Implementing Policy Management through BDI

Simon Miles and Juri Papay and Michael Luck and Luc Moreau
University of Southampton, Southampton, UK
email: sm@ecs.soton.ac.uk

Abstract

The requirement for Grid middleware to be largely transparent to individual users and at the same time act in accordance with their personal needs is a difficult challenge. In e-science scenarios, users cannot be repeatedly interrogated for each operational decision made when enacting experiments on the Grid. It is thus important to specify and enforce policies that enable the environment to be configured to take user preferences into account automatically. In particular, we need to consider the *context* in which these policies are applied, because decisions are based not only on the rules of the policy but also on the current state of the system. Consideration of context is explicitly addressed, in the agent perspective, when deciding how to balance the achievement of goals and reaction to the environment. One commonly-applied abstraction that balances reaction to multiple events with context-based reasoning in the way suggested by our requirements is the *belief-desire-intention* (BDI) architecture, which has proven successful in many applications. In this paper, we argue that BDI is an appropriate model for policy enforcement, and describe the application of BDI to policy enforcement in personalising Grid service discovery. We show how this has been implemented in the myGrid registry to provide bioinformaticians with control over the services returned to them by the service discovery process.

1 Introduction

The Grid is a truly heterogeneous, large-scale computing environment in which resources are geographically distributed and managed by a multitude of institutions and organisations [16]. As part of the endeavour to define the Grid, a service-oriented approach has been adopted by which computational resources, storage resources, networks, programs and databases are all represented by services [10]. Discovering services, workflows and data in this fluid and ever-changing environment is a real challenge that highlights the need for registries with reliable information content.

myGrid (`www.mygrid.org.uk`), a pilot project funded by the UK e-Science programme, aims to develop a middleware infrastructure that provides support for bioinformaticians in the design and execution of workflow-based *in silico* experiments [12] utilising the resources of the Grid [15]. For less skilled users, myGrid should help in finding appropriate resources, offering alternatives to busy resources and guiding them through the composition of resources into complex workflows. In this context, the Grid becomes egocentrically based around the Scientist: *myGrid*.

Middleware is generally regarded as successful if its behaviour and management remain largely invisible to the user, while still remaining efficient. However, such a requirement is in conflict with myGrid's philosophy according to which middleware should act in accordance with personal needs. Crucially, myGrid user requirements [14] have identified that final service selection ultimately rests with the scientist, who will select those to be included according to the goal of the experiment they are designing. Therefore, a service registry, the specific Grid service we consider in this paper, should be designed to provide a list of services adapted to the user's needs. Yet, realistically, users cannot be interrogated repeatedly for each operational decision pertaining to service discovery made when enacting experiments on the Grid. It is important that the environment in which the scientist works is configured prior to use so that preferences can be taken into account automatically.

The term 'user' should be understood here in the broadest sense. Indeed, the configuration may not necessarily be customised from the end-user's personal preference only, but also from their collaborators or institutions preference or recommendations. For instance, guidelines may be issued by a lab director about which services should or should not be used. System managers may also dictate service constraints, e.g., so that machines hosting them remain at an acceptable load level. All these generally complex preference requirements specify not only the functional behaviour of services, but also non-functional aspects such as security or quality of service. They cannot be programmed generically into the system, since they can vary dramatically from one deployment to the other and they potentially can change over time. Our belief is that definition of preferences with regard to a particular aspect of behaviour, in our case service discovery, should be achieved through a *policy*. Such a policy defines what is allowed, what is not allowed, and what to do (or not) in reaction to events, for all the stakeholders of the service.

While various policy definition technologies have been developed, largely focusing on the declaration of either what is or is not permitted, or what rule to follow in reaction to a given event, little consideration has been given to the context in which rules are applied. Conversely, work in artificial intelligence, planning and agent-based systems has developed approaches in which context is explicitly considered when deciding how to achieve a goal or react to the environment. One commonly-applied abstraction that balances reaction to multiple events with context-based reasoning in the way suggested by our requirements is the *belief-desire-intention* (BDI) architecture.

In this paper, therefore, we propose the application of agent-oriented engineering and the BDI architecture to policy enforcement in personalising Grid service discovery. Specifically, having adopted a message-passing architecture to implement the registry, we see such messages as events that can trigger plans, whose execution is decided based on the current state of the system and enact further goals by generating more messages. Primitive goals are executed by passing messages to the appropriate manually programmed message handlers. This provides us with a flexible architecture that is capable of routing and handling messages according to all requirements specified in the form of policies at deployment time. We show how this has been implemented in the myGrid registry to provide bioinformaticians with control over the services returned to them by the service discovery process.

In Section 2, we present the motivation for policy-controlled behaviour in Grid service discovery, give an example scenario in which policy enforcement is required and show how an intentional, agent-based perspective provides a useful way of modelling the problem. We develop the idea in Section 3 by showing how the Procedural Reasoning System (PRS) [17] is applied to the problem of policy enforcement in our design, and show the way in which this is implemented as part of the myGrid architecture in Section 4. We discuss the evaluation of our work in Section 5, related work on policies and BDI agents is discussed in Section 6 and conclusions drawn in Section 7.

2 Service Discovery Policy in myGrid

For a bioinformatician to discover tools and data reposistories available on the Grid, there must be a known *registry* in which they are advertised. However, as part of the user requirements, individual scientists have preferences about which services to use: they prefer some services over others, and sometimes want to take into account the opinions of trusted experts as to which services have performed well for them previously. In many labs, all members will use the same services and parameters to those services. This subjective information is not available in public registries and cannot easily be added (it cannot be added at all by third parties).

In myGrid, therefore, we have developed the idea of a *personalised* registry. In brief, an instance of a registry copies a filtered selection of entries from other registries and allows *metadata* to be attached to the service descriptions. Metadata is encoded information regarding, and explicitly associated with, existing data. This metadata can include personal opinions and can subsequently be used to ensure that the service discovery returns only the desired services.

To enable this personalised service discovery, a wide range of behaviours must be possible to specify, including the following:

1. Keep registry contents up-to-date with regard to another registry.
2. Add metadata to entries in the registry if allowed.
3. Perform discovery and return results to a client if allowed.
4. Keep subscribers notified of new services registered in the registry.
5. Favour some sources of data and metadata over others when multiple sources are used, retaining only the most favoured.
6. Resolve conflicts between information from different sources.

All of the above configuration options are personal to the owner of the personalised registry, and can include significant amounts of detail. In fact, all these requirements lead to a curated registry, which identifies multiple roles: owner, curator, user, with policies specifying the behaviour of the system, but also the actions permitted to each role (in role-based access control style). Other management issues to be enforced include the prioritisation of actions, which affects the responsiveness and latency of the system and an avoidance of cyclic subscriptions. To deploy a personalised registry, therefore, we need a policy must to define its behaviour, and a suitable mechanism to *enforce* policy during the operation of the personalised registry. We will argue that this is best done using BDI.

Example Scenario To better explain our approach to solving the problem of policy enforcement in this domain, in this section we present an example scenario and associated policy. Figure 1 illustrates the scenario in which the expert scientist in an organisation has a personalised registry (Registry 1) that copies the service adverts published in one or more public registries. The expert then adds a *trust value* as metadata to each service advert, indicating how reliable they have found that service. A novice in the same organisation also has a personalised registry (Registry 2) that copies the content of the expert's registry, but only where the trust value of a service is higher than a particular defined constant. The novice is the only user allowed to edit the metadata in Registry 2. This means that when the novice discovers services, they are only provided with services that the expert has judged to be regarded as trustable.

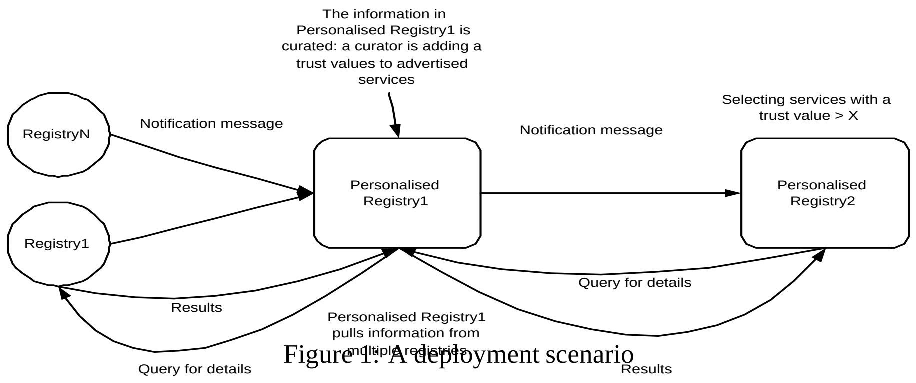
The information in Personalised Registry1 is curated: a curator is adding a trust values to advertised services
Selecting services with a trust value > X
RegistryN
Notification message
Notification message
Personalised Registry1
Personalised Registry2
Registry1
Query for details
Results
Personalised Registry1 pulls information from multiple registries
Query for details
Results

Figure 1: A deployment scenario

We now consider in detail the policy specified in order to define the desired behaviour of these personalised registries. First, on metadata being added or updated in the expert's registry, the novice's registry should query the expert's registry for the details of the service advert, and its metadata, and then save them if the metadata includes a trust value that exceeding the defined constant. Second, a service advert and associated metadata should only be kept in the novice's registry if it has been copied from the expert's registry, or if the novice has personally published it so that no other party is authorised to change the contents of the novice's registry.

In our system, the policy is specified in one or more *policy documents* following a structured XML format. Thus, the policy document for encoding the behaviour of the novice's registry is given in Figure 2, which is separated into three parts.

In the first part of Figure 2, we show the part of the policy document relating to authorisation. We group all users into a single "User" *role* and all trusted registries into a "TrustedRegistry" role. There is only one client in each specified role in the policy as it stands, but this could easily be extended later if the novice allows colleagues to use her personalised registry. The operations related to changing metadata, such as trust values, are grouped into the "Edit Metadata" operation type. We then specify that members of the "User" role are permitted to perform operations of this operation type (members of the "TrustedRegistry" role are also permitted to edit metadata).

```
<Role value = "User">
 <Client value = "Novice"/>
</Role>
<Role value = "TrustedRegistry">
 <Client value = "Expert"/>
</Role>
<OperationType value = "Edit Metadata">
 <Operation value = "AddMetadataToBusinessService"/>
 <Operation value = "DeleteMetadata"/>
</OperationType>
<Permission>
 <OperationType reference = "Edit Metadata"/>
 <Role reference = "User"/>
</Permission>
...
<Subscription>
 <Registry>
  <RegistryLocation value =
    "http://www.ecs.ac.uk/expert:8080/view"/>
   <RegistryType>
    <RegistryCallHandler value = "uk.ac.soton.ecs.views.server.impl.notifications
                                  .handlers.RegistryEventIncomingHandler"/>
   </RegistryType>
   <Client reference = "Expert" />
  </Registry>
  <Argument parameterName = "topic">
   <String value = "MetadataChanged"/>
  </Argument>
</Subscription>
...
<EntityFilter>
 <ComparisonHandler value = "GreaterOrEqualThanComparison" />
 <MetadataValueExtractor>
  <Operation value = "GetBusinessServiceMetadata" />
  <Argument parameterName = "type">
   <URI value = "http://www.ecs.ac.uk/expert/types/TrustRating"/>
  </Argument>
 </MetadataValueExtractor>
 <MetadataValueExtractor>
  <Operation value = "ReturnConstant" />
  <Argument>
   <String value = "0.7"/>
  </Argument>
 </MetadataValueExtractor>
</EntityFilter>
```

Figure 2: Policy document fragment with three parts

The second part of Figure 2 specifies that the expert's registry should notify the novice's registry whenever a piece of metadata is added or changed. This is because the metadata may include a trust value provided by the expert and we may wish to copy the service advertisement to which it is attached into the novice's registry. Such a request for notification is called a *subscription*. Finally, the third part of Figure 2 specifies that the novice's registry filter ensures only services with a trust value exceeding 0.7 should be included.

Interpretation in Intentional Terms Registries satifying the requirements above are crucial for several reasons. First, they are federated, with annotations made in one registry being communicated to another without guarantee of their inclusion in the latter: this is *flexible*, *social* behaviour. Second, the registries are *autonomous* in that they poll other registries according to some query, and *reactive* in that they may incorporate the results into their own repository, both within the current environmental context provided by the policy and through communication from other registries. Such flexible, autonomous and reactive behaviour, which shares many characteristics with the notions underlying agent systems, suggests that an agent approach may offer a useful framework.

In this way, the services other than the registry can also be viewed as agents, which may be represented in the system as automatic publishers (or re-publishers) of services into multiple registries, as automated discoverers of services to be included in workflows, as personal agents adjusting service discovery to a user's preferences and as automated executors that handle the invocation, composition and failure of services. This usefully identifies the application as a whole as a multi-agent system.

To allow this to be applied in the design of an individual agent, we can view the registry as an entity with multiple intentions it fulfills in line with environmental conditions (the *context* as defined earlier). Since registries themselves are tradionally viewed as passive repositories of information, it is perhaps clearer to view the agent in this system as an entity *managing* the content of the registry rather than the registry itself, so that the agent enforces the management *policy* of the registry. (Note, however, that there is also much work on agent brokers and mediators addressed in a different context [19, 22]. Although these address a different problem of matching, they might use policy-based registries to achieve such increased functionality.)

This policy enforcement agent can be seen as instantiating as intentions the goals expressed above. More specifically, the agent must take into account context in fulfilling its intentions. For example, in keeping the most recent version of a service advertisement in the registry, the currently stored version must be taken into account. Similarly, in copying service advertisements from the most trusted other registries, the sources of an advertisement and trust in those sources must be compared.

3 Using BDI to Enforce Policy

In order to enforce management policy in a registry, we create an agent that processes the goals of the policy and the operations performed on the registry using the *belief-desire-intention* model and its specific instantiations as the *procedural reasoning system* (PRS) and the *distributed multi-agent reasoning system* (dMARS) [6]. The choice of this policy enforcement mechanism was due to the particular requirements of registry management. In particular, decisions on policy are triggered by external events and different decisions are made depending on both the policy and the contents of the registry. Other policy enforcement technologies are discussed in section 6; in this section, we show how this BDI model fits well with our requirements.

The benefits of modelling registry policy enforcement in terms of agents are two-fold. First, at the level of the individual agent, we can map our requirements to proven agent technologies and re-use them for fulfilling intentions in a flexible, autonomous, context-sensitive way. In this way, we adopt the BDI model, which can manage multiple intentions achieved in different ways depending on context. We discuss how the BDI model has been applied to registry policy enforcement below. The second benefit is in easing the complexity of understanding the behaviour of multiple, federated registries. As the behaviour of a registry is controlled by a flexible policy enforcement agent, we can assume that each will robustly interpret any data sent to it by other registries, without consideration of how it is processed or how it may conflict with other operations the agent is performing.

The Procedural Reasoning System PRS is an agent architecture that balances reaction to the environment (i.e. doing something appropriate) with structured activity based on knowledge (i.e. planning in advance for complex actions). An agent's *beliefs* correspond to information the agent has about the world, *desires* (or goals, in the system) intuitively correspond to the tasks allocated to it. The intuition is that an agent is unable to achieve *all* its desires, even if these desires are consistent. Agents must therefore fix upon some subset of available desires (or *intentions*) and commit resources to achieving them until either they are believed satisfied or no longer achievable [5].

The model is operationalised by pre-defined *plans* that specify how to achieve goals or perform activities. Each agent has a *plan library* representing its *procedural knowledge* of how to bring about states of affairs. Plans include several components: a *trigger*, which indicates the circumstances in which plan should be considered for use; a *context*, which indicates when the plan is *valid* for use (and is a formal representation of the context of the registry as discussed in the introduction), and a body comprising a set of *actions* and *subgoals* that specify how to achieve the plan's goal or activity.

In the PRS view, triggering events, e.g. user commands, appear at unpredictable times in the external environment and within the agent. On perceiving an event, an agent compares it against the plans in its plan library and, for each plan with a matching trigger, the *context* is compared to the agent's current beliefs. When the trigger and context of plans match the current event and beliefs, plans are committed to being achieved, and they become intentions, which are active threads of control that enact plans. As an agent runs, it progressively enacts its intentions by performing the actions specified in the intended plan one at a time one. When all actions have completed, the intention is achieved. Subgoals in a plan allow parts of the plan to be tailored to the current circumstances that may have changed since the plan started (because the newly triggered subplans are chosen based on their contexts). The activity of triggering new plans and transforming them into intentions combines with enactment of intentions to allow such agents to balance reactivity with advance planning.

Broadly, we can see how each feature is useful in meeting our requirements. An operation call by a user to the registry (e.g. publishing or discovering a service, or information being received from another registry) is represented by a message, the receiving of which acts as a triggering event for PRS plans. The context of a plan can represent decisions on whether to allow an action, such as publishing a service, based on the authorisation of the caller, the content of the advertisement or the current content of the registry. The actions of a plan are performed by sending messages (either the ones received or new ones created according to the plan specification) to the appropriate handler, which implements the required business logic. Finally, subgoals can be used to both separate concerns and ensure the most recent context is taken into account on each decision. For example, copying information from another registry may involve checking both the information content and the authorisation applied to the registry, and these steps can be separated by the use of subgoals.

Example PRS Plan In our system, a set of policy documents, written in XML, are parsed on creating a registry and used to construct a running PRS agent with a set of PRS plans. The policy can then be altered at run-time to, for example, permit new users to perform service discovery.

In the example scenario given in Section 2, services are copied from the expert's registry into the novice's registry only if the trust value assigned by the expert is greater than 0.7. The policy requires several plans to be defined to ensure appropriate behaviour over time, including those encoding the following behaviours. On startup, the novice's policy agent should ask the expert's policy agent to inform it when trust values are added to or changed for the services in the expert's registry. Then, every time a new service advert is published, either by a user or copied from another registry, the agent should ensure that the user/registry is authorised to provide such information before it can be saved. When the agent is informed that a trust value has been added to a service in another registry, it should decide whether to include the copy of the service.

The `TrustValueAdded` plan for this final behaviour is shown in Figure 3, which expresses the plan structure clearly, based loosely on the AgentSpeak(L) language [20]. At the top of the plan is the trigger for considering when to form an intention to enact the plan. This trigger is a piece of metadata representing a trust value being added to a service in a remote registry, with the service being identified by a unique *service key* as is the convention in UDDI. In the second section, the context determines whether the plan is applicable whenever the trigger occurs. The context compares the trust value to the constant given in the policy (0.7), and if the trust value is high enough, the actions are performed. These are the last four lines of the plan, performing the following: retrieve the details of the service advert from the remote registry, save those details, retrieve the metadata attached to the service advert and save that metadata. The retrieval operations are primitive actions performed on the remote registry, while the save operations are subgoals to trigger the processing of new intentions. The value in using subgoals in this example is that the authorisation to copy the contents of a remote registry can be checked at the time of the save operation.

For each operation, the authorisation is checked before it can be performed. In the `SaveService` plan of Figure 3, we show the SaveService message being triggered, due to a subgoal from the plan in Figure 3 or because a user has called the saveService

```
TrustValueAdded(remoteRegistry,serviceKey,trust)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
? trust >= 0.7
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
details=getServiceDetails (remoteRegistry, serviceKey)
! SaveService (details)
metadata=getServiceMetadata(remoteRegistry, serviceKey)
! SaveMetadata (serviceKey, metadata)


SaveService (serviceDetails)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
? inRole (role, caller)
? isPermitted (SaveService, role)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
UDDIPublishHandler.saveService (serviceDetails)
```

Figure 3: Plans for copying services and saving service adverts

operation to publish a service. In the context the plan first finds in the agent's belief store the roles in which the client sending the message is active, and then checks that at least one role is permitted to perform SaveService. If so, then a message is sent to the UDDI publish handler to actually save the service.

4 Implementing Policy Enforcement

The above sections justify the use of PRS and describe how policy documents can be mapped to PRS plans, but we still need to address how the policy is practically enforced within the registry. In order to do this, we need to introduce the architecture of the myGrid registry.

The myGrid registry is modular and extensible, allowing deployers to choose which protocols will be supported. At the moment the registry can support several protocols including UDDI version 2, a protocol for parsing and annotating WSDL documents and a generic protocol for attaching metadata to service descriptions. Each operation call is processed by a chain of handlers performing the business logic of the operation. The PRS agents are inserted into the handler chains to ensure that the policy is enforced at all times.

As shown in Figure 4, a registry is created by providing a set of policy documents to a registry factory. The policy describes a set of policy objects controlling how the registry behaves. These policy objects are explicitly instantiated when the documents are parsed. They are then translated into Procedural Reasoning System plans to be interpreted by a set of PRS Enforcer agents, one for each protocol that the registry is configured to process. Whenever a user of the registry makes an operation call, the call is transformed into a message, that triggers a PRS Enforcer before the message (or a newly generate one) can be processed by the registry's handlers. The policy can be modified during the registry's lifetime by the registry administrator via calls to a generic policy object manipulation API.

The transformation of policy documents into PRS agents can best be illustrated by referring back to the policy fragments in Figure 2. The first fragment translates directly to terms in the context of plans to save metadata, denying the action if the permission

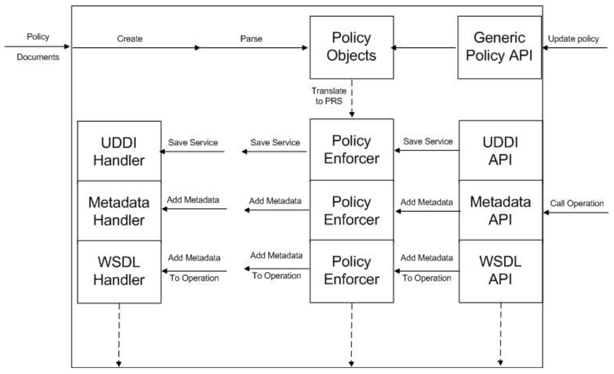

Figure 4: The architecture of a myGrid registry

does not exist, the second maps to the triggering event of the `TrustValueAdded` plan, and the third fragment is used as the context of that plan .

5 Evaluation

A small but increasing number of bioinformatics services are being made available as Web Services by public and commercial organisations. These include XML Central of DDBJ [4], who have exposed many popular bioinformatics tools, such as Blast, the gene sequence similarity search tool, and Protein Data Bank Japan [3]. In addition, the European Bioinformatics Institute [1], as part of the myGrid project, has exposed many of its tools as Web Services to support the day-to-day research by biologists at the University of Manchester and the University of Newcastle, and these are gradually being moved into the public domain.

We have successfully tested our system using the scenario described earlier, as well as more simple cases, using the interfaces and descriptions of these existing bioinformatics services (these are available from the websites of the organisations mentioned above). While this does demonstrate that our approach does as it states it should, the evaluation of personalisation is made difficult by the subjective and qualititative nature of judging it to be a success. The requirements for personalisation come directly from biologists [14] and, at the moment, we have anecdotal evidence from bioinformaticians that it would be a useful tool on transferring to a new organisation, for example. However, the true value cannot be interpreted until myGrid becomes more widely adopted within organisations. In addition, the personalised registry will continue to be developed and tested as part of the Open Middleware Infrastructure Institute programme [2], which has a wide user base including engineers, physicists, chemists and biologists.

6 Related Work

Policy languages have been the focus of much attention lately in the Grid and Web Services community. Bradshaw *et al.* [11] discuss the notion of policy in the context of autonomous entities that cannot always be trusted to regulate their own behaviour appropriately, because poorly designed, buggy or malicious. In this context, they introduce KAoS, a policy language that allows them to externally adjust the bound of autonomous behaviour in order to ensure safety and effectiveness of the system. Specifically, their policy language and associated enforcers allow them to dynamically regulate the behaviour of system components without changing their code, not requiring the cooperation of the components being governed. The KAoS policy language is based on the OWL ontology language, and distinguishes *permitted actions* from *obligated actions* [21]. KAoS can therefore be seen as a specification language, identifying some of the properties that are expected from a system. Interestingly, obligations and permissions policies may not necessarily be enforceable, and this remains an open problem at the moment.

The recent proliferation of policy-based specifications in the Web Services community is also relevant to this line of work. WS-Policy is a framework for indicating a service's requirements and policies [7]. WS-SecurityPolicy defines extensions for security properties for Web services [8]. Other proposals for security policy exist in the context of Web Services: XACML (eXtensible Access Control Markup Language) is an OASIS specification that defines an XML schema for an extensible access-control policy language [23]; Denker *et al.* [9] define an OWL ontology of security mechanisms, which they use to specify which security mechanisms are supported by Semantic Web Services. Such security requirements can be regarded as an obligation policy that client must satisfy in order to interact with a service. Similarly Kagal *et al.* [18] also define a policy language with constructs for obligations and authorisations. Their policy statements are also used during matchmaking. Ponder [13] is a declarative, object-oriented language for specifying security policies with role-based access control and general-purpose management policies for specifying what actions are carried out when specific events occur within a system or what resources to allocate under specific conditions. Enforcement is typically delegated to *enforcement agents* that intercept actions and check whether the access is permitted.

7 Conclusions and Further Work

The demand for personalisation of the Grid in a wide variety of applications has driven the need for policies which configure the resources available to the needs of individual users. However, the dynamic distributed environment requires policy enforcement mechanisms that act in a timely and context-dependent way. These requirements are exactly those addressed by agent-based systems and solved by existing agent technologies such as BDI. In this paper, we have presented our approach to policy enforcement in service discovery, as applied to the bioinformatics application domain. In our approach, policy documents are written by a user to define the behaviour of a personalised service registry. These are then translated into a set of PRS plans, enacted by a policy enforcement agent whenever triggered. Acting as part of the handling architecture of a registry, the agent can ensure that authorisation rights are observed and that content is shared between registries by continued communication. We will be testing our policy approach further within the context of the myGrid architecture in supporting bioinformaticians. We also intend to look at the verification of policies, by reasoning over their data representation. Additionally, we intend to look at applying our policy enforcement to configuring the agent matchmaking process, which can be seen as an extension of service discovery.

8 Acknowledgements

This research is funded in part by the myGRID e-Science pilot project grant (EPSRC GR/R677643), the Overseas Travel Scheme of the RoyalAcademy of Engineering, EPSRC project (reference GR/S75758/01) and the PASOA project (EPSRC GR/S67623/01).

References

[1] European bioinformatics institute. http://www.ebi.ac.uk, 2004.

[2] Open middleware infrastructure institute. http://www.omii.ac.uk/, 2004.

[3] Protein data bank japan. http://pdbj.protein.osaka-u.ac.jp/, 2004.

[4] Xml central of ddbj. http://xml.ddbj.nig.ac.jp, 2004.

[5] P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.

[6] M. d'Inverno et al.˙ The dmars architechure: A specification of the distributed multi-agent reasoning system. *Autonomous Agents and Multi-Agent Systems*, to appear, 2004.

[7] D. Box et al.˙ Web services policy framework (ws-policy). http://msdn.microsoft.com/webservices/default.aspx?pull=/library/en-us/dnglobspec/html/ws-policy.asp, 2003.

[8] G. Della-Libera et al.˙ Web services security policy (ws-securitypolicy). www.ibm.com/developerworks/library/ws-secpol/index.html, 2002.

[9] G. Denker et al.˙ Security for daml web services: Annotation and matchmaking. In *Second International Semantic Web Conference*, Sanibel Island FL, 2003.

[10] I. Foster et al.˙ The Physiology of the Grid — An Open Grid Services Architecture for Distributed Systems Integration. Technical report, Argonne National Laboratory, 2002.

[11] J.M. Bradshaw et al.˙ *Computational Autonomy*, chapter Dimension of Adjustable Autonomy and Mixed-Initiative Interaction. Springer, 2004.

[12] L. Moreau et al.˙ On the Use of Agents in a BioInformatics Grid. In S. Lee, S. Sekguchi, S. Matsuoka, and M. Sato, editors, *Proceedings of the Third IEEE/ACM CCGRID'2003 Workshop on Agent Based Cluster and Grid Computing*, pages 653–661, Tokyo, Japan, 2003.

[13] N. Damianou et al.˙ Ponder: a language for specifying security and management policies for distributed systems. *Imperial College Research Report DoC 2000/1*, 2000.

[14] R. Stevens et al.˙ Performing *in silico* Experiments on the Grid: A Users Perspective. In S. Cox, editor, *Proceedings of the UK OST e-Science Second All Hands Meeting 2003*, pages 43–50, Nottingham, UK, 2003.

[15] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid. Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 2001.

[16] Ian Foster. What is the grid? a three point checklist. http://www-fp.mcs.anl.gov/ foster/, 2002.

[17] M. Georgeff and A. Lansky. Procedural Knowledge. *Proceedings of the IEEE*, 74(10):653–661, 1986.

[18] L. Kagal, T. Finin, and A. Joshi. A policy based approach to security for the semantic web. In *Second International Semantic Web Conference (ISWC2003)*, Sanibel Island FL, 2003.

[19] M. Klusch and K. Sycara. Brokering and Matchmaking for Coordination of Agent Societies: A Survey. In A. Omicini et al., editor, *Proceedings of Coordination of Internet Agents 2001*. Springer, 2001.

[20] A. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Seventh European Workshop on Modelling Autonomous Agents in a M ulti-Agent World*, 1996.

[21] A. Uszok, J. Bradshaw, and R. Jeffers. Kaos: A policy and domain services framework for grid computing and semantic web services. In *Proceedings of the iTrust'04 conference*, 2004.

[22] C. Wong and K. Sycara. A Taxonomy of Middle Agents for the Internet. In A. Omicini et al., editor, *Proceedings of the Fourth International Conference on Multiagent Systems*, pages 465–466, 2000.

[23] Oasis extensible access control markup language. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml.