Managing the Policies of Non-Technical Users in a Dynamic World

Tim Owen, Ian Wakeman, Bill Keller, Julie Weeds, David Weir
Department of Informatics, University of Sussex, Brighton, UK
{timo,ianw,billk,juliewe,davidw}@sussex.ac.uk

Abstract

In this paper, we describe the use of description logic *as the basis for a policy representation language and show how it is used in our implementation of a policy managed pervasive environment. We compare our approach within this domain to conventional policy management in both implementation and analysis, and highlight the difficulties presented by dynamic environments.*

1. Introduction

The vision of pervasive computing promises to deploy computing devices and services throughout the everyday environment, offering the potential of a far greater level of automation and assistance to the general population than the traditional PC. This future provides an opportunity to apply policy-based management to the pervasive infrastructure, allowing users to tailor their environment by bringing together and configuring the services around them.

However, there are characteristics of this application domain that introduce new challenges to policy formulation and analysis. The target users are typically non-technical people, which contrasts with the more technical and structured environment of systems management, where policies are formulated and debugged by knowledgeable users using special purpose tools. Furthermore, the likely styles of policy that users have about their environment do not necessarily fit neatly into the general pattern of deontic policies of authorisation and obligation.

In this paper we discuss the complications that arise from this style of policy management, both in representation, implementation and analysis. In particular, the dynamic nature of the pervasive environment, where services and entities appear and disappear, and where the attributes and descriptions of the entities and services evolve over time, present very different challenges from the comparatively static domains assumed by much of the literature in policy management.

The remainder of this paper will briefly describe the overall structure of the system within which the policy representation acts, and describe how the policies are represented and executed using description logic (DL) [2]. We then show the limited extent to which we can statically analyse policies, and how we can determine potential conflicts at runtime.

2. System Overview

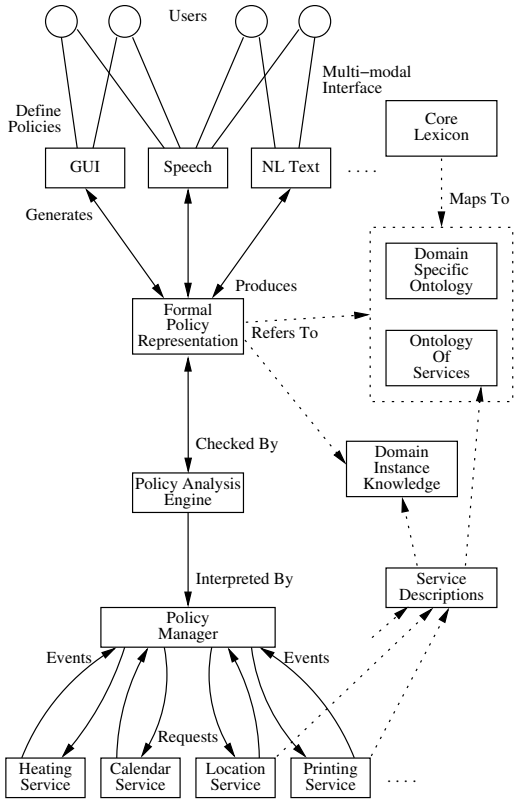

Figure 1. Overview of Policy Management System

Figure 1 shows the high-level architecture of the system we are building. There are a number of software services that provide useful functionality in the user's environment. These send out notifications of events, and accept requests for action from applications and other services. The policy manager stands as a central broker for notifications and requests, and creates, modifies or cancels requests in accordance with its set of active policies, before passing them on to the relevant services.

Users can configure and compose the services to personalise their environment by formulating policies that specify what actions should be taken when certain events occur. For example, a user may create a policy such as *Switch the heating down when there is nobody in the house* or another that specifies *Print documents on the nearest printer, by default*. The task of the policy manager is to enforce these policies, by acting upon incoming events such as notifications about person-movements from the location service or print requests from an application. The manager may modify a request (e.g. naming a target printer if not specified) or initiate a new request (e.g. asking the heating service to lower its setpoint) as required to implement the user's policies.

Currently, we have built a prototype system of most of the major components shown in Figure 1. The knowledge base module is implemented using the RACER system[7] which provides a DL reasoning engine, as well as storing the definitions of domain concepts and roles (the TBox) and the known individuals (the ABox).

3. Policy Representation and Implementation

We undertook a series of user studies to investigate how users would like to configure an imagined pervasive environment. From the corpus of results we identified that many configuration policies involve the incremental refinement and rewriting of action requests such as "print colour documents to the colour printer", whilst the remainder were mostly of the form of conditioned action requests, such as "send me an email when my printer goes offline".

We therefore designed a policy representation suited to this style of policy. In particular, the representation should be able to cope with the incremental modification of requests for action, including:

- the ability to partially specify how requests will be satisfied;
- the ability to fill in default constraints if they are not already specified;
- the ability for new constraints to override and have preference over previously specified constraints

The general form of a user policy is a rule with a precondition and postcondition, each expressed using DL terms that reference concepts, roles and individuals in the ontology. For example, a user's policy *Print colour documents on a colour printer* would be expressed formally as:

$$x \in \text{Print} \sqcap patient.colourness.\text{COLOUR} \Rightarrow$$
$$x \in target.colourness.\text{COLOUR}$$

where Print is a concept name of a request to print, which has roles called patient and target (which in turn each have colourness roles). This policy rule can be read as: if there is a Print request whose patient (i.e. the document to be printed) has a COLOUR value for the colourness role, then assert the postcondition that the target of the request must be a printer whose colourness role has the COLOUR value.

However, certain user policies require that the postcondition contradicts the preconditions of the rule. For example in the user policy *If a document is sent to printer LJA and it is offline, send to LJX instead* we expect the original target of the print request to be overwritten by the new assertion in the postcondition.

A variation of this style of policy is to use a more cautious, default, semantics where the postcondition is only asserted if it doesn't contradict existing knowledge about the request. This captures policies such as the example in the previous section *Print documents on the nearest printer, by default* where we don't wish to overwrite existing target information, but only add to it if not specified. Each policy rule is tagged by the creator to indicate whether overwrite or default semantics are required.

Action-style policies are expressed with a postcondition that triggers a new request, rather than asserting further changes to the original event that matched the precondition. For example, if the user's policy is *Email me whenever someone prints on my printer* can be captured in the following rule:

$$x \in \text{Print} \sqcap target.owner.\text{TIM} \Rightarrow$$
$$y \in \text{Email} \sqcap recipient.address.\text{"alert@me.com"}$$

where the variables x and y indicate different requests are being referred to.

Each formulated policy rule is passed to the policy manager, which is responsible for implementing the set of user policies. The manager examines incoming notifications and requests from services and applications, and tests the preconditions of each policy in turn to see if it matches. If the policy manager applies a policy to a request, it incorporates the postcondition terms into its knowledge about the request. Once there are no further policy applications to perform, requests are finally passed to the relevant service for execution. By this process a request is incrementally refined, by the addition of new assertions and the replacement of earlier knowledge by more recent information from policy postconditions. This allows users to configure the behaviour of their environment by filtering and modifying the descriptions of requests passing through the system.

User studies indicated that there was a wide disparity among users as to whether or not they anticipated that policies would be chained. Some users believed that only a single policy should ever fire, whilst other users believed that policies would chain together raising the problem of looping. We have therefore made the engine configurable in how policies are chained together, ranging from single policy application through to greedy chaining, albeit still restricting policies to single firing per request to prevent loops.

Whilst the choice of representation has been driven by the need to represent action policies, we are still able to represent deontic-style policies based on authorisation and obligation e.g. *Bill is not allowed to print* becomes

$x \in \text{Print} \sqcap\ agent.name.\text{BILL} \Rightarrow$
$x \notin \text{Print}$

The application of policies involves modification to the policy manager's knowledge about each request as it is processed. This raises a number of complications arising from conflicts and ordering dependencies among the set of policies. Since the system behaviour will depend on how the multiple interacting policies are managed, this will be particularly problematic for non-technical users who, in devising their policies, may be unaware of potential interactions.

4. Policy Analysis

As described in Section 3, the preconditions of policies are tested in order. The ordering of the policies therefore plays a key role in determining the behaviour of the system. We have adopted a number of heuristics to resolve the ordering.

Following other approaches in the literature such as Lupu and Sloman [11], policies with more specific conditions are given higher precedence than more general policies. Since we can call upon a DL reasoning engine, we use both subsumption and disjointedness tests upon the preconditions of pairs of policies to determine their relationship. If the preconditions of one policy subsumes another, it is given a higher precedence than the other. If the preconditions are disjoint, then policies will never interfere with each other and so can be tested in any order.

In the situation when the preconditions of policies overlap, we have used the following additional tests. If the preconditions of one policy are disjoint from the postconditions of the other, then the order in which the policies apply is obviously important, and in this case we query the user. Otherwise, we adopt the heuristic approach of testing the size of the realizable sets[1] of the combined pre- and postcondition pairs, and giving higher precedence to the policy whose preconditions generate smaller sets.

1 The *Realizable Set* of a set of conditions is calculated by using the current ABox to calculate how many entities are covered by these conditions

In the majority of policy managed systems, much attention is paid to determining a priori which policies will conflict with each other, e.g. through using syntactic checks as in the modality conflicts of Lupu and Sloman [11] or by using the event calculus to check explicit conflict definitions [4, 3]. Description logic has previously been used to check for feature interaction in telephone services [1]. However, because our system works within a dynamically evolving world, it is not possible to exhaustively identify all potential conflicts. Whether a policy will fire depends upon the particular set of entities within the ABox, and since this is a changing set, static analysis of interactions has a limited lifetime. It is also difficult to determine whether the possible interaction between two policies is good or bad — if policy A will cause policy B to fire, the system cannot tell without human intervention whether this is intended or not. We have provided a search capability to determine which policies can interact with each other based upon the current state of the ABox, but both the volume of interactions and our preliminary user studies indicate that this would not be useful for most users.

Rather than static analysis, we rely heavily upon runtime tests, and provide explanations to allow users to debug and modify their policies. After the application of a policy and the refinement of constraints for an action to fire, we test whether the current set of constraints will produce an empty realizable set. If this is the case, we flag to the user that there may be a potential problem with the policy definitions. The explanation as to how policies have interacted to produce this state can be used in policy refinement.

Finally, we envisage that deployment of any real system based on this style of policy management would come with a number of generic policies designed to prevent *bad* interactions. For instance, to ensure that all documents eventually went to a printer that was online, consider the policy *Do not send documents to offline printers*. This can be encoded as

$x \in \text{Print} \sqcap\ target.status.\text{OFFLINE}) \Rightarrow$
$x \in target.status.\text{ONLINE}$

The use of the overwrite rule to mask and retract contradictory conditions means that the conditions leading to offline printer choices are retracted.

5. Related Work

There has been work on allowing access to the configuration of pervasive environments from the HCI end of the pervasive computing research spectrum. For example, the work of Truong et al [14] provides a pseudo-natural language interface, using a fridge magnet metaphor. However, this has very limited expressibility, and requires the use of a screen for the editor, as does the jigsaw editor of Humble [8] in which typed services can be composed in pipelines similar to other techniques of visual programming, or the browser

approach of Speakeasy [12], where components are connected using a visual editor based on file-system browsers.

From the policy management community, description logic has been taken up and used in the Rei system [9] and the Kaos system [15]. However, as described above, we are geared to the policy management of actions, rather than the deontic approach taken in these languages and systems. Other research on pervasive systems such as Gaia [13], Interactive Rooms [5] and Aura [6] have said little about their approach to user configuration, concentrating in general on the programmatic interface and necessary middleware for easy construction of pervasive services.

6. Conclusions and Future Work

We view policy management as an important aspect of pervasive computing configuration. However, the nature of the pervasive computing domain requires different approaches to modelling and analysing policies. Not all configuration policies are deontic-style but involve changes and refinements of the constraints upon requests and notifications.

Further, the need to be accessible to non-technical users requires any policy management system to be able to explain how and why particular policies were applied, and to be adaptable in the extent to which policies are chained together. We have found that *description logic* is a useful formalism in representing policies, since it allows policies to incrementally accumulate knowledge about requests.

At the time of writing, we are integrating the policy engine with a natural language understanding system[16] after which we will be running a number of user studies to understand the benefits and limitations of our approach, to enable us to improve the design and implementation. Concurrent work is investigating how to best explain and debug the operation of the policy management system through a multi-modal interface.

The policies with which we currently work can be characterised as *action* and to a limited extent *goal* policies according to the taxonomy of Kephart [10]. In future work we intend to investigate how planning technologies can be incorporated into the policy engine to deal with policies such as "Keep average heating costs below $200 per annum" and eventually to deal with policies specified in terms of utility functions.

References

[1] C. Areces, W. Bouma, and M. de Rijke. Description logics and feature interaction. In *DL'99, the International Workshop on Description Logics*, Linkping, Sweden, July 1999.

[2] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.

[3] A. Bandara, E. Lupu, J. Moffett, and A. Russo. A goal-based approach to policy refinement. In *Proc of 5th IEEE Workshop on Policies for Distributed Systems and Networks*, IBM Thomas J Watson Research Center, Yorktown Heights, New York, June 2004.

[4] A. Bandara, E. Lupu, and A. Russo. Using event calculus to formalise policy specification and analysis. In *Proc of 4th IEEE Workshop on Policies for Distributed Systems and Networks*, Lake Como, Italy, June 2003.

[5] T. W. Brad Johanson, Armando Fox. The interactive workspaces project: Experiences with ubiquitous computing rooms. *IEEE Pervasive Computing*, 1(2), 2002.

[6] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste. Project aura: Toward distraction-free pervasive computing. *IEEE Pervasive Computing*, April 2003.

[7] V. Haarslev and R. Mller. Description of the racer system and its applications. In *Proceedings International Workshop on Description Logics (DL-2001)*, Stanford Ca, August 2001.

[8] J. Humble, T. Hemmings, A. Crabtree, B. Koleva, and T. Rodden. "playing with your bits": User configuration of ubiquitous domestic environments. In *UbiComp 2003*, 2003.

[9] L. Kagal, T. Finin, and A. Joshi. A policy language for a pervasive computing environment. In *Proc of 4th IEEE Workshop on Policies for Distributed Systems and Networks*, Lake Como, Italy, June 2003.

[10] J. O. Kephart and W. E. Walsh. An artificial intelligence perspective on autonomic computing policies. In *Proc of 5th IEEE Workshop on Policies for Distributed Systems and Networks*, IBM Thomas J Watson Research Center, Yorktown Heights, New York, June 2004.

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

[12] M. W. Newman, J. Z. Sedivy, C. M. Neuwirth, W. K. Edwards, J. I. Hong, S. Izadi, K. Marcelo, T. F. Smith, J. Sedivy, and M. Newman. Designing for serendipity: supporting end-user configuration of ubiquitous computing environments. In *Proceedings of the conference on Designing interactive systems*, pages 147–156. ACM Press, 2002.

[13] M. Romn, C. K. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. Gaia: A middleware infrastructure to enable active spaces. *IEEE Pervasive Computing*, pages 74–83, October 2002.

[14] K. N. Truong, E. M. Huang, and G. D. Abowd. Camp: A magnetic poetry interface for end-user programming of capture applications for the home. In *UbiComp 2004*, 2004.

[15] A. Uszok, J. M. Bradshaw, R. Jeffers, N. Suri, P. Hayes, M. R. Breedy, L. Bunch, M. Johnson, S. Kulkarni, and J. Lott. KAoS policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement. In *Proceedings of Policy 2003*, Como, Italy, 2003.

[16] J. Weeds, B. Keller, D. Weir, I. Wakeman, J. Rimmer, and T. Owen. Natural language expression of user policies in pervasive computing environments. In *Proceedings of OntoLex 2004 (LREC Workshop on Ontologies and Lexical Resources in Distributed Environments)*, Lisbon, Portugal, May 2004.