Middleware for User Controlled Environments

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

Abstract

In this paper, we describe the middleware that has evolved from our attempt to capture user descriptions of policies controlling devices and services from natural language. Description Logic (DL) provides a formal link between the natural language processing, the ontology and the middleware. We show that the use of a formalism such as DL opens useful avenues to detecting and resolving conflicts in policies, both in formulation and when resolving them against incoming events and requests. We finish by arguing that pervasive middleware needs to move closer to the users' abstractions to provide a service for what will be a highly dynamic environment.

1. Introduction

In pervasive computing environments, there is an obvious requirement to allow customisation of the various services to meet the needs of the users of the environment. If the customisation is to be anything more sophisticated than simple selection of a couple of options, a rich interface must be available to allow the user preferences to be explained. In the Natural Habitat project[22], we are exploring how we can capture, interpret and negotiate understanding of these preferences using natural language (NL), relying on the limited domain of discourse to make the NL problem tractable. Throughout this work, we use the term **policy** to mean a user-defined rule that specifies how some aspect of their environment should behave in response to events and circumstances. This meaning of policy is more general than its use in other research areas, such as defining access control or security permissions.

In this paper, we will be talking about how the emphasis on negotiating meaning from users has shaped the design of our middleware. In particular, our choice of Description Logic [1] (DL) to capture the formal representation of a user's policy has provided interesting insights into the design of middleware for pervasive systems.

We chose DL as our formalism for a number of reasons: it is a well-understood logical formalism, which is particularly well-suited to expressing ontological knowledge; is supported by widely used systems such as the Racer DL engine[8]; it supports partial descriptions that can be combined incrementally to form more specific descriptions; the standard DL operations of subsumption, disjointedness and inference have efficient implementations under certain conditions; and it provides a suitably abstract level of representation that is not tied to a particular interface modality such as natural language or graphics. We argue that DL is also an appropriate formalism for analyzing and reasoning about user policies, providing tools for debugging functionality both at "compile-time" — when the policies are defined — and at "run-time" when the policies are executed in response to a request or event.

This has shaped our design, pushing our middleware layer, the layer in which we provide abstractions linking the multifarious services together, into an implementation of a policy manager and an associated ontology, utilising a standard message-based event and action request system.

Figure 1 shows the high-level architecture of the system we are building. The set of available services and their functionality is described in terms of an **ontology**. Ultimately, the goal is that users will be able to describe, browse and edit their policies by interaction through a multi-modal interface in which policies can be expressed using a combination of text, speech and diagrammatic representations. Underlying these modes of input is a common representation of that policy, expressed in DL. We can conceive of grammar-based bi-directional mappings between the formal policy language and any of the interface modes, allowing the user to apply whichever forms of input are most appropriate to guide the production of the final policy. There are inevitably cases where ambiguity or conflicts arise in terms of how policies should be interpreted. Rather than using brittle AI techniques to guess what a user is thought to have intended, our strategy is to present the user with a simple, predictable and easily understood processing model.

Each policy is expressed in terms of the concepts and roles known in the ontology and the individuals known in the current state of the world. Having captured user policies

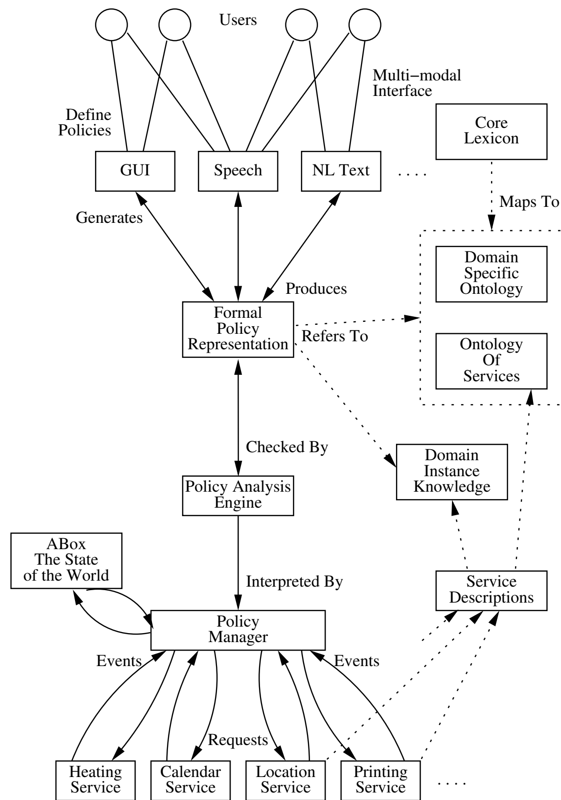

Figure 1. Overview of Service Composition System

TBox

Device	$\sqsubseteq$	Physical
ComputingObject	$\sqsubseteq$	Abstract
User	$\sqsubseteq$	Human
	$\sqcap$	*name*.String
DataType	$\sqsubseteq$	ComputingObject
String	$\sqsubseteq$	DataType
File	$\sqsubseteq$	ComputingObject
	$\sqcap$	*owner*.User
Event	$\sqsubseteq$	ComputingObject
Notification	$\sqsubseteq$	Event
Request	$\sqsubseteq$	Event
Printer	$\sqsubseteq$	Device
	$\sqcap$	*name*.String
	$\sqcap$	*owner*.User
	$\sqcap$	*colour*.ColourVal
	$\sqcap$	*status*.StatusVal
	$\sqcap$	*duplicity*.DuplexVal
Document	$\sqsubseteq$	File
	$\sqcap$	*security*.SecureVal
	$\sqcap$	*colour*.ColourVal
ColourVal	=	{ COLOR, MONO }
StatusVal	=	{ BUSY, ONLINE, OFFLINE }
DuplexVal	=	{ SINGLE, DOUBLE }
SecureVal	=	{ OPEN, CONFIDENTIAL }
Print	$\sqsubseteq$	Request
	$\sqcap$	*agent*.User
	$\sqcap$	*patient*.File
	$\sqcap$	*target*.Printer
DeadRequest	$\sqsubseteq$	Request

ABox

LJA	$\in$	Printer
	$\sqcap$	*colour*.MONO
	$\sqcap$	*duplicity*.SINGLE
		. . .

Figure 2. Ontology Structure and Typical Contents

in DL, these are analysed statically and then passed to the **policy manager**. This component provides a realisation of all the policies in the system, by receiving events from the environment and requests from user applications, and invoking service functionality in accordance with the policies. Our prototype implementation uses the Racer DL engine [8] to maintain the knowledge base. We are currently enhancing the natural language interface, and integrating the system with existing middleware infra-structure, such as the Scooby environment [17].

In the rest of this paper, we make the case for using ontologies within the middleware to allow dynamic configuration of services within the environment, describe the approach we have taken to defining policies, and show how the representation of policies in DL allows the detection of a number of policy conflicts, and simplifies the debugging of policy configurations at run-time.

2. Description Logic Representation and Ontologies

In order to capture user policies, two major components are required: a language formalism in which each policy can be expressed, and an ontology of concepts that the policy body can refer to. Alongside the concepts in the ontology, maintained within the *TBox*, we maintain a repository of knowledge about specific individuals in the domain, the *ABox*. Figure 2 shows the general structure of our ontological knowledge base, with a fragment of some typical concepts and roles in each part. We take a modular, layered approach, maintaining a domain-specific ontology and separate lower-level ontologies for each service in the domain. Together, these form a knowledge base that can be used by

the NLP analysis to help guide the interpretation of users' input, and the generation of a formal representation of each policy. Each ontology module contains definitions of concepts, and the roles played by those concepts.

In building the complete knowledge base, the ontological information is derived from a number of sources, both static and dynamic. Domain knowledge is prepared statically, whilst as services appear, the semantic service description is dynamically integrated into the ontology. We envisage eventually being able to add new concepts *learnt* from the user, such as their definition of a long document.

The ontology is necessary to process the natural language policy descriptions. Furthermore, the provision of service semantics in the TBox and ABox facilitates the composition of services to meet the demands of users. Without this information, each service could only be configured in isolation, a point well-understood within the Semantic Web community [5], and slowly becoming accepted within the pervasive middleware community[14].

3. User Policies

For the purpose of examining sample policies, we selected a concrete scenario: an office with a number of different printers, in which users can specify policies about which printer to use under which circumstances. Similar formulations could also provide access control rules by specifying policies on the *agent* and *owner* of requests and devices. We devised a small sample ontology that serves to represent the concepts and roles of services, devices and objects in the office printing environment. Figure 2 shows a fragment of this sample ontology, expressed in the DL syntax.

Our goal is to explore the design issues required to recognise and support different styles of policy. We introduce a distinction that will play a useful role in formulating our policies, which is the different modes of inference required to handle them. Because we are working with a specific domain, the interpretation can be guided and constrained by domain knowledge about the kinds of policies we expect to see.

3.1. Modes of inference in user policies

In general, policies involve a set of **pre-conditions** that express when the policy should be applied, and a set of **post-conditions** that express what should be true when the pre-conditions hold. For example, *If a printer is offline* (pre-condition), *then an e-mail should be sent to Support* (post-condition). However, the effect of applying the post-condition can vary in subtly different ways, which can be considered in terms of two modes of inference.

- *Over-write inference:* new facts are asserted when the pre-conditions apply; however in addition, any pre-existing facts that contradict the newly asserted facts are overwritten. To denote overwrite policies we use the $\Rightarrow$ symbol in the rule.
- *Default inference:* when the pre-conditions apply, the facts specified in the post-conditions are only asserted when they do not contradict pre-existing facts. For such policies, we use the $\stackrel{def}{\Rightarrow}$ symbol to denote a default inference rule.

Over-write policies In this style of policy, a request is modified under some specified pre-conditions. For example:

> *If a print request is sent to LJX and LJX is offline, then print to LJA instead.*

In this example, the post-condition changes the target of the print request referred to in the pre-condition from one value to another. To handle this type of policy, the post-condition must be made true and any contradicting statements should be over-written. We write this rule as:

$$x \in \text{Print} \sqcap target.(name.\text{`LJX'} \sqcap status.\text{OFFLINE}) \Rightarrow x \in target.name.\text{`LJA'}$$

Note here that is important that the target of x is changed from the printer with name 'LJX' to the printer with name 'LJA'.

Users may also express negative constraints:

> *Don't print colour documents on LJA*

In this example, the policy expresses a negative constraint which requires that a print job is redirected away from LJA to another (as yet unspecified) printer. Rather than cancelling the request, we need to cancel the value of the target role and leave it to be filled in by some other policy. This policy is expressed formally as:

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

Default policies In this style of policy, the user wishes to supply a default value for a role value to be filled in when it is left unspecified in a request. For example:

> *The default printer for printing colour documents is COLJX.*

Here, the pre-condition is that there is a print request involving a colour document and for which there is no printer specified. The post-condition is that the target of the print request should be COLJX. In this example, a print request which has a printer specified should not be modified. This corresponds to default inference, where the consequent should be inferred unless there is evidence to the contrary. This policy is then:

$$x \in \text{Print} \sqcap patient.colourness.\text{COLOUR} \stackrel{def}{\Rightarrow} x \in target.name.\text{`COLJX'}$$

In the same way as for over-write rules, a default rule may employ negative constraints. This enables users to specify policies such as "Unless explicitly specified, don't print documents on colour printers".

We note that the natural language expression of a policy will often be ambiguous between an **overwrite rule** and a **default rule** interpretation. For example, a user may say, *Send colour documents to COLJX*. In this case, the user may assume a default rule interpretation by the system or they may intend that *all* colour documents should be printed on COLJX regardless of the target specified in the print request. In such ambiguous cases, we intend to generate both possibilities and query the user as to which is intended. In future work, it may be possible to have **meta-policies** about which type or style of policy is most likely.

4. Policy Management

Once a user has formulated their policy using whichever modes of interaction they prefer, the result is a formally-specified policy representation: a rule, with DL terms for its pre- and post-conditions. These terms refer to concepts, roles and individuals in the Ontology, hence the policy representation is a relatively high-level intermediate form. We believe that this is an important design aspect, since our middleware components for analysing, managing and implementing policies need to work with abstractions closer to the user-level. This becomes apparent when considering the interactive and bidirectional nature of policy formulation: as the system performs analysis it can feedback information such as conflicts and ambiguity to the user, using structures and terms that are familiar. We cannot expect non-technical users to debug their policies at the system implementation level.

As indicated in the system overview of Figure 1, each user policy is passed into the middleware components (policy analysis and management) where it is integrated with the system and can then be applied to alter the behaviour of requests. These components are examined in more detail below.

4.1. Ordering and Analysis

In any realistic situation, a user is likely to have many policies about how to manage their pervasive computing environment. Furthermore, as discussed in Section 3, the post-condition of a policy can over-write information in service requests, and hence affect the applicability of other policies. This introduces the need for our system to consider the whole set of policies, and to analyse how adding a new policy affects the global behaviour. Dealing with multiple policies immediately gives rise to questions such as:

- *Ordering:* which policies should be considered before others, e.g. should specific policies be considered before general default policies?
- *Conflicts:* what should happen if two policies specify post-conditions that clash, for example.
- *Chaining:* is it reasonable for the application of one policy to have a knock-on effect that causes other policies to then be applied? Can we detect policies that may loop by feeding into each other?

To help us gain a better insight into users' expectations about the system's behaviour, we are conducting a small pilot study. The aim is to elicit users' conceptions of how a small set of policies would be applied in a simple scenario, and to improve our understanding of how policies should be ordered and applied in practice. This study is on-going work, but initial findings suggest that users do have expectations about how and when multiple policies should be applied, and we are able to feed this into the policy analysis task.

On the assumption that the policy manager must consider the set of policies in *some* order, the policy analysis component of our system has the task of determining where to place each new user policy into this ordering. It is during this process of analysis that we can detect certain conflicts and ambiguities in the set of policies, and pass this information back to the user interface to assist them in formulating their policies. By analogy with programming environments, the policy analysis phase performs static type checks on the fledgling policy.

Our use of DL for the formal policy representation is of great benefit for analysis, since there are well-defined logical tests for subsumption and disjointness that enable us to compare policies based on their pre- and post-condition descriptions. For example, this simplifies the creation of one possible policy ordering, where policies with more specific pre-conditions are considered before more general ones. Similarly, if the pre-conditions of two policies are not disjoint or related by subsumption, then we may consider this ambiguous and seek clarification from the user. The Racer system we are using as the knowledge base implements these logical operations as standard, allowing us to work with the higher-level, more abstract, DL.

Since our user studies are ongoing, the policy analysis engine has been implemented in a modular way, so that different policy ordering approaches can be tested. Our current analysis module places policies into an approximate linear order based on subsumption of pre-conditions, so that more specific policies will be examined before more general policies. This initial choice reflects our current intuitions about the system behaviour, but can be altered in the light of user study outcomes.

4.2. The Policy Manager

The policy manager is the other major component of the middleware in our system. It makes use of the ordered set of user policies produced by the analysis phase described above, by examining events and requests for service functionality and applying relevant policies to the requests before invoking the actual functions. Conceptually, the policy manager takes each request and considers policies in order, checking to see if the pre-condition is met and if so, altering the request so that it meets the post-condition of the policy. This process continues, for some sequence of policies, until the request has been modified according to the user's set of policies. The policy manager then determines from the resulting modified request what information to pass to the actual service functionality.

For example, recalling the printing example policies introduced in Section 3, an incoming Print request might initially only specify which document is to be sent. By considering and applying a sequence of policies, the policy manager alters the request so that the printer target is now specified, in accordance with the user's preferences. Ultimately, the policy manager then identifies from the resulting altered request enough information to invoke the printing service with a specific document and printer name.

Even given the existence of a policy ordering, there are a number of choices to be made in the design of the policy manager's core activity:

- How do we decide whether a request, in its current state, meets the pre-condition of some policy?
- Since policies can over-write and add new information to the request, how can these changes be managed?
- At what point does the policy manager stop applying policies and convert a request into an actual service invocation?

Again, these decisions are affected as much by user expectations of the system behaviour as by purely technical considerations. Hence, our current implementation is also modular with respect to the policy testing and application algorithms.

Our policy manager implementation takes a **constraint-based** approach, where we consider each DL term in policies and requests as an individual constraint. Then, the process of applying policies to modify requests involves building up a combination of these individual constraints that reflects the changes made to the request, in accordance with the post-conditions of policies. The use of DL is particularly helpful in this approach, because one of its key features is support for combining partial descriptions into a more complex whole. In essence, that is the task of the policy manager: taking partial descriptions (i.e. requests) and combining them with other partial descriptions (policy conditions) to produce a resulting request that satisfies the constraints made by the user's policies.

5. Discussion and Related Work

Policy management of distributed systems has been studied for many years. For example, the Ponder language [13] from Sloman et al has been used to provide a software engineering view of conflict resolution [11], whilst more recent work has used the Event Calculus approach of Bandara [3] to explicitly model how policies will interact in a specified environment. The KAoS system[19, 21] employs DLs to represent and analyse policies, but with the more specific notion of policies for access control. However, we believe we are the first to propose the use of Description Logic to model more general policies than access control, and it is part of our future work to discover mappings between our approach and these alternative approaches to policy management.

Recent interest in the Semantic Web[4] has led to a proliferation of potential ontology representation formalisms. Baader et al.[2] note that DLs are ideal candidates for ontology languages since they provide a well-defined semantics and powerful reasoning ability. Further, as noted by Stevens et al.[18], they are the underlying logical formalism of the web ontology languages OIL[7] and DAML+OIL[10]. Web Services[12] and the use of ontologies for the Semantic Web[2, 7, 15] have also examined the problem of service composition, although this has been focused primarily on *programmatic* access to services. For example, foundations such as WSDL[6] and DAML[9] enable programmable software agents to locate and compose services as required to carry out a higher-level goal. This involves the field of Artificial Intelligence (AI) Planning, by attempting to *automatically* determine how lower-level services can be built into a new higher-level service. While this approach is an interesting research area, the aim of having software agents automatically compose services is an ambitious one. In contrast, we take a complementary, user-centred approach: a key principle of our system is that the *user* is in control of how services are connected to implement a policy.

There has been work in the area of pervasive computing in integrating the ontology into the middleware [14], techniques for allowing the user to configure services [20] and approaches to using various logics to coordinate services [16]. However, there is still no commonly accepted middleware incorporating ontologies for service discovery and composition.

6. Conclusion and Future Work

We have presented evidence that typical natural language descriptions of policies controlling the disposition of re-

quests to use devices can be formalised using Description Logic, and that the DL can be used to provide an intuitive and efficient ordering for the checking of policies against incoming requests and events. Further, by checking the subsumption and disjointedness of policies, we can detect a limited set of conflicts at compile-time, and by checking the satisfiability and realisability of rewritten requests against the state of the world captured in the ABox, we are able to detect problems at run-time. We believe that this indicates that the way forward for pervasive middleware is to provide abstractions closer to the user, and thereby facilitate the users' control of their environment.

References

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

[2] F. Baader, I. Horrocks, and U. Sattler. Description logics as ontology languages for the semantic web. In *Lecture Notes in AI*. Springer, 2003.

[3] 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.

[4] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001.

[5] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1. Technical report, W3C, 2001.

[6] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1. Web Site: `http://www.w3.org/TR/wsdl`, Mar. 2001.

[7] D. Fensel, F. van Harmelan, I. Horrocks, D. McGuinness, and P. Patel-Schneider. OIL: An ontology infrastructure for the semantic web. *IEEE Intelligent Systems*, 16(2):38–45, 2001.

[8] 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.

[9] J. Hendler and D. L. McGuiness. The DARPA agent markup language. *IEEE Intelligent Systems*, 15(6), 2000.

[10] I. Horrocks and P. Patel-Schneider. The generation of DAML+OIL. In *Proceedings of the 2001 Description Logic Workshop*, pages 30–35, 2001.

[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] S. McIlraith and T. Son. Adapting golog for composition of semantic web services, 2002.

[13] E. L. N. Damianou, N. Dulay and M. Sloman. The ponder policy specification language. In M. Sloman, J. Lobo, and E. C. Lupu, editors, *Policies for Distributed Systems and Networks*, number 1995 in Lecture Notes in Computer Science. Springer Verlag, January 2001.

[14] D. O'Sullivan and D. Lewis. Semantically driven service interoperability for pervasive computing. In *International Workshop on Data Engineering for Wireless and Mobile Access*, San Diego, Ca, September 2003.

[15] A. Pease, I. Niles, and J. Li. The Suggested Upper Merged Ontology: a large ontology for the semantic web and its applications. In *Working Notes of the AAAI-2002 Workshop on Ontologies and the Semantic Web*, Edmonton, Canada, 2002.

[16] A. Ranganathan and R. H. Campbell. An infra-structure for context-awareness based on first order logic. *Personal and Ubiquitous Computing*, 7:353–364, November 2003.

[17] J. Robinson and I. Wakeman. The scooby event-based pervasive computing infrastructure. In *Proceedings of the 1st UK-UbiNet Workshop*, London, UK, September 2003.

[18] R. Stevens, I. Horrocks, C. Goble, and S. Bechhofer. Building a reason-able bioinformatics ontology using OIL. In *Proceedings of the IJCAI-2001 Workshop on Ontologies and Information Sharing*, pages 81–90, 2001.

[19] G. Tonti, J. M. Bradshaw, R. Jeffers, R. Montanari, N. Suri, and A. Uszok. Semantic web languages for policy representation and reasoning: A comparison of KAoS, Rei, and Ponder. Submitted to the International Semantic Web Conference (ISWC 03), 2003.

[20] 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.

[21] 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.

[22] 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.