not allowed to call Carl))])
% + The second user's policy
implies(conj(incoming(_G249, sip:alice@uottawa.ca), []), [proxy([location(sip:carl@pager.ottawahospital.com)], ordering(parallel))])
%-------------------------------------------------------------------
```

6.2. Applicability

Unlike PSTN, Internet telephony is not based on hierarchical network architecture, and thus it has no centralized services centre. As discussed in Xu (2003), users' CPL scripts may be located in signalling servers. On the one hand, from the point view of network components, a signalling server acts like a PBX in PSTN (there is no public switch in Internet telephony). On the other hand, from the point view of feature management, signalling servers are "most similar in functionality to service control or switching points in the circuit-switched network" (SCP and SSP) (Lennox and Schulzrinne, 2000). Either way, CPL scripts residing in different signalling servers have no knowledge of each other.

Hence, feature interaction detection could be performed in two stages: offline and online (Reiff-Marganiec and Turner, 2004). Offline detection means performing feature interaction detection at the time when features are uploaded to the signalling server but not activated yet. It can be implemented to detect conflicts between features located in the same signalling server, typically a PBX (these features could belong to the same user or to different ones). In practice, our detection rules could be run after a new feature is uploaded but before it is activated. This makes it possible to recognize both local feature interactions in one user's CPL scripts and pair-wise ones inside one domain.

For potential interactions between two features that are located in different signalling servers, it is neither possible nor necessary to scan every pair of CPL scripts beforehand, in consideration of the very large number of domains and URL on Internet. We propose online detection, at the time of call setup. However, since this requires accessing CPL scripts in different servers, a new concern arises: who should perform this detection, the caller or the callee? Few users will want their calling policies open to the public. Thus, in order to perform a "safe" detection, a trusted third party is needed. One can think that when a call is initiated, the pre-authorized third party will access the CPL scripts of the caller and callee, and then apply FI detection rules to check potential feature interactions. In the case where a call is forwarded from the original destination to a second one, the detection process will be executed twice, first considering the two scripts of the caller and original callee, then considering those of the caller and the second callee. This complex procedure could be justified in cases where the greatest dependability is required.

A more complete view of this problematic, not limited to CPL, but essentially in agreement with our view, has been presented in Blair and Turner (2005).

7. Related work

This research topic has been addressed by other work that was carried out independently and with different methods, roughly in the same timeframe as our own research.

7.1. Detecting script-to-script interactions in CPL

The motivation of M. Nakamura, P. Leelaprute, K. Matsumoto and T. Kikuno's work is very much the same as ours. Their approach addresses possible semantic warnings in individual CPL scripts, and then extends the analysis method to pairs of scripts based on "defining feature interactions as the semantic warnings over multiple CPL scripts" (Nakamura et al., 2003; Nakamura et al., 2004).

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

- *Multiple Forwarding Addresses (MFAD)*: More than one <location> tag is set before an action <proxy> or <redirect>. We did not take this case into consideration since it is more a compilation-time warning than a logical inconsistency.
- *Unused SUBactions (USUB)*: A subaction is defined but not used. Again, this is more a compilation-time warning than a logical inconsistency.

- *Call Rejection in All Paths (CRAP)*: All execution paths terminate at <reject>. This is not necessarily a semantic error since a user may want to block all calls in some cases, in addition, it is not a logical inconsistency.
- *Address Set after Address Switch (ASAS)*: As mentioned above, this scenario describes a conflict between two local intentions: the first intention blocks calls to a particular destination but the second one forwards a call to the same destination. If both intentions are located in the same incoming policy, it cannot be said that there is a conflict: we could block all incoming calls from user A while forwarding other calls to A. In fact, this conflict only occurs in a very special case where outgoing calls to A are blocked while other outgoing calls are forwarded to A. Our current local inconsistency detection rules do not cover this special case; however, it will be well handled if our BR1 is extended to the case of single scripts.
- *Overlapped Conditions in Single Switch (OCSS)*: This describes the scenario which is detected by our rule L1 (see Section 4.1).
- *Identical Actions in Single Switch (IASS)*: it describes the situation of feature redundancy which can be detected by our rule L3 (see Section 4.2).
- *Overlapped Conditions in Nested Switches (OCNS)*: This describes the situation of redundant conditions that is discussed in Section 4.2; however, as our example shows, indirect condition redundancy also can occur between two different switches, which does not fall under the definition of OCNS.
- *Incompatible Conditions in Nested Switches (ICNS)*: it describes the same situation of unexecutable actions that is discussed in Section 4.3.1.

The above comparison shows the overlap of the two approaches in the area of single CPL scripts. Pair-wise feature interactions detected by our rules D1, D2, D3 and F1 are also covered by ASAS and MFAD after the combination of two scripts. However, the interaction between Outgoing Call Forking and Call Forwarding which is detected by our rule D4 is not covered in their work.

A more detailed comparison will be found in Xu (2003).

7.2. The work of Amyot et al.

Amyot et al. (to appear) propose an approach to solve interactive conflicts for personalized services in Internet telephony, which is part of personalized services management architecture. This architecture includes creation of policies for personalized communication services, validation of services, and conflict handling. CPL was chosen as the possible service creation tool, and a translator from CPL to the input language of the tool FIAT (Gorse, 2000; Gorse et al., to appear) was developed so that potential conflicts can be detected by applying FIAT. However, this work concentrates on inconsistencies local to a single CPL script, while our work concentrates on inconsistencies between scripts.

7.3. The work of Calder et al.

Calder et al. (2003b) propose two hybrid approaches that combine off-line and on-line techniques to detect and resolve feature interactions. One of these two hybrid approaches, the user-centric hybrid approach, is related to our work. It presents six rules for feature

interaction detection. Some of these rules are similar to ours; however, they do not attempt to justify these rules on the basis of basic principles as we do.

Also, Calder et al.'s work is to combine off-line and on-line approaches to detect and resolve Feature Interactions, which is not a main concern of our paper.

7.4. The work of Blair et al.

Blair and Turner (2005) present insightful research on handling policy conflicts in call control. This approach relates to our work mainly in two aspects: it models policies in a very similar way as ours, for instance, policies are specified by means of triggers, conditions and actions. Secondly, it discusses conflicts handling on a single policy server and in distributed setting, which corresponds to feature interactions in single and pair-wise CPL scripts in our work. However, they are concerned with the language APPEL, rather than CPL, and their main emphasis is on conflict resolution, rather than detection.

8. Future work

Many improvements on this process are still possible. Some were discussed in Section 4.3. This section discusses a few others.

8.1. Domain intersection and inclusion

The problem of domain intersection and inclusion is illustrated by the following example. Suppose that Alice (Alice@uottawa.ca) does not want to take any calls from people whose name contains "Carl" and she gives this feature the highest priority; also suppose that calls from "ibm.com" are so important to Alice that she wants all these calls handled by Bob in case she is absent, as shown below:

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

incoming(x, "sip:Alice@site.uottawa.ca")
⋀ address-switch(x.original.host = "ibm.com")
⋀ outcome(proxy(x, "sip:Alice@site.uottawa.ca"), "noanswer")
→
proxy(x, "sip:Bob@uottawa.ca")
```

If somebody named "Carl" in ibm.com (say "Carl@ibm.com") calls Alice@site.uottawa.ca when Alice is absent, then this call will be rejected rather than taken by Bob. There is no logical error but it would be helpful to remind Alice of this potential problem. Dealing with this kind of interaction would require providing the analyzer with information concerning domain intersection and inclusion, or simply asking the user questions. We leave this for further study.

8.2. Multi-way interactions detection

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

We believe that multi-way interactions could be detected by generalizing the method proposed in this paper; however, a more complicated notation would be required.

8.3. New types of features and policies

We have mentioned that CPL is a limited language. Extensions of CPL and more advanced policy languages will create new possibilities for feature creation, with their own possibilities for feature interaction (Dini et al., 2004; Reiff-Marganiec and Turner, 2004; Wu and Schulzrinne, 2005).

8.4. Feature interaction resolution

Feature interactions should be resolved after they are detected. As we mentioned earlier, when feature interactions are local logical inconsistencies within a CPL script, we need to offer choices to users to "correct" them. In this case, we refer readers to Amyot et al. (2005) although further work is necessary to adapt their method to ours. For pair-wise interactions detected at run time, negotiation may be necessary. The negotiation between two users could be done automatically by user agents with pre-set preferences and priorities. If negotiation cannot be done because of priority conflicts, we need to find a compromise between the two possibilities of inviting users to make decisions or simply terminating the call. Either way, a global services management model with a trusted third party requires further research. This topic is explored in Blair and Turner (2005) and Dini et al. (2004).

9. Conclusions

We have studied the problem of feature interaction in CPL from a logical point of view, on the basis of the concepts of policy, intention and logical contradiction. In some cases, basic contradiction rules identifying generally recognized conflicts of intentions were given, and then more specific detection rules were derived. We have also implemented a tool to translate CPL scripts into a logic-based language and to check conflicts on the basis of the rules. The tool provides clear diagnostics on the conflicts identified. The derived rules are simpler to check than the basic rules, thus detection is quick, although we have identified situations where detection would be more accurate by using additional information, such as domain structures, which would of course affect efficiency.

It is important to note that our detection rules are only proposals, which can be replaced by others. Our purpose is to demonstrate a logical process, and present the results obtained according to our assumptions. Accordingly, the set of feature interactions that we find can be expected to be complete only with respect to our rules. A comparison with the results

obtained by using a completely different method is given in Section 7. Feature interaction is a complex problem and different methods will find different types of interactions.

Acknowledgements

This project was supported by the Communications and Information Technology Ontario (CITO) and the Natural Sciences and Engineering Research Council of Canada (NSERC). Discussions with Tom Gray and Ramiro Liscano greatly enhanced our understanding of the feature interaction problem in Internet telephony. Furthermore, we would like to thank our colleagues at SITE, in particular Amy Felty and Daniel Amyot. The referees provided many useful comments that led to improvements in the paper.

References

Amyot D, Logrippo L. Feature interactions in telecommuications and software systems VII. In: Proceedings of the seventh international feature interaction workshop, Ottawa, 2003. Amsterdam: IOS Press; 2003.

Amyot D, Gray T, Liscano R, Logrippo L, Sincennes J. Interactive conflict detection and resolution for personalized services. J Commun Networks 2005;7:353–66.

Blair L, Turner KJ. Handling policy conflicts in call control. In: Reiff-Marganiec S, Ryan MD, editors. Proceedings of the eighth international conference on feature interaction VIII. Amsterdam: IOS Press; 2005. p. 39–57.

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

Calder M, Kolberg M, Magill E, Marples D, Reiff-Marganiec S. Hybrid solutions to the feature interaction problem. In: Amyot D, Logrippo L, editors. Feature interactions in telecommunications and software systems VII. Amsterdam: IOS Press; 2003b. p. 295–312.

Cameron EJ, Griffeth ND, Lin YJ, Nilson ME, Schnure WK, Velthuijsen H. A feature interaction benchmark for IN and beyond. IEEE Commun Mag 1993;31:64–9.

Dini P, Clemm A, Gray T, Lin FJ, Logrippo L, Reiff-Marganiec S. Policy-enabled mechanisms for feature interactions: reality, expectations, challenges. Comput Networks 2004;45(5):585–603.

Felty A. Temporal logic theorem proving and its application to the feature interaction problem. In: Giunchiglia E, Massacci F, editors. Issues in the design and experimental evaluation of systems for modal and temporal logics, Technical report DII 14/01, University of Siena, June 2001.

Gorse N. The feature interaction problem: automatic filtering of incoherences and generation of validation test suites at the design stage. Master thesis in Computer Science, University of Ottawa, 2000.

Gorse N, Logrippo L, Sincennes J. Formal detection of feature interactions with logic programming and LOTOS. Software Syst Model, to appear.

Huth M, Ryan M. Logic in computer science: modelling and reasoning about systems. Cambridge: Cambridge University Press; 2000.

International engineering consortium. Voice over Internet protocol: definition and overview; 2005. http://www.iec.org/online/tutorials/int_tele.

Lennox J, Schulzrinne H. Feature interaction in Internet telephony. In: Calder M, Magill E, editors. Feature interactions in telecommunications and software systems VI. Amsterdam: IOS Press; 2000. p. 38–50.

Lennox J, Schulzrinne H. CPL: a language for user control of internet telephony services. Draft-ietf-iptel-cpl-06, Internet draft, Internet Engineering Task Force, January 2002.

Nakamura M, Leelaprute P, Matsumoto K, Kikuno T. Detecting script-to script interactions in call processing language. In: Amyot D, Logrippo L, editors. Feature interactions in telecommunications and software systems VII. Amsterdam: IOS Press; 2003. p. 215–30.

Nakamura M, Leelaprute P, Matsumoto K, Kikuno T. On detecting feature interactions in programmable service environment of Internet telephony. Comput Networks 2004;45(5):605–24.

Reiff-Marganiec S, Ryan MD. Feature interactions in telecommunications and software systems VIII. Amsterdam: IOS Press; 2005.

Reiff-Marganiec S, Turner K. Feature interactions in policies. Comput Networks 2004;45(5):569–84.

Stepien B, Logrippo L. Representing and verifying intentions in telephony features using abstract data types. In: Cheng KE, Ohta T, editors. Feature interactions in telecommunications, III. Amsterdam: IOS Press; 1995. p. 141–55.

Turner KJ, Magill EH, Marples DJ. Service provision. New York: Wiley; 2004.

Wu X, Schulzrinne H. Handling feature interactions in the language for end system services. In: Reiff-Marganiec S, Ryan M, editors. Feature interactions in telecommunications and software systems VIII. Amsterdam: IOS Press; 2005. p. 270–87.

Xu Y. Detecting feature interactions and feature inconsistency in CPL, Master thesis in Computer Science. University of Ottawa; 2003.

Zave P. FAQ sheet on feature interaction; 2004. http://www.research.att.com/~pamela/faq.html.

Zisman A. An overview of XML. Comput Control Eng J 2000;11(4):165–7.