Interactive Conflict Detection and Resolution for Personalized Features

Daniel Amyot, Tom Gray, Ramiro Liscano, Luigi Logrippo, and Jacques Sincennes

Abstract: **In future telecommunications systems, behaviour will be defined by inexperienced users for many different purposes, often by specifying requirements in the form of policies. The call processing language (CPL) was developed by the IETF in order to make it possible to define telephony policies in an Internet telephony environment. However, user-defined policies can hide inconsistencies or feature interactions. In this paper, a method and a tool are proposed to flag inconsistencies in a set of policies and to assist the user in correcting them. These policies can be defined by the user in a user-friendly language or derived automatically from a CPL script. The approach builds on a pre-existing logic programming tool that is able to identify inconsistencies in feature definitions. Our new tool is capable of explaining in user-oriented terminology the inconsistencies flagged, to suggest possible solutions, and to implement the chosen solution. It is sensitive to the types of features and interactions that will be created by naive users. This tool is also capable of assembling a set of individual policies specified in a user-friendly manner into a single CPL script in an appropriate priority order for execution by telecommunication systems.**

Index Terms: **Call processing language, features, interactions, Internet telephony, personalization, policies, services.**

I. INTRODUCTION

Telecommunications systems offer many features and services to users. *Features* are capabilities offered to users to manage the handling of their calls. Well-known examples of features are call forwarding, call screening, and call waiting. *Services* are marketable entities that bundle features together for offer to the user. For example, a call forwarding service might include call forward unconditional, call forward no answer and call forward on busy. Commercially available private branch exchange (PBSx) offer hundreds of such features.

In general, features can be parameterized to give the user control over how they operate. Users for example are allowed to set the phone number to which their calls are forwarded in call forwarding features. Recent technological developments are now offering users much greater programmability of their features [1]–[3]. Services can be offered in which users may specify *policies* that indicate how incoming and outgoing calls should be handled, according to certain conditions. These conditions can classify calls on the basis of the users' social and business environment. For example, calls could be classified on the basis of the business relationship with the caller (VIP customer, boss, friend, subordinate, etc.), the user's current location (out of the office, in the office, in the car, etc.), her current activity (in a meeting, working alone, etc.), etc. Users may then specify how each call category should be treated (reject, forward to voice mail, notify user and ask caller to wait, etc.). In other words, users will be given the capability of personalizing the handling of their calls. Services will be adapted to the environment of users and will function to their specific needs.

This abundance of possibilities will carry risks, however. Multiple personalized features may interact in ways that will cause them to defeat each other or create system malfunctions (unexpected or unpredictable behaviour, loss of connection, etc.). This is the *feature interaction* problem, which has been identified as a research problem in its own merit and has generated some amount of literature [4], [5]. If care is not taken, the combination of features may cause unwanted results in features that work flawlessly in isolation.

Feature interactions were under control in traditional switching, because the creation and combination of features were limited and in the hands of qualified designers. Personalized policies were nearly non-existent. New features were evaluated extensively in the design process by experts and tested in conjunction with all other features in order to prevent undesirable interactions from reaching the end user. In the new world of personalized features, end users will design their own features and policies, possibly many of them, to cover many different types of operations and applications. These users may have a clear idea of how each feature or policy will behave on its own, but will be puzzled by the fact that they won't work as expected in conjunction. It is common for people to specify features that are on the surface inconsistent. For example, an executive may tell her secretary that no calls should be put through on Tuesday afternoons and also tell her that if an important customer named Terry March calls to put him straight through. What happens if Terry March calls a Tuesday afternoon? Whatever decision the secretary might take, the boss may be surprised. It may be easy to see a conflict if only two policies are specified, but surely detection will be more complex in the case of many policies specified at different times. Means must be provided to bring such conflicts to the attention of ordinary users, while at the same time suggesting possible resolutions.

In this paper, a method and a tool are proposed to flag inconsistencies in a set of personalized features, to explain them with user-oriented terminology, to suggest possible solutions, and to implement the chosen solution. Section II introduces basic concepts and a pre-existing feature interaction detection tool used in our approach. High-level system goals and functionalities are presented in Section III. The main elements and function-

Manuscript received January 6, 2004; approved for publication by Hyeong Ho Lee, Division III Editor, July 13, 2005.

D. Amyot, R. Liscano, and J. Sincennes are with the School of Information Technology and Engineering, University of Ottawa, Canada, email: {damyot, rliscano, jack}@site.uottawa.ca.

T. Gray is with PineTel, Canada, email: digroup_codec@direcway.com.

L. Logrippo is with the Déartement d'informatique et ingéierie, Universitédu Québec en Outaouais, Canada, email: luigi@uqo.ca.

alities of our system are detailed and illustrated in Sections IV–VII. Related work and potential extensions are discussed in Section VIII, followed by our conclusions.

II. BACKGROUND CONCEPTS

This section introduces concepts (feature interactions), a language (CPL), and a relevant pre-existing tool for policy-based call control and feature interaction detection (FIAT).

A. Feature Interactions and Policy Conflicts

Traditionally, communication services have been created by service providers. This is the approach used in the intelligent network [6]. Engineers have two choices for detecting and resolving conflicts or undesirable interactions [5].

- *At design time (offline)*: When different features (often created independently) are integrated to the system, before it is made available to the end-user. Techniques such as testing and proofs based on formal models can be applied at the requirements, design, and code level to detect the interactions; feature precedence or tighter feature integration (where features become aware of each other) can be used to solve them.
- *At run time (online)*: When not all conflicts or undesirable interactions can be eliminated at design time (e.g., call forwarding loops), service providers rely on run-time mechanisms that monitor the system and react to detect and resolve problems.

In both cases, the resolution is the same for all users, whatever their preferences, intentions, context, or understanding of the features.

New types of communication and collaboration services and applications, enabled by recent IP telephony protocols and languages such as IETF's session initiation protocol (SIP) [7] and call processing language (CPL) [8], are emerging. These features will benefit from the availability of various sources of contextual information (e.g., user's location, presence, schedule, relationships, preferences, etc.) to satisfy a wide variety of requirements [1], [3]. End-users will have the capability of creating their own features through policies, and they should be involved in defining their own solutions to resolving conflicts among these policies. If an increased number of undesirable interactions were anticipated because even amateur programmers can implement Internet telephony features [9], the situation could deteriorate rapidly without tools to guide *naïve* end-users (non-programmers) into creating their own features. We can distinguish two main situations.

- *Conflicts among policies of a single user*: Such conflicts can be detected at design time, when a user inputs policies in the system. Since the system and the user have the knowledge of all the policies of the user, conflicts can be solved according to user intentions, rather than with fixed mechanisms as in traditional systems.
- *Conflicts among policies of multiple users*: Since there can be a large number of users, each with their own customized set of policies, the set of potential conflicts can be unmanageable. But each one of these potential conflicts will cause difficulties only if the users that have conflicting policies actually become involved in the same call. The method and tool discussed in this paper do not cover this type of conflict.

Policy conflict detection is present in various domains, principally in *network management*. Much of the existing work in that domain is based on concepts developed by Sloman and Moffett [10]. In particular, Thebaut *et al.* [11] have embedded in their system conflict resolution approach rules that are triggered at run time. Lupu and Sloman presented several challenges related to policy conflicts in distributed systems [12]. Fu *et al.* tailored Sloman's work to detect and solve IPSec policy conflicts, at run-time, while trying to comply with security requirements [13]. However, as noted in [14], techniques developed for network management are not necessarily well tailored to personalized communication services.

B. Call Processing Language (CPL)

CPL is a language that can be used to describe and control Internet telephony services [8]. CPL uses the extensible markup language (XML) [15] to describe personalized features in terms of policies applicable to incoming and outgoing calls. CPL is powerful enough to describe a large number of services and features, but is limited in power so that untrusted users may develop services that can run safely in Internet telephony servers. Notably, the language provides no way to define loops or recursive scripts, and therefore it is not Turing-complete.

CPL scripts can be created by end-users to proxy, redirect, or reject calls. Policies for one user can be combined into a single CPL script. Such a script expresses a decision tree where the conditions are based on different types of *switches*, which group conditions based on addresses, call priority, time, languages, and any string found in the header of a request. There are two decision trees: One for incoming calls and one for outgoing calls.

Since they order the feature preconditions, CPL decision trees provide an implicit and absolute ordering of priorities among the features. This prevents indeterminacy among features with overlapping preconditions. However this comes with the price of requiring that the user decide the ordering of feature priorities on her own. To do this she must understand all of the implications that any chosen priority ordering will have on the execution of her features. She is required to detect and understand the effect of ordering on the interactions among features.

CPL, and variants such as the language for end system services (LESS, [16]), have no mechanism of their own to detect feature interactions [17]. A CPL based telecommunication system just traverses the user's CPL decision tree and triggers a feature it encounters if all preconditions are met. Therefore, CPL uses a totally ordered priority system for resolution. This is sufficient in an execution environment, but is inadequate in a design environment because individual policies cannot be distinguished in CPL scripts and hence their management becomes problematic. A small change to one individual policy could require a complete restructuring of the CPL script taking into consideration all of a user's individual policies.

C. Policy-Based Call Control

Call control engines in IP-based PBXs, switches, gateways, and proxy servers can make use of *policies* to implement, control or restrict various types of communication services (e.g.,

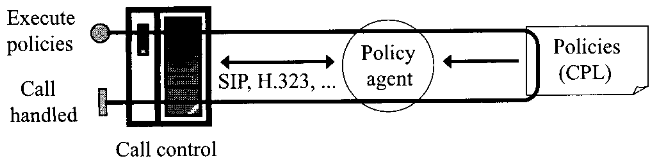


Fig. 1. Policy execution in call control.

VOCAL[1]). In a sense, policies can be *executed* by the call control engine. For instance, a generic policy-based architecture for executing policies is presented in [3].

The use case map[2] scenario in Fig. 1 illustrates a simple situation where a conventional call control engine asks its *policy agent* what to do upon receiving or prior to sending a communication by some user (e.g., a telephone call). The policy agent accesses the relevant policies (described as CPL scripts or in another representation) and executes them by telling the call control what to do with that communication (e.g., redirect the call, reject the call, log the call, etc.). The protocol used between the call control engine and the policy agent is outside the scope of this paper, being understood that SIP, H.323, or proprietary signalling protocols can be used.

A thesis [18] and a paper [19] propose a logic representation of features and a tool called FIAT (feature interaction analysis tool) for "filtering" (i.e., detecting the possibility of) inconsistencies among these features, at the design stage. This representation is generic enough to support personalized communication policies and we have used the FIAT tool in this context to detect policy conflicts.

In FIAT, a feature is described as a four-part tuple, using a prolog-based syntax [20].

- *Preconditions*: Mandatory conditions or system state under which the feature is activated (e.g., this policy is active from Monday through Friday only).
- *Triggering events*: Action(s) triggering the feature (e.g., there is an incoming call from Bob).
- *Results*: Represent the actions produced by execution of the feature and the state in which the system is after such execution (e.g., forward the call to the voice mail system).
- *Constraints*: Restrictions relative to the possible values of the variables used in the preconditions, triggering events, and results describing this feature (e.g., originator is different from terminator).

The vocabulary used for conditions and actions is not predetermined and is hence adaptable to various contexts and domains such as telephony, network management, or presence and availability.

FIAT is a Prolog program that takes pairs of feature descriptions as input and uses *filtering rules* to detect *inconsistencies* (also called *incoherencies* in [18] and [19]). Three main filtering rules are currently part of the tool. These rules have sub-cases (not described in detail here), and other rules could easily be added.

- Two features can be triggered by the same event with the same preconditions and yield different and possibly contradictory results (non-determinism). A classical example is the case of two concurrent features reacting in different ways to the same signal. For instance, for the conventional features *incoming call screening* and *call forwarding*, if a call comes in from a subscriber in the screening list, should it be forwarded or simply blocked?
- The results of a feature trigger another feature and the results of the two features present a contradiction (transitive inconsistency). A classical example of this case is *originating call screening*, after which *call forwarding* is possible, leading to the possibility that a subscriber gets connected to another subscriber in the first subscriber's screening list.
- The results of a feature trigger another feature and vice-versa (feature loop). The typical example here is a *call forwarding* loop.

FIAT makes use of Prolog's backtracking and unification mechanisms to catch inconsistencies under these criteria. A finite set of constants representing users is considered, and for each possible binding of such constants to the variables in pairs of feature descriptions, it is checked if one of the filtering rules above is satisfied. If this happens, then there is an inconsistency and a potential interaction and this fact is reported in a helpful plain language diagnostic message. Note that contradictions can be explicitly defined by the user of the tool, between user-defined terms: For instance, *busy*(*A*) can be defined as being in contradiction with *idle*(*A*) as *contradiction_pair*(*busy*(*A*),*idle*(*A*)), where busy and idle are user-defined terms, and *A* is a common parameter.

For each inconsistency detected, FIAT can generate an example of a specific situation or scenario that can lead to an undesirable interaction. These scenarios can be used as user information to better illustrate the problem, or be converted to test cases. FIAT reports these in a text-based output. Note that the presence of an inconsistency does not necessarily imply an interaction, but enough information is given to the user to determine if an interaction exists.

Finally, FIAT supports a database of previously reported inconsistencies. This database can be used to restrict reported inconsistencies to those not discovered before. This database can be adapted to hide problems (e.g., policy conflicts) that the user would have been told about already and chose to tolerate.

It should be noted that the general problem of detecting all inconsistencies in a set of features (e.g., a CPL script) is of exponential complexity, because in principle it involves deciding the truth of arbitrarily complex propositional logic formulae. Therefore, in practice heuristics must be used. It has been shown in [18] and [19] that the rules of FIAT, which are of polynomial complexity only, are powerful enough to flag a very good number of well-known interactions in conventional telephony. The polynomial bound is due to the fact that only finite (in fact, usually small) sets of variables and values need to be considered. As well, this analysis has been found to be adequate for the real-life features that were taken into consideration in the case studies for this paper.

[1]Vovida Open Communication Application Library, available at http://www.vovida.org/applications/downloads/vocal/.

[2]A UCM scenario start with a filled circle (triggering event), progresses along the path (superimposed on entities and components), and terminates with a bar (result). Such scenarios are independent of the types of messages exchanged between the components. See http://www.UseCaseMaps.org.

III. GOALS AND SYSTEM OVERVIEW

This section enumerates the goals of our system and then gives an overview of its main functionalities.

A. System Goals

In order for a system to allow users to define personalized communication policies, to detect and report undesirable interactions in a comprehensible way, and to allow users to resolve them with minimum effort, we have identified several goals and requirements, which we have grouped under five categories.

Policy management

G1. To provide a user-friendly policy specification environment for personalized policies. This must include means to create, modify, delete, activate, deactivate, and prioritize policies. This can be done by using well-known web interfaces (Section IV) and could be enhanced to voice interfaces (Section VIII-B).

G2. To support several levels of policies (e.g., user, group, and enterprise policies) and check for consistency between these levels. This has not yet been done in our implementation and remains a goal for further study (Section VIII-B).

Individual and integrated policy descriptions

G3. Policies for individual features defined by users may be unsuitable for execution. There is a need to go from user-suitable individual policies to an execution-suitable integrated version of these policies (Section VII).

Conflict detection

G4. In principle, conflicts between policies of a single user are impossible in a sequential language such as CPL. This is because in each case the list of policies is examined top-down and the first applicable policy is executed (see Section II-B). Later policies applicable for the same situation are ignored. However the coexistence of simultaneously applicable policies in a set of policies should be addressed (Section V). At best, it could indicate poorly maintained policies. At worst, it could indicate ambiguously specified user intentions, of which the user should be informed.

Reporting and suggestions

G5. To report inconsistencies at the user-level (natural) language used to define the policies involved. This requires the translation of the reporting obtained from tools for inconsistency identification such as FIAT into a more user-friendly reporting. It also requires the illustration of inconsistencies by providing examples of problematic situations. Each inconsistency should be classified based on its characteristics and reported (for example) as conflict, shadowing, redundancy, or specialization (Sections VI-A to VI-D).

G6. The reporting should be able to be modulated according to levels of sensitivity defined by the user. For instance, the user may wish not to be informed about shadowing in certain situations (Section VI-E).

G7. To provide suggestions on how to resolve the conflicts detected. These suggestions need to be adapted to the nature of the policies involved as well as to the type of conflict. Suggestions could include editing a policy, disabling a policy, setting priorities between policies, adding exceptions, tolerating the conflict by allowing the system to resolve it as reported (Sections VI-A and VI-D).

Support for conflict resolution

G8. Conventional approaches often implement one resolution mechanism (at design time or at run time) for all users, but this is not acceptable in a context where users have the opportunity to create their own features. Resolution mechanisms must be allowed to vary from user to user (Section VI-F).

G9. Suggesting solutions solves only half of the problem. These solutions need to be supported by user-friendly means to implement them (e.g., a simple click of a mouse in a multiple-choice box). The user must be able to escape the suggested solution and implement her own (Section VI-F).

References to these goals will be included where relevant mechanisms are discussed.

B. System Overview

Overview of the creation validation process

Following is an overview of the policy creation and validation process supported by our system, which will be further explained in the rest of the paper.

1. Enter user policies described in a user-friendly manner (for instance, using a web browser).
2. Translate these policies into an executable feature language such as CPL.
3. Prompt the user with an option to validate the overall policy set before it is uploaded to the execution system.
4. If validation is selected, translate the policies from CPL into the FIAT format for which it is possible to detect common feature specification errors. Otherwise, go to step 8.
5. Take the errors detected by FIAT and interpret them with awareness of the expectations and common errors of naïve users.
6. Report these errors to the user (e.g., via a web interface) in terms which are understandable to naïve users and consistent with the way the policies were originally described.
7. Provide the user with options to either accept the interactions as they are, to repair them manually or to accept a recommendation of automatic correction. Unlike conventional systems, where feature interactions are solved in the same way for all users, the selected resolution is personalized to satisfy the user's intentions, independently of how others may solve similar conflicts.
8. Upload the set of policies to the execution system.

Fig. 2 illustrates the general system flows of our process. It supplements existing policy-based call control (introduced in Fig. 1) by providing support for policy management, conflict detection, and conflict resolution.

Policy management

web-based policy management enables the creation, modification, deletion, activation, deactivation, and prioritization of policies. A policy management interface is presented to the end-user through a web browser (e.g., Netscape, Internet Explorer, Opera, etc.). To a large extent, this makes the access to the policy management system device independent.

web-based interfaces are easily adaptable to home or work environments (e.g., hospitality, medical, engineering, legal, etc.) and can communicate with the end users using a language and

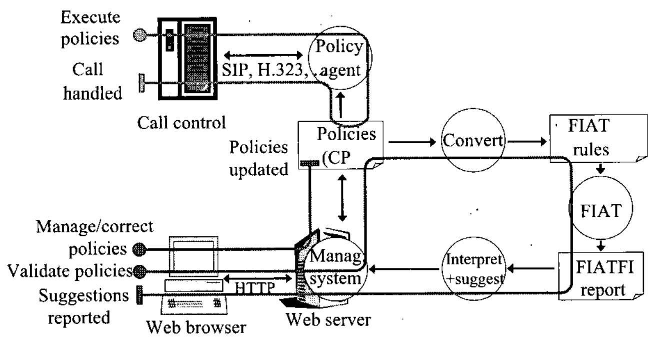


Fig. 2. System overview.

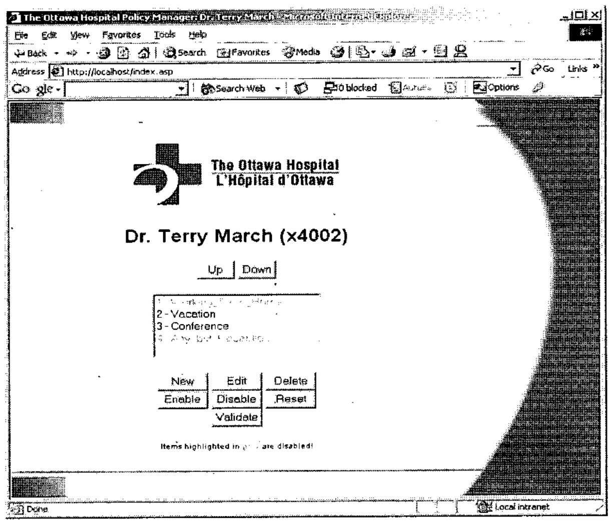


Fig. 3. Example of web interface for policy management.

a terminology they are familiar with. Policies can be described interactively in these terms, and translated to a particular representation (e.g., CPL) by a conversion application running on the web server (as shown by the *manage policies* scenario in Fig. 2). The web-based interface can represent individual policies internally (e.g., in proprietary format or in a database) or using a standard representation such as CPL. Through the same interface, a single script (e.g., in CPL) that integrates all the activated policies that belong to a user can be synthesized. This integrated script is used at run time by the policy-based call control engine (as shown by the *execute policies* in the scenario of Figs. 1 and 2). Therefore, the user never needs to directly see CPL code.

Validation and conflict reporting

The *validate policies* scenario in Fig. 2 adds a policy conflict detection functionality to the policy management system. Upon a request (e.g., by clicking a validate button on the web interface), a user can ask the system to detect potential problems in her list of activated policies. To enable this, the policies are converted to a formal representation suitable for analysis, such as FIAT rules. Once translated, individual policies can be processed by the FIAT tool, which reports various types of conflicts and examples as described in Section II-C. The conflicts need to be interpreted and reported in a way understandable to the user, using the language and concepts she used to create the policies initially. This conversion also produces HTML code [21] for the web-based policy management interface.

In addition to conflict reporting, suggestions on how to resolve these conflicts are generated and reported as well (Section VI-A).

Automated support for policy corrections

The conflict resolution suggestions are reported to the user in a hyperlinked list in the policy management interface (Section VI-A). Selecting a suggestion (see the *correct policies* scenario in Fig. 2) activates the proper sequence of requests to the web application (on the web server) required to repair the problem according to the option chosen (Sections VI-B and VI-F). The individual and integrated policies are then generated again (Section VII).

IV. POLICY CREATION AND MANAGEMENT SYSTEM

The system of Fig. 2 provides a mechanism for the creation, management, testing, and provisioning of policies. This section presents the details of our system's interface and operations, as well as of the structure of the users' policies.

A. Interface and Operations

The web interface enables a user to manage her list of policies (e.g., in a list box or in a different panel or frame), typically sorted by name or by priority. Fig. 3 presents one such interface, which was used as a prototype for doctors, nurses, and other employees in a hospital (our ongoing example). The policies are sorted according to their priority.

The following operations on policies are supported (via buttons, links, menus, or possibly voice activation) in accordance with goal **G1**.

- *Create*: Create a new policy, add it to the list, and activate it.
- *Modify*: Modify the selected policy.
- *Delete*: Delete the selected policy from the list.
- *Duplicate*: Make a copy of the selected policy (with the intention of modifying it later).
- *Deactivate*: Deactivate the selected policy. Policies that are inactive are still kept on the list but are marked as such (e.g., different color, shadowed text, or special icon). They are not used for validation or execution, but they are kept for future re-activation.
- *Activate*: Activate the selected inactive policy.
- *Set priority*: Set the priority of the selected policy. This could be an absolute priority or a relative one (e.g., move the policy higher or lower in the list when sorted by priorities).
- *Validate*: Detect and report conflicts in the list of active policies.
- *Approve*: Approve the current list of policies and enable them for execution (e.g., through the generation of a single CPL script uploaded to the call control switch or to the policy agent).

B. Policy Structure

Policies can be created using structured text, obtained in various ways (free form, lists, pop-up menus, etc.) from the user. A policy is composed of seven elements.

- A *name*, used as unique identifier.
- A *priority*, expressed as a numerical value. The lowest numerical value is associated with the highest priority.
- The *operation*, which is typically one of forward an incoming call, reject an incoming call, or block an outgoing call.
- A *precondition*, based on the characteristics of the caller or callee (e.g., phone number, role, domain, location, organization/business, name, device, presence information, etc.). Many characteristics can be combined in a logical expression.
- The operation *target* (for forward operations), typically a phone number, a person, a role, a device, voice mail, etc.
- An optional list of *exceptions* to a general precondition, based on the same types of characteristics.
- A *time constraint*, where the policy is active. This can be a specific time interval (e.g., from 1:00 to 2:00) or recurring intervals (e.g., every Tuesdays). "Forever" can be used to specify the absence of time constraints.

The web interface introduced in Fig. 3 is used to create such policies. Hyperlinks and context-driven menus are used to ensure the syntactical correctness of policies and the access to up-to-date domain knowledge (e.g., the list of doctors, with their specialities).

A policy is *general* when the precondition is a domain of values (e.g., all calls, all calls from Canada, all calls from a certain company). If the precondition relates to particular enumerated values, then the policy is *specialized* (e.g., how to proceed for calls from specified individuals).

A general policy may have exceptions, which are enumerated values. For example,

MyCompany_To_Pager (2):
forward calls from my company to my pager except if the call is from Terry March from 09:00 on Sunday, November 21, 2004 to 10:00 on Friday, November 26, 2004.

It means that

- the policy *name* is MyCompany_To_Pager and the priority is 2;
- the *operation* is the forwarding of an incoming call, and its target is my pager;
- the *precondition* is all calls from my company, with the exception of Terry March;
- the *time constraint* is from 9:00 on 2004/11/21 to 10:00 on 2004/11/26.

On a web interface, various elements can be hyperlinked to the form where they have been defined (see the underlined elements above).

V. TRANSLATION OF POLICIES TO CPL AND FIAT RULES

In order to enable formal analysis and detection of undesirable interactions (goal **G4**), each individual user policy is translated into CPL specifications that consist of a single branch (instead of the more conventional decision tree). A specialized policy will lead to one CPL script, whereas a general policy will lead to one or more CPL scripts. This section explains this translation process, as well as the conversion to a logic representation (FIAT rules). Section V-C also addresses the conversion of more general CPL scripts (with decision trees) to a collection of FIAT rules.

Table 1. Mapping between policy elements and FIAT rules.

Policy	FIAT Rule
Name	Rule name
Priority	Rule number
Operation	Rule result
Precondition	Rule triggering event
Target	Rule result
Exceptions	Rule constraint
Time constraints	Rule precondition

A. Specialized Policies

The translation of specialized policies (Section IV-B) into FIAT rules (Section II-C) is as follows. The single branch is visited downward, collecting the condition (e.g., time, caller, domain, etc.), the trigger (e.g., incoming call parameters) and the result (e.g., redirect, location). This information is then used to produce the corresponding FIAT rule, using the mapping found in Table 1.

For example, here is a policy (with name conference and priority 1) that redirects incoming calls from reception to my pager (a user-friendly alias to the address terry_march@pager.ottawahospital.com) during a conference in November.

Conference (1):
Forward calls from reception to my pager (no exceptions) from 09:00 on Sunday, November 21, 2004 to 10:00 on Friday, November 26, 2004.

The CPL script generated from this policy and called Conference_1 is the following[3]

```
<cpl>
  <incoming>
    <time-switch>
      <time dtstart="20041121T090000"
            dtend="20041126T100000" >
        <address-switch>
          <address contains="Reception">
            <location url="sip:terry_march@
                      pager.ottawahospital.com">
              <proxy/>
            </location>
          </address>
        </address-switch>
      </time>
    </time-switch>
  </incoming>
</cpl>
```

The translation of the CPL script into a FIAT rule results in

```
feature(['Conference',1],
[subs(user,Any),time([dtstart(d(2004,11,21,9,0,0)),
                      dtend(d(2004,11,26,10,0,0)),
```

[3]The initial <cpl> tag usually includes a reference to the CPL XML Schema. However, for simplicity, this information will not be included in the CPL scripts used in this paper.

```
                    interval('1'),
                    wkst('MO')])],
  [incoming([address(contains('Reception'))])],
  [proxy([location('sip:terry_march@
                   pager.ottawahospital.com')])])
  :- true.
```

In the previous FIAT rule,

- `['Conference',1]` is the rule *identifier*, with the rule *priority*.
- `[subs(···), time(···)]` is the rule precondition, with the *subs* part (required by FIAT but not really used in our mapping) and *time constraint*, which states that the policy applies from 9:00 am on November 21, 2004 and ends at 10:00 am on November 26, 2004. The values of *interval* and *wkst* (week start day), two fields used in the definition of periodical time constraints, are default values that appear in the translation when not specified explicitly by the user.
- `[incoming(···)]` is the rule triggering event, corresponding to the policy *precondition*.
- `[proxy(···)]` is the rule result, corresponding to the policy *operation* and its *target*.
- `true` is the rule constraint. It is set to true for specialized policies (no constraint), but it is used to represent *exceptions* in general rules.

B. General Policies

A general policy that contains exceptions generates a separate CPL specification for the general case, and one for each exception (as opposed to being combined into a single CPL script using the *otherwise* construct). In our system, exceptions bear a higher priority than the general case, which we believe to be the normal interpretation. This is reflected by the numerical naming scheme mentioned earlier (see also Section VII).

The CPL scripts that constitute general policies are translated to FIAT rules one at a time. While the translation of the first script is identical to the translation of specialized policies, the translation of the subsequent parts of that policy varies: They are further refined by adding the negation of the conditions of all previously translated parts. This process can be depicted by the following algorithm.

```
Algorithm 1 (translateToFIAT) // Translates CPL to FIAT
Input: list_of_CPL_parts
Output: // FIAT rules

previous_conditions = $\emptyset$;
foreach part in (list_of_CPL_parts) {
  let part $\rightarrow$ (conditions, triggers, results) // Tuple structure
  produceFIAT(conditions && not(previous_conditions),
              triggers, results)
  previous_conditions = previous_conditions $\cup$ conditions
}
```

For example, here is a policy that forwards all incoming calls to Jim Darling (general part), except if the call originates from the reception (exception part).

Any_but_Reception (3):
Forward any call to Jim Darling except if the call is from reception forever.

The exception part would first be converted to the following CPL script.

```
<cpl>
  <incoming>
    <address-switch>
      <address contains="Reception">
      </address>
    </address-switch>
  </incoming>
</cpl>
```

Note that the absence of specific actions to be taken in this CPL script results in the default behaviour of the system, namely to accept the incoming call.

The general part would result in the script below, and would be tagged as being of *lower* priority than the exception part. Priority information is not encoded in the CPL script but becomes part of the script name itself.

```
<cpl>
  <incoming>
    <location url="sip:jim_darling@
                   ottawahospital.com">
      <redirect/>
    </location>
  </incoming>
</cpl>
```

According to algorithm 1, the translation of the CPL scripts into FIAT rules becomes, respectively.

```
feature(['Any_but_Reception',3],
   [subs(user, _G728)], % _G728 is a dummy variable
   [incoming([address(contains('Reception'))])],
   [defaultAction]) :- true.
```

and

```
feature(['Any_but_Reception',4],
   [subs(user, _G834)],
   [incoming(ANY)],
   [redirect([location('sip:jim_darling@
                        ottawahospital.com')])]) :-
        ANY \= [address(contains('Reception'))]
               % not(previous_conditions)
        | ANY = anyUser.
```

The priority of CPL scripts is preserved within the FIAT rules such that later interpretation of the importance of the inconsistency detected may be qualified.

C. Translation of Full CPL Scripts to FIAT

The approach presented so far assumes the presence of a policy creation environment. However, for applications where a standard CPL script integrating multiple features or policies is provided (e.g., generated manually or by other means), our validation approach can still be used. The method for extracting individual CPL scripts and FIAT rules with priorities for processing by the system is described below.

As mentioned, CPL specifications are totally ordered. Hence, in a strict sense, there can be no conflicting instructions during execution. However, the burden of establishing this total ordering is carried by the user. In order to help the user establish this ordering and to uncover possible integration problems, the CPL specification is flattened into a set of FIAT rules, which are then processed by FIAT to detect and report inconsistencies.

The trees for incoming and outgoing calls are flattened separately. Starting at the top, each tree is traversed while collecting conditions and triggers defined in the elements of decision of the nested sub-trees. Conditions and triggers are contextual pieces of information that establish preconditions. But for the sake of interpretability and user-friendliness, contextual information is collected as conditions (e.g., time) while triggers are based on specific and connection-related facts (e.g., who is calling). Actions to be performed (e.g., redirect) are collected as results.

When the visit of a tree reaches a leaf (i.e., an action to be performed), the conditions, triggers and results collected so far are output as a FIAT rule. The visit continues by returning to the closest upward decision point with an unvisited branch, resetting the collection of conditions, triggers and results as they were at that point, and carrying on with the visit of the next unvisited branch (a *depth-first* traversal). If the branch is an `otherwise`, the corresponding condition is negated; if the branch is a `not-present`, the corresponding condition is removed. Then, the traversal of the tree is resumed.

As FIAT rules are produced, they are assigned a priority so that later analysis may provide more precise information on the inconsistencies based on the ordering used in the CPL specification.

VI. INTERPRETATION OF FIAT RESULTS FOR REPORTING AND SUGGESTIONS

As shown in Fig. 2, the generic results produced by FIAT need to be interpreted in the context of personalized policy conflicts, appropriate suggestions need to be provided, and automated corrections have to be supported. Sections VI-A, VI-B, and VI-D explain and illustrate our mechanisms for interpreting conflicts and making suggestions. Section VI-C gives a special attention to the problem of detecting overlapping time conditions (which is not supported natively by FIAT). Section VI-E presents two mechanisms to minimize the number of conflicts reported to the user (levels of sensitivity and database of previously tolerated/known conflicts). Finally, the support for interactive and personalized resolution of undesirable interactions is introduced in Section VI-F.

A. Interpretation of Inconsistencies and Suggestions

The inconsistencies reported by FIAT contain information identifying the features, their priority, and the nature of the inconsistency itself (i.e., which detection rule was violated). The first step toward the interpretation of the inconsistency is determining the type of policy, namely whether it is *general* or *specialized* (Section IV-B). In some cases, further understanding of the problem is obtained by comparing the relative priorities of the policies. Then the problem is reported to the user, starting by identifying the category of inconsistency. The role of each policy in the problem is exposed, an example of the possible misbehaviour resulting from the presence of the two policies is given and, when applicable, methods to correct the situation are proposed. As mentioned, all inconsistencies are cases of two policies that apply to the same situation, due to domain intersection, but which lead to different results. The following categories of inconsistencies are reported to the user (goal **G5**).

1. *Redundancy*: Two general policies target members of user-defined domains that have a non-empty intersection. A special case, *conflict within redundancy*, occurs when a general policy defines an exception whose domain intersects the domain of another general policy.
2. *Shadowing*: The domain of a general policy includes one or more elements enumerated by a specialized policy of lower priority.
3. *Specialization*: The domain of a specialized policy is included in the one of a general policy of lower priority.
4. *Conflict*: Two specialized policies address the same situation (i.e., the policies have intersecting enumerations).

Table 2. Suggestions offered for resolving conflicts.

Conflict	Suggestions
Redundancy	• Add (duplicate) exception to general policy • Disable first general policy • Disable second general policy
Shadowing	• Set priority of specialized policy above that of general policy • Set priority of general policy below that of specialized policy • Disable general policy • Disable specialized policy
Specialization	• Notice/warning (no suggestion other than a or e)
Conflict	• Disable first policy • Disable second policy

Let us elaborate on shadowing (the other inconsistencies can be explained using similar examples). Shadowing occurs when the preconditions of a general policy and those of a specialized policy overlap while their respective results differ. The specialized policy is unreachable and never executed for the elements in the intersection because all calls that meet its preconditions are handled by the general policy with higher priority. For example, a policy that forwards all calls that arrive from 2:00 pm to 4:00 pm to voice mail shadows a policy where all calls from John Doe from 2:30 pm to 3:00 pm should be forwarded to the user's mobile phone.

Upon the detection of an inconsistency, some of the following suggestions may be provided by the system (goal **G7**).

a) Edit a policy (enables the user to modify one of the conflicting policies).
b) Disable a policy (deactivate a policy, without deleting it).
c) Set the priority of a policy above/below the priority of another policy.
d) Add an exception to a general rule.
e) Tolerate the conflict and no longer report it (the system will proceed as explained in the conflict report).

The comments explaining the conflicts have hyperlinks to the policies involved, hence they can at any time be edited (suggestion a). Also, each detected conflict, whatever its nature, could be tolerated and put in a database so it would no longer be reported (suggestion e).

The other types of suggestions (b, c, d) are adapted to the particular conflict detected, and only the relevant suggestions are offered (see Table 2).

To facilitate and accelerate the correction of the policies, the suggestions proposed to the users are hyperlinked to commands and parameters that instruct the system to apply the one selected (e.g., clicked on). This will be further explained in Section VI-F.

B. Inconsistency Interpretation

Each inconsistency detected by FIAT can be interpreted according to the following procedure, which is based on the nature (specialized or general) and the priority of the features involved.

```
Algorithm 2 (interpretInconsistency) // gives a diagnostic
Input: list_of_inconsistencies
Output: // Interpretation
foreach inconsistency in (list_of_inconsistencies) {
  let inconsistency → (Feature1, Feature2, problem)
                         // Tuple structure
  determineCategory (inconsistency, Category)
  if priority (Feature1) > priority (Feature2) then
    // To report on shadowing
    swapNumbersOfFeatures (inconsistency)
  endif
  reportConflict (inconsistency, Category)
}
```

where

```
Algorithm 3 (determineCategory) // used by Algo. 2
Input: inconsistency // tuple
Output: Category // see Section VI-A
let inconsistency → (Feature1, Feature2, problem)
                       // Tuple structure
if generalPolicy(Feature1) then // Feature1 is general
  if generalPolicy (Feature2) then // Feature2 is general
    Category = 1 // Redundancy [+Conflict]
  else // Feature2 is specialized
    if priority (Feature1) > priority (Feature2) then
      Category = 2 // Shadowing
    else
      Category = 3 // Exception/Specialization
    endif
  endif
else // Feature1 is specialized
  if generalPolicy (Feature2) then // Feature2 is general
    if priority (Feature1) < priority (Feature2) then
      Category = 2 // Shadowing
    else
      Category = 3 // Exception/Specialization
    endif
  else // Feature2 is specialized
    Category = 4 // Conflict
  endif
endif
```

and

```
Algorithm 4 (reportConflict) // used by Algo. 2
Input: Inconsistency // tuple
Input: Category // see Section VI-A
Output: // Interpretation
let inconsistency → (Feature1, Feature2, problem)
                       // Tuple structure
case category of
  1: If generalTrigger (problem) then
       "Redundancy: Both policies provide directives for
                  all incoming calls"
     else
       "Conflict within Redundancy: Exception collides"
     endif
  2: "Shadowing: General policy Feature1 overrides policy
                  Feature2"
  3: "Specialization: Policy Feature1 specializes general
                  policy Feature2"
  4: "Conflict: Both specialized policies address the same
                  case"
endcase
```

Using such a procedure, HTML code can be produced to format the report and provide appropriate hyperlinks to the user, via the interface used to generate the policies.

C. Time Constraints

Time constraints, specified with semantics based on the iCalendar standard [22], may represent unique occurrences as well as recurring intervals. These time constraints become members of the set of preconditions of the FIAT description. Although FIAT does not support the concept of time per se, it does offer general mechanisms to define interaction of parameters. As mentioned in Section II-C, it is possible to state that two given parameters are in direct contradiction, such that when FIAT encounters these parameters in the precondition parts of two different rules, the analysis of that pair is skipped altogether. However, this method does not apply directly to the specification of time because time specifications in iCalendar are complex character chains that cannot be immediately compared.

For example, it is not immediate to know whether the 28-th of February will fall on a Monday within the next two years. In principle, one can assume that any two time intervals that are not obviously exclusive will overlap at some point in the future, but it would be useful to see if this can happen within the lifetime of the policy or in a foreseeable future. Our solution to this problem is a pragmatic one: We have created the `notOverlap()` predicate, which assumes that each rule has a life expectancy (two years by default) and exhaustively searches for overlaps in this time interval. A contradiction pair that uses this complex predicate is automatically included in the list of descriptions.

```
contradiction_pair(time(T1),time(T2)) :-
        notOverlap(T1,T2).
```

where T1 and T2 are two time descriptions. As a result, FIAT will avoid analysing any pair of rules for which time does not overlap (they are disjoint in time). For example, given that one rule applies to Mondays and another to Tuesdays, their time specifications do not overlap and therefore no further analysis is carried on that pair. However, a rule that applies to a particular week and another rule that applies to Tuesdays will not contradict each other (as their time specification overlap) and hence they will need to be further analysed. To avoid problems in the farther future, these evaluations could be automatically repeated at fixed time intervals.

D. Illustration

As an example of conflict reporting with suggestions, we will use the following set of four policies (prioritized as they appear). Note that words underlined are hyperlinks to definitions (in policies and conflicts) or to correction procedures (in suggestions).

Conference (1):
Forward calls from reception to my pager (no exceptions) from 09:00 on Sunday, November 21, 2004 to 10:00 on Friday, November 26, 2004.

Working_From_Home (2):
Forward any call to My Home Phone (no exceptions) forever.

Any_but_Reception (3):
Forward any call to Jim Darling except if the call is from reception forever.

Appointment (4):
Forward calls from reception to my pager (no exceptions) from 08:00 on Thursday, November 25, 2004 to 17:00 on Monday, November 29, 2004.

The following five problems are uncovered by FIAT and interpreted by the algorithm in Section VI-B. The conflicts and suggestions are formatted in HTML and displayed on the web interface of Fig. 3.

Specialization
Policy Conference specializes general policy Working_From_Home.

When a call comes in from 'reception', it will be forwarded to location 'sip:terry_march@pager.ottawahospital.com' by policy Conference, instead of being forwarded to location 'sip:terry_march@home.ottawahospital.com' by the general policy Working_From_Home.

This is just a notice.

Suggestions:
- TOLERATE this conflict

Conflict
Both policies Conference and Any_but_Reception address the situation where a call comes in from 'reception', but they react differently.

Since policy conference has priority over policy Any_but_Reception, the call will be forwarded to location 'sip:terry_march@pager.ottawahospital.com' (instead of being [defaultAction].)

Suggestions:
- If the currently prioritized policy is preferred
  - DISABLE policy Any_but_Reception
- If the alternative is preferred
  - DISABLE policy conference
- TOLERATE this conflict

Conflict within Redundancy
The general policies Working_From_Home and Any_but_Reception specify conflicting actions to be taken when a call comes in from 'Reception'.

The call will be forwarded to location 'sip:terry_march@home.ottawahospital.com' since Working_From_Home has higher priority, while policy Any_but_Reception would have let it through.

Suggestions:
- See the related Redundancy warning for details.

Redundancy
The general policies Working_From_Home and Any_but_Reception both provide directives for all incoming calls.

Policy Working_From_Home has higher priority and will have calls forwarded to location 'sip:terry_march@home.ottawahospital.com'. Policy Any_but_Reception will never have calls forwarded to location 'sip:jim_darling@ottawahospital.com'.

Suggestions:
- If the currently prioritized policy is preferred,
  - ADD EXCEPTION for reception to policy Working_From_Home
  - DISABLE policy Any_but_Reception
- If the alternative is preferred,
  - DISABLE policy Working_From_Home
- TOLERATE this conflict

Shadowing
General policy Working_From_Home overrides policy Appointment.

When a call comes in from 'reception', it will be forwarded to location 'sip:terry_march@home.ottawahospital.com' by policy Working_From_Home instead of being forwarded to location 'sip:terry_march@pager.ottawahospital.com' by policy Appointment.

Suggestions:
- SET PRIORITY of Appointment above that of Working_From_Home
- SET PRIORITY of Working_From_Home below that of Appointment
- DISABLE policy Working_From_Home
- DISABLE policy Appointment
- TOLERATE this conflict

Note that these results are returned to the user within a couple of seconds, which is a typical response time for web-based environments. Since users are not expected to have more than 15 or 20 policies activated at the same time, and since modifications and validation are performed by users only sporadically, performance is adequate for practical purposes.

E. Levels of Sensitivity

Various levels of sensitivity for the detection of conflicts can be defined by the user, similar to compiler options where one can choose the types of warning to be reported while compiling the source code of a program. Each level is essentially a subset of the four types of conflicts defined in Section VI-A. Predefined levels can easily be defined for inexperienced users (goal **G6**).

- *Complete*: Redundancy, shadowing, specialization, and conflict.
- *Errors only*: Redundancy, shadowing, and conflict.
- *Conflicts only*: Redundancy and conflict. This option is particularly useful in a system where individual policies are in-

tegrated in such a way that a specialized policy always has priority over a (otherwise conflicting) general policy.

Additionally, a database containing previously detected (and tolerated) conflicts could be turned off or be reset. This would enable end-users to get a complete list of conflicts for a particular level of sensitivity or just those introduced by the latest changes to the policies (see Section II-C).

F. Support for Policy Correction

As mentioned, in the previous examples of suggestions, all underlined actions (e.g., <u>SET PRIORITY</u>, <u>DISABLE</u>) are in fact hyperlinks with appropriate addresses and arguments to request from the policy server that the modifications selected by the user (i.e., clicked on) be executed. This automation level addresses goal **G9** and prevents users from having to modify the policies manually (and potentially to make mistakes along the way). The conflict explanations also have hyperlinks to the policies involved, hence they can at any time be edited if desired.

Note that the conflict resolution chosen by a user is based on local priorities and local activation of policies, and hence is independent of that of other users (goal **G8**). Such personalized policy conflict resolution enables the same conflict to be addressed in different ways by different users, which is a major improvement over conventional telephony systems.

VII. PRODUCTION OF SINGLE SCRIPTS INTEGRATING INDIVIDUAL POLICIES

Common CPL execution engines in PBXs and other telecommunication servers require that a user be limited to one single CPL script (goal **G3**). Once the inconsistencies are removed and the individual policies (CPL branches) ordered according to the priorities desired by the user, the individual policies defined within the system may be integrated into a single CPL script according to the recursive procedure described in algorithm 5.

```
Algorithm 5 (integrate) // integrates a list of policies
Input: list_of_branches // prioritized CPL policies
Output: // Single CPL script

outputBranch(head(list_of_branches))
rest = tail(list_of_branches) // remove the first branch
if rest $\neq \emptyset$ then
    open_otherwise
    integrate(rest)
    close_otherwise
endif
```

This procedure produces a nested list of branches using CPL `otherwise` constructs. A more advanced procedure could prioritize the policies to remove all shadowing conflicts automatically, before the integration (see algorithm 6). Such integration would implement the naïve view that specialized policies always have priority over (conflicting) general policies, at the cost of decreased flexibility.

```
Algorithm 6 (removeShadowingAndIntegrate) // alternative
Input: list_of_branches // CPL policies
Output: // Single CPL script

ConflictList = detectConflicts(list_of_branches)
new_list = list_of_branches
foreach conflict in (ConflictList)
    if type(conflict) = shadowing then
        // Adjust the priorities in new_list
        givePriorityTo(specializedPolicy(conflict), new_list)
    endif
integrate(new_list)
```

In the example of shadowing of Section VI-A, it would be commonly understood that the more specialized policy relating to John Doe would have higher priority. If it did not, then the policy could never operate and the user's probable intention would be defeated.

VIII. DISCUSSION

A. Related Work

Recent contributions by Nakamura *et al.* [23]–[25] contain an analysis of the feature interaction problem in CPL and present a tool to detect feature interactions in single CPL scripts and in pairs of CPL scripts. We limit our analysis to single CPL scripts, and the detection techniques differ considerably. The authors of these papers base their method on an analysis of the problems that can arise in CPL scripts. They identified eight possible semantic warnings and wrote a program to detect the presence of these situations in scripts. Interactions involving scripts of two different users are identified by combining the scripts and looking for the same situations (with limitations for practical use, as explained in Section II-A). Our method is based on a general-purpose tool (FIAT) that uses logic programming to detect logical inconsistencies in feature descriptions, on the basis of the general filtering rules described in Sections II-C or VI-A.

In particular, our tool does not deal with the following warnings, which are identified in those papers.

- Multiple forwarding addresses;
- unused sub-actions;
- call rejections in all paths;
- identical actions in a single switch;
- overlapped conditions in nested switches;
- incompatible conditions in nested switches.

Such warnings are useful, however it can be claimed that they are more CPL programming errors than policy inconsistencies. In fact, our CPL code generator could prevent most of these situations from happening.

Situations covered by both tools are

- Address set after address switch;
- overlapped conditions in a single switch.

Apart from these situations, the tool in discussion does not seem to deal with all the other cases covered by the rules of Sections II-C and VI-A. Further, this approach does not mention a policy creation, integration, and management system (our Sections IV and VII), or a system to help the user remove undesirable inconsistencies (our Section VI). Finally, their system seems to be specific to CPL, while ours can be easily adapted to other policy languages.

In his recent thesis, Xu suggested a conversion from CPL to a logic-based representation [26], [27]. The approach includes a Prolog theorem prover used for feature interaction detection in single scripts and pairs of scripts. The analysis warns users about unreachable code, redundant actions, and shadowing for single scripts. Policy management and conflict resolution are not addressed. This work does not use FIAT directly, but adapts the ideas of FIAT to the analysis of CPL. Clearly, it would still be useful to produce a tool that combines the strengths of these two approaches with ours, but we leave this to further research.

B. Potential Extensions

Enterprise and group policies

Our assumption so far has been that one end-user has control over all of her call processing policies. However, in an enterprise several layers of policies could be involved (as implied by goal **G2**). For instance enterprise policies imposed to all employees (e.g., no outgoing calls to pay-per-call 1–900 numbers), and group policies applicable to a group of individuals (e.g., no long-distance calls, obligation to answer calls from the group manager (no voice mail), etc.). Personal policies defined by the end-user may obviously conflict with group and enterprise policies.

In this context, conflicts can be detected in the same way as with personal policies. A sorted list of policies can be produced out of the combination of enterprise, group, and personal policies. A typical priority scheme for these layers would be 1-enterprise, 2-group, and 3-personal, but more fine-grained arrangements could be supported. For instance, group policies may be split into two sets, the first with a priority higher than personal policies (mandatory), and the second with lower priority (can be overridden by personal policies). The analysis mechanism could take such scheme into consideration when reporting the conflicts (e.g., do not report conflicts between personal policies and group policies of lower priority).

End-users would not have any means to modify group or enterprise policies. These would be managed separately by designated administrators with sufficient access privileges.

Voice interfaces

It was shown how web interfaces (running on personal computers, PDAs, mobile phones, phones with HTML browsers, etc.) can be used to manage personal policies. Voice-activated interfaces can also be used as they nowadays provide audio menus and speech-recognition capabilities enabling anyone to manage policies through their phone (wired or wireless), anytime, anywhere. To support voice, the web server needs to be coupled to a voice server and to generate the interface in a language such as voice XML [28] instead of (or additionally to) the conventional HTML used for web interfaces.

Interactions between policies of different users

Many feature interactions are between policies of different users, and our method does not address these. Research done on this issue concurs on the following main ideas [25], [27], [29]

- Multi-user interactions can be handled with similar methods as those available for a single user. It is necessary to do the union of the sets of policies of the users involved and to look for interactions in this union, at run-time. If forwarding features are present, it may be required to include, transitively, the policies of the participants where the calls may be forwarded.
- Who does this remains a major issue. It appears to be necessary to rely on a trusted third party, which is invoked at the time a new call is proposed. This party would collect all the policies of all the users that are being connected and would check for inconsistencies by using methods similar to the one described above.
- The resolution mechanism could be predefined or described with policies. Again, resolution schemes proposed by different participants could themselves conflict.
- The issue of privacy in the reporting and resolution of conflicts between multiple parties is a serious one. A solution to this issue that is not sensitive to sociological requirements could cause severe embarrassment to users and rejection of the tool.

The elaborate procedures required to handle such situations go beyond the scope of this paper and still require further research to be usable not only by expert system administrators but also by naïve users.

IX. CONCLUSIONS

In the context of a policy-based feature creation environment, several goals have been identified to enable naïve users not only to create and manage policies in a usable way, but also to detect conflicts and resolve them interactively (Section III). We presented a method and a tool that achieve most of these goals.

- Manage feature policies, described in natural language via a web interface, for inclusion in executable CPL scripts (Sections IV, V, and VII).
- Detect inconsistencies between these policies, with several degrees of sensitivity, and report conflicts with user-oriented terminology (Section VI).
- Help the user remove the inconsistencies interactively, with minimal effort (Section VI).

We also discussed possible extensions that would cover the remaining goals (Section VIII).

We believe that this framework is not limited to Internet telephony. In fact, the issues we have identified in this paper are pervasive in policy systems. They are encountered in many other contexts such as email, active databases, web services, routers, and firewalls.

It is worth recalling that the approach takes into consideration common principles, such as the implicit priority of specialized policies over general ones. Naïve users are very comfortable specifying features that are both shadowed and specialized (see Section VI-B). To be useful, a tool must support this natural attitude rather than trying to force users in an artificial pattern. The matching of the technique to the natural tendency of users is the reason why inconsistencies such as specialization can be filtered out easily, giving users the opportunity to focus on more relevant conflicts. The system is open to the application of other ergonomic principles to make it more sensitive to implicitly specified user preferences. For example, more recently specified policies tend to be a more accurate reflection of the user's current intentions and can be given higher priority.

The tool is part of a prototype for an advanced IP-based PBX developed by the company Mitel and has been used in a number of industrial demonstrations. The method is the subject of a patent submission.

ACKNOWLEDGMENTS

This work was funded in part by grants from Mitel, Communication and Information Technology Ontario, and the Natural Sciences and Engineering Research Council of Canada. Kathy Baker and Jean-Marc Seguin participated in the design and implementation of the system, and are listed as co-authors on the patent submission.

REFERENCES

[1] M. Amer, A. Karmouch, T. Gray, and S. Mankovskii, "Feature interaction resolution using fuzzy policies," in *Feature Interactions in Telecommunications and Software Systems VI (Proc. FIW 2000)*, M. Calder and E. Magill, Eds., IOS Press, May 2000, pp. 94–112.

[2] T. Gray, R. Liscano, B. Wellman, A. Quan-Haase, T. Radhkrishnan, and D. Choi, "Context and intent in call processing," in *Feature Interactions in Telecommunications and Software Systems VII (Proc. FIW 2003)*, IOS Press, 2003, pp. 177–184.

[3] S. Reiff-Marganiec and K. J. Turner, "Feature interactions in policies," *Computer Networks*, vol. 45, pp. 569–584, 2004.

[4] D. Amyot and L. Logrippo, Eds., *Feature Interactions in Telecommunications and Software Systems VII (Proc. FIW 2003)*, IOS Press, 2003.

[5] M. Calder, E. Magill, M. Kolberg, and S. Reiff-Marganiec, "Feature interactions: A critical review and considered forecast," *Computer Networks* vol. 41, pp. 115–141, 2003.

[6] U. Black, *The Intelligent Network*, Prentice-Hall, 1998.

[7] A. B. Johnston, *SIP: Understanding the Session Initiation Protocol*, Artech House, 2001.

[8] L. Lennox, X. Wu, and H. Schulzrinne, "Call processing language (CPL): A language for user control of Internet telephony services," *RFC 3880*, Internet Engineering Task Force, Oct. 2004.

[9] J. Lennox and H. Schulzrinne, "Feature interaction in Internet telephony," in *Feature Interactions in Telecommunications and Software Systems VI (Proc. FIW 2000)*, M. Calder and E. Magill, Eds., IOS Press, May 2000, pp. 38–50.

[10] J. D. Moffett and M. S. Sloman, "Policy conflict analysis in distributed system," *J. Organizational Computing*, pp. 1–22, 1994.

[11] S. Thebaut, W. Scott, E. Rustici, P. Kaikini, L. Lewis, R. Malik, S. Sycamore, R. Dev, O. Ibe, A. Aggarwal, and T. Wohlers, "Policy management and conflict resolution in computer networks," US Patent 5,889,953, 30 Mar. 1999.

[12] E. Lupu and M. S. Sloman, "Conflicts in policy based distributed system management," *IEEE Trans. Software Eng.*, vol. 25, Nov./Dec. 1999.

[13] Z. Fu, S. F. Wu, H. Huang, K. Loh, and F. Gong, "IPSec/VPN security policy: Correctness, conflict detection, and resolution," in *Proc. IEEE Policy Workshop 2001*, Jan. 2001.

[14] S. Reiff-Marganiec and K. J. Turner, "Use of logic to describe enhanced communication services," in *Formal Techniques for Networked and Distributed Systems (Proc. FORTE 2002)*, D. A. Peled and M. Y. Vardi, Eds., *LNCS 2529*, Springer-Verlag, 2002, pp. 130–145.

[15] T. Bray, J. Paoli, and C. M. Sperberg-McQueen, *Extensible Markup Language (XML) 1.0, 2nd ed.*, W3C Recommendation REC-xml-20001006, World Wide Web Consortium (W3C), Oct. 2000.

[16] X. Wu and H. Schulzrinne, "Location-based services in Internet telephony," in *Proc. IEEE CCNC 2005*, Jan. 2005.

[17] X. Wu and H. Schulzrinne, "Feature interactions in Internet telephony end systems," *Technical report*, department of computer science, columbia university, NY, U.S.A., Jan. 2004.

[18] N. Gorse, " The feature interaction problem: Automatic filtering of incoherences & generation of validation test suites at the design stage," M.Sc. thesis, SITE, University of Ottawa, Canada, Sept. 2000.

[19] N. Gorse, L. Logrippo, and J. Sincennes, "The feature interaction problem: Automatic filtering of incoherences and generation of validation test suites at the design stage," *J. Software & Syst. Modeling*, to be published.

[20] A. Colmerauer, "Prolog in 10 figures," *Commun. the ACM*, vol. 28, pp. 1296–1310, 1985.

[21] D. Raggett, A. Le Hors, and I. Jacobs, *HTML 4.01 specification*, W3C Recommendation REC-html401-19991224, World Wide web Consortium (W3C), Dec. 1999.

[22] F. Dawson and D. Stenerson, "Internet calendaring and scheduling core object specification (iCalendar)," *RFC 2445*, Internet Engineering Task Force, Nov. 1998.

[23] M. Nakamura, P. Leelaprute, K. Matsumoto, and T. Kikuno, "Semantic warnings and feature interaction in call processing language on Internet telephony," in *Proc. IEEE SAINT 2003*, Jan. 2003, pp. 283–290.

[24] M. Nakamura, P. Leelaprute, K. Matsumoto, and T. Kikuno, "Detecting script-to-script interactions in call processing language," in *Feature Interactions in Telecommunications and Software Systems VII (Proc. FIW 2003)*, pp. 215–230.

[25] M. Nakamura, P. Leelaprute, K. Matsumoto, and T. Kikuno, "On detecting feature interactions in the programmable service environment of Internet telephony," *Computer Networks*, vol. 45, pp. 605–624, 2004.

[26] Y. Xu, "Detecting feature interactions and feature inconsistencies in CPL," M.Sc. thesis, SITE, University of Ottawa, Canada, Sept. 2003.

[27] Y. Xu and L. Logrippo, "Detecting feature interactions in CPL," submitted for publication.

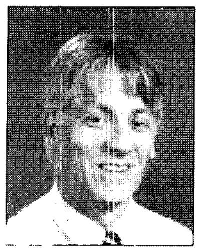
[28] S. McGlashan *et al.*, *Voice Extensible Markup Language (VoiceXML) Version 2.0*, W3C Candidate Recommendation, 20 Feb. 2003.

[29] L. Blair and K. J. Turner, "Handling policy conflicts in call control," in *Feature Interactions in Telecommunications and Software Systems VII (Proc. ICFI 2005)*, S. Reiff-Marganiec and M. D. Ryan, Eds., IOS Press, June 2005, pp. 39–57.

[30] P. Dini, A. Clemm, T. Gray, F. J. Lin, L. Logrippo, and S. Reiff-Marganiec, "Policy-enabled mechanisms for feature interactions: Reality, expectations, challenges," *Computer Networks*, vol. 45, pp. 585–603, 2004.

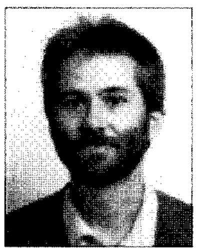
Daniel Amyot is Assistant Professor at the School of Information Technology and Engineering, University of Ottawa. He joined the Communications Software Engineering Research Group in 2002. He was previously senior researcher in software engineering for Mitel Networks, in the Strategic Technology group. Daniel has a Ph.D. and an M.Sc. from the University of Ottawa (2001 and 1994), as well as a B.Sc. from Laval University (1992). His general research interests include scenario-based software engineering, formal methods, requirements engineering, and feature interactions in emerging applications. Daniel has several contributions in the area of specification and validation of telecommunication services and features with use case maps and LOTOS. He is also ITU-T Rapporteur for Requirements Languages (URN and MSC). He co-chaired the 7-th Feature Interaction Workshop in 2003, the 4-th SDL and MSC Workshop (SAM) in 2004, and the 4-th NOTERE conference in 2005.

Tom Gray has had a 30-year career in the telecommunications industry. He has worked for Bell Northern Research, Mitel Corporation, and Mitel Networks. He has over 40 patents issued and pending, and over 30 papers published in international conferences and archival journals. At Mitel, he was instrumental in developing a consortium of university projects that explored the types of services that will be useful and profitable in new converged multimedia networks and how they would be developed, operated, and maintained. Over 20 university research groups participated in this project in Canada and the United Kingdom. This effort resulted in useful new technology, many papers and patents, and the education of numerous students. Prior to this, he was the prime mover in the group that created the vision and technology behind the development of the Mitel Light family of PBXs. The group realized that the assumptions that had been seen as fundamental to the design of telephone switches were no longer valid and based a new architecture on the capabilities of new technology. The success of this product changed the industry-wide model of PBX architecture. The basis of this architecture has guided the development of PBXs and other telecommunication switches up to the present day.



Ramiro Liscano is a professor at the School of Information Technology and Engineering at the University of Ottawa. His areas of research are in spontaneous networking, service discovery, distributed call control, mobile computing, and policy management. Previous to that he held a Senior Research Engineering position with the Strategic Technology Group at Mitel Networks. From 1995–2000, he was a research scientist in the Network Computing Group at the Institute for Information Technology at the National Research Council. He received his Ph.D. from the Systems Design Engineering Dept. at the University of Waterloo in 1998. He has published over 80 papers in the past 20 years. He is a senior IEEE member and member of the Association of Professional Engineers of Ontario.

Luigi Logrippo received a degree in law from the University of Rome (Italy) in 1961, and in the same year he started a career in computing. He worked for several computer companies and in 1969 he obtained an M.Sc. in Computer Science from the University of Manitoba, which was followed by a Ph.D. in Computer Science from the University of Waterloo in 1974. He was with the University of Ottawa for 29 years, where he was Chair of the Computer Science Department for 7 years. In 2002, he has moved to the Université du Québec en Outaouais, Département d'informatique et ingénierie, while remaining associated with the University of Ottawa as an Adjunct Professor. His interest area is formal and logic-based methods and their applications in the design of communications systems. For a number of years he worked on the development of tools and methods for the language LOTOS. Current research deals with the formal analysis of advanced communications services made possible by internet telephony, of the policies that govern them, and of their interactions, in application areas such as presence features and e-commerce contracts.



Jacques Sincennes is a research programmer/systems analyst at the University of Ottawa, School of Information Technology and Engineering. He has joined the Telecommunication Software and Engineering Research Group in 1988. He was involved in the development of tools for the specification and validation of protocols using the formal language LOTOS. In collaboration with Nortel Networks, he participated to the specification of the Wireless Intelligent Network standard and to the elaboration of formal methods. In the realm of user-defined call control and in association with Mitel Networks, he assembled a system for policy interaction detection that provides customized problem resolution. He coauthored a number of papers and a patent application.