# Use of Logic to describe Enhanced Communications Services

Stephan Reiff-Marganiec and Kenneth J. Turner

University of Stirling,
Department of Computing Science and Mathematics,
Stirling FK9 4LA, Scotland, UK
{srm,kjt}@cs.stir.ac.uk

**Abstract.** New functionality is added to telecommunications systems in the form of features or services. However, this is a very provider-centric approach, not giving much control to the user. We consider a logic that allows the user to express preferences as to how they wish calls to be handled. This logic is encapsulated in a user-friendly policy description language. The transferability of a policy description language (Ponder) developed for system management and access control is discussed.

**Keywords:** Enhanced Communications Services, Policies, Policy Description Language, Ponder, SIP.

## 1 Introduction

Call processing systems are large, distributed systems. Traditionally, they were either telephone exchanges or private branch exchanges under the control of a single provider or owner. Recent developments have shifted the whole area of call processing towards an open market and a more unified view of communications. We now consider more than POTS (Plain Old Telephone System) calls. Furthermore, the functionality has been largely enhanced by the provisioning of features. Features are extensions to the basic service, developed independently of the basic service. The latest developments allow users to develop and deploy their own features.

In this paper we consider policies as a mechanism for enterprises and individuals to specify their preferences for call processing. Policies are defined as *information which can be used to modify the behaviour of a system* [LS99]. Considerable interest has been aroused by policies in the context of multimedia and distributed systems. Policies have also been applied to express the enterprise viewpoint of ODP (Open Distributed Processing) [SD99] and agent-based systems [BGM98].

### 1.1 Feature Interaction

With the growing number of features that might be developed by different providers, a problem known as the feature interaction problem gains importance.

A feature interaction is encountered when two or more features that each independently work as expected do not do so if provided together in the system. A typical example is a user subscribing to a Call Forwarding on Busy feature (which redirects incoming calls to a busy line to a different phone) and a Call Waiting feature (which plays a tone when a call comes in to a busy line). Assume the user is busy and an incoming call arrives. This raises the question of which feature's behaviour should be activated, as both together do not provide sensible behaviour.

Feature interaction has been considered by an active research community composed of academics and industrialists, since the late 1980s. The proceedings of the series of International Feature Interaction Workshops, e.g. [CM00,KB98] provide a good overview.

### 1.2 New Call Processing Architectures

We have already mentioned the convergence of communications systems. The buzzword often heard from communications technology developers is "everything over IP". Hidden behind this is essentially the fact that data is delivered over IP networks and that the same technology can be used to deliver voice or in fact any mixture of communications media.

There are several aspects of delivering communications media over IP networks, such as session control and quality of service. SIP (Session Initiation Protocol) [HSSR02] is concerned with establishing, adapting and terminating sessions. Mapping traditional telecommunications onto this framework, a call is essentially a session.

Call processing is primarily concerned with session control, and hence call processing policies will influence session control. However, they might range beyond that and stipulate conditions on call content (such as minimal quality requirements).

In SIP, users exchange information about call content, required media and quality upon setup or adaption of a session. Thus, the information required by call processing policies is available in the SIP message exchange.

We distinguish our work from ongoing work on CPL (Call Processing Language) [LS00]. CPL essentially allows users to define traditional features in the SIP environment (although the intention behind CPL was to provide more than that). We see our work at a higher level, although CPL scripts can be seen as a subset of call processing policies.

We return to SIP in section 5, when we consider how call processing policies can be enforced in a communications system.

### 1.3 Logic and Policies for Call Processing

We consider policies in the context of call processing. In this context we consider a policy to be *a high level statement as to how calls should be processed in the system.* This can be seen as a refinement of Lupu et al.'s definition[LS99]:

information is considered to be high-level statements, and the behaviour of the system is processing calls.

Our work is novel in that policies have not been considered much for call processing previously. The notable exception is work by Amer et al. [AKGM00] which considers fuzzy policies to resolve feature interactions. Related work by Buhr et al. [BAE+98] considers the use of OPI (Obligations, Permissions, Interdictions) [BGM98] to improve scalability in a similar (feature-oriented) context. However, the work of Amer et al. does not follow this and instead adapts the techniques developed by Mariott et al. [MMSS94] to handle policies with associated truth values to express fuzziness.

This related work is distinguished from ours in that the authors allow users to define preferences as to how their *subscribed* features behave rather than lifting the user's view completely to policies (i.e. away from features) as we do. In addition policies can express concepts such as presence and availability, neither of which need necessarily follow a trigger-response pattern, as traditional features do.

## 2 Policies for Call Processing

Traditionally features are either *subscribed* to by the users of the system (user features) or added as additional system services (e.g. billing, system features). In contrast, policies are *defined* by the subjects that desire their enforcement. This set of subjects contains many entities: users, system providers, enterprises and roles, as well as agents representing any of these.

In more detail, users are individuals, usually identified with one or more addresses. An example user is Bob, associated with a phone number, but also with his email address and his mobile phone number. Enterprises are collections of users, often with an internal hierarchical structure. An example is a company, which has a management board and then a number of departments, which in turn might be subdivided. At each level, policies for call processing could be in place, applying to members at that level or below. Roles are a different, orthogonal classification of users. A user might be in a certain department in a company, but usually each user also plays a number of roles (within and outside the enterprise). Examples are that Bob is a programmer, but he also provides technical customer support and is member of the company's wine club.

Policies can be associated with each of the above subjects, and clearly policies might be contradictory. Maybe a user does not want to answer calls before 9:30 as he is dealing with mail, but the enterprise structure or his role therein requires that calls be answered after 9:00 (we return to this later).

### 2.1 Telephony Examples

Subjects can wish for a wide range of policies. Let us consider some examples in the domain of telephony (though the application domain is larger):

*Example 1.* Never forward emergency calls.

This policy would need to be applied network-wide, thus it can be seen to be a service provider policy. It requires the ability to recognise a call type, here *emergency*. Other service provider policies deal with quality of service and other service guarantees.

*Example 2.* My secretary will handle all my calls.

Considering that a call forwarding feature is by now standard in most telephone systems, this policy could easily be judged to be a feature. However, we claim that it is indeed a policy, since *secretary* as a role allows for independence from a particular user or physical phone number (as is required by a call forwarding feature).

*Example 3.* Members of the wine club are available at lunchtime to accept calls to discuss fine wines.

This policy again considers the concept of role. However, here a person has a role by being member of the wine club. Further, notions of time and the content of the call are relevant. In this example the content can be extracted from the context. If another wine club member rings at lunchtime, it is presumed that they want to discuss wines. Note that this policy has an element beyond simple call control, it contains a notion of presence or availability which can be queried by other users.

*Example 4.* I prefer to speak to Jane or Paul if John is busy.

John might forward all calls to Mary when he is busy, but the policy holder expresses a particular, different treatment in this case. This example is interesting in that it does not express an absolute weight (i.e. never or always) but rather has the fuzzy notion of *prefer*.

*Example 5.* If my call is not returned within one hour, send an email reminder to the callee.

Policies might also relate calls. Note that we consider the action of sending an email also as a call, thus leaving the territory of plain telephony (and end-to-end connectivity).

*Example 6.* Always notify me when an expected visitor arrives at reception.

This example expresses a policy that is to be activated when a certain event occurs. Again, we could see this in a much wider context than plain telephony – the reminder could be a telephone message, but equally an alarm going off or a computer alert could be considered.

*Example 7.* All callers phoning regarding project X should be connected to the project web page.

Leaving the territory of traditional telephony allows for interesting behaviour to be requested. Here a caller is redirected to a web page which hopefully contains the required information.

### 2.2 Policies and Features

The examples highlight an important difference between policies and features. Features can be described as specific, prescriptive and imperative. For policies the terms generic and declarative are more fitting. This distinction highlights that features exactly prescribe how a very specific situation is to be handled (forward my calls to extension 55 when I am busy), whereas policies require an interpretation: forward calls to my *secretary*.

Crucially, features leave no space for preferences. Upon subscribing to a feature the action will always be taken. For example a call forwarding on busy feature will always forward calls when the user is busy. This means from a user's perspective that a feature is either enabled or disabled. It is not possible to express that the user would *prefer* the call to be forwarded when busy. Often when two features interact the only resolution is to disable one, indicating that one user (at random from a user perspective) does not receive the expected behaviour. However, assuming the policy version with preference, the user has expressed that not receiving the expected behaviour would be acceptable, thus the user would be less unhappy if his preference is not satisfied. The occurrence of an inconsistency still means that only one user's expected behaviour can be satisfied, but the system has additional information to a obtain a resolution that is most acceptable by the users involved.

## 3 Towards A Logic for Call Processing Policies

Feature descriptions are usually very imperative: the trigger events and corresponding responses are defined – often by means such as state machines. Due to the descriptive nature of policies this is not a suitable approach. We consider a logic approach and refer to the result as a Policy Description Language (PDL).

Such a language or logic must be able to express a large number of concepts to be usable for call processing policies. On the other hand, limiting the complexity of the language is an important goal as it must be decidable under realistic time constraints as well as simple enough for the average user to use.

We propose that simple policies be formulated and then composed to formulate more complex cases.

### 3.1 Policy Formulation

A simple policy can be defined by a policy rule. Each policy rule is composed of four parts: a "policy owner", a preference, an action block and a condition block. Note that not every policy rule will have all four of these parts (we can imagine general policies that do not have a condition block).

The policy owner is a subject. An enterprise policy is owned by the enterprise, whereas a personal policy is owned by the person that wishes it to be enforced.

We assume a set of preferences, the elements of which should be at least partially ordered. We can see boundaries with "never" and "always" expressing

the strongest preferences and some form of "don't care" the weakest. All other values in the set take a place somewhere in between. Examples of such other values are "wish", "may", "prefer" or "rather not". Clearly each of these ordered elements expresses either a positive or negative preference, so a mapping onto the real numbers from -1 to 1 might be a suitable representation of this set. It is important to know how strong preferences are in relation to each other, as the strength needs to be taken into account when resolving conflicts between policies.

The action block simply contains one or more instructions. These instructions are typically actions as provided by the target system. In a call processing system they can be actions such as "forward call", "originate call" or "contact". Each action usually has a number of associated parameters (e.g. "forward call" will have a target user as parameter), but again those are defined by the target system.

Conditions can further restrict the applicability of a policy rule. Typical conditions are equalities or inequalities on parameters associated with the call. There is a large set of these parameters and we consider them next.

In summary, while the four parts of each policy allow us to define all simple policies that we might require, the language must be flexible enough to allow for domain-specific information to be plugged in. That is, the available actions and condition elements are domain-specific and might need to be replaced if the domain changes.

### 3.2 System Parameters for Conditions

To refine the applicability of a policy rule, each rule contains a condition part. We have said that this considers equalities of many system parameters. The system parameters applicable in a typical call control system are as follows. Note that these parameters might apply to the subjects engaged in a call as well as to the call itself.

- caller (a user or agent on behalf of a user)
  We need to be able to consider originator-based conditions.
- callee (any subject)
  Similarly, we require destination-based conditions. For this we require notions of what calling enterprises or multiple users means.
- devices (mobile phone, PDA)
  With growing capability differences among communications devices, it becomes important to be able to distinguish them when considering preferences.
- call content (email, video, language)
  Depending on call content a user might wish different handling, e.g. video might only be acceptable if a high resolution display is available.
- media (fixed, mobile, high speed)
  The media used for data transfer impacts on other aspects, e.g. Quality of Service (QoS), and so users might have different preferences depending on the available media.

– call type (emergency, long distance, intra-company, local)
  Conditions can be based on the type of call (e.g. see Example 1).
– cost
  A user might not be willing to conduct calls above a certain cost threshold.
– quality
  With more than plain voice being communicated, quality becomes more important: users need the possibility to express conditions based on the quality of the transmitted information.
– topic (wines, project x, weekend plans)
  Users might accept calls with a certain topic despite being involved in another activity and hence not accepting calls – e.g. a user in a project meeting might accept calls concerning the project but no others.

Note that some of these classes are overlapping, or might have implications that influence others (e.g. sending video call content via a mobile network will probably be adverse to quality).

When considering conditions on subjects, the following are important, especially when availability and presence are considered:

– locations (my office, at customer site)
  Depending on the location of a user, different call processing options might be chosen.
– identity, role(s) (Mary, customer service representative)
  Different identities and (more importantly) roles will require distinct call processing
– interests, capabilities (programming, Java expert)
  Users should be able to provide information about their capabilities. This is very important in call centres, where the caller should be routed to a person who speaks the right language or is an expert in the field of the query.

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. Roles for example, may be defined in a company organisation chart, the location can be established from the user's diary or mobile home location register. Note that we are not concerned with privacy issues at this moment.

In addition to the properties of a call or subject, the more global condition of time is relevant. Time can be relative or absolute (i.e. *in one hour* or *at 8am every Monday*) and is also subject to different time zones. We need to talk about time spans and might also consider laxer notions of time, such as lunch time or Easter.

### 3.3 Policy Composition

Policy rules are relatively simple, but a user might wish to express more complex situations. This should be possible by combining policy rules. Our work uses a number of combinators for policy rules as described next.

**Sequencing.** Sequencing simply applies one rule after the other.

**Parallel Composition.** Two versions are seen: true parallel composition where the conditions of the rules are matched before the rules are applied. A "don't care about order" sequencing essentially allows the system to identify a suitable sequence. The former might lead to the two rules giving incompatible results, which can be handled like any other policy conflict and as such is not necessarily a disadvantage.

**Choice.** Choice can be unguarded or guarded. A user might not have a preference between a number of rules, so any one can be chosen or there might be extra conditions on each of the rules that must be evaluated first. *I prefer to speak to John or Mary* is an example of an unguarded choice. We have two policies (*speak to John* and *speak to Mary*), but the English sentence suggests an indifference between which is actually chosen. Note that there are at least two interpretations of this choice: try John and if he is unavailable try Mary, or try both John and Mary and chose the one who responds first. In any case, if one of the two is available the user should be connected to them.

**Loops.** Loops are problematic as they might not terminate, so we will only allow bounded loops. It could be argued that loops should be avoided, but we can imagine that a user wishes to have a policy which is best described by a loop: *If the other party is busy, I wish to try again in 30 seconds.*

We have shown the structure of a simple policy rule, discussed the elements thereof, and considered conditions in more detail. We concluded by showing how policy rules can be combined to build more complex policies.

## 4 Using an Existing Policy Description Language

In the previous section we have outlined our requirements for a policy description language. We have considered the essential elements and structure of such a language. As policies have been used extensively in the context of system management, some PDLs exist already. We will now consider the applicability of one such language in the context of call processing.

### 4.1 Ponder

We consider the Ponder policy description language in some more detail now. Ponder is being developed at Imperial College (London). Ponder provides a language for the specification of policies [DDLS01], a framework to deploy Ponder policies [DLSD01], and a toolkit to support the policy life-cycle in the context of the framework [Pon]. Ponder provides 3 types of policies:

1. positive and negative authorisation policies (`auth+`, `auth-`)
2. obligation and refrain policies (`oblig`, `refrain`)
3. delegation policies (`deleg`)

(1) expresses whether a subject is allowed or forbidden to apply an action to a target. (2) describes which actions a subject has to perform/refrain from

performing on a target. (3) allows for a subject to pass its own authorisation (temporarily) to other subjects.

Authorisation policies are seen to be enforced by the target, delegation and refrain policies are enforced by the subject. This distinction becomes important as two different kinds of objects are used to ensure the enforcement of the policies.

The Ponder framework comprises the *language* and the *deployment* model. The semantics of the language is defined by the enforcement model. Thus while maintaining the syntax, the semantics of the language can be changed significantly by using a different enforcement model. We consider the impact of this in more detail in section 4.2.

### 4.2 Ponder for Call Policies

It is non-trivial to define a mapping between typical call processing policies and the management policies for which Ponder was essentially created. However, it would be preferable to use an existing notation rather than develop a new one, if this is possible.

Before attempting to express the example policies using Ponder, we need to consider some issues. In Ponder the distinction between subject and target is significant. However, we will simply say the subject is the entity establishing the policy, whereas the targets are given by the call(s) and the user(s) involved in the call. Thus, we have two basic types: `<call>` and `<domain>`, where the latter is a domain description (essentially a graph relating subjects in a hierarchical fashion) as used by Ponder. `<call>` is a type for calls. For our purposes a variable of type `call` has the following fields (though more can be extracted from the list of system parameters (see section 3.2): `calltype` (to denote the type of call, e.g. emergency), `caller` and `callee` (the originator and terminating user of the call), `topic` (the subject of the call) and `cost` (the cost of the call).

We will use authorisation policies for absolute statements such as never or always. Obligation policies are used for cases when a trigger event exists. Using Ponder policy types we can define generic policies that can then be specified by defining instances.

The subject domain hierarchy contains `everyone` as root and then somewhere in the domain structure we find elements for `mysecretary`, `self` and `wineClub`, each of which will have one or more children of the leaf elements such as `John` or `Mary`. Note that `self` is to be instantiated in such a way that it represents the entity that established the policy.

Here we will simply show the syntactic formulation of the policies in section 2.1, assuming that the semantic interpretation is the same as for the previous description of the examples. We return to the semantics aspect afterwards (in section 4.3). Before describing policies, we briefly introduce the Ponder notation.

The Ponder notation allows the user to define a `type` for policies which then can be instantiated with particular values in an `inst` statement. Both the `type` and `inst` keywords are followed by the policy type (see section 4.1) and an identifier. Finally parentheses contain the parameters (formal parameters for the type and actual parameters for inst). Parameters are of the form `<type of`

parameter> identifier [, identifier], preceded by the optional keywords subject and target if a parameter represents one of these.

The body of a type contains a do action() statement, where action is some action as given by the underlying domain. This represents the action to be executed as a result of a policy application. The when keyword allows one to phrase conditions. Obligation policies have an extra body part called on which introduces the event that triggers the policies.

*Ponder Code for Example 1*

```
type auth- fwdEmergencyT (subject <domain> s, target <domain> t,
                          <call> c) {
    do forward(c,t);
    when c.calltype = "emergency"}
inst auth- fwdEmergency (everyone, everyone, c){}
```

This example is clearly recognisable as a negative authorisation policy and thus can be implemented in the expected straightforward fashion. It must be assumed that the subject or target everyone is resolved in such a way that it applies to any element in the domain. Forwarding is achieved by the forward function which forwards a call c to a new target t.

*Ponder Code for Example 2*

```
type oblig secretaryT (subject <domain> s, target <domain> t, <call> c) {
    on incoming();
    do forward(c,t);}
inst oblig secretary (self, mysecretary, c){}
```

The implementation here is slightly more difficult, but this is mainly due to the fact that the policy in Example 2 does not specify the trigger event. However, by assuming *incoming* as trigger event, we might have changed the intention of the original plain language policy. The user could have wanted this policy to mean that the secretary does in fact also make outgoing calls in the policy owner's name. It is necessary to resolve who will receive the call if mysecretary is a group of people – possibilities include the first person in the list or the one who answers first.

*Ponder Code for Example 3*

```
type auth+ wineT (subject <domain> s, target <domain> t, <call> c,
                  <time> t1, t2){
    do acceptCall(c,t), available();
    when t1 < time < t2
         and c.topic="fine wine"}
inst auth+ wine (wineClub, wineClub, c, 12.00, 13.00)
```

We allow other members of the wine club to contact us at lunchtime (which in this particular instance is assumed to be from 12.00 to 13.00, but in general could be derived from a diary). In addition to just accepting calls we also provide other members with a query function *available()* to check whether we are indeed available to accept a call.

*Ponder Code for Example 4*

```
type oblig speakToT (subject <domain> s, target <domain> t, t1,
                     <call> c) {
    on busy(t)
    do forward(c,t1);}
inst oblig speakTo (self, John, (Jane or Paul), c)
```

Caused by the occurrence of a busy notification from John, we wish the call to be forwarded to one of Jane or Paul. Note that this might easily interfere with John's forwarding preferences. What we are not able to express in the Ponder language is that we *prefer* this behaviour rather than allowing it as the only possibility. A potential solution for this is to provide a new argument to the policy expressing the preferability and adapting the deployment to take this into account when conflicts are detected. The language also does not provide an interpretation as to how the choice between Jane and Paul is made. Possible options would be the first who answers or a random choice. If neither is available, the call can always revert to John's forwarding preference. Again, this must be resolved in the enforcement model (see section 5).

*Ponder Code for Example 5*

```
type oblig noReturnT (subject <domain> s, target <domain> t,
                      <call> c, <int> p) {
    on returnCallActivated(startT)
    do email(t,remind);
    when returnStillActive(startT+p)}
inst oblig noReturn (self, everyone, c, 60)
```

When the return call is set up, *returnCallActivated(time)* is issued, thus triggering this policy and providing the start time `startT`. If the call is not answered and a period `p` has passed, the action is executed.

*Ponder Code for Example 6*

```
type oblig notifyT (subject <domain> s, target <domain> t, <call> c) {
    on arrived(t)
    do notify(s);}
inst oblig notify (self, visitor, c)
```

The last two examples are straightforward. Depending on the occurrence of an event, an obligation to perform an action is created.

*Ponder Code for Example 7*

```
type oblig projectXT (subject <domain> s, target <domain> t, <call> c,
                      <String> topic, wp){
    on incoming()
    do connectToWebpage(c, wp);
    when c.topic = topic}
inst oblig projectX (self, everyone, c, "Project X", "www.projectX.org")
```

In this policy we have a mixture of media. A caller is not connected to the expected phone but is instead diverted to a web page if he was ringing with respect to a project. This policy requires a more flexible interpretation of call than traditionally thought about. It also requires information about the caller's device: if it can display web pages there is no issue, but if it can only produce audio output a solution such as Text-to-Speech might be required.

### 4.3 Discussion

From the above examples we can see that Ponder indeed allows us to formulate the example policies. However, this formulation is often not intuitive, and we do not think that it could be expected that the average user could perform the required mapping. Most actions depend heavily on the underlying system, as is to be expected.

Ponder's policy types allow policies to be formulated in a generic way and then to be specialised by instantiations. This is certainly of benefit to the user, as policy templates can be provided that are then adapted by the user.

Some ideas cannot be readily expressed in the language, and we have pointed these out. For example, there is no concept of how strongly a user feels about a policy being respected.

It is often not intuitive what the target and the subject are in call processing policies. Sometimes we could phrase the policy without clearly identifying either (e.g. the policy in Example 1 needs only to consider the call as it applies to everyone). We have shown earlier that we only require to know who owns the policy. The deployment should then place the policy in a network node that is always involved in the owner's calls. This is also the place where the policy will be enforced. More detail will be given in the next section.

So far we have mainly considered the language, but Ponder comprises the *language*, the *deployment model* (concerned with installing policies in the enforcement points) and an *enforcement model*. We have been able to formulate the English language versions of our policies in Ponder syntax, but we have not discussed the semantics in detail. The semantic interpretation of the policies is tightly connected to the enforcement model.

The enforcement model for Ponder was originally defined in the context of management policies, where a clear distinction between subjects and targets is possible. Both gain importance when the enforcement of policies is considered: authorisation policies are enforced by the targets, and obligations by the subjects. Thus a negative authorisation policy gains the meaning that the target blocks the prohibited action, whereas an obligation policy means that the subject will indeed perform the required action when the conditions are met.

The example provided in [DDLS00, p37] considers the patient as target, showing a case where it is hard to see how the target can actually enforce a policy (say, a drug treatment). This case is more common in call processing policies, where a proxy or policy server must enforce the policies rather than the target user or the call object.

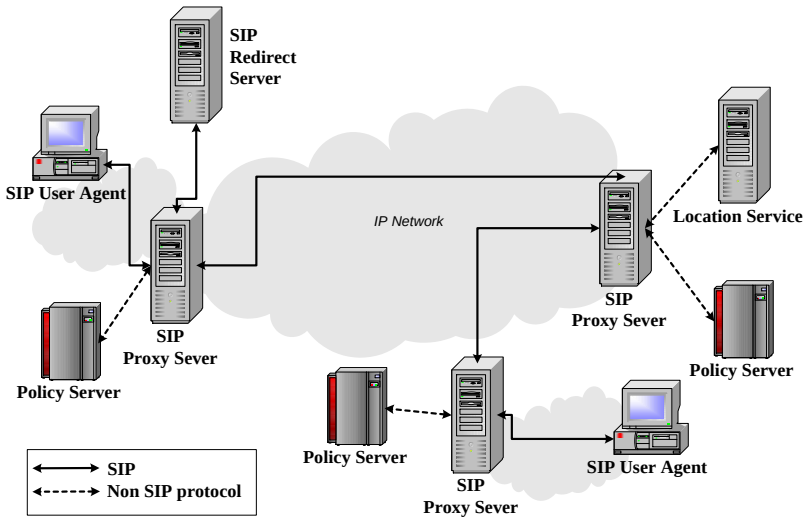


**Fig. 1.** SIP Architecture with Policy Servers

This motivated us to use the classes of obligation and authorisation policies more loosely, namely authorisation was used for absolute statements, obligation to describe reactions to trigger events. This is more natural than the original Ponder interpretation in the context of call processing policies, but does violate the enforcement model.

We will now outline the deployment and enforcement model for call processing policies considering a particular call processing mechanism: SIP. The deployment in other call processing systems is similar and will be discussed in section 6.

## 5 Deployment and enforcement in a SIP environment

As we have seen, the ability to formulate policies is one half of the process of enabling policy-based call processing. The other half consists of embedding the policies into the telecommunications system in a suitable fashion. For the purpose of this paper, we considered only how the policies could be enabled in a SIP environment. Ongoing work is concerned with the deployment and enforcement of policies in other environments such as a PBX or H.323 [ITU00].

Recall, SIP is a protocol to establish, modify and terminate multimedia call sessions. The policies that we formulated can influence how these three actions are actually performed, e.g. a policy might disallow establishing certain sessions. In order to see how policies can be enforced in a SIP environment it is necessary to understand the SIP architecture (Figure 1).

The figure highlights the essential components of the SIP architecture. User agents are essentially SIP-enabled communications devices such as a SIP telephone, a PDA or a softphone (i.e. a software implementation of a phone). They can issue and receive SIP commands and thus establish sessions. The path between two (or more) users is established via a series of proxy servers. Communication is via the IP protocol. The architecture includes two further kinds of servers: redirect servers, which essentially redirect incoming calls to a different SIP address, and location services servers, which can detect the current location of registered users.

The SIP protocol is based around a textual format similar to HTML. SIP proxy servers are equipped with SIP CGI [LRS00] which permits new services to be defined. This interface can be used to intercept messages and make the required changes to them before passing them back into the network. Clearly, it can be used to intercept messages and pass them to a policy server for processing.

Our policy server is a new component added to the architecture. It will usually be co-located with a proxy server. Its purpose is to evaluate the information in a SIP message, either to make the necessary changes (if the path is to be changed) or to generate a suitable response (if the call needs to be blocked). This is then passed back to the proxy server in order to enforce the policy.

In SIP users can potentially connect directly to their call partners, provided they know the exact address. It must be ensured that this is not allowed if policies are to be enforced successfully. The SIP architecture usually provides users with an outgoing and incoming proxy server, and it is desirable that these are in fact in the communications path. Similar techniques such as those used for firewalls can enforce that all communication passes the incoming and outgoing proxy servers, thus policies can indeed be enforced in these points.

The enforcement model for policies in SIP is based around a policy server that receives all communicated messages that are relevant. It considers how they need to be handled in the context of the activated policies. Policy conflicts are resolved at the policy server. It might be necessary to send some policy information to the other party's policy server, which can be done by attaching it (in a suitable format) to the body of the SIP message. Negotiation between policy servers is a more advanced option for conflict resolution. When considering conflict during enforcement, is will usually be necessary to violate some aspects of a users policies, but it is desirable to minimize such violations.

The deployment model is simply concerned with delivering policies into a policy repository (associated with the policy server). Several options are possible here. One is to piggy-back the policy onto a SIP register message and have the SIP-CGI interface handle the policy upload. However, due to the possibility of better feedback mechanisms, it would be preferable to have a direct interface (e.g. web) to the policy server. The user would then go to a web page and configure his/her policies. This would also allow resolution of conflicts within one user's policies by giving the user a chance to adapt the conflicting policies directly.

## 6 Summary and Further Work

We have considered call processing policies. Call processing is performed by large distributed systems and thus seems a natural setting to apply policies. However, this has not previously been done. A number of telephony policies have been indicated in the paper.

We have identified the components and constructs that a language or logic for call processing policies needs to contain. This was done in an abstract fashion, essentially to enable us to compare the requirements with the functionality offered by existing PDLs.

The evaluation of the Ponder framework leads to a twofold conclusion. The Ponder language seems a suitable mechanism to express call processing policies, despite a number of shortcomings being detected. However, the standard enforcement model of Ponder is unsuitable for call processing policies.

It must be evaluated whether it is productive to enhance Ponder for our purposes and develop a new or tailored enforcement model. The benefits of using the existing compiler and tools are removed, as they are centred on the standard enforcement model. However, it seems preferable to reuse existing work as much as possible, rather than providing another new language. Currently we feel that, based on the described requirements, a new language needs to be developed. We expect to use an XML schema to structure policies, allowing for the use of a growing repertoire of XML tools. This will be investigated further.

We are aware that policies can interact, and that these interactions must be resolved. Policies can interact in two different ways: they might contradict each other or not. It is the former case that the policy server needs to resolve. There are two dimensions to interactions. As indicated earlier, policies of the same user can be statically analysed, whereas a dynamic analysis and resolution is required for policies of different users, as they might only become aware of each other when an actual call takes place.

Resolution might mean that some policies need to be violated. In this case a penalty scheme could enable fairness. Preferences provide a means to satisfy a user expressing stronger preferences at the cost of a user who is indifferent (and thus not really losing much). It might prove necessary to experiment with negotiation schemes to resolve complex cases or gain the best outcome.

Ongoing work is concerned with building a prototype SIP policy server which will enable us to obtain empirical results. Furthermore, the policy server will be built in such a way that it is possible to include it in different network architectures. In particular we also consider inclusion in a PBX or H.323 setting. The latter uses gatekeepers rather than proxies, but they essentially provide the same extensibility. Inclusion in PBX products is facilitated by our collaboration with Mitel, as access to the PBX's proprietary interface is required.

## Acknowledgements

This work has been supported by EPSRC (Engineering and Physical Sciences Research Council) under grant GR/R31263 and Mitel Networks Corporation. We thank all

people who contributed to the discussion of policies in this context. Particular thanks are due to Daniel Amyot and Tom Gray (both of Mitel Networks Corporation), and Prof. Evan Magill and Mario Kolberg (both of Stirling University).

## References

[AKGM00] M. Amer, A. Karmouch, T. Gray, and S. Mankovskii. Feature interaction resolution using fuzzy policies. In *[CM00]*, pages 94–112, May 2000.

[BAE+98] 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 *[KB98]*, pages 135–149, September 1998.

[BGM98] 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.

[CM00] M. Calder and E. Magill, editors. *Feature Interactions in Telecommunications and Software Systems VI*. IOS Press (Amsterdam), May 2000.

[DDLS00] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. Ponder: A language specifying security and managements policies for distributed systems. *Imperial College (London) Research Report*, 2000.

[DDLS01] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. *Lecture Notes in Computer Science*, 1997:18–39, 2001.

[DLSD01] N. Dulay, E. Lupu, M. Sloman, and N. Damianou. A policy deployment model for the Ponder language. *Proceedings of IEEE/IFIP Symposium on Integrated Network Managament 2001*, 2001.

[HSSR02] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. SIP: Session initiation protocol. *Request for Comments 2543 bis*, March 2002.

[ITU00] International Telecommunications Union ITU. *ITU-T Recommendation H.323: Packet Based Multimedia Communications Systems (Version 4)*. ITU-T, November 2000.

[KB98] K. Kimbler and L. G. Bouma, editors. *Feature Interactions in Telecommunications and Software Systems V*. IOS Press (Amsterdam), September 1998.

[LRS00] J. Lennox, J. Rosenberg, and H. Schulzrinne. Common gateway interface for SIP. *IETF Internet-Draft*, 2000.

[LS99] E. Lupu and M. Sloman. Conflicts in policy based distributed systems management. *IEEE Transactions on Software Engineering*, 25(6), November/December 1999.

[LS00] J. Lennox and H. Schulzrinne. CPL: A language for user control of internet telephony services. *IETF Internet-Draft*, 2000.

[MMSS94] D. Mariott, M. Mansouri-Samani, and M. Sloman. Specification of management policies. *Proceedings of the fifth IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, 1994.

[Pon] Ponder. A Policy Language for Distributed Systems Management, `http://www-dse.doc.ic.ac.uk/Research/ policies/ponder.html`.

[SD99] M. W. A. Steen and J. Derrick. Formalising ODP enterprise policies. *Proceedings of 3rd International Enterprise Distributed Object Computing Conference (EDOC '99)*, September 1999.