Handling Feature Interactions in the Language for End System Services

Xiaotao Wu[a] *and Henning Schulzrinne[b]

[a] Avaya Labs Research, Avaya Inc., USA
[b] Department of Computer Science, Columbia University, USA

June 25, 2006

Abstract

One big difference between Internet telephony and the public switched telephony network (PSTN) is that the PSTN usually assumes dumb endpoints, while Internet telephony has intelligent endpoints that can perform services. Performing services on endpoints may introduce many new communication services, make telecommunication services more distributed, and make the entire telecommunication network more robust and efficient. At the same time, it may also make telecommunication services more difficult to manage and require new techniques to create and compose services. In this paper, we first introduce the new scripting language, called the Language for End System Services (LESS), which we define specifically for end system service creation. LESS is extended from the Call Processing Language (CPL) and uses a tree-like structure to represent telecommunication services. Based on this tree structure, we propose a method based on LESS action conflict tables and a tree merging algorithm to detect potential feature interactions and help to resolve any conflicts detected. We integrated the method for handling feature interactions into our LESS-based end system service creation environment, which is part of our Session Initiation Protocol- (SIP) based user agent, SIPc. SIPc can execute LESS scripts for end system services and contains a service manager that can handle feature interactions.

1 Introduction

Telecommunication networks are moving from the circuit-switched public switched telephone network (PSTN) to packet-switched Internet telephony. A major difference between Internet telephony and the PSTN is that the PSTN usually assumes dumb endpoints while Internet telephony incorporates intelligent endpoints. In Internet telephony, endpoints usually have CPU and memory, so they are programmable and can perform services such as call forwarding, transfer, and call screening.

*Work mainly carried out while the author was at Columbia University. Email addresses: xwu@avaya.com (Xiaotao Wu), hgs@cs.columbia.edu (Henning Schulzrinne)

Performing services on end systems may introduce many new communication services, make telecommunication services more distributed, and make telecommunication networks more robust and efficient overall. At the same time, it may also make telecommunication services more difficult to manage, thus require new techniques for creating and composing services.

In this paper, we first introduce a new scripting language we defined for end system service creation, *the Language for End System Services* [1] [2], or LESS. As an extension of the Call Processing Language (CPL) [3], LESS enables comparatively inexperienced end users to create services that run on Internet telephony endpoints. This language is designed to make it impossible for users to do anything more complex (or dangerous) than describe Internet telephony services, but powerful enough to handle many commonly used services. In our technical report on end system service examples [4], we show that LESS can handle a big part of the services defined in AT&T 5ESS switches [5], ITU-T recommendation Q.1211 [6], and CSTA Phase III [7]. We also discuss how to use LESS to describe services involving other Internet applications such as email, web, presence notification, instant messaging, and networked appliance control.

Like PSTN services, Internet telephony end system services also have problems with feature interaction. "Feature interaction is an inevitable by-product of feature modularity." [8] While modularity enables efficient service creation, it may also cause feature interactions among multiple service scripts running on one or more devices. When users create new services, they often focus on their immediate needs without checking existing services, so feature interactions are likely to arise between newly created scripts and the old ones. One design goal of LESS is to facilitate the easy detection and resolution of feature interactions among LESS scripts.

LESS inherits CPL's tree-like structure to represent communication services. This structure enables the back-and-forth translation between graphical and textual representations of LESS scripts and makes it easy to handle feature interactions among LESS scripts. In this paper, we introduce a tree-merging algorithm to detect interactions among LESS scripts. The algorithm is based on the LESS action conflict tables, which we carefully define for analyzing LESS-based feature interactions. Once a feature interaction is detected, our algorithm can clearly identify the conditions that may cause the interaction, and our service management system can then guide users in resolving any feature conflicts detected.

Several methods already exists to handle feature interactions in CPL, but because feature interactions in LESS are very different from those in CPL, those methods cannot fully handle the feature interactions in LESS. In Section 1.1, we detail the differences between LESS and CPL in handling feature interactions and discuss the existing methods. We then briefly introduce LESS in Section 2. Section 3 shows the LESS action conflict tables. Section 4 discusses our tree-merging algorithm based on those action conflict tables. In Section 5, we present the implementation of our LESS-based service creation and execution environment for handling feature interactions. Section 6 concludes the paper and discusses our future work.

1.1 Related work

Most of the work discussed in this paper is based on the Session Initiation Protocol (SIP) [9]. SIP is an IETF standard used to setup Internet telephony call sessions. With extensions, such as those for presence [10] and for instant messaging [11], SIP can also handle functions beyond multimedia session setup. The definition of LESS is based on SIP.

LESS extends CPL and inherits many CPL characteristics, such as the tree-like structure, without user-defined variables and loops, and the inability to run external programs. However, there are many differences between LESS and CPL. CPL was designed to run on network servers such as SIP proxy servers; therefore, CPL focuses on call signaling routing. On the other hand, LESS is designed for endpoints, which are usually at the end of call signal routing paths. Therefore, LESS focuses on initiating, or terminating call sessions and controlling endpoints. Some CPL-defined signaling actions, such as `proxy`, cannot be used on endpoints, while LESS incorporates many new actions that are not in CPL but are required by endpoints such as `call`, `accept`, `transfer` and user interaction actions. LESS also defines new triggers for user interactions and timer event handling. Furthermore, LESS can handle presence information, instant messaging, and other Internet applications such as networked appliance control. In addition, because LESS runs on endpoints, it can acquire the context information that is only available to endpoints for service handling such as the availability of audio devices. For these reasons, feature interaction handling in LESS is very different from that in CPL.

Several methods exist on feature interactions in CPL. Xu et al. proposed to translate CPL into a formal language to check feature interactions [12]. Amyot et al. developed a tool called FIAT for filtering inconsistencies among features, then implemented a translator to convert CPL to the FIAT input language [13]. FIAT has a user-friendly web-based user interface for handling conflicts detected among CPL scripts. Nakamura et al. analyzed possible semantic warnings in an individual CPL script, then extended the analysis to multiple scripts by defining an operator to combine multiple scripts into one [14]. Because LESS is extended from CPL, we can use these approaches to detect and resolve feature interactions. However, because of the many differences between LESS and CPL, the existing work cannot fully handle feature interactions in LESS.

All the existing work focuses on call routing services because they are CPL-based. Thus, they do not handle signaling actions for endpoints such as `accept`, `terminate`, and `call`. These new signaling actions in LESS can introduce many interesting feature interactions that could not occur among CPL scripts. Though the existing work may be extended to handle feature interactions in LESS, the work in this paper, especially the action conflict tables, can be of great help for their extensions.

We also noticed that both Nakamura and Xu's work handle feature interactions among multiple users' scripts because CPL scripts run on proxy servers that are usually owned by ISPs. ISPs may have the right to access multiple users' service scripts. However, LESS scripts usually run on endpoints that cannot access other users' scripts. Thus, in general, we do not expect to handle feature interactions among multiple users' LESS scripts.

In fact, even if there were a centralized server that could access all the LESS scripts

of multiple users, privacy concerns would make it difficult to resolve the detected interactions. For example, user Bob has a script that keeps calling Alice every 10 minutes, but Alice has a script that rejects all calls from Bob. According to Nakamura's work, the combined script causes a CRAE (call rejection in all execution paths) semantic warning. But can we inform Bob that his calls to Alice will always be rejected? The answer depends heavily on the social context surrounding Bob and Alice. For example, in an enterprise environment that requires employees to share service information, Bob may be informed about the conflict. In this case, multi-user feature interaction handling is possible and very useful. But in a residential environment, to protect Alice's privacy, Bob should not be informed. In the latter case, the conflict cannot be resolved.

Neither Nakamura nor Xu addressed how to resolve feature interactions. The FIAT system of Amyot et al provides a viable way to resolve feature interactions. FIAT has a web interface and involves end users in resolving feature interactions. It provides suggestions like `Add EXCEPTION`, `DISABLE`, `SET PRIORITY`, and `TOLERATE`, as well as human-understandable explanations to guide users in resolving the detected feature interactions. We consider it one practical way to involve end users in resolving feature conflicts. Our implementation also uses a popup dialog that shows users the detected feature conflicts and asks users to make a choice. Our feature interaction detection algorithm can clearly identify the conditions that may cause the detected interactions. We can then present the conditions to users in a human-readable way and ask users to make decisions. The users' decisions can be saved in our service management system. However, our user interface for resolving feature interactions is different from the FIAT system in that we do not define different suggestions for users, but instead ask users to make a choice among the actions they originally defined in their own services. We then automatically prioritize, disable, or merge services, or tolerate the conflicts based on users' choices. In addition, we integrated our service learning and service risk management work into our feature interaction handling implementation. Users can receive suggestions that have been inducted from their call histories and have more options for reduced-risk call handling actions. We detail our implementation in Section 5.

There are also some existing works on feature interaction in policies [15], and on using logic programming [16][17] to detect feature interactions. These works can handle event-based call processing, which is also the call processing model for LESS and CPL. But those works do not address the feature interactions on endpoints among end user created services, which is the focus of this paper.

Some researchers proposed to use architectural approaches to deal with feature interactions in general. Architectural approaches attempt to clearly define the relationships among features to make feature composition possible. One simple architectural approach statically defines the precedence of all the features, then executes features in order. This approach is far from enough to deal with complex communication services because in many cases, the precedence of features is dynamic when performing communication services.

Other more sophisticated architectural approaches are available to handle interactions among well-modularized features, such as the pipe-and-filter architecture [18], the Distributed Feature Composition (DFC) [19], and the agent-based architecture [20]. We will not describe these architectural approaches in detail but note one fact

that makes them unsuitable for end-user-created LESS scripts. All these approaches assume that features are carefully designed and modularized, and thus handle feature interactions based on this assumption. They assume features following the pipe-and-filter architectural style: "Feature components are independent, they do not share state, they do not know or depend on which other feature components are at the other ends of their calls (pipes), they behave compositionally, and the set of them is easily enhanced [18]" [19]. This assumption can be held for many existing PSTN services, which are designed by professional service designers, and are carefully checked to make them compositional.

However, for services created by inexperienced service creators, such as CPL or LESS scripts, this assumption is unlikely to hold. Users may create an ill-formatted feature that overlaps with existing features or a monolithic script that should be divided into multiple modules. It is difficult to use architectural approaches to handle these ill-formatted features in end systems. Sometimes, architectural approaches may be required to reconstruct the ill-formatted features, and compose the reconstructed features to achieve the expected results. The reconstruction certainly gives the features a better format, but it is not the original format the users are familiar with. For example, in Figure 1, *feature1* and *feature2* conflict when the time is between 2:00PM and 3:00PM on Dec 25, 2004. The *preferable service* shown in the figure takes *feature1*'s decision if the caller is *sip:t@a.com*, but takes *feature2*'s decision if not. Defining the precedence between the two services cannot resolve the conflicts. However, merging two trees into one, as the figure shows, can easily resolve the conflicts.

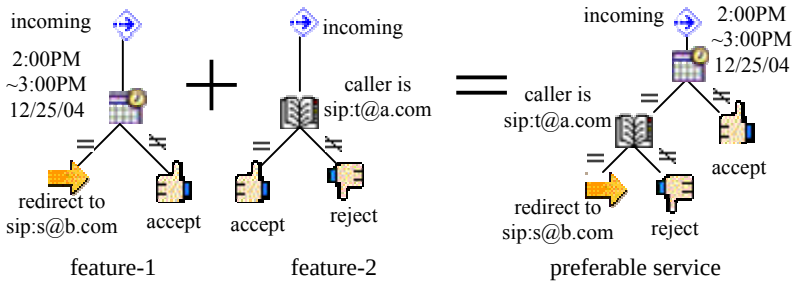


Figure 1: Merging two trees to get a preferable service

Part of the work in this paper was published in our paper [21] in *International Conference on Feature Interactions in Telecommunications and Software Systems (ICFI'05)*. This paper corrects many errors in the action conflict tables in the ICFI paper. It also illustrates in great detail the relationship between action conflicts and end system capabilities and introduces the concept of using service risk management and service learning to handle feature interactions. In addition, this paper introduces LESS and presents our implementations in much more detail.

2 Language for End System Services (LESS)

2.1 Requirements for an end system service creation language

An end system service creation language or a subset of that language must be simple, easy to understand, safe for creating error-free services, and powerful enough to allow comparatively inexperienced users to build a wide range of services. The language should also be extensible to allow the introduction of more complicated concepts and elements in order to enable experienced service programmers to build more powerful services.

Because users may want to have the same services on different devices, a platform-neutral high-level language is called for. The language must be able to express user interactions and control media streams. It should also be extensible to accommodate new Internet services. Because we must restrict its power to ensure its simplicity and safety, we do not expect that the language to handle all kinds of services. Thus, it need not be Turing-complete.

In Internet telephony, services are not restricted to a small number of named services such as "call forwarding busy"; rather, it appears more plausible to have policy-based or rule-based services, e.g., certain events under certain conditions invoke certain actions, similar to the way email user agents, for example, support message filtering and forwarding. Using rules to describe a service is easier to understand than using C/C++ or Java to program a service; however, those rules must be powerful enough to represent a wide range of telecommunication services [4].

2.2 High-level abstraction

Figure 2 shows how LESS abstracts communication services. A LESS script is a collection of event handling rules. The events can be an incoming call, an outgoing call, or a timer event. For example, a LESS event handling rule can be *"when an incoming call arrives, check the caller of the call, if the caller is sip:tom@example.com, reject the call."* LESS uses a structured way to describe the rules so a LESS interpreter can interpret the rules and perform the desired communication services. A LESS event handling rule consists of a trigger, zero or more switches, and one or more actions. Modifiers are used to provide arguments for actions. A rule is invoked only when a signaling or nonsignaling event matches its trigger, e.g., an incoming call event matches an `incoming` trigger.

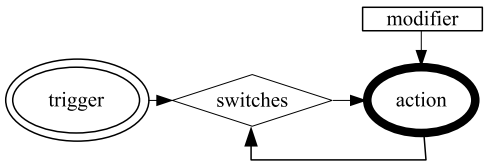


Figure 2: Call decision making process in LESS

Once a trigger is invoked, switches check the status of triggers and their context for decision making. For example, an `address-switch` checks the caller and the callee's addresses and makes call decisions based on those addresses; for example, `accept`, `reject`, or `redirect` calls. Modifiers can provide action arguments. For example, a `location` modifier can specify the target uniform resource identifiers (URI) of a `redirect` or a `call` action. One action may be followed by additional processing, e.g., more switches for condition checking and more actions for event handling. Multiple actions can also be executed in parallel.

The relationship between triggers and actions is important in defining LESS. Actions are used to handle the event that invokes a trigger. Not every action can apply to every trigger. In LESS definition, we clearly define the triggers each action can apply to. For example, an `accept` action, which automatically accepts an incoming call, can only be used in an `incoming` trigger. We cannot use an `accept` action in an `outgoing` trigger, which handles outgoing SIP INVITE requests. In addition, in a LESS script any LESS action may not invoke another LESS trigger, thus avoiding infinite loops. In this way, a LESS `call` action does not invoke a LESS `outgoing` trigger.

LESS inherits the CPL tree-like structure in which LESS elements, such as triggers, switches, and actions are called nodes. Each node has one or more outputs that can connect to additional nodes to further process an event. For example, the script fragment in Figure 3 shows an `address-switch` with two outputs. The first output handles incoming calls from `sip:tom@abc.com`. The second output handles calls from others.

```
<address-switch field="origin">
  <address is="sip:tom@abc.com"> <accept/> </address>
  <otherwise> <reject/> </otherwise>
</address-switch>
```

Figure 3: Address-switch example

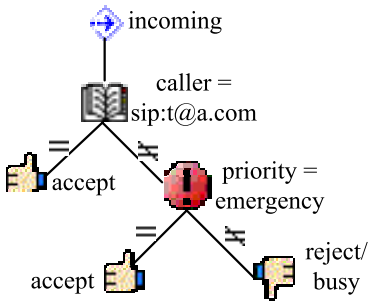


Figure 4: An example of the tree-like structure for trigger handling

Each LESS script starts at a trigger node, which then connects to a switch node or an action node. The outputs of the switch or action node are connected to additional nodes until a node with no specified outputs is reached. There are no back references from a child node to its ancestors or to itself. Later in this paper, we use the term *LESS decision tree* to refer to a LESS script. A LESS decision tree can be converted to a set of event handling rules if we define a rule as the path from the root of a decision tree (a trigger node) to a leaf of the decision tree (an action node). Figure 4 shows a decision tree, which represents the LESS service script shown in Figure 5.

```
<?xml version="1.0"?>
<less> <incoming>
  <address-switch field="origin">
   <address is="sip:t@abc.com"> <accept/> </address>
   <otherwise> <priority-switch>
     <priority equal="emergency"> <accept/> </priority>
     <otherwise> <reject status="486" reason="Busy"/>
     </otherwise>
   </priority-switch> </otherwise>
  </address-switch>
</incoming> </less>
```

Figure 5: LESS script example

2.3 LESS elements

As we describe in Section 2.2, LESS has four kinds of elements: triggers, modifiers, switches, and actions. We briefly introduce these elements below. The complete definitions are in the Internet draft we submitted to the IETF [1]. A trigger is an entry point to every service, and is the same as the `toplevelaction` defined in CPL. Table 1 shows the definitions of the triggers in LESS.

Trigger	Basic/Extension	Events to handle
`incoming`	basic definition	incoming calls
`outgoing`	basic definition	user made outgoing calls
`timer`	basic definition	timer events
`subscription`	event extension	incoming event subscriptions
`notification`	event extension	incoming event notifications
`message`	IM extension	incoming instant messages

Table 1: LESS triggers

In end systems, an outgoing call is usually triggered by a user interaction, e.g., dialing a number and pressing a call button. The `outgoing` trigger in fact handles user interactions. Thus, in a LESS script the `call` action, which automatically generates outgoing calls, does not invoke the `outgoing` trigger.

Modifiers can provide action parameters. Table 2 shows the definitions of the modifiers in LESS.

Modifier	Basic/Extension	Definition
`location`	basic definition	add a URI in the location set
`remove-location`	basic definition	remove a URI from the location set
`lookup`	basic definition	lookup URIs and add to the location set
`media`	media extension	change media attributes of calls

Table 2: LESS modifiers

Switches represent the choices a LESS script can make. Table 3 shows the switches.

Switch	Basic/Extension	Make decisions based on
`time-switch`	basic definition	the time of executing a LESS script
`address-switch`	basic definition	the addresses, e.g., the caller's URI
`priority-switch`	basic definition	the priority of the original call
`string-switch`	basic definition	the free-form strings in a call request
`language-switch`	basic definition	the languages in which the originator of a call wishes to communicate
`status-switch`	basic definition	people's status, e.g., busy
`event-switch`	event extension	the event values in an event subscription or notifications
`where-switch`	location extension	the physical locations of people

Table 3: LESS switches

Actions represent users' decisions for trigger handling. Table 4 shows the actions.

LESS can be easily extended to have more triggers, switches, modifiers, and actions. In the LESS XML schema, we define four abstract elements: `'trigger'`, `'switch'`, `'action'`, and `'modifier'`. We constructed the LESS tree structure based on these abstract elements. We then require that all the new triggers, switches, modifiers, and actions must be the `'substitutionGroup'` of the abstract elements. Using `'substitutionGroup'` for LESS extensions can preserve the tree structure so that the tree-merging algorithms defined here can also be applied to LESS extensions in the future. However, because the LESS action conflict tables, which we introduce in the next section, can only be constructed by experts, we require that people who define a LESS extension must also be able to define action conflict tables for that extension.

3 Feature interaction detection in LESS

Feature interactions among multiple LESS scripts may occur if multiple actions are invoked at the same time. There may be no interactions among the features; sometimes, the interactions are desired, but in many situations features conflict. For example, for

Action	Basic/Extension	Definition	Parameters
`accept`	basic definition	accept a call	none
`reject`	basic definition	reject a call	code, reason
`redirect`	basic definition	redirect a call	permanent
`call`	basic definition	make a call	none
`terminate`	basic definition	terminate all calls	none
`mail`	nonsignaling	send an email	none
`log`	nonsignaling	log events	none
`wait`	nonsignaling	wait a duration of time for subsequent actions	duration
`media-update`	media extension	change media setup for ongoing sessions	none
`transfer`	mid-call extension	transfer existing calls	none
`merge`	mid-call extension	merge existing calls into a conference call	none
`alert`	UI extension	play alerting messages	uri, ...
`sendmsg`	IM extension	send instant messages	none
`approve`	event extension	approve event subscriptions	none
`deny`	event extension	deny event subscriptions	none
`defer`	event extension	defer the decision on event subscriptions	none
`subscribe`	event extension	send an event subscription	none
`notify`	event extension	send an event notification	none

Table 4: LESS actions

an incoming call, one script performs an action to accept the call, while another script performs an action to log the call. The `accept` and `log` actions do not interact. In another example, when a user is already in a call session and receives a new call, one script performs an action to transfer the existing calls, while another script performs an action to automatically accept the new call. If there is only one audio input/output resource in the user agent, the `accept` action must be performed after the `transfer` action. In this case, the feature interaction is desired. Based on this observation, the relationships among actions must always be analyzed. Because we must also consider the availability of resources of an endpoint, such as the number of audio devices, in deciding whether two actions conflict, we must analyze the relationship between feature interactions and end system capabilities.

E.J. Cameron et al. [22] classified feature interactions into three dimensions – customer-system, single-multiple user, and single-multiple component dimension – and five categories – SUSC (single-user-single-component), SUMC (single-user-multiple-component), MUSC (multiple-user-single-component), MUMC (multiple-user-multiple-component), and CUSY (customer-system). End system services usually experience single-user interactions. In other words, in this paper we focus only on SUSC and SUMC feature interactions. One exception is the interaction between the caller's pref-

erences and the callee's service scripts, which involves multiple users. We discuss this kind of multi-user feature interaction in Section 3.4.

In the PSTN, feature interactions may occur when multiple users share one end device. However, in Internet telephony, different people usually have different URIs even if they own the same end device. This is different from the PSTN situation, in which multiple persons sharing one end device also share one logical address – the phone number. Because of this difference, in Internet telephony end systems, we can still perform single-user feature interaction handling even if multiple users use one device.

For single-user feature interactions, we can define preconditions and expected results for LESS actions. Based on the preconditions and expected results, we can construct action conflict tables and use the tables to detect feature interactions. In the sections that follow, we categorize LESS actions into call control actions, presence notification actions, and other actions such as instant messaging and networked appliance control. For each set of actions, we first analyze the preconditions and their expected results, then check both SUSC and SUMC feature interactions.

3.1 End system call control actions

The call control actions can be signaling or nonsignaling actions, and can be in different call stages. Table 5 shows the actions.

stage/action	Signaling actions	Nonsignaling actions
Incoming call handling	accept, reject, redirect	log, mail, wait (all stages)
Outgoing call setup	call	
Mid-call stage	transfer, media-update, merge	
Call termination	terminate	

Table 5: Call control actions

For signaling actions, the actions that belong to the same call stage usually conflict. For example, an end system can only choose among the `accept`, `reject`, or `redirect` actions to handle an incoming call. Actions at different call stages may also interact. For example, accepting an incoming call then transferring the call is a desirable interaction; however, rejecting a call then transferring the call is an undesirable interaction. In this case, the nonsignaling actions do not conflict with the signaling actions.

To check feature interactions between two actions, we must define the execution order of the actions and check possible interactions in different orders. For example, if we want to check the interactions between action A and action B, we first check the situation in which A is performed before B. It consists of two steps: checking whether A's result changes or conflicts with the precondition of B, and checking whether B's result changes the expected result of A. We then check the interactions in a different execution order with B performed first. The preconditions and expected result of each action are shown in Table 6.

action	precondition	call states	device states
accept	incoming call setup pending, media devices available	call setup finalized, a session is setup	media devices occupied
reject	call setup pending	call setup finalized	no change
redirect	call setup pending	call setup finalized	no change
call	media devices available	if accepted, a session is setup	if accepted, media devices occupied
transfer	one or more sessions, media devices occupied	if succeeds, all sessions terminated	if succeeds, media devices available, otherwise, media devices occupied
media-update	one or more sessions, media devices occupied	all sessions alive	media transmission changed, e.g., held, or muted
merge	one or more sessions, media devices occupied	all sessions merged	media devices occupied
terminate	one or more sessions, media devices occupied	all sessions terminated	media devices available

Table 6: The context assumption and expected result of call control actions

We further investigate the cause of feature interactions and find five kinds of interactions. We call the first *action conflict*, which has the expected result of two actions conflicting, e.g., the conflict between an `accept` and a `reject` action.

We call the second *attribute conflict*, in which two actions have the same name but different attributes. LESS modifiers should be treated as action attributes. Two actions with the same name conflict if their modifiers are different. For example, two scripts conflict if they both perform `redirect` actions, but to separate locations.

We call the third interaction *resource competition conflict*. Two actions may compete for resources such as audio devices. For example, if there is only one audio device in an end system, two calls using the one audio device will cause a conflict such as accepting an incoming call and making an outgoing call to another address at the same time.

We call the fourth interaction *disabling conflict*. It occurs when one action's expected results make another action's preconditions impossible. For example, if a script `terminates` existing sessions, another script cannot execute `media-update` action because the precondition of `media-update` assumes one or more existing sessions. The disabling conflicts are similar to the semantic warnings in the paper by Nakamura et al. [14].

We call the last kind of interactions *"enabling interactions"*. It occurs when one action's expected results enable another action's preconditions. This kind of interaction

is desirable. For example, the `accept` action can enable the `transfer` action for the same call.

3.1.1 SUSC feature interactions

	accept	reject	redirect	transfer	merge
accept	A(m)	C	C	E	E
reject	C	A(r)	C	D	D
redirect	C	C	A(a)	D	D
transfer	E	-	-	A(a)	C
merge	R	-	-	C	-
m-update	R^1,E^1	-	-	C	C
term	E	-	-	D	D
call	R	-	-	E	E

	m-update	term	call
accept	E	E	R
reject	D	D	E
redirect	D	D	E
transfer	C	C	E
merge	C	C	R
m-update	A(m)	C	R^1,E^1
term	D	-	E
call	E	E	R

m-update: `media-update`, term: `terminate`, -: no interaction, C: action conflict,
A(m): attribute conflict on media, e.g., using video or not
A(r): attribute conflict on code/reason, e.g., using 4xx or 6xx response in SIP
A(a): attribute conflict on address, e.g., redirect to different addresses
E: enabling, D: disabling conflict, R: resource competition
R^1: `media-update` for unholding calls competes resources with `call` or `accept`
E^1: media-update for holding calls enables `call` or `accept`

Table 7: Call control action conflict table for handling `incoming` trigger

Table 7 shows the conflict table for call control actions. One assumption in the table is that a call usually requires audio, and there is also one audio device in an end device. For video and text conversations, people can watch multiple video windows and handle multiple instant messaging sessions simultaneously if the CPU power or network bandwidth allows, so we do not consider resource competition for video and text conversations.

The table is asymmetrical. The cell of row `m`, column `n` and the cell of row `n`, column `m` may not have the same values. In this table, row actions are performed before column actions. For example, row 1, column 4 means *`accept` then `transfer`*.

3.1.2 SUMC feature interactions

A user's service scripts can be hosted on the user's end devices and signaling servers in the network. However, scripts in different places may interact. For example, if the scripts on a SIP proxy server reject all calls, the scripts on the destination end devices can never be executed. When a proxy server proxies a call to all the end devices of a user in parallel, if one of the end devices (e.g., the voicemail server) automatically accepts the call immediately, the other end devices never have the chance to accept the call. These examples involve one user but multiple end devices for the user so they are SUMC feature interactions. We divide these kinds of SUMC feature interactions into two categories: *end system–proxy server* and *end system–end system* feature interactions.

End system–proxy server feature interactions: End system–proxy server feature interactions are caused when the CPL scripts on proxy servers interact with the LESS scripts on end systems. There are only three signaling actions for CPL: `proxy`, `redirect`, and `reject`. Every action may interact with the actions on end systems. For incoming calls, proxy server scripts are executed before end system scripts. They may interact in two ways: proxy server scripts may block the execution of end system scripts or proxy server scripts may overlap with end system scripts. For outgoing calls, end system scripts are executed before proxy server scripts, and proxy server scripts may modify the result of end system service scripts. Table 8 shows the possible interactions for incoming call handling.

server/end	accept	reject	redirect
reject	block	overlap	block
redirect	block$^+$	block$^+$	block/overlap$^+$
proxy	block$^+$	block$^+$	block$^+$

+: depending on the URI to which to redirect or proxy a call

Table 8: Interactions between services on end systems and proxy servers for incoming call handling

As shown in the table, if a proxy server uses the `proxy` action to handle an incoming call, and an end system tries to use the `accept` action to handle the same call, the `proxy` action blocks the execution of the `accept` action. The `accept` action is executed only if the target URI of the `proxy` action is equal to the end system's URI. So we use block$^+$ to mark the interactions that depend on the target URI of the `redirect` or `proxy` actions.

Table 9 shows the possible interactions for outgoing calls and call termination handling. In this table, end system actions are performed before proxy server actions.

End system–end system feature interactions: End system–end system feature interactions involve multiple end devices belonging to one user. These kinds of feature interactions are the most complicated feature interactions for end system services. They sometimes also involve service scripts running on proxy servers. For example,

end/server	reject	redirect	proxy
call	block	modify	-
transfer	block	modify	-
terminate	N/A	N/A	-

Table 9: Interactions between services on end systems and proxy servers for outgoing call and call termination handling

if a SIP proxy server sequentially proxies a call with a voicemail server at the last, the auto-accept script running on the voicemail server does not affect other end devices' behavior. However, a proxy server can also proxy a call to a number of locations at the same time. This type of parallel proxying is known as forking [9]. When a proxy server does forking, if the timeout value of the auto-accept script running on the voicemail server is set as zero or as a very small value, it may render other end devices unable to accept incoming calls. Thus we must also consider proxy server scripts when we handle end system–end system feature interactions. Table 10 summarizes the action conflicts between two end systems.

end1 / end2	accept	reject	redirect
accept	C	-	C
reject	-	-	A(r)
redirect	C	A(r)	A(am)

C: action conflict, A(r): conflict based on reason attribute,
A(am): conflict based on address and media attribute

Table 10: Action conflicts between two end systems for incoming call handling

The table shows that during the call setup stage, the forking proxy in Internet telephony systems may cause multiple end devices to receive an incoming call setup request at the same time. If they all try to accept the call, or try to redirect the call to separate locations, a feature conflict occurs. To resolve this kind of feature conflict, the forking proxy must choose a single best response.

In SIP, there are different status codes for rejecting a call. If one end system rejects a call using a `603 Busy everywhere` status code, other end systems should not try to redirect the call. This is similar to the Basic Rule BR1 in [12]. Typically, a person would not talk with another person using two different end devices of the same media type. The A(am) in the table indicates such feature interactions. The `redirect` action may also cause call forwarding loops, which can be detected by checking the locations of the actions based on the table.

Action conflicts between two end systems can be more complicated, and not just for incoming call handling. For example, two end systems that handle two different calls but transfer their calls to the same destination may cause the destination to reject one call. Table 11 shows the more complicated action conflicts between two end systems.

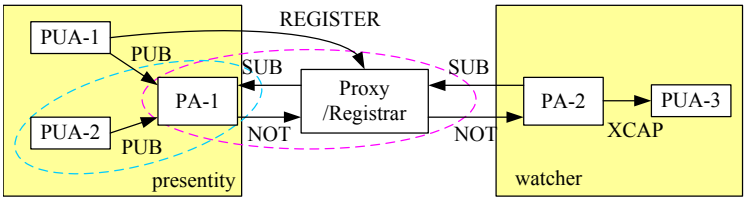
end1 / end2	call	transfer	terminate
accept	A(am)	A(am)	-
reject	-	-	-
redirect	A(am)	A(am)	-
call	A(am)	A(am)	-
transfer	A(am)	A(am)	-
terminate	-	-	-

A(am): conflict based on address and media attribute

Table 11: More complicated action conflicts between two end systems

3.2 End system presence and event notification actions

We base our event-based services on the SIP event notification architecture [23]. Figure 6 shows the architecture.



SUB: SUBSCRIBE, NOT: NOTIFY, PUB: PUBLISH

Figure 6: SIP event notification architecture

In this architecture, a Presence User Agent (PUA) manipulates presence information for a presentity. A PUA is an entity that can acquire the presence information of a human, program, or collection of humans and/or programs and can transmit the state information to a presentity. A presentity is an entity that provides presence information to a presence service. The presence service can then send the presence information to watchers interested in the information. A watcher is a logical entity that requests presence information about a presentity, or watcher information about a watcher, from presence services. There can be multiple PUAs per presentity. A presence agent (PA) can access presence data manipulated by PUAs for the presentity, and can then receive SUBSCRIBE requests, respond to them, and generate notifications of changes in presence state. One way for a PA to access the data is by co-locating the PA with the proxy/registrar. Another way is to co-locate the PA with the PUA of the presentity [10]. In Figure 6, PUA-2 and PA-1 may co-locate together so PA-1 can directly access the presence information of PUA-2. PA-1 can also co-locate with the proxy/registrar and access the presence information through SIP REGISTER requests sent by PUAs. If a PUA and a PA are not co-located, the PUA can use SIP PUBLISH [24] requests to send their status to the PA. PA-2 may use SIP SUBSCRIBE requests to ask for presence status stored in PA-1, and PA-1 will use SIP NOTIFY requests to send the status.

PUA-3 may use XCAP [25] to retrieve the status information stored at PA-2.

Feature interactions for event-based services can be SUSC or SUMC interactions, depending on whether the PUA and the PA of a user are co-located or not. If a PUA and a PA are co-located, we must deal with SUSC feature interactions; otherwise, we must deal with SUMC interactions.

Actions related to event-based services can be divided into two categories. One category handles incoming subscriptions, such as `approve` or `deny`. The other sends outgoing messages, such as the `subscribe` and `notify` actions. Feature conflicts can be categorized into action conflicts and action attribute conflicts, the same as those we define in Section 3.1.

3.2.1 SUSC feature interactions

For an incoming subscription, `approve` and `deny` actions conflict. `Approve` actions with different attributes such as different subscription expiration times also conflict. Similarly, `deny` actions having different reasons conflict. `Subscribe` and `notify` actions do not conflict with other actions, but they may cause action attribute conflicts. Two `subscribe` actions will conflict if they have the same destination and the same event package, but differ in other attributes such as the expiration time. Two `notify` actions will conflict if they have the same destination and the same event package (e.g., `presence`), but different event descriptions (e.g., one presence status is `open`, the other is `closed`).

3.2.2 SUMC feature interactions

For an incoming subscription, a PA can decide whether to `approve` or `deny` it. A PA can also set the subscription status as *pending*, and send a notification to the PUA about the watcher-list changes [26]. Once the PUA gets the watcher-list, the PUA will use XCAP to update the watcher-list document on the PA to authorize the subscription. A user may have multiple PUAs. Feature interactions may occur among the PA and all the PUAs involved in handling incoming subscriptions. For an incoming subscription, if a PA and a PUA make different decisions (for example, the PA approves a subscription but the PUA denies the subscription), an action conflict occurs.

3.3 Other end system services

In Internet telephony, end systems can support instant messaging and networked appliance control. Therefore, we also analyze feature interactions for these services.

3.3.1 Feature interactions for instant messaging

LESS has only one action for instant messaging, namely `sendmsg`, which is used to send a message. If we ignore the content of messages, there are no conflicts between multiple `sendmsg`s. However, the content of a `sendmsg` may have special meanings in some circumstances. For example, if we use SIP MESSAGE to perform shared web browsing [27], the message content is used to convey URL information. In general, we consider that two `sendmsg`s of differing content conflict.

Instant messaging may also experience SUMC feature interactions. One incoming message may be sent to multiple contacts of a user. If more than one contact can automatically send a message back, SUMC feature interactions may occur. There is little difference between SUSC and SUMC instant messaging feature interactions: both depend on whether the message content conflict.

3.3.2 Feature interactions for networked appliance control

Internet telephony user agents can control networked appliances. For example, when getting an incoming call, a UA can automatically turn off a nearby stereo. Networked appliance control services can be very complicated. Different sensors may trigger different control actions, and the actions performed by multiple networked appliances may conflict. For example, turning on an air conditioner to cool a room and turning on a heater to warm the room conflict. Kolberg et al. described these kinds of feature interactions in detail in [28].

If multiple LESS scripts try to control a networked appliance to perform different actions, feature interactions may occur. Different networked appliances may have different interactions. For example, when controlling a lamp, `power on` and `power off` conflict. To control a stereo, the `power on` action enables the `tune` action. To analyze feature interactions, we must first identify the appliance we want to control and its available control actions. We must then build the preconditions and expected result tables for the control actions. From these tables, we can build the action conflict tables for the device. We choose to use lamp control as an example for feature interaction analysis. Keep in mind that networked appliance control actions may interact with call control actions. For example, the brightness of the lamp in a person's room may affect the perceptual quality of the person's video communications.

The commands for a lamp can be `power on`, `power off`, `dim`, and `bright`. Table 12 shows the preconditions and expected results of the lamp control actions.

	precondition	expected result
power on	-	The lamp is on.
power off	-	The lamp is off.
dim	The lamp is on.	The lamp is dimmer and still on.
bright	The lamp is on.	The lamp is brighter and still on.

Table 12: Context assumption and expected results of lamp control actions

Table 13 shows the conflict table, which assumes that multiple scripts are trying to control the same device at the same time. We also added two call control actions, `accept` and `call`, to show that networked appliance control actions may cause feature interactions with call control actions. In the table, the `power on` action makes `dim` and `bright` possible, and the `bright` or `power on` action may provide a better environment for a video call, so we use *enabling* to mark this kind of interactions.

Networked appliance control intrinsically involves multiple components, the controller and the appliances. If multiple users try to control one networked appliance at

	power on	power off	dim	bright	accept	call
power on	-	C	E	E	E(V)	E(V)
power off	C	-	C	C	C	C
dim	-	C	A	C	C	C
bright	-	C	C	A	E(V)	E(V)

A: attribute conflict, C: conflict, E: enabling, E(V): enabling video communications

Table 13: Interactions between lamp control actions

the same time, MUMC feature interactions may occur. If all users access one device through the same appliance controller, e.g., a networked appliance gateway, the policies residing on the controller may help resolve the conflicts. For example, the controller administrator may define user priorities. User actions having higher priority may override user actions having lower priority. If multiple users access a device through different controllers, intercontroller communication is required to solve possible feature conflicts. This kind of feature interaction is beyond the scope of the LESS-based feature interaction handling.

3.4 Feature interactions between caller's preference, end system's capabilities and users' service scripts

Sometimes, a caller may explicitly express preferences in a call signaling message [29]. These preferences include the ability to select which URI a request is routed to, and to specify certain request handling directives in proxy servers. For example, the `Reject-Contact: *;mobility="mobile"` header in a SIP INVITE request expresses a desire not to route a call to a mobile device. The caller's preferences may conflict with the callee's service scripts. These kinds of conflicts cannot be detected offline. However, they are easy to detect by checking the callee's service script actions and the value of the Reject-Contact header in the caller's SIP message. If a feature interaction occurs, the caller's preferences should override the callee's service script actions. For example, Alice uses a mobile phone and has a service script that automatically `accept`s calls from Bob. However, Bob does not want to talk to a mobile phone so he puts `Reject-Contact: *;mobility="mobile"` in his request. In this case, Alice's phone should not accept the call. Sometimes, service script actions may also conflict with end system capabilities. For example, for an incoming video only call to an end system with only audio capability, an `accept` action is not appropriate. In this case, system capabilities should always override script actions. In this example, the end system should prompt the user for proper handling.

The action conflict tables in previous sections contain several assumptions about end system capabilities, e.g., with one audio input/output device, or with enough bandwidth and CPU resources to handle multimedia calls. Although these assumptions hold for many Internet telephony endpoints, some endpoints may have more or fewer resources. Most of the action conflict table elements will not be affected by the difference. But some action conflicts, such as the conflicts caused by resource competition,

should be adjusted to reflect the difference. In our implementation, we collect system information and adjust action conflict tables based on that information.

4 Using tree-merging to detect and resolve feature interactions

Because LESS has a tree-like structure, it would be straightforward to merge multiple LESS decision trees into one to resolve feature conflicts. After merging, there is only one active LESS script for an end device for each trigger. The merging algorithm holds for services running on the same device. However, for service scripts on different devices, such as the SUMC feature interactions discussed in Section 3.1.2, the merging algorithm can only detect feature interactions, it cannot resolve them. We will still keep the original scripts after merging so that users can modify them independent of the merged script. Although users can edit services based on a merged script, editing services based on the original scripts could be easier since they are created by the users themselves, while the merged script is created by a machine. In this way, no conflicts arise when we execute service scripts, and we can still keep service modularity so users can easily maintain their services and create new services efficiently. One potential problem in this approach is that modifications to the original scripts may interact with users' decisions for resolving earlier conflicts. This problem can be easily resolved by using the same algorithm to detect feature conflicts between the merged tree and the modified script.

4.1 Tree merging algorithm

```
<less> <incoming>
 <string-switch field="organization"> <string is="ABC Inc.">
  <address-switch field="origin"> <address is="sip:tom@abc.com">
   <string-switch field="subject"> <string is="group meeting">
    <accept/>
   </string> </string-switch>
  </address> <otherwise>
   <location url="sip:tom@vmail.abc.com"> <redirect/> </location>
  </otherwise> </address-switch>
 </string> </string-switch>
</incoming> </less>
```

Figure 7: Sample script for defining decision rules

In a LESS decision tree, the path from the root of the tree to each leaf node is called a decision rule. A rule is composed of a trigger, the actions in accordance with the trigger, and a list of switch nodes and action nodes along the path from root to action node. We call the list of switch nodes and action nodes a rule path. For example, for the script in Figure 7, a decision rule can be represented as
`{incoming, accept, {{string-switch,organization="ABC Inc."},`

{address-switch, origin="sip:tom@abc.com"}, {string-switch, subject="group meeting"}}}. A decision consists of the items in a rule path and the actions in accordance with the trigger. A rule path can be validated based on Nakamura's work [14]. Changing the orders of items in a valid rule path does not affect the decision.

To facilitate rule merging, we must normalize the rules generated from LESS decision trees. The normalization process sorts switches in a rule path, e.g., ordered as address-switch, time-switch, status-switch, string-switch, priority-switch, where-switch, language-switch, and event-switch. It also merges the switch nodes with the same switch name into one node in a rule path. For example, a normalized rule for the script above is {incoming, accept, {address-switch, origin="sip:tom@abc.com"}, {string-switch, subject="group meeting", organization="ABC Inc."}}. Because switches are independent of each other, normalized rules are functionally equal to the original rules. The overall multiscript merging process is shown in Figure 8.

```
set base-rule-set empty
foreach LESS-tree {
  convert the LESS-tree into a rule set
  foreach rule in the rule set {
    normalize the rule
  }
  merge the normalized rule set into base-rule-set
}
convert merged base-rule-set into a decision tree
```

Figure 8: Tree merging process

```
if (two rules have different triggers) {
  no rule conflict
} elseif (actions in two rules do not conflict) {
  no rule conflict
} elseif (no overlap between rule path in two rules) {
  no rule conflict
} else {
  two rules conflict,
  return the rule path overlap and action conflict information
  prompt to the script owner to judge
}
```

Figure 9: Checking two rules conflict or not

Figure 9 shows the conflict checking process of two rules. During the process, we can use the action conflict table in Section 3 to check whether two actions conflict. If the actions in two rules conflict, we must check whether any conditions match both rule paths. We call the conditions matching two or more rule paths an overlap. Figure 10

shows the algorithm for determining an overlap.

```
set overlap-set empty
foreach switch-node1 in rule-path1 {
  if (there is a switch-node2 in rule-path2
  that has the same switch name) {
    if (the overlap between switch-node1
    and switch-node2 is empty) {
      return empty overlap-set
    } else {
      insert the overlap into overlap-set
    }
  } else {
    insert switch-node1 into overlap-set
  }
}
foreach switch-node2 in rule-path2 {
  if (there is not a switch-node1 in rule-path1
  that has the same switch name) {
    insert switch-node2 into overlap-set
  }
}
return overlap-set
```

Figure 10: Determining overlap between two rule paths

Once we find the overlap and the conflicting actions, we can present the information to users to make decisions. It is straightforward to form a human-understandable description because LESS defines a very limited number of switches and actions. We can simply design the description format for each switch and action, and then compose a sentence based on the description format. In general, there is no need to use complicated natural language processing techniques to present the conflict information. For example, for the situation in Figure 1, we can ask the user "For an incoming call, if the time is between 2:00PM and 3:00PM on Dec 25, 2004, and if sip:tom@abc.com calls you, what would you like to do?". We can provide two choices for the user: *redirect the call to sip:conf@abc.com* or *automatically accept the call*. We can record the user's decision and build a normalized rule set without conflicts. Any resolution of feature interactions for end system services must involve end users because only they can make decisions about what they need. Because end systems can directly interact with end users, and end users can directly modify their scripts, involving end users in feature interaction resolution is practical and necessary.

After resolving feature interactions, the normalized rules should be converted back to a decision tree for service execution. The conversion is straightforward. We start by setting an empty tree, then go through every rule and incorporate the rule's switches into the tree. Finally, we put the rule's actions at appropriate branches of the tree.

4.2 Feature interactions caused by multiple triggers

In most cases, no feature interactions arise between decision trees having different triggers (root nodes). However, occasionally, actions caused by different triggers compete for resources. For example, a `timer` trigger may invoke a `call` action at the same time as an incoming call is automatically accepted. Both the outgoing call caused by the `timer` and the accepted incoming call will try to use the one audio device and cause a conflict. For an end system with limited resources, if multiple scripts have different triggers, we can use the tree-merging algorithm to detect possible resource competition. Figure 11 shows the algorithm.

```
for decision trees with different triggers {
  rename every trigger as 'common-trigger'
  store the original trigger information
}
perform regular tree-merging algorithm for 'common-trigger'
if (there is resource competition feature conflict) {
  present the conflict along with the original trigger information
  modify service scripts based on the decision of the user
}
```

Figure 11: Detecting feature interactions among the scripts with different triggers

5 Implementation

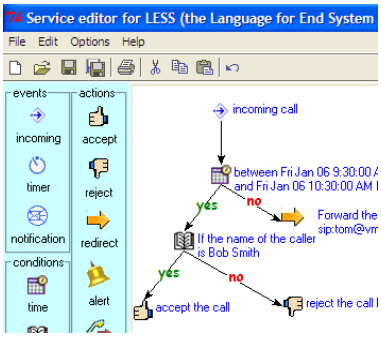


Figure 12: CUTE

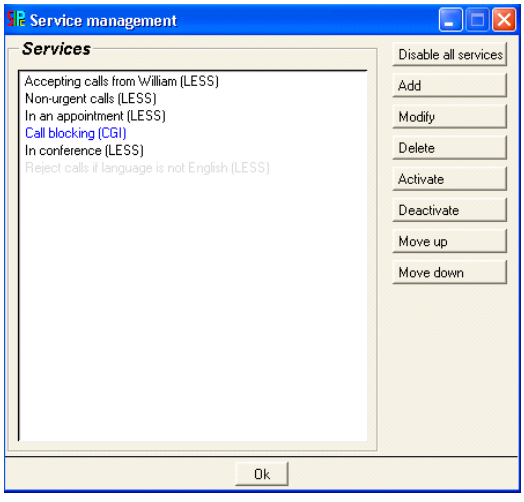


Figure 13: SIPC's service manager

We have implemented a SIP based user agent, SIPC, which supports a range of

media types such as audio, video, whiteboard and desktop sharing. In addition to multimedia communications, SIPC also supports the SIP event notification architecture [23] for presence notification, the SIP MESSAGE method for Instant Messaging [11], location sensing, the Service Location Protocol (SLP) [30] for networked resource detection, and the SIP DO [31] method for networked appliance control. SIPC can also invoke and control web browsers by using dynamic date exchange (DDE) to handle email and web browsing. With all these functions, SIPC can perform all the services mentioned earlier.

SIPC has two interfaces for service programming, a SIP common gateway interface (SIP CGI) [32], and a built-in LESS interpreter to handle LESS service scripts. The SIPC service manager cannot detect feature interactions between SIP CGI programs and LESS scripts because SIP CGI programs can be written in a variety of programming languages, such as C/C++, Perl, and Java. It can, however, detect and help resolve feature interactions among LESS scripts. We built a service creation tool and integrated it into SIPC. The tool is called *CUTE*, which stands for *Columbia University Telecommunication service Editor*. Figure 12 shows the CUTE user interface. A user can drag triggers, switches, and actions to the CUTE drawing panel and build LESS decision trees. Once a LESS service script is created, SIPC can show the script in its service manager, as shown in Figure 13, and use it to handle users' communication events. A user can activate or deactivate services in the SIPC service manager. If there is more than one active LESS script and they are in conflict, the service manager can use the algorithm in this paper to detect the interactions and ask the user to make a decision.

Figure 14 shows the user interface for resolving a feature conflict. As the figure shows, the SIPC service manager translates the context that can cause feature conflicts into human-readable language, then asks users to choose an action to perform. The translation is made straightforward by using the trigger and rule-path overlap information of two services. Once a user makes a decision, the SIPC service manager tries to prioritize services to resolve the conflict because the prioritization is easy for people to understand.

As Figure 13 shows, services are listed in order with the services listed first executed first. Figure 14 shows that if a user chooses to `reject` a call according to the *In conference* service, the *In conference* service is listed at the top of the *Accepting calls from William* service. Users can also manually change the order of services by clicking the `Move up` or `Move down` button. This is similar to the method defined in FIAT[13], but we do not explicitly ask users to set priority among or disable features. If all the branches of a feature are redundant or shadowed [13] by features having higher priority, the feature will be automatically disabled. Figure 13 shows the disabled feature in light gray.

Compared with merging services, prioritizing services can help improve the feature conflict resolution. In general, end users are not expected to create a very complicated service manually. We assume that the depth of LESS decision trees created by end users will be $<= 10$. In our performance test, which was run on a modest PC (2.0 GHz AuthenticAMD processor, 1.0 GB memory, running Windows XP Professional), with a modest load (around 50 percent CPU usage by a load generator), and using Tcl (an interpreted language) to implement our feature interaction detection algorithm, the longest delay we observed for detecting feature conflicts between two LESS decision

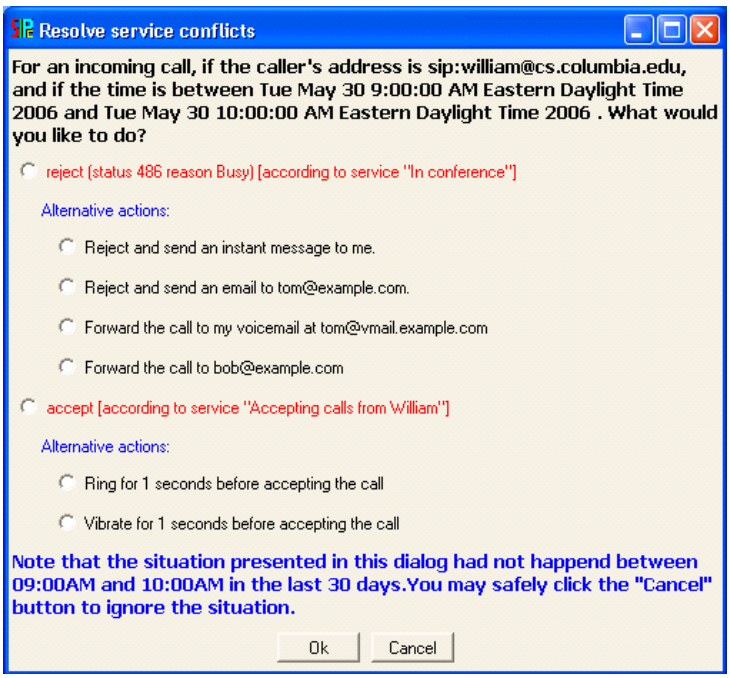


Figure 14: Resolve feature interactions

trees, each with a tree depth of 10, was 2.498ms. However, after merging, in the worst case the depth of a merged tree can be the summation of the depth of all the trees joining the merge. The outcome can cause longer delays in calculating feature conflicts. Thus, the SIPC service manager always tries to prioritize services to resolve feature conflicts.

However, as we discussed in Section 1.1, some conflicts cannot be resolved by prioritizing services, especially when three or more services are in conflict. For example, a user has three features in conflict: `Fa`, `Fb`, and `Fc`. Because the conflicts may be on different branches of the decision trees of the features, the user's choices may cause `Fa>Fb`, `Fb>Fc`, and `Fc>Fa` (here, `Fa>Fb` means feature `Fa` has higher priority than feature `Fb`). We cannot prioritize `Fa`, `Fb`, and `Fc` here to resolve the feature interactions. In fact, the orders of features implies a directed graph. If the graph contains a cycle, we cannot prioritize features to resolve the feature interactions. Instead, we must merge the features in the cycle into one by using the tree-merging algorithm. Figure 15 shows the algorithm to find the cycle.

The SIPC service manager does not list the merged services, instead, it still lists the original scripts but groups them in the list. Each original script has a reference to the merged script. And every action in the merged script also has a reference to its original script. This way, users can still edit their original scripts but use the merged script for call handling.

The options provided by the SIPC service manager are not limited to the services defined by the user. Integrating our service risk management implementation[33] into

```
// For a feature 'F', we use 'F.higher' to represent the set of
// features that have higher priority than 'F', and 'F.lower' for
// the features with lower priority than 'F'.
for every two features Fa and Fb in a user's feature set {
  if (Fa has higher priority than Fb) {
    if (Fa.higher contains Fb) {
      there is a cycle
      use DFS (Depth-First Search) to find out the cycle and merge
    } else {
      add Fa into Fb.higher
      foreach feature Fx in Fb.lower, add Fa into Fx.higher
      add Fb into Fa.lower
      foreach feature Fy in Fa.higher set, add Fb into Fy.lower
    }
  }
}
```

Figure 15: Finding the scripts to merge

the SIPC service manager offers users additional options. These options gives users a chance to choose a reduced-risk action to handle a conflict. For example, as Figure 14 shows, the services in red are defined by the user. For each user-defined option, there are several alternatives within that option. For example, a conference attendee has an incoming call from `sip:william@cs.columbia.edu`, but the user does not want to answer the call during a conference session. However, the attendee also does not want to risk losing important calls from William by simply rejecting the call. The additional options allow the user to transfer the call to a contact person, or to reduce risk by using another communication method such as email or instant messaging. Figure 13 shows the alternative options for rejecting and accepting calls. The additional contact information can be configured in the configuration dialog of the SIPC service manager, as shown in Figure 16. If a user chooses an alternative option, he or she, in fact creates a new service. The new service is listed in the SIPC service manager, and receives higher priority than the two existing services.

The configuration dialog also shows system properties. We use the `systeminfo` command to obtain CPU and memory information, the `ping` command to obtain approximate bandwidth information, the `ICreateDevEnum` interfaces to obtain the number of available video capture devices, and the `waveInGetNumDevs()` and `waveOutGetNumDevs()` functions to obtain the number of available audio devices. From this information, the SIPC service manager calculates the maximum audio and video sessions the system can support and adjusts the conflicts that may cause resource competition shown in Tables 7, 10, and 11 to reflect the end system's capabilities. Those capabilities are also used to handle feature interactions between a caller's preferences and a callee's capabilities, and are sent to other parties by using SIP NOTIFY method.

If a user does not want to handle the detected feature conflicts, he or she can simply click the `Cancel` button shown in Figure 14 to tolerate the conflicts. This action keeps

Figure 16: Configuration dialog

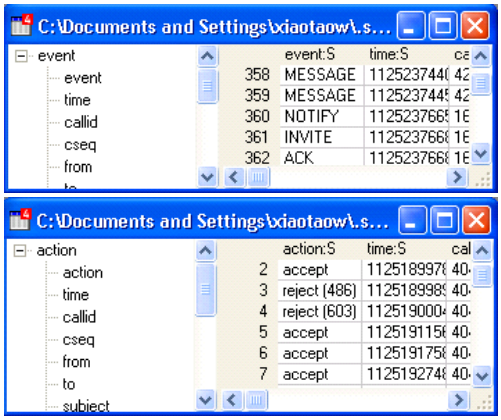


Figure 17: Events and actions log

the current order of the features unchanged.

To make feature interaction resolution transparent to users, we are integrating our service learning implementation [33] into the feature interaction handling process. By learning from users' call histories, we can infer users' preferences for handling feature conflicts and suggest a default choice for each user. In our current implementation, we check the probability of the occurrence of a feature conflict from users' event logs and action logs as shown in Figure 17. The SIPC service manager can offer suggestions to users based on probabilities. Currently, as a default setting, the SIPC service manager checks events for the last 30 days. If a situation that caused a feature conflict did not occur within the last 30 days, the SIPC service manager suggests that the user ignore the feature conflict. Otherwise, it tells the user the number of times the situation occurred in the last 30 days and suggests that the user choose an option instead of clicking the `Cancel` button. As shown at the bottom of Figure 14, the service manager finds out that `sip:william@cs.columbia.edu` did not call the user between 09:00AM and 10:00AM in the last 30 days, the service manager then infers that `sip:william@cs.columbia.edu` may not call during the same time period in the future and suggests that the user ignore the feature interaction.

6 Conclusion and future work

This paper introduces the Language for End System Services (LESS) which we defined specifically for VoIP end system communication service creation. One LESS design goal is to make it easy to detect feature interactions among LESS scripts. We propose a tree-merging algorithm to handle LESS feature interactions based on the action conflict tables presented in this paper. We investigated a variety feature interactions among LESS scripts and showed that our method can easily handle feature interaction detection and resolution for a language, such as LESS and CPL, having a tree-like structure.

We have developed a LESS-based service creation and management environment with our feature interaction handling method built-in. We then integrated this service creation and management environment into our SIP user agent, SIPC. We also did some preliminary work on building a user-friendly interface to help end users better understand feature interactions and resolve any interactions they encountered. One solution we are working on is to integrate our service learning implementation into the feature interaction handling process. By service learning, we may make the feature interaction resolution transparent to users in some situations, although we believe that in many cases, feature interaction handling for end system services still requires involving the end users.

7 Acknowledgements

The work is supported by a grant from FirstHand Technologies.

References

[1] Xiaotao Wu and Henning Schulzrinne. LESS: language for end system services in Internet telephony. Internet Draft draft-wu-iptel-less, IETF, February 2005. Work in progress.

[2] Xiaotao Wu and Henning Schulzrinne. Programmable end system services using SIP. In *Conference Record of the International Conference on Communications (ICC)*, pages 789–793, May 2003.

[3] Jonathan Lennox, Xiaotao Wu, and Henning Schulzrinne. Call processing language (CPL): a language for user control of Internet telephony services. RFC 3880, Internet Engineering Task Force, October 2004.

[4] Xiaotao Wu and Henning Schulzrinne. End system service examples. Technical Report CUCS-048-04, Columbia University Department of Computer Science, New York, New York, December 2004.

[5] AT&T. 5ESS switch, the premier solution, feature handbook, issue 4, September 1987.

[6] International Telecommunication Union. General recommendations on telephone switching and signaling – intelligent network: Introduction to intelligent network capability set 1. Recommendation Q.1211, International Telecommunication Union, Geneva, Switzerland, March 1993.

[7] Ecma International. Services for computer supported telecommunications applications (CSTA) Phase III. Standard 269, Ecma International, June 2004.

[8] Pamela Zave. An experiment in feature engineering. In *Programming Methodology*, February 2003.

[9] J. Rosenberg, Henning Schulzrinne, G. Camarillo, A. R. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: session initiation protocol. RFC 3261, Internet Engineering Task Force, June 2002.

[10] J. Rosenberg. A presence event package for the session initiation protocol (SIP). RFC 3856, Internet Engineering Task Force, August 2004.

[11] Session initiation protocol (SIP) extension for instant messaging. RFC 3428, Internet Engineering Task Force, December 2002.

[12] Yiqun Xu, Luigi Logrippo, and Jacques Sincennes. Detecting feature interactions in CPL. *Journal of Network and Computer Applications*, article in press, December 2005.

[13] Daniel Amyot, Tom Gray, Ramiro Liscano, Luigi Logrippo, and Jacques Sincennes. Interactive conflict detection and resolution for personalized features. *Journal of Communications and Networks*, 7(3), September 2005.

[14] Masahide Nakamura, Pattara Leelaprute, Ken ichi Matsumoto, and Tohru Kikuno. On detecting feature interactions in the programmable service environment of internet telephony. *Computer Networks*, 45(5):605–624, 2004.

[15] Petre Dini, Alexander Clemm, Tom Gray, Fuchun Joseph Lin, Luigi Logrippo, Stephan Reiff-Marganiec, and Kenneth J. Turner. Policy-enabled mechanisms for feature interactions: reality, expectations, challenges. *Computer Networks*, 45(5):585–603, 2004.

[16] T. Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[17] Nicolas Gorse, Luigi Logrippo, and Jacques Sincennes. Formal detection of feature interactions with logic programming and LOTOS. *Journal of Software and Systems Modeling*, article in press, December 2005.

[18] David Garlan and Mary Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, pages 1–39. World Scientific Publishing, 1993.

[19] Michael Jackson and Pamela Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, August 1998.

[20] N. Griffeth and Hugo Velthuijsen. The negotiating agents approach to runtime feature interaction resolution. *Feature Interactions in Telecommunications Systems, IOS Press*, pages 217–235, 1994.

[21] Xiaotao Wu and Henning Schulzrinne. Handling feature interactions in the Language for End System Services. In *International Conference on Feature Interactions in Telecommunications and Software Systems (ICFI)*, June 2005.

[22] E. J. Cameron, N. Griffeth, Y. Lin, Margaret E. Nilson, William K. Schure, and Hugo Velthuijsen. A feature interaction benchmark for IN and beyond. In *Feature Interactions in Telecommunications Systems*, pages 1–23, Amsterdam, Netherlands, 1994.

[23] Adam Roach. Session initiation protocol (SIP)-specific event notification. RFC 3265, Internet Engineering Task Force, June 2002.

[24] Aki Niemi. Session initiation protocol (SIP) extension for event state publication. RFC 3903, Internet Engineering Task Force, October 2004.

[25] Jonathan Rosenberg. The extensible markup language (XML) configuration access protocol (XCAP). Internet Draft draft-ietf-simple-xcap-08, Internet Engineering Task Force, October 2005.

[26] J. Rosenberg. A watcher information event template-package for the session initiation protocol (SIP). RFC 3857, Internet Engineering Task Force, August 2004.

[27] X. Wu and Henning Schulzrinne. Use SIP MESSAGE method for shared web browsing. Internet draft, Internet Engineering Task Force, November 2001. Expired.

[28] Mario Kolberg, Evan H. Magill, and Michael Wilson. Compatibility issues between services supporting networked appliances. In *IEEE Communications Magazine*, November 2003.

[29] Jonathan Rosenberg, Henning Schulzrinne, and Paul Kyzivat. Caller preferences for the session initiation protocol (SIP). RFC 3841, Internet Engineering Task Force, August 2004.

[30] E. Guttman, C. E. Perkins, J. Veizades, and M. Day. Service location protocol, version 2. RFC 2608, Internet Engineering Task Force, June 1999.

[31] Stan Moyer, Dave Marples, and Simon Tsang. A protocol for wide-area secure networked appliance communication. *IEEE Communications Magazine*, 39(10):52–59, October 2001.

[32] J. Lennox, Henning Schulzrinne, and J. Rosenberg. Common gateway interface for SIP. RFC 3050, Internet Engineering Task Force, January 2001.

[33] Xiaotao Wu and Henning Schulzrinne. Service learning and service risk management in Internet telephony. In *Conference Record of the International Conference on Communications (ICC)*, IEEE, May 2005.