Kenneth J. Turner and Lynne Blair. Policies and Conflicts in Call Control, Computer Networks, 51(2):496-514, February 2007

Policies and Conflicts in Call Control

Kenneth J. Turner[a] and Lynne Blair[b 1]

[a] *Computing Science and Mathematics, University of Stirling, Stirling FK9 4LA, UK*

[b] *Computing, InfoLab 21, South Drive, Lancaster University, Lancaster LA1 4WA, UK*

Abstract

Policy-based management is introduced and related to the specific needs of call control. It is explained how policies differ in important ways from features. Related work on policy-based management is reviewed, leading to the conclusion that a different approach is required for call control. A general architecture is presented for a policy system. This includes an overview of the major policy components, relating them to the system under control and to the context system that provides additional information.

As a framework for explaining how policy conflicts are handled, the policy language for call control is briefly presented along with some sample policies. The paper then focuses on how policy conflicts are defined and resolved, using sample resolutions to illustrate the approach. Pointers are given to future enhancements to policy support, including new applications of policies to novel domains.

Keywords: Call Control, Internet Telephony, Policy, Policy Conflict

1 Introduction

This paper presents a novel approach to policy-based control of calls, and to handling conflicts that may arise among such policies. This can be seen as a significant step forward from features and interaction handling in conventional telephony. The paper offers the following enhancements to previous publications on this approach:

- An extended review is given of policy-based management. This is used to explain the distinctive needs of policies to call control, and why no existing policy system was found to be suitable for this application.

[1] Work carried out while the author was on leave at the University of Stirling.

Email addresses: kjt@cs.stir.ac.uk (Kenneth J. Turner), lb@comp.lancs.ac.uk (Lynne Blair).

- A unified and generalised view is presented of the entire policy system. This demonstrates that the approach is not tied to call control. The description also provides the necessary framework for understanding how policy conflicts in call control are handled.
- The mechanisms for conflict detection and resolution have been generalised, allowing a wider variety of conflicts to be handled.
- Compared to [7], much more detail of resolution policies has been given, along with a number of illustrative examples.
- An overview is given of new enhancements being developed for the policy system, and how it is being adapted for new applications.

1.1 Policy-Based Systems

Policy-based management has become popular for controlling a variety of systems. As examples, policies are commonly used for access control, quality of service, and system management. Policies capture high-level goals that can be automatically enforced. Using predefined policies, a system can dynamically adjust its behaviour without requiring manual intervention.

Suppose it is necessary to control access to a networked printer. The system manager can define which users may access this printer, change its settings, or upgrade its firmware. As another example, suppose a streaming video system must adapt to changing workloads. The system manager can allocate default resources such as processing, bandwidth and buffering. As demands vary, policies can decide how to modify these resources: frame rate or size might be altered, compression parameters might be adjusted, and colour depth might be changed.

Policy conflict is an almost inevitable consequence of policy-based management. Such conflicts may arise at different levels. Even the policies of a single user may interfere with each other. The user of a network printer, for example, may have high quality and low cost as goals. The policies of peer users may also disagree. For example one user in a videoconference might desire high-quality video, while the other requires low-quality due to limited device capabilities. Policies may also be defined hierarchically within an organisation. Conflicting policies may occur at all levels, e.g. individual (high-quality video needed), department (H.261 video codec preferred), organisation (video bandwidth should be limited).

This paper reports on work to develop a policy-based system for call control. One aim was to allow end users (telephone subscribers) to define policies for how they wish their calls to be handled. Another aim was to allow system administrators to define higher-level policies for handling policy conflict. Although several existing policy languages might have been suitable, it will be seen that applications like call control require a different approach. Few options existed for dealing with conflicts, so again a distinctive solution was required.

1.2 Call Control

In telephony, the basic call is extended through features. These are relatively self-contained additions of functionality, e.g. for call diversion, call waiting or charge card calling. An important aspect of features is that they are automatically invoked, usually at well-defined trigger points in the basic call state model. This means that features can readily be added with little disturbance to the basic call. Unfortunately, the same mechanism means that features may interfere with each other – the well-known feature interaction problem [10].

There is a good analogy between features and policies, and between feature interaction and policy conflict. In a sense, a feature is a low-level policy. In fact, the authors are working on policy refinement to allow higher-level policies to be expanded into lower-level ones. This will bridge the gap between policies and features. However, features have a number of characteristics that limit their flexibility. In contrast, policies are higher-level and more malleable. There are similarities between them, but also important differences [13,29]:

- Features and policies are both intended to allow users to control their calls.
- Feature interaction and policy conflict may both be handled statically (at definition time) or dynamically (at call time).
- Features are low-level and imperative, whereas policies are higher-level and declarative. Suppose the user does not wish to receive calls from the press. In a feature-based approach, terminating call screening would be required with a list of blocked numbers. A comparable policy could simply reject calls from the press, identified by the caller domain or topic.
- Features have limited parameters, whereas policies can be much more flexible. For example, a call diversion feature would typically be parameterised by the affected number, the forwarding number, and the condition for diversion. A comparable policy could be much more subtle, choosing different forwarding numbers according to the caller, the time of day, the subject of the call, the capabilities and devices of the call parties, etc.
- Features are fixed and managed by the network operator or equipment supplier, whereas policies are open-ended and defined (mostly) by end users. A typical network or switch may have tens to hundreds of features. Although this may offer the user many options, the range of choices is nonetheless fixed and customisation is limited. If the user's requirement is not met by an existing feature, there is no alternative.
- Because features are defined by engineers, a technically complex approach may be followed. In contrast, policies should be definable by users to meet their needs. Although the policy language necessarily limits what users may do, the range of policies is much wider and is in fact infinite. Since policies should be accessible to ordinary users, a user-friendly and non-technical approach must be adopted. As a special case, more complex policies may also be defined by network operators.

- Feature interaction handling is essentially under the control of one network operator or equipment supplier. (Although this may not apply to multi-carrier calls, a common approach is often adopted.) This makes it much easier to identify and manage feature interactions. Policies, however, are largely user-defined. Furthermore, the policies applying to a call may stem from any pair of users (who may have never called each other before). Detecting and resolving policy conflicts is thus a much more challenging and dynamic task.
- One thing that helps resolution is that policies are closer to user needs, so it is easier to determine what the user's intention was. Not knowing the intention is a well-known problem in resolving feature interactions. Consider Do Not Disturb in conjunction with Alarm Call. Is the intention to block all calls (including wake-up calls), or to avoid being bothered by calls from other people?

Some network operators have introduced additional flexibility by allowing third parties to add external functionality. This is the approach taken by Parlay/OSA (Open Service Architecture, *www.parlay.org*). For example, this more flexible treatment of features like Automatic Call Distribution and Freephone Routing. It is arguable whether such an approach can be described as feature-based or policy-based.

Policy support for call control was developed by the ACCENT project: Advanced Call Control Enhancing Network Technologies, *www.cs.stir.ac.uk/accent*. The approach to policies for call control is discussed in [26,27,37]. In fact, a broad view was taken of what a 'call' might be. This includes traditional telephony, but also allows for newer developments such Internet telephony, Interactive Voice Response, multimedia calls, and Web/Grid services. Although the domain of call control has been of considerable influence, a generic approach has been developed that can be adapted for other domains.

1.3 Related Work

CPL (Call Processing Language [19]) allows users to define how they wish calls to be handled. However CPL is limited in a number of ways that make it unsuitable for general call control:

- It is limited in its network bindings (currently H.323 and SIP).
- It gives limited control over calls, specifically just call setup. There is also a need for mid-call control (e.g. when a new party is added to a call) and call tear-down control (i.e. when a call is disconnected). CPL also supports only limited checks, e.g. on the caller or the current time.
- It does not support a range of preferences (positive or negative, with different strengths).
- It is not (yet) integrated with context systems that provide presence and availability information.
- It does not offer any mechanisms for detecting and resolving conflicts among call preferences.

Some of the limitations of CPL have been addressed in work on LESS (Language for End System Services [41]). New developments in this include support for presence-based services and consideration of feature interactions.

Call centres and CTI (Computer Telephony Integration) support flexible call handling; see [16] for a survey of the approaches. Call centres rely on mechanisms such as Calling Line Identification and Automatic Call Distribution to route callers to appropriate agents. Call centres are designed for large businesses, unlike the work reported here which is intended for individual end users. Call centres essentially deal with routing within one organisation, whereas call policies handle calls on a global basis. Call centres also do not support the kinds of capabilities discussed in this paper. Policy-based support of calls is thus complementary to the techniques used in call centres.

Although call centres are not appropriate for ordinary subscribers, the policy system was designed to allow third-party policy support. This allows a user to offload policy definition and enforcement to a separate organisation, much as they might employ an answering service. This means that end users can then benefit from policies, without being exposed to the technical issues. As will be seen, the policy wizard allows an administrator to define policies on behalf of users. An administrator can also predefine policies that users simply select (such as ‘on holiday’ or ‘out of the office’). Again, the goal is make policies useful to those who wish to use them indirectly.

Policies have been used in many kinds of management tasks. Example applications, with one representative citation each, include access control [5], admission control [42], agent-based systems [8], content distribution [39], distributed trust [34], group collaboration [24], healthcare [1], network management [22], Open Distributed Processing [35], QoS (Quality of Service) [25], security [33], and systems management [12].

[21] defines policies as information that can be used to modify the behaviour of a system. This is a very general and open-ended definition. In the context of this paper, policies are interpreted as the goals for how calls should be handled. Policies lend themselves well to networked applications, where the very distribution demands careful management. Despite this, call handling systems have attracted little policy support. [2] uses fuzzy policies as a means of resolving feature interactions. Many researchers see policies as important in future call handling [13].

Policy language developments in industry have largely focused on network management and QoS. For example, Cisco have developed policy support for control of security and QoS in routers. Lucent and Bell Labs developed PDL (Policy Description Language) for network management. Hewlett-Packard’s PolicyXpert (now discontinued) was also focused on network management. The IETF standard for COPS (Common Open Policy Service) is intended as a protocol for managing QoS. None of these efforts is of direct relevance to call control.

[27] discusses the kind of policies that are needed in call control. Initially, some existing policy languages were evaluated to determine their suitability for this application. For example, a detailed evaluation [27] was made of Ponder [12]. It was found that Ponder was only partly suitable for this purpose. Nonetheless, Ponder has been influential on the work reported on call control.

A new policy approach was defined to overcome limitations of existing languages in a call control context:

- The focus of on call control is distinctive. It places different demands on a policy system, and of course it requires specialised support in a communications setting. The language developed for call control falls into the general category termed ECA (Event-Condition-Action). However the events, conditions and actions that arise in call control are completely different from, say, those required in network management.
- Ideally a policy language should be capable of specialisation for various application domains. This is true of only some existing languages. Although a language for call control has been developed, the core of the language is separate and can be adapted for other uses. Even when used for call control, the language has to be largely independent of the underlying communications system.
- In systems management, a useful distinction can be often made between the subject of a policy (that performs an action) and the target of a policy (that is acted upon). A number of policy languages such as Ponder reflect this. In call control, the nature of subject and target becomes unclear. It can be argued that the subject is the caller, the call or the network, while the target is the callee, the call or the network. Suppose a caller wishes video as well as voice. Is this achieved by the caller, the call instance or the network? Is it the callee, the call instance or the network state that is altered? Because of this issue, it was found difficult to apply Ponder effectively to call control.
- In many application domains, the entities involved in policies are fairly static and predictable. This does not apply to call control, where any user (previously unknown) may call any other user. As a result, call control introduces a much more dynamic set of policies. In addition, policies may be introduced by the underlying networks as well as the call parties.
- Most policy languages require specialised technical expertise, being designed for programmers or technicians. In contrast, policies for call control must be accessible to the ordinary subscriber. This presents a major challenge, because the policy language and the supporting policy system must be usable by non-technical people. Communication is global, so policy support must also be truly international – specifically, multilingual.
- Call control is more likely to lead to policy conflict because very many users with unpredictable policies may wish to communicate. Conflict handling needs to be meaningful to ordinary end users.
- Many policy languages support modal or deontic aspects. In the OPI language

[3], these are obligation, permission and interdiction. Ponder has obligation, authorisation and refrain policies. Obligation and interdiction apply to the subject, while permission applies to the target. Since the notions of subject and target do not map so readily to call control, these modalities need some rethinking for call control. Furthermore, obligations placed on end users have limited value since they cannot be enforced.

- A policy language should ideally have a form that is readily parsed by many tools. XML is widely employed for structured information, but is used by only a few policy languages.

For these reasons, it was concluded that no existing policy system would adequately serve for call control. It was therefore necessary to develop a new policy language and policy support, inspired by the unique needs of call control. However the language has been cleanly separated into a core and its specialisation for various application domains (here, call control). This allows the policy system to be largely re-used in other contexts. In this respect, the call control policy language resembles some others such as Ponder.

Distributed definition of policies can lead to incompatibilities among them. Policy conflict resembles the extensively studied feature interaction problem. A general discussion of this problem appears in [9–11]. It is argued in [28] that some techniques from feature interaction can be adapted for detection and resolution of policy conflicts. Nonetheless, conflict handling is still a challenging task.

Apart from feature interaction, policy conflict also resembles work on interactions among requirements [32] and on conflicts among goals [38]. Policy conflict has been studied for some years, but without any general solution emerging; it is easier to resolve policy conflicts in particular domains. [20] considers conflict analysis for management policies. A Role Based Management framework includes tool support for determining significant policy conflicts. The use of meta-policies has been considered in distributed systems management [21]. This work applies meta-policy checks when policies are specified and when they are executed. The POLICE language [14] aims to simplify conflict handling by avoiding negative policies. This is possible because policies in this language automatically lead to prohibition unless explicit authorisation is given.

[2] aims to define hierarchical policies such that, by definition, the subordinate policies cannot conflict. Conflicts are, however, still possible if one policy in the hierarchy is changed. The need for policies to control agents is examined in [40]. Multi-agent conflicts are avoided either through negotiation between agents or by appropriate sequencing of their tasks. [15] recognises but does not address conflicts that arise in policy-driven adaptation mechanisms. [6] tackles the problem of authorisation policies leading to conflict. This is resolved by providing a function to compare policies and decide which should take precedence.

1.4 Overview of Paper

Section 2 introduces the policy system, both in general and in its specialisation for call control. Section 3 overviews the policy language, and discusses how it has been specialised for call control and conflict resolution. Illustrative examples are given of both call policies and resolution policies. Section 4 summarises and evaluates the work, indicating how it is being developed further.

2 The Policy System

This section describes the architecture and main components of the policy system. An overview of the architecture appears in [28]. The implementation and APIs of the policy server are specified in [30]. The implementation and customisation of the policy wizard are described in [36]. A general overview of policies in call control is presented in [37]. Its specialisation for H.323 Internet telephony is covered in [17,18]. Conflict handling for call control is discussed in [7].

2.1 Policy System Architecture

The generic policy system architecture is shown in figure 1: the arrows represent socket interfaces. This gives considerable flexibility, allowing the components of the policy system to be distributed as required. It is also easy to replicate the components for resilience or load-sharing. For example an organisation might use an external policy server, might have a single policy server that manages multiple departments, or might have one policy server per department. Since the interfaces are logical ones, the components may be on separate physical systems or might share the same equipment. The policy system components have been run on anything between one and five separate systems.

All the code is written in Java, so multi-platform operation is possible; the policy system has been demonstrated without change on four different platforms. The policy system is also designed to be as independent as possible of the underlying communications services. This is essential because the communicating users cannot predict what network technology might be used. For example, at different times a call to the same user might connect to a conventional telephone, a mobile telephone, an Internet telephone or to voicemail. The policies must be independent of this. Of course, the potential penalty is that network-specific capabilities cannot be exploited (for good reasons). In fact, it is possible to define policies that depend on a particular network – but the user can be warned of this.

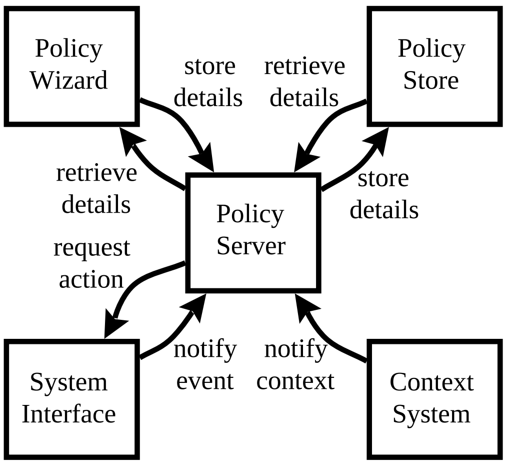


Fig. 1. Generic Policy System Architecture

The *policy server* is the heart of the system. The server retrieves and enforces policies, detecting and resolving policy conflicts. A *policy wizard* provides a user-friendly, natural language interface to the policy system. Apart from defining and editing policies, the wizard also supports policy variables, policy templates, voice clips, and a simple interface to presence and availability. Policy information is held in a *policy store* that includes regular policies, resolution policies, policy variables and user profiles. The *system interface* notifies the policy system of significant events in the system being managed, and performs the actions dictated by the policy system. The *context system* provides the policy system with contextual information that may influence policies. For example, this might identify a call party's availability and capabilities.

2.2 Policy Server

The policy server is triggered by external events, mostly from the system interface but also from the context system. The event interface provides information in a domain-independent format. For example, the system interface may notify the policy server of a request for a networked printer, a proposed videoconference, or an incoming call.

Event notifications can be as fine-grained or as coarse-grained as desired. Events are normally high-level triggers, such as a call being made. However individual keypresses on a telephone could also be reported as events, if it were desirable to have policies at this level. For example, use of the hash or square key might be disabled by a policy in order to restrict user actions.

At a minimum, the system interface must identify the event and the users involved in the event. For an incoming call, for example, it would report at least the terminating call event, the caller and the callee. This is used to interrogate the policy store for policies relevant to the event. In addition, higher-level policies are also implicitly identified (e.g. those for the user's department and organisation). Policies are associated with users, and implicitly with the devices or facilities they control. In call control, for example, policies can be associated with the user's telephone or with the user's voice mailbox.

The policy server retrieves a collection of policies that govern an event. These are checked for conflict using the resolution policies that apply in this context. Typically, resolution policies are defined by an administrator for groups of users (e.g. a department). However, they may also be defined by individuals or related organisations (e.g. the provider of a telephone service). As will be seen later, a resolution policy defines a class of conflicts that it handles. If no conflicts exist among the actions proposed by the policies, these actions are sent to the system interface for execution. If conflicts exist, the resolution policies dictate which actions result.

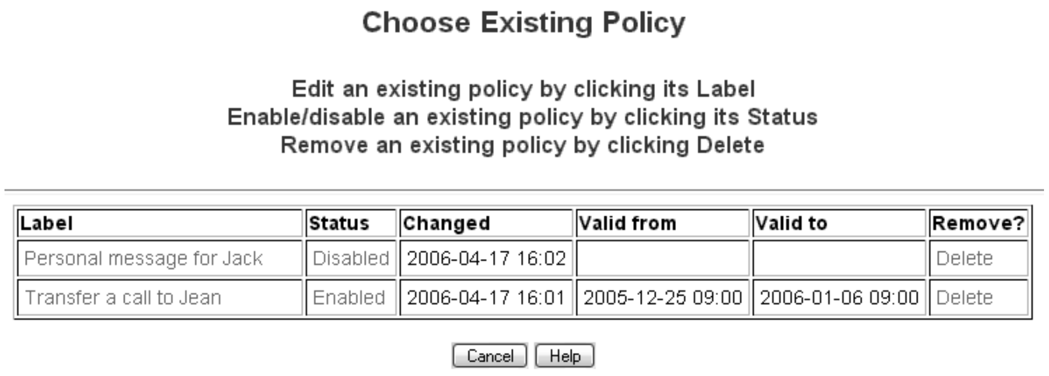
Normally, the effect of resolution is to create a subset of the proposed actions. That is, incompatible actions are eliminated. However, resolution might also result in completely different actions. Suppose Anne likes to have video calls, while Bert likes to have a complete recording of a call. Since this could result in very large call records, their administrator Cath might define the 'add video' and 'record call' actions as conflicting. The resolution might be to conference Cath into the call initially to decide whether video may be used or recorded.

2.3 Policy Wizard

Internally, policies are XML documents defined by a schema. Although they are thus text files, they are usable only by specialists. The policy wizard therefore exists to present and edit policies in a user-friendly manner. This is particularly important when dealing with end users such as subscribers. The policy wizard is web-based, being supported by JSP scripts (Java Server Pages) in a web server. Apart from the familiarity of a web interface, this has the advantage that policies may be modified from anywhere. A user away from the office, for example, may remotely log into the policy system and change how calls are handled (e.g. forward them to the current location, send them to voicemail).

The policy wizard interface shows policies in structured natural language. This was deemed to be the most appropriate way of interacting with end users. Note that natural language processing is not required because the interface is carefully structured. Because of the international nature of computing, the policy wizard was designed to be multi-lingual. Currently it supports English, French and German, but is readily extended for many other languages. The wizard also supports variants on the languages, e.g. American English and British English.

Other forms of interface were considered for the policy wizard. The authors like


Choose Existing Policy

Edit an existing policy by clicking its Label
Enable/disable an existing policy by clicking its Status
Remove an existing policy by clicking Delete

Label	Status	Changed	Valid from	Valid to	Remove?
Personal message for Jack	Disabled	2006-04-17 16:02			Delete
Transfer a call to Jean	Enabled	2006-04-17 16:01	2005-12-25 09:00	2006-01-06 09:00	Delete

Fig. 2. Screenshot of Policy List in The Wizard

Interactive Voice Response as an alternative, because it would allow a user on the move to define and modify policies (with perhaps just a mobile telephone). A graphical representation of the underlying XML was also considered. Since policies have a simple tree structure, this would be easy to achieve. However the effect would essentially be draw place boxes around the phrases that are currently rendered in natural language. It is unclear whether this would help much.

As an example of the policy wizard in action, figure 2 lists some policies for call control. These are the existing policies of English speaker Mark. Figure 3 shows what Mark sees when he clicks on the label of the second policy, allowing him to modify it. Currently it defines the following:

Applicability: The policy is defined as part of Mark's 'In the office' profile. Assigning a policy to a profile allows Mark to quickly enable different set of policies, e.g. 'At home' or 'On holiday'.

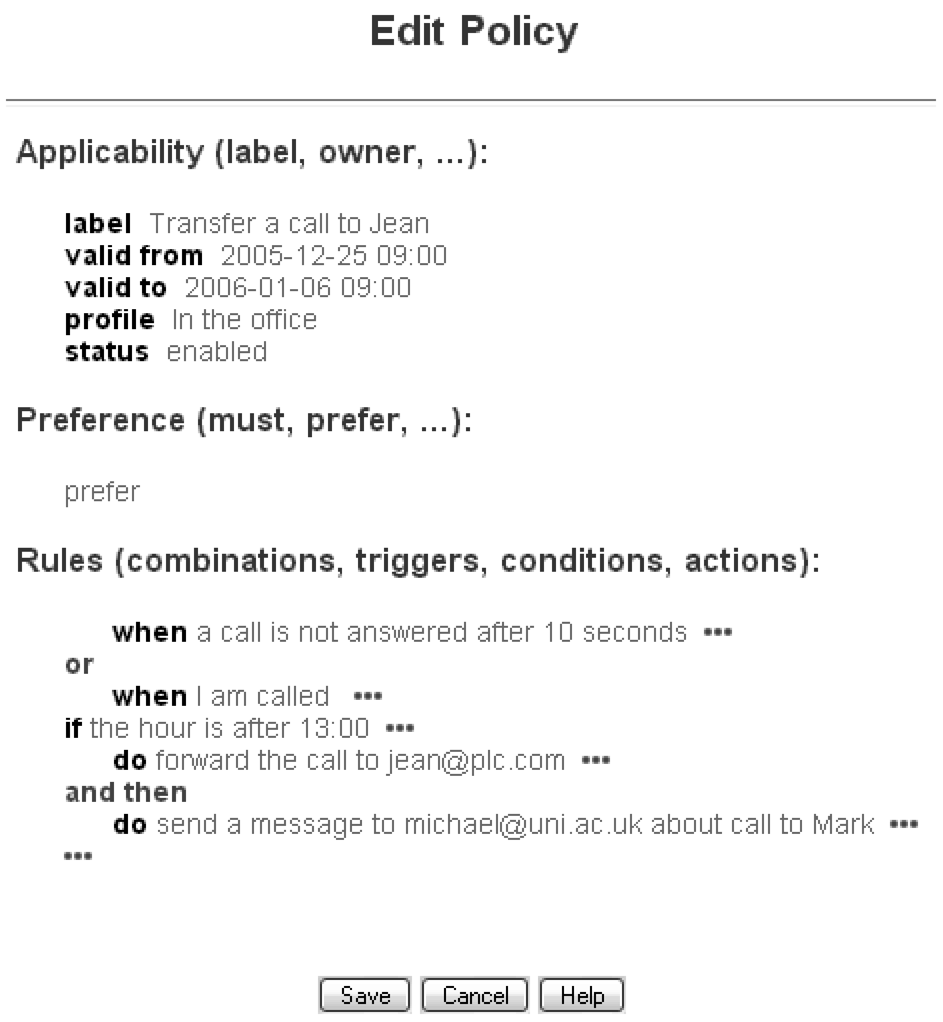
Preference: Mark *prefers* to have this policy apply. A policy can alternatively have a *must* or *should* preference, negative preferences, or an empty ('don't care') preference.

Rules: When there is no response to a call after 10 seconds or if someone calls, check if it is after 1PM. In that case, forward the call to jean@plc.com, then send a message to michael@uni.ac.uk that there has been a call to Mark.

All the elements of a policy are hyperlinked; clicking on an element takes the user to a page where the element can be changed. The '· · ·' symbol indicates where the policy can be extended. For example, clicking on the first instance of '· · ·' in figure 3 allows the user to add a further trigger, combined with *and* or *or*.

2.4 System Interface

The system controlled by policies depends, of course, on the particular domain. For access control to a printer, it might be a print spooler. For call control, it might be a proxy server for SIP (Session Initiation Protocol, used in Internet telephony).


Edit Policy

Applicability (label, owner, ...):

label Transfer a call to Jean
valid from 2005-12-25 09:00
valid to 2006-01-06 09:00
profile In the office
status enabled

Preference (must, prefer, ...):

prefer

Rules (combinations, triggers, conditions, actions):

when a call is not answered after 10 seconds •••
or
when I am called •••
if the hour is after 13:00 •••
do forward the call to jean@plc.com •••
and then
do send a message to michael@uni.ac.uk about call to Mark •••
•••

Fig. 3. Screenshot of Wizard Policy Editor

The system interface has to be created by adding a policy interface module to the server. Experience has shown that this is feasible in many cases, though it requires the server API to be defined. An interface module is typically about 1,000 lines of code. As an example for call control, interfaces were created to the MKC 7000 ICS softswitch and to the SER proxy server (SIP Express Router).

The system interface is bidirectional. It is designed to trigger on significant events: requesting a printer, excessive jitter in a videoconference, initiating a call, etc. The relevant event parameters are collected and sent to a policy server determined by a configuration parameter. While this event is being handled, processing of the event in the server is suspended. (Of course, the server continues to handle other activities normally.) Once the policy system has decided which actions apply (a possibly empty list), these are sent to the system interface for execution. Actions might deny a print request, add bandwidth to a videoconference, or divert a call.

The policy server is designed to be generic, so it does not need to have knowledge of specific system interfaces. The exchange with the policy server has a uniform format: key-value pairs for the event parameters and the resulting actions. The interpretation of events and actions is domain-specific, and thus defined by the specialisation in effect for the policy language. However, the policy server is driven by a database table giving the mapping between policy terminology and domain terminology. For example, an *INVITE* in SIP is mapped to a *connect* event in policy terms. The events and actions for call control are discussed in section 3.2.

As an example, information from a communications server might include:

- the time of the event
- the type of event (e.g. no answer to a call)
- the type of network (e.g. SIP Internet telephony)
- the addresses involved: the user who triggered the event, the caller and the callee
- the topic of the call.

The resulting actions might include:

- reject the call, forward the call, or fork the call (i.e. try multiple destinations)
- add or remove a third party (i.e. another subscriber)
- add or remove some medium (e.g. video)
- play a clip in some medium (e.g. audio or video).

2.5 Context System

The context system provides additional information that policies may act upon. The difference between this and the system interface is that context includes information to supplement system events. As a specific example, information from a context system for call control might include:

- the capabilities of a call party (e.g. a French speaker, a Java expert)
- the role of a call party (e.g. the callee's manager, a press agent)
- the presence and availability of a call party (e.g. present in Building 7, available for budget discussions).

The context system is outside the scope of the policy system, though it has a defined interface to it. The context system may obtain information from any source such as an organisation chart (for roles), an active badge system (for presence), or a user's schedule (for availability). Like system events, context events may trigger policies. For example, Anne can define a policy that notifies her (by email, pager or call) when Mark becomes available.

As a demonstration of a simple context system, the authors have implemented a link to a user's calendar as stored by Microsoft Outlook. This allows presence and availability information to be fed automatically into the policy system. For example, it may be determined that Anne is off-site or in room 5, and that she has a meeting or is free.

2.6 Policy Store

The policy store is used to hold dynamically changing information such as policies, policy variables and conflict details. In addition, the policy system stores more static information such as login data, as well as the terminology mapping between policies and domains. In fact, the use of the dynamic and static information is quite different. Although a single policy store could be used, the current implementation uses two different kinds of databases. Dynamic information is held in an XML database (the IBM TSpaces tuple space server). Static information is held in an SQL database (MySQL). Internally, both forms of database are implemented via an abstract interface. Alternative databases can therefore be readily used in place of the current solutions.

3 The Policy Language

This section discusses the policy language APPEL (ACCENT Project Policy Language Environment/Language, the French word for ‘call’). APPEL is specified in [31]. Its use for SIP is described in [37], for H.323 in [18], and for conflict handling in [7].

3.1 Core Language

APPEL is a family of policy languages with a common core. As illustrated in figure 4, the core language provides a structure for policies without commitment to any particular application domain. The core language is then specialised for each domain by defining its particular triggers, conditions and actions. At present two APPEL derivatives have been defined, one for call control and one for call conflict resolution. Others are currently under development.

The core APPEL language is specified in [31]. As APPEL is defined by an XML schema, a call policy document requires a wrapping of the form:

```
<?xml version="1.0" encoding="UTF−8"?>
<policy_document xmlns:xsi="http://www.w3.org/2001/XMLSchema−instance"
 xsi:noNamespaceSchemaLocation="http://www.cs.stir.ac.uk/schemas/appel_call.xsd">
 ...
</policy_document>
```

(Resolution policies conform to the *appel_resolution* schema.) For brevity, this wrapping plus the obvious XML closing tags are omitted in the sample policies given in this paper.

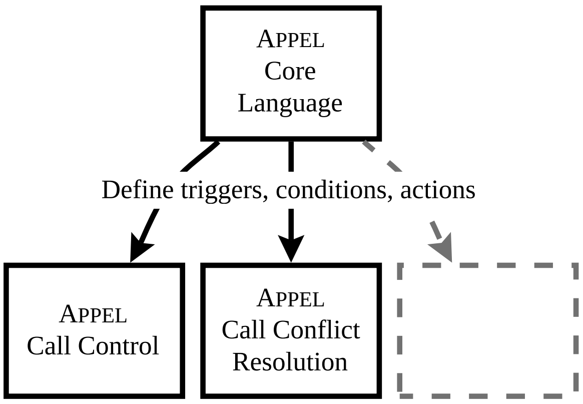

APPEL Core Language
Define triggers, conditions, actions
APPEL Call Control
APPEL Call Conflict Resolution

Fig. 4. APPEL Core and Derived Languages

Simple policies use <**trigger**>, <**condition**> or <**action**> elements. Where a combination of these is required, the plural form of the tags is used. Following this, a combination operator and a pair of elements are given. If more than two elements need to be combined, they are progressively grouped in pairs.

Policy elements are required to have a fixed name so that XML validation can be performed. If an element has parameters, these have to be written as argument place-holders like *arg1* or *arg2*. The argument values are then written as attributes. As an example the *fork_to* action takes one parameter, written as:

<**action** arg1=″anne@home.co.uk″>fork_to(arg1)

APPEL is intended as a general language for expressing policies in a variety of application domains. The core language is therefore cleanly separated from its specialisations. Unlike many policy languages, APPEL is designed for end users rather than technicians or administrators. This has significantly influenced the design of the language, e.g. it is closer to natural language than to programming. The motivation was to ensure that policies could readily be formulated and understood by ordinary users.

Policies have owners and apply to domains. These are the same when a person defines individual policies. However it is possible for an administrator to define policies that apply to others, typically in the same organisation. The owner is always a person, identified by an email-like address (e.g. anne@cs.stir.ac.uk). The domain to which a policy applies may be an individual, a symbolic name for a group of individuals, or a list of both. Individuals may belong to several domains.

The approach supports generic policies that are instantiated as required. This allows a policy administrator to define a range of re-usable policies that can be easily adapted by end users. For example, a policy to forward calls on no answer requires only the forwarding address and the timeout to be defined. APPEL also supports policies that are parameterised by policy variables. This allows the user to vary behaviour by defining the variables independently of the policies. For the same example, the forwarding address and the timeout could be defined by variables rather than fixing them in the policy. Apart from being simpler for a novice user, this allows a single change (e.g. in the forwarding address) to apply to a range of policies.

Generic policies are directly supported by the policy wizard as policy templates. Although a range of template policies is predefined, this can be modified for use within an organisation. This lends itself to vertical markets, i.e. classes of business applications. For example, different suites of policies might be provided for use in medical practices or in legal offices. The same approach also allows for localisation, so policies may differ from country to country according to local practice.

Following the focus on end users, APPEL optionally allows a simple preference to be associated with a policy: *must*, *should* or *prefer* (plus the negative forms of these). Preferences come into play only when conflicts have to be resolved.

A policy document defines one or more policy rules. The applicability of a rule depends on a number of factors: whether it is activated, whether it falls within its period of validity, whether it matches the current user profile, whether its trigger has occurred, and whether its conditions are satisfied. An inapplicable rule is simply ignored.

Policy rules may be composed in various ways: unguarded (unconditional), guarded (conditional), sequential (use first applicable rule) , or parallel (try rules concurrently). A rule body contains an optional trigger, an optional condition, and a compulsory action. Triggers are caused by external events from the system interface or the context system. Omitting a trigger means that a rule does not need an explicit event to occur; such a rule is a goal. A triggered rule must have its conditions satisfied for it to execute. Omitting a condition means that only a trigger is needed to enable the rule. If both the trigger and the condition are omitted, the action can be executed without a trigger. However, the validity period for the policy may delay the action.

Triggers may be combined using *and* and *or*. Conditions may be combined with *and, or* and *not*. Actions are the outcome of a policy, and are sent to the system interface for execution. Actions may be composed in various ways: *and* (both executed in a system-defined order), *andthen* (both executed in the given order), *or* (one executed by system choice), *orelse* (the first executed if permitted), or *else* (subject to the preceding condition).

3.2 Policy Language for Call Control

Figure 5 shows the triggers, condition parameters and actions defined for call control. Most of the elements here should be understandable, but see [31] for a detailed explanation. These are interrelated in that only certain combinations of triggers, conditions and actions are permissible. As an example, a *present* trigger establishes the date, time and user location. A policy with this trigger may refer to these conditions, and may invoke actions such as making a connection or sending a message. The policy wizard enforces such restrictions. It also deals with combinations of triggers: *and* forms the union of the permitted conditions and actions, while *or* forms their intersection.

3.3 Sample Policies for Call Control

The use of APPEL has been illustrated elsewhere [18,31,37]. Only some brief examples are therefore given here as a context for understanding the conflict resolution policies discussed in section 3.5.

Fork Incoming Calls: Anne wishes to be called both at the office and at home. However, she does not feel strongly about this and so omits a preference. Incoming calls to the office are therefore also forked her home address, i.e. both are tried.

```
<policy owner=″anne@cs.stir.ac.uk″ applies_to=″anne@cs.stir.ac.uk″
 id=″Fork Incoming Calls″ enabled=″true″ changed=″2005-12-24T11:20:05″>
  <policy_rule>
    <trigger>connect_incoming
    <action arg1=″anne@home.co.uk″>fork_to(arg1)
```

Forward On Unavailable: While Anne is unavailable, incoming calls should be forwarded to Bert. The empty argument for *unavailable* means the current user is unavailable, i.e. busy. Presence and availability events on their own can trigger a policy. When used in conjunction with another trigger, they are implicitly and automatically generated according to the user's status.

```
<policy owner=″anne@cs.stir.ac.uk″ applies_to=″anne@cs.stir.ac.uk″
 id=″Forward On Unavailable″ enabled=″true″ changed=″2005-12-24T11:38:16″>
  <preference>should
  <policy_rule>
    <triggers>
      <and/>
      <trigger>connect_incoming
      <trigger arg1=″″>unavailable(arg1)
    <action arg1=″bert@cs.stir.ac.uk″>forward_to(arg1)
```

Video For Outgoing Calls: Anne must have video for any calls she makes.

```
<policy owner=″anne@cs.stir.ac.uk″ applies_to=″anne@cs.stir.ac.uk″
 id=″Video For Outgoing Calls″ enabled=″true″ changed=″2005-12-24T16:25:47″>
  <preference>must
  <policy_rule>
    <trigger>connect_outgoing
```

Trigger	**Condition Parameters**	**Actions**
–	date, day, time	note_availability, note_presence, send_message
absent	date, day, time	log_event, note_presence, send_message
available	date, day, time, topic	connect_to, log_event, note_availability, send_message
bandwidth_request	bandwidth, callee, caller, date, day, medium, network_type, time	confirm_bandwidth, reject_bandwidth
connect, connect_incoming, connect_outgoing, no_answer, no_answer_incoming, no_answer_outgoing	active_content, bandwidth, call_content, call_type, callee, caller, capability, capability_set, cost, date, day, destination_address, device, location, medium, network_type, priority, quality, role, signalling_address, source_address, time, topic, traffic_load	add_caller, add_medium, add_party, fork_to, forward_to, log_event, note_availability, note_presence, play_clip, reject_call, remove_medium, remove_party, send_message
disconnect, disconnect_incoming, disconnect_outgoing	callee, caller, date, day, medium, network_type, time	log_event, note_availability, note_presence, play_clip, send_message
event	caller, date, day, network_type, time, topic	note_availability, note_presence, send_message
present	date, day, location, time	connect_to, log_event, note_presence, send_message
register, register_incoming, register_outgoing	caller, date, day, network_type, time	note_presence, reject_call
unavailable	date, day, time	log_event, note_availability, send_message

Fig. 5. Triggers, Condition Parameters and Actions for Call Control

```
    <action arg1="video">add_medium(arg1)
```

Still Unavailable After Call: A user might normally be considered available after a call ends. Anne prefers to stipulate explicitly when she is available, so on call disconnection she notes herself as still unavailable. This is the meaning of the empty argument for *note_availability*.

```
<policy owner="anne@cs.stir.ac.uk" applies_to="anne@cs.stir.ac.uk"
 id="Still Unavailable After Call" enabled="true" changed="2005-12-24T13:18:01">
  <preference>prefer
  <policy_rule>
    <trigger>disconnect
    <action arg1="">note_availability(arg1)
```

3.4 Policy Language for Call Control Conflict

3.4.1 Detecting Conflicts

Policy conflicts may arise statically (when policies are defined) or dynamically (when policies are executed). Although the design of the policy server allows for both, the focus of the work reported here has been on dynamic conflicts. In fact this is a much more demanding task, partly because the relevant policies cannot be determined in advance, and partly because conflict detection resolution has to work in real time.

Conflicts are handled by resolution policies, distinguished from regular policies. Resolution policies are higher-level policies that deal with clashes among policy actions. An important design issue was to externalise the handling of conflicts. Detection and resolution are therefore defined outside the policy server, and are not built into it. Apart from handling conflicts in a more transparent manner, this allows conflicts to be dealt with in a localised way. Although there is a predefined set of resolution policies, these may be varied according to the needs of the organisation.

A resolution policy defines what conflict means, and specifies how to resolve it. Conflict handling is specific to an application domain. In some cases, conflict detection could be generic. For example, an *add* action will probably conflict with a *remove* action for the same parameters. However this is not inevitable, and could depend on the domain. For example H.323 allows additional video codecs, but H.261 must be supported by everyone. Videoconferencing software typically allows users to select codecs. Suppose the users have contradictory policies about which ones to use. It is not necessary to handle this as a policy conflict, since the underlying network will manage the negotiation.

For this reason, APPEL does not have an in-built notion of conflict. Instead, all conflicts must be explicitly defined. Conversely, if two actions are not defined as conflicting then they are regarded as compatible. Although this makes the approach more flexible and generic, it potentially means extra effort to identify conflicts. To alleviate this, a library of predefined resolution policies is provided. This can be adapted and extended for local use with little extra work.

Of course, this begs the questions of what should be considered a conflict and how such conflicts should be resolved. Since conflicts arise from actions, then all pairwise combinations of these need to be considered. The resolution approach guarantees that only pairs need to be considered; in feature interaction, some three-way interactions are known. Many conflict-prone pairs can be identified mechanically, e.g. *add-add* or *add-remove* for related actions. However, it requires human judgment to identify whether these really are conflicts and how best to resolve them. More seriously, some conflicts involve combinations of apparently unrelated actions. As a case in point, consider the example given later of adding a caller and adding video to a call. The identification of conflicts can therefore be only semi-automated. However, this is also true in the more mature area of feature interaction.

In call control (figure 5), there are 16 possible actions and therefore $\frac{16\times16}{2} = 128$ possible pairs. Some of these (*log_event*, *send_message*) are compatible with all other actions, while some (*note_availability*, *note_presence*) can conflict only with each other. In practice, there are thus about 50 combinations that would need to be examined. It is believed that a degree of automation is possible, though human judgment must be the final arbiter. Future work will include a means of semi-automating this analysis (similar to interaction filtering for features, e.g. [23]).

Conflict detection is defined to be commutative and associative. If action 1 conflicts with action 2, then action 2 conflicts with action 1. The way in which action 1, action 2 and action 3 are combined does not affect the conflict outcome. The policy server exploits this when it checks a set of proposed actions against the resolution policies. Since resolutions may be defined in various user domains, they are (partially) ordered by domain before they are applied. This ensures that higher-domain resolutions (e.g. for stir.ac.uk) are applied in preference to lower-domain ones (e.g. for cs.stir.ac.uk).

When dealing with conflicts in the call domain, the policy system can take advantage of the possibility to play voice clips. For example, a user whose policy is overridden can be informed that this happened and why. However, care is needed to ensure that privacy is not breached. Suppose Anne has a policy that she does not wish to receive calls from Bert. If Bert is thwarted in calling Anne, it could be embarrassing to Anne if he were told exactly why! This is similar to the Terminating Call Screening feature. Network operators are careful to block the caller without revealing why. Because of the greater flexibility of policies, a wider range of responses may be given. For example colleagues might be told that their callee will next be available at lunch-time, but an external caller might just be told that the person is unavailable.

Another interesting possibility, not currently implemented, is that the conflict resolution system can learn what resolutions are acceptable. If Bert's call to Anne is rejected due to policy conflict, the system log will record this. A user might even get to 'vote' by dialling a digit from 0 to 9, indicating the degree of satisfaction with an outcome.

3.4.2 Resolving Conflicts

The triggers of a resolution policy are the actions of regular policies. Often, just a pair of triggers is used such as *add_medium* and *remove_medium*. However, multiple triggers may be used as required. For example, conflict may be defined to arise on adding video to the call, adding a third party to the call, and forwarding the call.

A resolution policy explicitly binds the parameters of actions to resolution variables named *variable* and numbered 1 to 9. The preferences associated with the corresponding policies are implicitly bound to resolution variables named *preference* and numbered 1 to 9. The conditions of a resolution policy are typically based on these resolution variables. However, the conditions that are used in regular policies may also appear in resolution policies.

A resolution aims to take a set of conflicting actions and replace them with compatible actions. In order to avoid infinite regress, there is a single level of resolution. That is, resolution policies are not considered to conflict with each other. The policy server applies the first resolution policy that is enabled by the actions (i.e. is triggered by the actions and has satisfied conditions). If overlapping resolutions in the same domain are defined, this will not be noticed. Automated discovery of such potential problems will be addressed in future work.

The actions of a resolution policy may be generic or specific. Examples of both are given later. A generic resolution decides among the conflicting actions. It may choose among the actions on the following basis:

- *apply_callee* or *apply_caller*
- *apply_older* or *apply_newer*, based on the policy definition time
- *apply_inferior* or *apply_superior*, based on the policy domain (e.g. cs.stir.ac.uk is inferior to stir.ac.uk)
- *apply_negative* or *apply_positive*, based on the policy preference (e.g. *must_not* is negative)
- *apply_weaker* or *apply_stronger*, based on the policy preference (e.g. *should_not* is weaker than *must*, and *should* is weaker than *must_not*)

There are also some generic resolutions mainly intended for internal use by the policy server. A generic resolution may not result in a definite outcome (e.g. *apply_superior* will not eliminate actions from policies in the same domain). If resolution is unsuccessful, the policy server uses the *apply_default* action to achieve some resolution. This is a last-resort strategy that tries *apply_stronger*, and then *apply_newer* if that does not achieve resolution. If a unique action is still not obtained, one is chosen at system discretion. Having to do this is logged as a warning to the system administrator that the resolution policies are incomplete.

Resolution policies commonly deal with pairs of actions, checking their parameters and the policy preferences. In general this requires consideration of four cases: equal/similar and equal/opposite (same action parameters, preferences in a similar or opposite sense), unequal/similar and unequal/opposite (different action parameters, preferences in a similar or opposite sense).

As will be seen in the examples of section 3.5, sometimes all four cases need to be specified explicitly and sometimes fewer. If a resolution does not deal with all four cases, the others are handled implicitly. It may be that no resolution is required because the actions are compatible. For example, each user can be allowed to fork a call to a different address. When resolving conflicts, the policy server does not perform actions with negative preferences (e.g. the action 'must not add video' is ignored if generated by conflict resolution).

Preferences are internally mapped to integers: *must* (+3), *should* (+2), *prefer* (+1), empty or 'don't care' (0), *prefer_not* (-1), *should_not* (-2), *must_not* (-3). Preferences may therefore be ranked by the usual comparison operators (e.g. *lt*, *ge*). However, it is usual to employ two operators that make a broader comparison: *in* (read as 'in keeping with') and *out* (read as 'out of keeping with'). Positive and negative preference values are considered to be opposites. A zero value is similar to a positive or a negative value. Thus *must* is in keeping with *should* or empty, and is out of keeping with *prefer_not* or *must_not*. Similarly *must_not* is in keeping with *should_not* or empty, and is out of keeping with *prefer* or *must*.

3.5 Sample Policies for Call Control Conflict

Fork-Fork Conflict – Generic Resolution: Almost any call control action may conflict with itself if its arguments are the same and the preferences of each party are opposite. As an example, suppose one party wishes to fork the call to an alternative address (e.g. to try a home number in addition to the dialled office number). However, suppose the other party does not wish to fork the call to this address (e.g. because the callee must be called only in the office). The following detects this conflict, and resolves it through a generic action: choosing the stronger of the two preferences. Note that this policy applies to a domain (@cs.stir.ac.uk) rather than to the owner. In general, an administrator can define regular and resolution policies for groups of users identified in this way.

<**resolution** id="Call Fork-Fork Conflict"
  owner="admin@cs.stir.ac.uk" applies_to="@cs.stir.ac.uk" enabled="true"
  changed="2005-12-24T15:40:00">

```
<policy_rule>
   <triggers>
      <and/>
      <trigger arg1=″variable1″>fork_to(arg1)
      <trigger arg1=″variable2″>fork_to(arg1)
   <conditions>
    <and/>
    <condition>
      <parameter>variable1
      <operator>eq
      <value>variable2
    <condition>
      <parameter>preference1
      <operator>out
      <value>preference2
   <action>apply_stronger
```

This resolution explicitly deals with only the equal/opposite case. The equal/similar case is not explicitly handled since the preferences are compatible: 'must fork to address *A*' and 'should fork to address *A*' will result in forking to *A* since one of the two equivalent actions will be selected by default. The unequal/opposite case is not explicitly handled since the actions do not conflict: 'must fork to address *A*' and 'should not fork to address *B*' will result in forking to only *A* since actions with negative preferences are not performed. The unequal/similar case does not need explicit description: 'must fork to address *A*' and 'should fork to address *B*' will result in forking to both *A* and *B* since both actions are compatible. Similar resolutions could be defined for pairs of *add_caller*, *add_party*, *add_medium*, etc.

Forward-Forward Conflict – Generic Resolution: Call forwarding is another example of a call control action conflicting with itself. However, the resolution is more complex. There is conflict if the forwarding addresses are the same and the preferences are opposite (equal/opposite case), or if the forwarding addresses differ and the preferences are similar (unequal/similar case). The generic resolution given here is to apply the caller's preference.

```
<resolution id=″Call Forward-Forward Conflict″
 owner=″admin@cs.stir.ac.uk″ applies_to=″@cs.stir.ac.uk″ enabled=″true″
 changed=″2005-12-24T14:51:20″>
  <policy_rule>
     <triggers>
        <and/>
        <trigger arg1=″variable1″>forward_to(arg1)
        <trigger arg1=″variable2″>forward_to(arg1)
     <conditions>
        <or/>
        <conditions>
           <and/>
        <condition>
          <parameter>variable1
          <operator>eq
          <value>variable2
        <condition>
          <parameter>preference1
          <operator>out
          <value>preference2
      <conditions>
        <and/>
        <condition>
          <parameter>variable1
          <operator>ne
          <parameter>variable2
        <condition>
          <parameter>preference1
          <operator>in
          <value>preference2
    <action>apply_caller
```

The equal/similar and unequal/opposite cases are handled implicitly. Similar resolutions could be defined for pairs of *note_availability*, *note_presence*, *reject_call*, etc.

Medium Add-Remove Conflict – Generic Resolution: A number of call control actions are inverses of each other, and are an obvious source of conflict. For example a conflict arises if one party wishes to add a digital whiteboard during the call, while the other party wishes to omit this. The following checks if the medium in question is the same for both actions, and whether the associated preferences are similar. If so, this policy selects the weaker preference as a generic resolution. (This choice is just for illustration, as an example of favouring less aggressive policies!)

```
<resolution id=″Medium Add-Remove Conflict″
 owner=″admin@cs.stir.ac.uk″ applies_to=″@cs.stir.ac.uk″ enabled=″true″
 changed=″2005-12-24T13:29:1″>
  <policy_rule>
    <triggers>
      <and/>
      <trigger arg1=″variable1″>add_medium(arg1)
      <trigger arg1=″variable2″>remove_medium(arg1)
    <conditions>
      <and/>
      <condition>
        <parameter>variable1
        <operator>eq
        <value>variable2
      <condition>
        <parameter>preference1
```

```
            <operator>in
            <value>preference2
        <action>apply_weaker
```

This resolution explicitly deals with only the equal/similar case. The equal/opposite case is not explicitly handled; 'must add medium *M*' and 'should not remove medium *M*', for example, will result in adding *M* since actions with negative preferences are not performed. The unequal cases are handled implicitly. Similar resolutions could be defined for *add_party* vs. *remove_party*, *confirm_bandwidth* vs. *reject_bandwidth*, etc.

Bandwidth Confirm-Reject Conflict – Specific Resolution: This example is a straightforward conflict: one party wishes to confirm the requested bandwidth, while the other wishes to reject the request. This time the resolution is specific: the bandwidth request is confirmed, and the conflict is noted in an event log. Although *variable2* is set to the reason for rejecting the bandwidth request, it is not in fact used here.

```
<resolution id="Bandwidth Confirm-Reject Conflict"
 owner="admin@cs.stir.ac.uk" applies_to="@cs.stir.ac.uk" enabled="true"
 changed="2005-12-24T17:41:32">
   <policy_rule>
      <triggers>
         <and/>
         <trigger>confirm_bandwidth
         <trigger arg1="variable2">reject_bandwidth(arg1)
      <condition>
         <parameter>preference1
         <operator>in
         <value>preference2
      <actions>
         <and/>
         <action>confirm_bandwidth</action>
         <action arg1="Overruled bandwidth conflict by confirming it">log_event(arg1)
```

The resolution explicitly deals with only the equal/similar case. Other cases are handled implicitly.

Caller Add-Medium Add – Specific Resolution: Suppose one party wishes to add video to the call, while the other wishes to include a third party in the call (*add_caller*). This might be considered undesirable, since the third party would be able to view the call parties and their workplaces. The resolution is specific: allow both actions, but conference in cath@cs.stir.ac.uk to oversee the call (*add_party*). Note that the triggers and actions are all of different types.

```
<resolution id="Caller-Medium Add-Add Conflict"
 owner="admin@cs.stir.ac.uk" applies_to="@cs.stir.ac.uk" enabled="true"
 changed="2005-12-24T11:40:00">
```

```
<policy_rule>
  <triggers>
    <and/>
    <trigger arg1="variable1">add_caller(arg1)
    <trigger arg1="variable2">add_medium(arg1)
  <conditions>
    <and/>
    <condition>
      <parameter>variable2
      <operator>eq
      <value>video
    <condition>
      <parameter>preference1
      <operator>in
      <value>preference2
  <actions>
    <and/>
    <actions>
      <and/>
      <action arg1="variable1">add_caller(arg1)
      <action arg1="variable2">add_medium(arg1)
    <action arg1="cath@cs.stir.ac.uk">add_party(arg1)
```

The resolution explicitly deals with only the equal/similar case. Other cases are handled by default.

4 Conclusions

A policy system has been created to support a call policy language and its associated conflict handling. The work has concentrated on call control in Internet telephony. The software is written in Java and has proven to be very portable. Interfaces have been created between the policy system and servers for SIP (MKC 7000 ICS and SER softswitches) and H.323 (Gnu GK gatekeeper). The policy system has been demonstrated in conjunction with a variety of systems, including the PSTN (Public Switched Telephone Network), PBXs (Private Branch Exchanges), mobile phones, wireless PDAs (Blackberry), and email servers.

Considerable efforts have been put into making the policy system usable by ordinary end users. The policy wizard has been the focus of this goal. Although the run-time performance of the policy system has not yet been formally assessed, empirical studies have shown that the use of policies adds only a small overhead to call processing – under a second to retrieve policies, detect and resolve conflicts, and control the underlying communications system in accordance with these policies.

The policy system has been demonstrated to several industrial audiences. Three companies are currently considering adapting it for their product lines. Although use by non-technical people has so far been limited, it is believed that the right ingredients are present to make it usable. The system architecture is designed for scalability in the number of users and in the deployment of physical systems to support policies.

Future work will enhance the current implementation in various ways:

- More extensive usability trials will be conducted. These will include a formal assessment of performance and scalability. Currently this is awaiting industrial commitment, since only telephony companies have access to a representative user base.
- The emphasis in conflict handling has been on dynamic (online) aspects. Although static (offline) conflict handling has been studied, it is still to be implemented.
- It is possible to define overlapping resolution policies. Automated determination of such problems will be addressed.
- Resolution policies are defined manually, though the predefined resolutions are a good basis for this. A semi-automatic method will be investigated for determining conflict-prone combinations of policy actions.
- Policy refinement is being studied as a means of realising higher-level policies (particularly goals) using lower-level policies. Planning techniques from Artificial Intelligence are the preferred approach. This will help to bridge the gap between policies and features.
- Although conflict resolution has been designed for the case of multiple policy servers [7], it is currently implemented only for the single-server case.

The core policy language has been defined to be extensible, and has been instantiated for call control and for call conflict handling. However, this is currently achieved by manual editing of the core language schema. In follow-on work, the elements of the policy language are defined using OWL (Web Ontology Language). Apart from generalising the approach, this allows automated extension of the policy system into new domains. Although the policy server is very largely domain-independent, this is not true of the policy wizard (which currently has an intimate knowledge of call control). The policy wizard has been adapted to read the policy elements from an OWL description of the domain.

Currently, the policy system is tied into SIP and H.323 Internet telephony. Both of these are open international standards. An obvious possibility would be to integrate the policy system with Skype (*www.skype.com*). This is an Internet telephony solution that has rapidly gained popularity; see [4] for an analysis of Skype. However Skype is proprietary and closed, making it difficult to integrate with new techniques.

The policy work is also being extended into two new areas. One of these concerns policies for control of sensor networks – specifically in wind farms. The other is applying policies to management of technology and network services that deliver care to users in the home. It is hoped that the flexibility demonstrated by the work so far will carry over into these novel domains.

Acknowledgements

The work reported here was funded by EPSRC (the UK Engineering and Physical Sciences Research Council, grant R31263) and by Mitel Networks (Canada). The authors are very grateful to Peter Perry and Tom Gray of Mitel Networks for their technical advice and support throughout. Integration of the policy system with the 7000 ICS softswitch was made possible through the active involvement of Joe Ireland, Peter Musgrave, Ryan Waters and their colleagues at MKC Networks. The authors also thank their co-workers on ACCENT: Stephan Reiff-Marganiec and Jianxiong Pang. Mario Kolberg (Stirling) and Grégory Estienne (Ottawa) contributed to the development of multi-language support in the policy wizard.

References

[1] S. Aljareh and N. Rossiter. Towards security in multi-agency clinical information services. In R. N. Procter and M. Rouncefield, editors, *Proc. Dependability in Healthcare Informatics*, pages 33–41, UK, Mar. 2001. University of Lancaster.

[2] M. Amer, A. Karmouch, T. Gray, and S. Mankovskii. Feature interaction resolution using fuzzy policies. In M. H. Calder and E. H. Magill, editors, *Proc. 6th. Feature Interactions in Telecommunications and Software Systems*, pages 94–112. IOS Press, Amsterdam, Netherlands, May 2000.

[3] M. Barbuceanu, T. Gray, and S. Mankovskii. How to make your agents fulfil their obligations. In H. S. Nwana and D. T. Ndumu, editors, *Proc. 3rd. Conference on Practical Application of Intelligent Agents and Multi-Agents*, pages 255–276, London, UK, Mar. 1998.

[4] S. A. Baset and H. Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. Technical Report CUCS-039-04, Computer Science Columbia Universit, New York, May 2004.

[5] A. Belokosztolszki and K. Moody. Meta-policies for distributed role-based access control systems. In J. B. Michael, J. Lobo, and N. Dulay, editors, *Proc. 3rd. International Workshop on Policies for Distributed Systems and Networks*, pages 106–115. IEEE Computer Society, Los Alamitos, California, USA, June 2002.

[6] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A system to specify and manage multipolicy access control models. In J. B. Michael, J. Lobo, and N. Dulay, editors, *Proc. 3rd. International Workshop on Policies for Distributed Systems and Networks*, pages 116–127. IEEE Computer Society, Los Alamitos, California, USA, June 2002.

[7] L. Blair and K. J. Turner. Handling policy conflicts in call control. In S. Reiff-Marganiec and M. D. Ryan, editors, *Proc. 8th. Feature Interactions in Telecommunications and Software Systems*, pages 39–57. IOS Press, Amsterdam, Netherlands, June 2005.

[8] R. J. A. Buhr, D. Amyot, M. Elammari, D. Quesnel, T. Gray, and S. Mankovski. Feature-interaction visualization and resolution in an agent environment. In K. Kimbler and W. Bouma, editors, *Proc. 5th. Feature Interactions in Telecommunications and Software Systems*, pages 135–149. IOS Press, Amsterdam, Netherlands, Sept. 1998.

[9] M. H. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks*, 41:115–141, Jan. 2003.

[10] E. J. Cameron, N. D. Griffeth, Y.-J. Lin, M. E. Nilson, W. K. Schnure, and H. Velthuijsen. A feature-interaction benchmark for IN and beyond. *IEEE Communications Magazine*, pages 64–69, Mar. 1993.

[11] E. J. Cameron and H. Velthuijsen. Feature interactions in telecommunications systems. *IEEE Communications Magazine*, pages 18–23, Aug. 1993.

[12] N. Damianou, N. Dulay, E. C. Lupu, and M. Sloman. Ponder: A language specifying security and management policies for distributed systems. Technical report, Imperial College, London, 2000.

[13] 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*, 45(5):585–603, Aug. 2004.

[14] T. Dursun and B. Örencik. POLICE: A novel policy framework. Number 2869 in Lecture Notes in Computer Science, pages 819–827. Springer, Berlin, Germany, 2003.

[15] C. Efstratiou, A. Friday, N. Davies, and K. Cheverst. Utilising the event calculus for policy driven adaptation on mobile systems. In J. B. Michael, J. Lobo, and N. Dulay, editors, *Proc. 3rd. International Workshop on Policies for Distributed Systems and Networks*, pages 13–24. IEEE Computer Society, Los Alamitos, California, USA, June 2002.

[16] N. Gans, G. Koole, and A. Mandelbaum. Telephone call centers: Tutorial, review, and research prospects. *Manufacturing and Service Operations Management*, 5:79–141, Sept. 2002.

[17] T. Huang. Policies for H.323 internet telephony. Technical Report CSM-165, Department of Computing Science and Mathematics, University of Stirling, UK, May 2005.

[18] T. Huang and K. J. Turner. Policy support for H.323 call handling. *Computer Standards and Interfaces*, 28(2):204–217, Nov. 2005.

[19] J. Lennox and H. Schulzrinne, editors. *Call Processing Language Framework and Requirements*. Internet Draft CPL-Framework-02. The Internet Society, New York, USA, Jan. 2000.

[20] E. C. Lupu and M. Sloman. Conflict analysis for management policies. In *5th. International Symposium on Integrated Network Management*, pages 430–443. Chapman-Hall, London, UK, 1997.

[21] E. C. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE Trans. on Software Engineering*, 25(6):852–869, Nov. 1999.

[22] D. Marriott, M. Mansouri-Samani, and M. Sloman. Specification of management policies. In *Proc. 5th. IFIP/IEEE Int. Workshop on Distributed Systems: Operations and Management*, 1994.

[23] M. Nakamura, T. Kikuno, J. Hassine, and L. M. S. Logrippo. Feature interaction filtering with Use Case Maps at requirements stage. In M. H. Calder and E. H. Magill, editors, *Proc. 6th. Feature Interactions in Telecommunications and Software Systems*, pages 163–178. IOS Press, Amsterdam, Netherlands, May 2000.

[24] L. Pearlman, V. Welch, I. Foster, and C. Kesselman. A community authorization service for group collaboration. In J. B. Michael, J. Lobo, and N. Dulay, editors, *Proc. 3rd. International Workshop on Policies for Distributed Systems and Networks*, pages 50–59. IEEE Computer Society, Los Alamitos, California, USA, June 2002.

[25] A. Ponnappan, L. Yang, and R. Pillai. A policy based QoS management system for the IntServ/DiffServ based Internet. In J. B. Michael, J. Lobo, and N. Dulay, editors, *Proc. 3rd. International Workshop on Policies for Distributed Systems and Networks*, pages 159–168. IEEE Computer Society, Los Alamitos, California, USA, June 2002.

[26] S. Reiff-Marganiec. Policies: Giving user control over calls. In M. D. Ryan, J.-J. C. Meyer, and H.-D. Ehrich, editors, *Objects, Agents and Features*, number 2975 in Lecture Notes in Computer Science, pages 189–208. Springer, Berlin, Germany, May 2004.

[27] S. Reiff-Marganiec and K. J. Turner. Use of logic to describe enhanced communications services. In D. A. Peled and M. Y. Vardi, editors, *Proc. Formal Techniques for Networked and Distributed Systems (FORTE XV)*, number 2529 in Lecture Notes in Computer Science, pages 130–145. Springer, Berlin, Germany, Nov. 2002.

[28] S. Reiff-Marganiec and K. J. Turner. A policy architecture for enhancing and controlling features. In D. Amyot and L. Logrippo, editors, *Proc. 7th. Feature Interactions in Telecommunications and Software Systems*, pages 239–246. IOS Press, Amsterdam, Netherlands, June 2003.

[29] S. Reiff-Marganiec and K. J. Turner. Feature interaction in policies. *Computer Networks*, 45(5):569–584, Aug. 2004.

[30] S. Reiff-Marganiec and K. J. Turner. The ACCENT policy server. Technical Report CSM-164, Department of Computing Science and Mathematics, University of Stirling, UK, Dec. 2005.

[31] S. Reiff-Marganiec, K. J. Turner, and L. Blair. APPEL: The ACCENT project policy environment/language. Technical Report CSM-161, Department of Computing Science and Mathematics, University of Stirling, UK, Dec. 2005.

[32] W. N. Robinson, S. D. Pawlowski, and V. Volkov. Requirements interaction management. *ACM Computing Surveys*, 35(2):132–190, 2003.

[33] T. Ryutov and C. Neuman. The specification and enforcement of advanced security policies. In J. B. Michael, J. Lobo, and N. Dulay, editors, *Proc. 3rd. International Workshop on Policies for Distributed Systems and Networks*, pages 128–138. IEEE Computer Society, Los Alamitos, California, USA, June 2002.

[34] K. E. Seamons, M. Winslett, T. Yu, B. Smith, E. Child, J. Jacobson, H. Mills, and L. Yu. Requirements for policy languages for trust negotiation. In J. B. Michael, J. Lobo, and N. Dulay, editors, *Proc. 3rd. International Workshop on Policies for Distributed Systems and Networks*, pages 68–79. IEEE Computer Society, Los Alamitos, California, USA, June 2002.

[35] M. W. A. Steen and J. Derrick. Formalising ODP enterprise policies. In *Proc. 3rd. International Enterprise Distributed Object Computing Conference*. Institution of Electrical and Electronic Engineers Press, New York, USA, Sept. 1999.

[36] K. J. Turner. The ACCENT policy wizard. Technical Report CSM-166, Department of Computing Science and Mathematics, University of Stirling, UK, Dec. 2005.

[37] K. J. Turner, S. Reiff-Marganiec, L. Blair, J. Pang, T. Gray, P. Perry, and J. Ireland. Policy support for call control. *Computer Standards and Interfaces*, June 2005. In press.

[38] A. van Lamsweerde, R. Darimont, and E. Letier. Managing conflicts in goal-driven requirements engineering. *IEEE Trans. on Software Engineering*, 24:908–926, 1998.

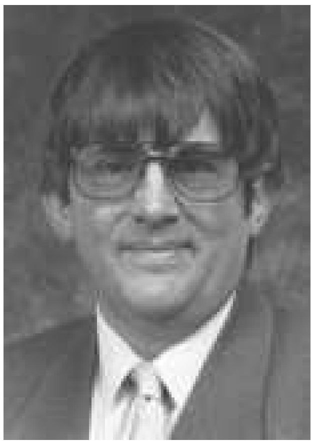
[39] D. C. Verma, S. Calo, and K. Amiri. Policy-based management of content distribution networks. *IEEE Network*, pages 34–39, 2002.

[40] T. Wagner, J. Shapiro, et al. Multi-level conflict in multi-agent systems. In *Proc. AAAI Workshop on Negotiation in Multi-Agent Systems*, 1999.

[41] X. Wu and H. Schulzrinne. Programmable end system services using SIP. In *Proc. International Conference on Communications 2003*. Institution of Electrical and Electronic Engineers Press, New York, USA, May 2003.

[42] R. Yavatkar, D. Pendarakis, and R. Guerin. *A Framework for Policy-Based Admission Control*. RFC 2753. The Internet Society, New York, USA, July 2000.

Author Bibliographies



Ken Turner holds a B.Sc. in Electrical Engineering and a Ph.D. in Artificial Intelligence. After 12 years working in the communications industry, he became Professor of Computing Science in 1987 at the University of Stirling, Scotland. His research interests include policy-based management, communications services, formal methods, and systems architecture. He conducts research in areas such as distributed systems, networking, healthcare and hardware design. His teaching interests include communications, compiler design, programming and software engineering.

Lynne Blair is a senior lecturer in the Computing Department at Lancaster University. She has a background in the formal specification and verification of distributed multimedia systems. Currently her research interests focus on issues of interaction that occur in software systems and aspect-oriented software development. Of particular interest is research into dynamically adaptive systems, especially the development of such systems via dynamic aspect-oriented techniques.