Policies: Giving Users Control over Calls

Stephan Reiff-Marganiec

University of Leicester[*],
Department of Computer Science,
Leicester LE1 7RH, UK
srm13@le.ac.uk

Abstract. Features provide extensions to a basic service, but in new systems users require much greater flexibility oriented towards their needs. Traditional features do not easily allow for this. We propose policies as the features of the future. Policies can be defined by the end-user, and allow for the use of rich context information when controlling calls. This paper discusses an architecture to allow for policy definition and call control by policies and describes the operation of a system based on this architecture. One aspect is policy definition, the APPEL policy description language that serves this purpose. An important aspect of the architecture is integral feature interaction handling. It is in this last aspect that we foresee a role for agents, and hope that this paper will stimulate some collaboration between the two mostly distinct research areas of feature interaction and agent technologies.

1 Motivation

Telecommunication has gained a central role in our daily life, be it private or within the enterprise. We have come a long way from just connecting two users and allowing them to have a verbal conversation. Over recent years we encountered a trend towards merging different communication technologies such as video conferencing, email, Voice over IP as well as new technologies like home automation and device control. This is combined with a move to greater mobility, e.g. wireless communications, mobile telephony, *ad hoc* networking, as well as traditional landlines. *Features* such as conference calling and voice mail have been added to help users deal with situations beyond simple telephony.

Currently such features only support communication, that is they allow the user to simplify and more closely integrate telecommunication in their activities. In the future, features will make use of the merged technologies on any device. We believe that features then will enable communications by allowing the user to achieve particular goals.

From this we conclude that the end-user must have a central place in communication systems. An obvious extension to this is that any feature must be highly customizable by end-users, as it is only the individual who is aware

[*] The material published in this paper has been developed in the ACCENT project during the authors previous employment at the University of Stirling.

of his requirements. It might even be desirable to allow the end-user to define his own features. However, end-users are usually not computer experts, so any customization or feature development must be simple and intuitive.

Policies have been used successfully in system administration and access control as a way for users to express detailed preferences. We believe that policies written by – and thus under the control of – individual users provide the functionality required.

We discuss the advantages of policies over features, and define an architecture in which policies can be used to control calls. We will find that policies do not remove the feature interaction problem, and hence conflict detection and resolution mechanisms will form an essential part of the proposed system. Resolution of conflicts at runtime requires flexible mechanisms, such as those that *agent* technologies can provide. The system is user-oriented and addresses the fact that future call control needs to enable individuals to achieve their goals.

We provide the background to this work in section 2, in particular introducing the key concepts: features, policies, feature interaction and policy conflict. In section 3 we propose an architecture and discuss its components. Sections 4 and 5 are used to define the operation of a system built on the architecture. We then return to discuss the link between policies and agents in section 6. Finally we summarise the approach and indicate further work in section 7.

2 Background

2.1 Policies and Features

Policies are defined as *information which can be used to modify the behaviour of a system* [16]. We interpret *modify* in this definition, to include specification of default behaviour, restriction of the system and enhancements of the functionality. Considerable interest in policies has developed in the context of multimedia and distributed systems. Policies have also been applied to express the enterprise viewpoint [22] of ODP (Open Distributed Processing) and in agent-based systems [4].

In telecommunication systems, customized functionality for the user is traditionally achieved by providing features, i.e. capabilities that have been offered on top of a basic service. Features are imperative by nature, in contrast to policies which are descriptive. Features are supplied by the network provider and thus do not offer completely customized functionality. Consider a call forwarding feature: the user choices are whether the feature is available or not, and which telephone number the call gets forwarded to.[1] There is no possibility for the user to forward only some calls or to treat certain calls differently, e.g. forward private calls to the mobile and others to a voicemail facility.

However, recent technological advances mean that users are connected nearly anywhere and nearly all the time. In this context, users should have much

[1] Note, however, that large organisations can have "call plans" that allow for routing based on time and location.

more control over their communications. Users should be able to define their availability, that is when they are available to receive communications and which ones. However, a usable mechanism for this must go much further than static rules; it must take the current context of the user into account. After all, the goal of new features should be to provide communications that are least intrusive to the user, but at the same time trying to satisfy all parties involved.

It is exactly the flexibility, adaptability and end-user definition of policies that makes them an ideal candidate technology for features of the future.

2.2 Policies and Feature Interaction

One might hope that policies would remove the feature interaction problem, simply by being higher-level. However, this would be far too optimistic, and in fact interaction problems are a harsh reality even at the more abstract level. The policy community has recognised that there are issues, which they refer to as *policy conflict*, but has not considered any practical solutions (the best effort to classify the problem being [16]). Most people in the policy community appear to think that this is a minor problem and will simply vanish, an opinion shared by many others outside the feature interaction community. We will use the terms policy conflict and feature interaction interchangeably, as essentially the two refer to the same problem but in two different domains.

The difference between policy conflict and feature interaction is twofold. The future will see many more policies than we have seen features. This is due to policies being defined by end-users. In addition to the increase in the number of policies, users are not aware of the feature interaction problem, or at least are generally not expert enough to anticipate problems. It should be obvious that this only increases the size of the feature interaction problem. It requires more workable, automatic solutions to detecting and resolving interactions. On the other hand, policies can contain preferences and the context can also provide priorities which can be used to resolve conflicts. The new communication architectures provide richer underlying network technologies that more readily allow information exchange between parties. Thus one could argue that the feature interaction problem is actually easier to deal with in new systems.

2.3 Feature Interaction and Agents

One of the strong aspects of agents is their ability to cope collaboratively with unexpected events in the environment due to some of their basic criteria: autonomy, pro-activeness, reactivity, collaboration and negotiation. To achieve this agents use standard AI techniques derived from knowledge engineering and machine learning, but add distributed coordination and negotiation facilities. An introduction to intelligent agents is provided in [27]. The occurrence of feature interaction is an unexpected event in a distributed environment, so agent technology might provide solutions here.

There is some work in the feature interaction community discussing the role of agents. Features and resources are represented by agents able to communicate with each other to negotiate their goals; successful negotiation means that an interaction has been resolved. The *a priori* information required is hidden in the concept of successful negotiation – it must be known when goals interfere.

In an early paper [25], Velthuijsen evaluated a number of distributed artificial intelligence techniques (DAI) to help solve the feature interaction problem. Several approaches have since been developed using the techniques. For example, Griffeth and Velthuijsen use negotiating agents to detect and resolve interactions in [12]. A resolution is a goal acceptable by all parties, and is achieved by exchanging proposals and counter-proposals amongst the agents. Different methods for negotiation have been envisioned: direct (agents negotiate directly without a negotiator), indirect (a dedicated negotiator controls the negotiation and can propose solutions based on past experience) and arbitrated (an arbitrator takes the scripts of the agents and has sole responsibility to find a solution). Griffeth and Velthujsen concentrate on indirect negotiation. The approach has been implemented in the experimental platform "Touring Machine" [3], although no conclusive report about the success is provided.

Rather than using direct negotiation, Buhr et al. [6] make use of a blackboard. Features are represented by agents which exchange information by writing to a public data space. Other agents can change the information written to the blackboard and a common goal can be negotiated. Amer et al. [1] also use the blackboard technique, but extend their agents to make use of fuzzy policies. Agents set truth-values (0 to 100) to express the desirability of certain goals. These values are then adapted as the call progresses, depending on the values of other agents. In the case of a conflict, an event with the highest truth-value is executed.

The suggested approaches are promising. But the existing telecommunication network architectures do not provide feature-to-feature communication. The approaches require a significantly different network architecture. We will discuss such an architecture in the policy context and consider the link between agents and policies later in section 6.

3 The Policy Architecture

In [21] we proposed a three layer architecture consisting of (1) the communications layer, (2) the policy layer and (3) the user interface layer. We will discuss the relevant details of the architecture again in this paper. A three-tier architecture is used in completely different ways for other applications. However, a three-tier policy architecture emerges naturally.

This architecture provides a number of key aspects, the details of which we discuss throughout the paper. With reference to Fig. 1:

- The Policy Servers can communicate with each other to negotiate goals or solutions to detected problems. This is not possible in existing communication architectures. It is this element that makes the use of agent technologies possible.

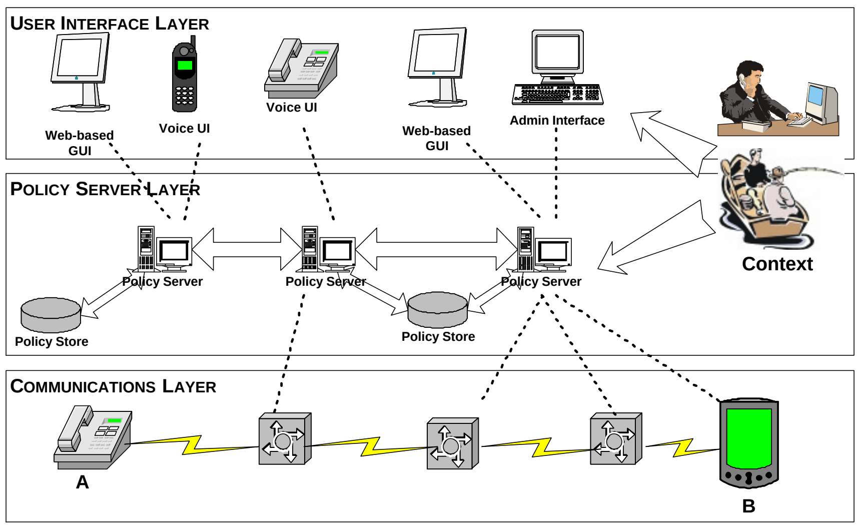

Fig. 1. Overview of the Proposed Architecture

– The User Interface Layer provides end-users with a mechanism to define functionality using a policy definition language. Such a mechanism does not exist in traditional telecommunication architectures, but recent architectures such as SIP (the Session Initiation Protocol) allow for caller preferences. A similar layer could be used in agent architectures to define the agent's goals.
– The User Interface Layer and the Policy Servers make use of context information (e.g. the user is in the office) to provide new capabilities. To our knowledge this has not been done before in the communications domain, but is somewhat natural to agent technologies where the environment influences the behaviour.
– The Policy Server layer is independent of the underlying call architecture. That means that we can work across several communication architectures if required. Advances at the communications layer will not adversely impact on the higher layers.
– All intra-layer signalling can be achieved by methods based on protocols such as SOAP (Simple Object Access Protocol).

The architecture makes the underlying communication network transparent to the higher layers. It enables end-user configurability and provides communication between policy servers which can be used for interaction handling. We will briefly discuss next the details of each layer.

3.1 The Communications Layer

The communications layer represents the call architecture. Details are not relevant for this paper. However, we impose two crucial requirements on the communications layer. (1) The policy servers must be provided with any message that arrives at a switching point; routing is suspended until the policy server has dealt with the message. (2) A mapping of low-level messages from the communications layer into more abstract policy terminology must be defined (usually developed by a domain expert, and in its simplest form is a dictionary).

In this paper we assume that both requirements are fulfilled. Our investigations have shown that this is a realistic assumption.

3.2 The Policy Server Layer

The policy server contains a number of policy servers that interact with the underlying call architecture. It also contains a number of policy stores, that is database or tuple space servers where policies can be stored and retrieved by the policy servers as required. We assume that several policy servers might share a policy store, and also that each policy server might control more than one switching point or apply to more than one end device in the communications layer. One user's policies are stored in a single policy store. For reliability replication could be considered (however this is outside the scope of the current work).

The policy servers interact with the user interfaces in the policy creation process as will be discussed in section 4. They also interact with the communications layer where policies are enforced; this will be discussed in detail in section 5.

A further role of the policy server is to detect and resolve (or suggest resolutions of) conflict among policies. To enable richer resolution mechanisms, communication between policy servers for information exchange is permitted. Further, the policy servers have access to up-to-date information about the user's context details which are used to influence call functionality.

3.3 The User Interface Layer

The user interface layer allows users to create new policies and deploy these in the network. A number of interfaces can be expected here: graphical or text-based interfaces, voice-controlled mechanisms, or administrative interfaces, each providing functionality targeted at certain types of users and devices.

The normal user will use a text-based interface for most functions. However mobile users might not have the capability to use such technology, so voice-controlled interfaces are more appropriate. These also suit users that simply want to activate or deactivate policies. A voice interface is essential for disabled or partially sighted users. Both text and voice interfaces should guide the user in an intuitive way, preferably in natural language or in a graphical fashion. We discuss a text-based interface in more detail in section 4. We also envision libraries of policy elements that users can simply adapt to their requirements and combine to obtain the functionality required, in a similar way that for example clip-art libraries are common today.

The administrative interfaces are system-oriented and exist mainly for system administrators to manage more complex functionality.

The policy definition environment provides access to contextual variables which are instantiated at runtime by the policy servers with actual context information. For example, "secretary" in a forwarding policy could be filled in from a company organisation chart in conjunction with holiday and absence information. Also, the location of a user could be determined from an active badge system.

4 Defining Policies

Policies should provide the end-user with capabilities to get the most out of their communication systems. End-users usually use their communication devices in a social or commercial context that imposes further policies. For example an employee is often subject to company policies. Hence policies will be defined at different domain levels (users, groups, companies, social clubs, customer premises, etc.) by differently skilled users. Any policy definition process needs to take this into account.

4.1 A Policy Description Language

In previous work [20] we have introduced initial ideas for a policy description language (PDL) that allows us to express call control policies. Here we present more details on APPEL (the Accent Project Policy Environment/Language).[2] We have analysed a set of more than 100 policies before defining a language that is able to express these. Further, any traditional feature can also be expressed.

The syntax of APPEL is defined by an XML grammar which we present using a graphical notation. We will only provide an informal insight into the semantics where appropriate.

A policy document (`policydoc`, see Fig. 2) forms the highest level in which one or more policies are encapsulated.

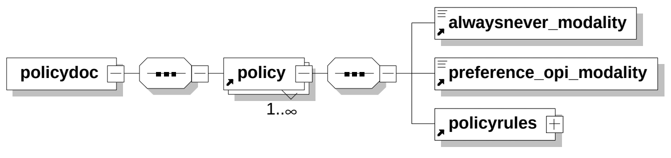

Fig. 2. The Policy Document Element and its immediate Children

[2] appel – from French: a call

A `policy` is the object of highest interest in the policy description language. A policy has a number of attributes, whereby `owner` highlights the person that the policy belongs to and `appliesTo` highlights the person/group that the policy applies to. Here one could argue that the two are mostly the same. However, in an enterprise context this might not be so: a policy might have been created (and be owned by) a system administrator but applies to one or more people in the enterprise. Further each policy can be enabled or not, determining whether the system should apply the policy.

A policy contains a `policyrules` element and might contain two kinds of modalities. Note that the user interface will typically also provide for temporal modalities – however these are encoded as conditions. A policy might specify an always/never modality. If always is specified it is assumed that the policy actions should be applied. If never is specified the user intends that the actions do not occur. Policies might also have a preference or obligation modality. These modalities will be used when policy conflict is detected and must be resolved. We have currently not specified a weight for the preference modalities.

A policy contains one or more policy rules (`policyrules`), where the policy rules are concatenated using one of four operators: guarded choice, unguarded choice, parallel application or sequence. All these operators are binary; larger chains can be achieved by appropriate nesting of policy rules.

When evaluating policies, it is relevant to be able to determine if a policy applies and if so, what actions should be taken. This depends fundamentally on the meaning of the combination operators. We discuss this next:

Guarded choice. When two policy rules are joined by the guarded choice operator, the execution engine will first evaluate the guard. If the guard evaluates to "true", the first of the two rules will be applied, otherwise the second. Clearly once the guard has been evaluated it is necessary to decide whether the individual rule is applicable and whether there is no conflict that prohibits the enforcement. The guard could be emulated by suitable conditions on the level of a policyrule, however there are uses for having guards at the higher level.

Unguarded choice. Unguarded choice provides more flexibility, as both parts will be tested for applicability. If only one of the two policy rules is applicable, this will be chosen. If both are applicable, the system can choose one at random (the current implementation will select the first).

Sequential composition. With sequentially composed policy rules, the rules are enforced in the specified order. That is we traverse the structure, determining whether the first rule is applicable. If so, we apply the first rule, and then move on to check the second rule. Note that the second rule will only be checked if the first rule is applicable.

Parallel composition. Parallel composition of two rules allows for a user to express an indifference w.r.t. the order of two rules. Both rules are enforced, but the order in which this is done is not important. An example would be a rule that leads to the attempt to add a video channel and the logging of the fact that this is attempted: we would allow the system to resolve the order of the two actions.

A policy rule (see Fig. 3) defines the triggers and conditions that need to hold for the action to be applied. We allow for policy rules that do not have trigger events (so-called goals) and also for rules that do not specify conditions. However, an action always needs to be specified.

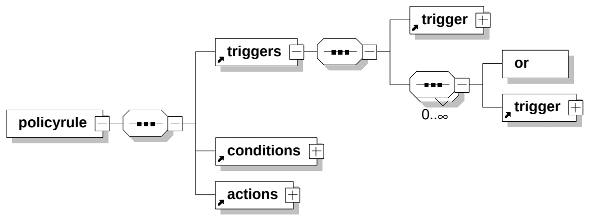

Fig. 3. The PolicyRule Element and its immediate Children

A policy might be activated by one or more triggers. The semantics here is that if any of the stated triggers occurs during enforcement, the policy is considered to have been triggered – whether the action should then be applied depends on the conditions provided. Goals are always considered to be triggered.

The enforcement of a policy usually means that the actions stated within the policy are applied to the underlying system. A user can specify a single action or a number of actions. If a number of actions is specified it is important to determine their application order. Four operators are provided for this purpose; we discuss their respective meaning next.

And specifies that the policy should lead to the execution of both actions. However, the order is not specified. This means that in general we will execute `action1` and `action2` in any order. The current implementation will execute the first action first, but this is only a design decision and one could execute the actions in parallel.

Or specifies that either one of the actions should be taken. The current implementation will assume that `action1` will be taken. Note that "or" is important when a conflict is detected: it provides the system with an alternative action to be taken which might not be conflicting.

Andthen is a stronger version of "and", as here the order is prescribed in that `action1` must precede `action2` in any execution.

Orelse is the "or" operator with a prescribed order. This specifies that a user requires `action1` to be tried first.

We have discussed the actions and triggers elements of a policy rule; now we consider conditions. A condition might be a simple condition or a more complex combination of `conditions` (see also Fig. 4). In the latter case, conditions can be combined with the usual boolean "and", "or" and "not" operators. These operators have their usual meaning.

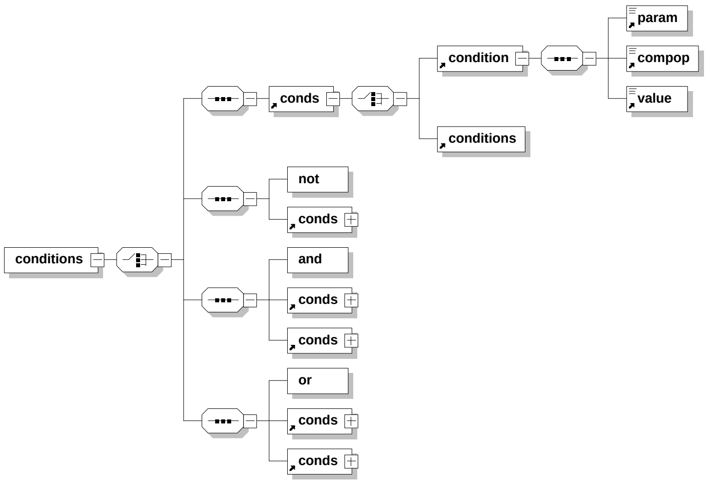

Fig. 4. The Conditions Element and its immediate Children

A `condition` is a simple test where a parameter is compared to a value using a number of operators. The operators are predefined in the grammar, but might have subtly different meaning depending on the arguments they are applied to. For example `in` applied to time might mean that the actual time falls within an interval, but `in` applied to a domain object might mean that the entity is in the domain. Conditions in policies refer to concepts from the context of the user/call. These concepts are encapsulated in parameters. It is possible to add further parameters.

Note that while the language has been developed in the context of call control systems it is thought to be more generally applicable. The only aspects of the language that are domain-specific are the available parameters, triggers and actions. However, these are clearly separated and thus can be readily adapted to other domains.

4.2 User Interfaces

The presented language is quite expressive. However, we do not expect the end user to define policies directly in the policy description language (i.e. raw XML. This would be unrealistic, as we expect the user to be less knowledgable and certainly not to be a computer expert. In order to make the policy language usable, we have developed a simple wizard that aids the user in creating and handling policies (currently a prototype). Also, this wizard is specific to policies in the call control context, but can be adapted to other domains. The interface is web-based, so users can access it from anywhere. The users are authenticated, and depending on their credentials will be allowed more or less possibilities in the interface. In general the interface provides functionality to create new policies, edit or delete existing ones as well as enabling and disabling policies that are currently deployed.

Once the user submits a change request the system gathers the relevant information. For creation and editing of policies, the XML representation of the policy is generated. The policy identifier, and if required the XML version, are submitted to the policy server. We consider the details in section 4.3.

It is intended that further interfaces will be added. In particular we will investigate VoiceXML based interfaces to allow users to handle policies on devices such as mobile phones where speech is the most appropriate input medium. At the lowest level, administrators might encode policies directly in XML using off-the-shelf tools.

4.3 Upload of Policies

A policy server provides an interface to receive changes to policies (with appropriate authorisation). The current implementation receives simple text messages on this interface, although we consider using SOAP [26] for future enhancements. The user interface layer uses this interface to submit the gathered information and to receive any feedback on success or failure of the operation.

In the absence of the feature interaction problem, all that would be required now is to insert the new policy into the policy store or to update the existing policy. This is exactly what happens if no conflict is detected. However as we have indicated, conflicts are rather likely. Hence, before inserting policies into the store a consistency check is performed on the set of policies applicable to the user. If a conflict is detected the change is not committed to the policy store, but rather an informative response is sent to the user. Such a message will state that there was a problem and suggest possible solutions, if they are available.

4.4 Interactions

It would be desirable if the policy architecture would not give rise to conflicts. However, this holy grail has not been achieved in many years of feature interaction research. It appears to be possible only at the cost of reducing the expressiveness of the policy language to trivial levels. We have taken a more realistic stance by introducing a guided design process that automatically checks policies for the presence of conflict and presents any detected problems to the user, together with suggested resolutions.

When the user has attempted to upload a new policy this is checked against other policies from the same user, but also against policies that the user might be subject to (e.g. due to her role in the enterprise). We can however only check for static interactions, i.e. those that are inherent to the policies independent of the contextual data. This suggests the use of offline detection methods and filtering techniques, which have been developed by the feature interaction community.

Any inconsistencies detected need to be reported to the user. The resolution mechanism is typical of offline methods – a redesign of the policy base. We consider the following methods most appropriate for the policy context, although an implementation is required to determine which is most suitable.

Anise [24] applies pre-defined or user-defined operators to build progressively more complex features. This neatly maps onto the philosophy of the policy definition language. The composition mechanism is able to detect certain forms of interaction arising from overlapping triggers. Zave and Jackson's [28] Pipe and Filter approach detects conflict of features that occur in specific, pre-defined order. An interesting approach is presented by Dahl and Najm [9] which allows for ready identification of interactions as occurrence of common gates in two processes. Thomas [23] follows the more common approach of guarded choices. The occurrence of non-determinism highlights the potential for interactions. Available tools allow the automatic detection of non-determinism, and these tools can be incorporated into the policy server. Many of the off-line techniques require analysis. This is done by either checking the models against specified properties (e.g. feature and system axioms) or by finding general properties such as deadlock, livelock or non-determinism (or a combination of the three). Again automatic tools for detecting these problems can be incorporated as part of the policy server. Note that these methods are applied at policy definition time, so execution times are less of an issue (as long as they stay within reason).

In fact we are not restricted to static interactions: we can also detect the potential for conflict. That is, we can filter cases where an interaction might occur, depending on the particular instantiation of contextual data. A suitable approach might be derived from the work of Kolberg *et al.* [14] Here the user has a choice of redesigning the policy or simply to accept that conflict might occur and let the enforcement process handle it when it occurs. In the latter case some additional information might be included in the policies to indicate the potential conflicts to guide any online resolution technique. The question, as to what information is required remains unanswered in our work so far.

One might suggest that the consistency check be performed at the user's device rather than at the policy server – a valid point which has been considered. Performing the check in the policy server has the major advantage that all details of the user's policies, as well as those from the same domain, are accessible and can be considered without transferring all policies to the user's device. Furthermore the user side of the implementation is kept light-weight, which is important for less powerful end-devices such as mobile phones or PDAs.

4.5 A Worked Example

To illustrate what we said in this section, consider the following example with a typical user, say Alice.

Alice already has a policy that asks to "forward calls to my voicemail in evenings". However, she is expecting an important call and sets up a new policy that specifies "forward calls to my mobile after 16:00".

To set up the policy, she will chose an appropriate policy type *forward calls to target*, which will allow her to set *target* to "mobile", simply by selecting values from lists. She then can add an extra condition, again by choosing a prototype *when it is later than time* and refining time to "16:00".

On receiving an upload command, the policy server checks the policies for consistency. Under the intended semantics, Alice's new policy conflicts with her existing policy: 'in the evening' both apply and they specify different forward targets. A possible solution that might be suggested is that Alice disables her usual evening policy temporarily to allow for the exceptional circumstances.

5 The Call Process

In the previous section we have discussed how users can define their policies and upload them to the policy server. In this section we discuss how policies are enforced to achieve the goals they describe.

5.1 Applying Policies to a Call

Let us consider a typical call setup. When attempting to set up a call from A to B, A's end device will generate a message that is sent to A's switching point (or proxy). From there the message gets forwarded through a number of further switching points until it reaches B's end device.

If we assume that policy servers are associated with switching points, then at every switching point that the message reaches it is sent to a policy server. Routing is suspended until the policy server allows continuation, and only then is continued with respect to the response from the policy server. In SIP, we can intercept, modify, fork and block messages by using SIP CGI [15]. It is easy to extract the complete message and pass it to the policy server for processing.

Once the policy server has obtained a message it needs to process it. The general process is to retrieve all applicable policies and to change the message as required. If there are no applicable policies, the message is simply returned to the proxy unchanged for onward routing. More interestingly, if one or more policies apply the required changes can be made to the original message. Alternatively new or additional messages can be created. The example in Fig. 5 shows "progress info" messages that are generated by the policy server. Also, if a policy requires forking of a call (e.g. "always include my boss in calls to lawyers") the respective messages to set-up the extra call leg need to be created.

Any messages returned to the proxy server are then routed onwards to the next proxy server, where the same process of forwarding the message to a policy server and application of policies is performed. Figure 5 shows an example of this process for a call between two parties where policies are enforced in one direction. We can also have scenarios where the policy at the remote end affects

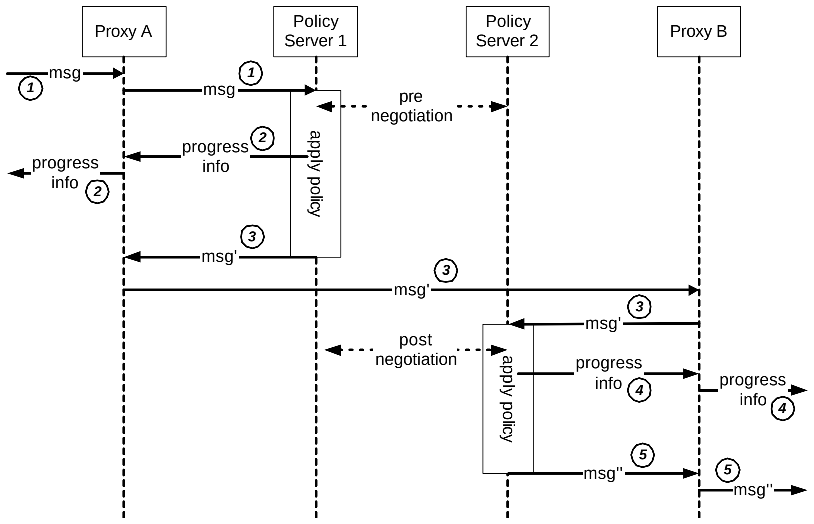

Fig. 5. Policy Enforcement Process

the originating end. Note that the figure shows two message exchanges between the policy servers, labelled pre- and post-negotiation. These allow for resolution of conflicts; we return to them later in section 5.3.

The process of applying a policy is somewhat complex. It requires the received message to be analysed, then the retrieval of any applicable policies as well as the retrieval of any required context information. Conflicts of policies must be determined, and new messages must be generated and sent.

It is impractical to require the user to always provide the relevant information when establishing a call, but this is not necessary. Most of the required information can be inferred from the context. For example, roles may be defined in a company organisation chart, the location can be established from the user's diary or mobile home location register. Or better, "mobile devices have the promise to provide context-sensing capabilities. They will know their location – often a critical indicator to the user's tasks." [11]. From the location, further facts can be derived, depending on the quality of the location information, GPS does not allow fine grained positioning, but smart badges could precisely locate users inside buildings. Hence, from you being in your boss's office, it can be derived that you are probably in an important meeting and do not wish to be disturbed. When having lunch with friends or colleagues, a lecturer might not wish to be disturbed by students. Note that we are not concerned with privacy issues at this moment.

The analysis of the messages must extract the relevant information, such as sender and recipient, as well as any other information that might occur in policies. Here it is important to point out that this involves a translation of communications layer messages into policy events. Clearly, the amount of call status information that can be used is dependent on what the underlying communications layer supports. For example, in the IN the topic of a call is usually not identifiable and hence cannot be used. Messages that are issued from the policy server to the communications layer require a reverse translation, that is policy events need to be converted into communications layer messages.

A number of policies can be applicable in any scenario. All goals and those policies relating to the trigger event can be activated by the call, provided that the sender or target of the message applies to the user of the policy.

In a next step further conditions must be evaluated. Usually this requires the context variables in the conditions to be instantiated with current values extracted from context information. Those where the conditions are not satisfied can be discarded from the set of applicable policies.

The remaining set of policies needs to be analysed for conflicts, and these must be resolved before the response messages can be generated. We consider an example before reconsidering interactions.

5.2 A Worked Example

We return to Alice for a further example. We now also have a company competition.com with three employees Paul, John and Janine.

Alice regularly speaks to Paul but also knows John. She prefers to speak to people she knows, so she has a policy that states "I prefer to speak to John if Paul is busy". But Paul also has a policy to handle redirection when he is busy: "I expect my calls to be redirected to Janine when I am busy".

Alice rings Paul while Paul is busy, this causes a conflict as Alice prefers to be forwarded to John rather than Janine. This conflict is somewhat new with respect to feature interactions, it is caused by a caller preference. This was not possible with traditional features.

For a resolution we can see several options here. One could always give precedence to caller expectations (after all, they are usually the paying party) – in which case Alice gets her wish. We also have preferences in the policies, and here Alice has expressed "prefer", whereas Paul has expressed a rather stronger concept of "expect" – thus, Paul will be granted his goal.

Another option is to start a negotiation process. Details of how to do this are not clear at this point. It is here that we hope for agent technologies to provide solutions; we return to this when discussing interactions in the next section.

5.3 More Interactions

In the example we have seen an interaction that only occurred while a call was actually taking place. Any detection mechanism incorporated in the enforcement part of the policy server needs to be able to detect and resolve such conflicts. The introduction of policy support must also not create unreasonable delays in call setups. However, if users are aware that complex steps are needed to resolve sophisticated policies, they should learn to live with the delays – and probably are happy to do so when the outcome is productive for them (hence the intermediate information messages are produced by the policy application process).

On-line and hybrid feature interaction approaches could provide possible solutions. A special category of on-line techniques, so called hybrid techniques, provides information gathered by an off-line phase to improve the on-line technique. This is of particular relevance when the available information at run-time is too limited to resolve detected interactions. However, we believe that in a policy context the available information is sufficient to resolve conflicts. The underlying architecture provides protocols that are rich enough to facilitate exchange of a wide range of information. There are essentially two classes of run-time approaches. One is based around the idea of negotiating agents, the other around a feature manager. Examples of the former are presented by Velthuijsen [25] and Buhr et al. [6]. Examples of the latter are presented by Cain [7], Marples and Magill [17], and Reiff-Marganiec [19].

In the feature manager approach a new entity is included in the call path, a so-called feature manager that can intercept events occurring in the system (such as messages sent, access to resources, etc.). Based on this information, the detection and resolution of conflict is possible. In [7], the feature manager knows specific undesired behaviour patterns and potential resolutions. The feature manager in [17] detects interactions by recognising that different features are activated and wish to control the call. The resolution mechanism for this approach [19] is based on general rules describing desired and undesired behaviour.

Feature manager approaches lend themselves to the policy architecture, as their main requirement is that the feature manager is located in the call path. This is naturally the case with policy servers. However, feature manager approaches so far have suffered from a lack of information to resolve interactions (though some progress has been reported in [19]).

Negotiation approaches have been considered in section 2. Their handicap in current architectures is the sophisticated exchange of information required which is not supported. However, our architecture provides for this by allowing communication channels between policy servers.

Two forms of negotiation are practical in the policy architecture: pre- and post-negotiation. In the former case, the policy server contacts the policy server at the remote end and negotiates call setup details to explore a mutually acceptable call setup. The communications layer is then instructed to set up the call accordingly. In post-negotiation, the policies are applied while the call is being set up, thus potentially leading to unrecoverable problems.

We hope that the agent community can provide solutions for negotiation in this context – or help with their development. Clearly, negotiation requires the exchange of information. It is here that we have a number of immediate points: it must be researched what information needs to be exchanged for negotiations to be successful. Also, as we are dealing with personal policies it might be undesirable to make information available to the other party. Clearly issues of privacy (which could be seen as "appropriate sharing" rather than hiding of information) need to be considered in any solution. Also, any approach developed will need to be part of the production system and hence must be efficient and scalable.

6 Policies and Agents

This short section simply serves the purpose of bringing the discussion from the background full circle. We have earlier described how policies and features are related, as we discussed the relation between feature interaction and agents and have also (throughout the paper) highlighted the similarities between feature interaction and policy conflict. Here we will summarise the relation between policies and agents.

We can consider the relation between policies and agents in two ways: *how can agents help with policy conflict?* and *can policies be useful for agents?* We will not attempt to provide definite answers to either question, but rather give our general view – in this sense the following is rather speculative.

In section 5.3 we have discussed the role that we foresee for agents in the policy architecture. Our hope is that agents will provide the solution to the negotiation required at runtime when a policy conflict is detected, we called this post-negotiation. We have also indicated that agents might attempt to prevent conflict, by determining outcomes before the actual actions are committed. We referred to this as pre-negotiation.

In the former case it is required of the agents to exchange the relevant details to come to a solution. However, note that we are not dealing with just two agents: there might be many more with each representing an entity that is interested in the call. In particular there are the end-users, but there might also be companies that impose rules and network providers that have to ensure that the communication network remains stable. One could envision decentralised negotiation or, if this is more suitable, centralised negotiation via some trusted "black board" or other entity. However, we can see that negotiation is relatively independent from the underlying call architecture as all that must be known about the network is the entities that have an interest in the negotiation.

The latter case does make matters somewhat more complex: a closer integration between the call architecture and the agents is required. In addition to what was said above the entities that have an interest in the communication must be found. While a remote end of the call (usually) knows the route the call took to reach it, the originating end cannot know the route. This is largely because the obvious route might not be appropriate in the context of policies that change the route (for example a forwarding policy).

However, we can also see that a policy framework and language (like APPEL) might be useful for users to describe their goals for agents. In general agents are acting on behalf of a user to achieve goals that the user wishes to achieve. However, as far as the author understands, these goals are often encoded in the agent in a way that is similar to what features provide for telecommunications.

For example, a search-bot has the goal of finding information regarding a certain query, and it is the query that can be specialised by the user. We think that user defined policies could provide a means of making agents more generic and dynamically reconfigurable. In particular they could be controlled by high-level goals set by the user. These goals will then automatically provide a framework for negotiation, as all policies together specify what is and what is not acceptable for the user. This requires for the policy language to be connected to agent ontologies in a generic way.

We believe that both aspects require some further investigation, which might best be conducted by teams composed of members of both the agents and the feature interaction communities.

7 Conclusion and Further Work

7.1 Evaluation

We have considered how policies can be used in the context of call control, especially how they can be seen as the next generation of features. The policy architecture allows calls to be controlled by policies. Each policy might make use of the context of a user. This allows context-oriented call routing, but goes far beyond routing by allowing for availability and presence of users to be expressed. In this way we can achieve truly non-intrusive communications that enable users to achieve their goals.

Policies can be easily defined and changed. We have introduced the APPEL policy description language. More importantly, changes and definition of new policies can be performed by the end-user via a number of interfaces. Web and voice interfaces have been discussed.

We have suggested some off-line techniques from feature interaction for policies to be checked for consistency when they are designed. They can be applied to new policies as well as to existing ones where detailed information is available (e.g. within the same domain). However, calls will eventually cross domains and then it is not possible to determine the policies that might conflict due to the sheer number of policies and users; the actual connections between users cannot be predicted. This requires that we make use of on-line techniques to resolve any such issues. To detect conflicts automatically, both off-line and on-line methods require an understanding of the conditions and actions to determine when policies are invoked simultaneously and when their actions lead to inconsistencies.

7.2 Future Work

A prototype environment to create and enforce policies has been developed on top of a SIP architecture. Thus the proposed architecture has been implemented.

The methods for detecting and resolving policy conflict identified in this paper need to be implemented in the prototype such that empirical data on their suitability can be gathered. That is, policy servers need to be equipped with mechanisms that can perform static consistency checking of uploaded policies. They also require mechanisms to detect and resolve conflicts at run-time. It is here that we envision the use of agent technologies. We have discussed some of the requirements and open questions in section 5.3.

In the future we would like to test the top layers on top of other systems. For example, in the telecommunication domain we are considering H.323, but we would also like to move the concept outside the telecommunication domain and have planned an investigation in the context of web services. Further development of additional user interfaces should strengthen the prototype. Another research area is the automatic gathering of context details, which is interesting in itself but beyond the scope of our work.

Other ideas for further work, include the linking of agent ontologies to the policy language in a generic way (as suggested in section 6). Also, the issue of privacy has not been addressed in this paper. For the acceptance of the system, secure storage and controlled access to user policies and profiles is highly important. Also, legal reasons prescribe for any system containing such private information to have a clear privacy "policy".

Acknowledgements

This work has been supported by EPSRC (Engineering and Physical Sciences Research Council) under grant GR/R31263 and Mitel Networks Corporation. The work has been conducted as part of the ACCENT project during the authors previous employment at the University of Stirling. I thank all people who contributed to the discussion of policies in this context. Particular thanks are due to colleagues at Mitel Networks Corporation and at Stirling University, in particular to Ken Turner who strongly influenced this work. Thanks are also due to Jennifer Paterson for implementing the policy wizard.

References

1. M. Amer, A. Karmouch, T. Gray, and S. Mankovskii. Feature interaction resolution using fuzzy policies. In *[8]*, pages 94–112, 2000.
2. D. Amyot and L. Logrippo, editors. *Feature Interactions in Telecommunications and Software Systems VII.* IOS Press (Amsterdam), 2003.
3. M. Arango, L. Bahler, P. Bates, M. Cochinwala, D. Cohrs, R. Fish, G. Gopal, N. Griffeth, G. E. Herman, T. Hickey, K. C. Lee, W. E. Leland, C. Lowery, V. Mak, J. Patterson, L. Ruston, M. Segal, R. C. Sekar, M. P. Vecchi, A. Weinrib, and S. Y. Wuu. The Touring Machine System. *Communications of the ACM*, 36(1):68–77, 1993.
4. M. Barbuceanu, T. Gray, and S. Mankovski. How to make your agents fulfil their obligations. *Proceedings of the 3rd International Conference on the Practical Applications of Agents and Multi-Agent Systems (PAAM-98)*, 1998.
5. L. G. Bouma and H. Velthuijsen, editors. *Feature Interactions in Telecommunications Systems.* IOS Press (Amsterdam), 1994.

6. 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 *[13]*, pages 135–149, 1998.
7. M. Cain. Managing run-time interactions between call processing features. *IEEE Communications*, pages 44–50, February 1992.
8. M. Calder and E. Magill, editors. *Feature Interactions in Telecommunications and Software Systems VI.* IOS Press (Amsterdam), 2000.
9. O. C. Dahl and E. Najm. Specification and detection of IN service interference using LOTOS. *Proc. Formal Description Techniques VI*, pages 53–70, 1994.
10. P. Dini, R. Boutaba, and L. Logrippo, editors. *Feature Interactions in Telecommunication Networks IV.* IOS Press (Amsterdam), 1997.
11. A. Fano and A. Gersham. The future of business services in the age of ubiquitous computing. *Communications of the ACM*, 45(12):83–87, 2002.
12. N. D. Griffeth and H. Velthuijsen. The negotiating agents approach to runtime feature interaction resolution. In *[5]*, pages 217–236, 1994.
13. K. Kimbler and L. G. Bouma, editors. *Feature Interactions in Telecommunications and Software Systems V.* IOS Press (Amsterdam), 1998.
14. M. Kolberg and E. H. Magill. A pragmatic approach to service interaction filtering between call control services. *Computer Networks: International Journal of Computer and Telecommunications Networking*, 38(5):591–602, 2002.
15. J. Lennox, J. Rosenberg, and H. Schulzrinne. Common gateway interface for SIP. *Request for Comments 3050*, Jan 2001.
16. E. C. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE Trans. on Software Engineering*, 25(6):852–869, 1999.
17. D. Marples and E. H. Magill. The use of rollback to prevent incorrect operation of features in intelligent network based systems. In *[13]*, pages 115–134, 1998.
18. D. Peled and M. Vardi, editors. *Formal Techniques for Networked and Distributed Systems – FORTE 2002, LNCS 2529.* Springer Verlag, 2002.
19. S. Reiff-Marganiec. *Runtime Resolution of Feature Interactions in Evolving Telecommunications Systems.* PhD thesis, University of Glasgow, Department of Computer Science, Glasgow (UK), May 2002.
20. S. Reiff-Marganiec and K. J. Turner. Use of logic to describe enhanced communications services. In *[18]*, pages 130–145, November 2002.
21. S. Reiff-Marganiec and K. J. Turner. A policy architecture for enhancing and controlling features. In *[2]*, pages 239–246, June 2003.
22. M. W. A. Steen and J. Derrick. Formalising ODP Enterprise Policies. In *3rd International Enterprise Distributed Object Computing Conference (EDOC '99)*, University of Mannheim, Germany, September 1999. IEEE Publishing.
23. M. Thomas. Modelling and analysing user views of telecommunications services. In *[10]*, pages 168–182, 1997.
24. K. J. Turner. Realising architectural feature descriptions using LOTOS. *Networks and Distributed Systems*, 12(2):145–187, 2000.
25. H. Velthuijsen. Distributed artificial intelligence for runtime feature interaction resolution. *Computer*, 26(8):48–55, 1993.
26. W3C. Simple Object Access Protocol (SOAP) 1.1, `http://www.w3.org/TR/SOAP/`. Visited on 09-12-2002.
27. M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.
28. P. Zave and M. Jackson. New feature interactions in mobile and multimedia telecommunication services. In *[8]*, pages 51–66, 2000.