Behavioural Specification of Grid Services with the KAoS Policy Language

Luc Moreau[1] Jeff Bradshaw[2] Maggie Breedy[2] Larry Bunch[2] Pat Hayes[2]
Matt Johnson[2] Shri Kulkarni[2] James Lott[2] Niranjan Suri[2] Andrzej Uszok[2]
(1) Electronics and Computer Science, University of Southampton
(2) Institute for Human and Machine Cognition (IHMC)
L.Moreau@ecs.soton.ac.uk, jbradshaw@ihmc.us, mbreedy@ihmc.us, lbunch@ihmc.us, phayes@ihmc.us
mjohnson@ihmc.us, skulkarni@ihmc.us, jlott@ihmc.us, nsuri@ihmc.us, auszok@ihmc.us

Abstract

Complex services in Service-Oriented Architectures such as the Grid typically require to be configured in multiple ways that cannot be anticipated by service designers; we illustrate this requirement by studying the myGrid registry, a Grid Registry capable of supporting annotations of service descriptions by third-party users. Instead, services have to be conceived so that they can be configured at deployment and run time. We argue that KAOS *is a powerful and flexible language that can help define such configurations. Using our registry case study, we examine the requirements that the definition of such complex configurations brings on policy languages and explain how they can be satisfied. Specifically, we use* role-value maps *to express constraints between property values; we introduce a notion of* PolicySet *with associated parameters that support constraints within a well defined scope; finally, we define a notion of* Context *that allows us to refer to property values that were extant in past execution environments. Essentially, these concepts allow us to add constraints to values in policy definitions, to organise policies in coherent and structure blocks, and to refer to the execution history. The paper discusses these concepts and how they are implemented in a binding of the* KAOS *policy language to the myGrid Registry.*

1 Introduction

Complex services in Service-Oriented Architectures such as the Grid typically require to be configured in multiple ways that cannot be anticipated by service designers. Consider the myGrid registry [17], which is capable of hosting service descriptions and third-party annotations; annotations include semantic descriptions of services, trust or accuracy information about them, their usage policies or simply general statistics about their use. Such Grid registries may be deployed in many different ways: e.g., they may be federated or replicated, they may hold annotations to be curated by experts, their access control may be role based. As registries operate semi-autonomously, some additional external constraints may be set by institutions that host them: e.g., about the computing resources they use or about the domains of services being registered.

The variety of service configurations is such that service designers cannot anticipate them all. It is therefore a requirement that such services be provided with the means to specify their configurations, hence giving them the flexibility that is desired at deployment or run time. We argue that policy languages, such as KAOS [22], Ponder [5], or Rei [14] can be utilised for configuring complex grid services. In particular, the KAOS language, which has been used for grid security [13, 23] or mobile agents, is an interesting candidate since it is conceived to constrain behaviour of (semi-)autonomous entities through the use of enforcement mechanisms [4]. Another benefit that is relevant to this context is that KAOS is based on the ontology language OWL [1] and therefore OWL-based reasoning can be applied to policies for enforcing them or for detecting conflicts they may entail.

We therefore undertook to use the policy language KAOS to specify configurations of the myGrid registry [17], which had already been designed to be modularly organised around a set components [18]. Three different usages of KAOS policies have been adopted in our endeavour. Policies are used to *(i)* to express the behavioural specification of components; *(ii)* to define components instances and aggregate them in meaningful compositions; and, *(iii)* to characterise overarching constraints over registries.

Doing so, we have encountered an interesting set of new requirements, which we have addressed by providing extensions to KAOS. These extensions will be discussed in this paper: *(i)* We made extensive use of role-value maps [20] to add constraints to property values in concept definitions, which would otherwise be not expressible in OWL. *(ii) PolicySets*, defined as first-class OWL concepts, can be extended with properties and can be referenced by policies belonging to these sets. *(iii)* Finally, we recognise that a policy language, in particular one that contains obligation policies, has an underlying computational model; we make past execution history explicit through an OWL-concept of *context*.

This paper is organised as follows. We provide further motivation for this work, by detailing a flexible Grid Registry, whose behaviour needs to be specified and constrained at deployment- and run- times (Section 2). We summarise key concepts of the policy language KAoS, which we see as a powerful fit for such an application (Section 3). We then discuss how policies can contribute to the configuration of such a Grid Registry (Section 4). This is followed by an analysis of the requirements that this application introduces on a policy language and the KAoS solutions to address them (Section 5), and an overview of our on-going implementation (Section 6). Finally, we conclude the paper by a discussion and a summary.

2 Background Motivation: a Flexible Grid Registry

The Grid is a large scale computer system that is capable of coordinating resources that are not subject to centralised control, whilst using standard, open, general-purpose protocols and interfaces, and delivering non-trivial qualities of service [10]. 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 [11]. In this context, a service is a network-enabled entity capable of encapsulating diverse implementations behind a common interface. A service-oriented view is powerful since it allows the composition of services to form more sophisticated services.

Service discovery is a difficult task in large-scale open distributed systems such as the Grid and Web, due to the potentially large number of services advertised. In order to filter out the most suitable services for the task at hand, many have advocated the use of semantic descriptions that qualify functional and non-functional characteristics of services in a manner that is amenable to automatic processing [2, 7, 24].

As part of the myGrid project (www.mygrid.org.uk), a service directory was designed for hosting semantic descriptions of services, including their functionality and their semantic inputs and outputs [15]. Semantic descriptions are currently expressed in an OWL ontology [24]. A key characteristics of our approach is that semantic descriptions need not be published by service providers, but can be made available by third party users [15]; such an approach supports collaborative practices, since users can utilise services because others found them useful or efficient to perform specific tasks.

We have identified different ways of deploying a Grid registry; the following use cases illustrate configurations that are deemed desirable in different circumstances. *(i)* First, a Grid registry can be deployed as a standalone service, presenting the set of interfaces required by its deployer, and whose availability to clients is prescribed by its security policy. *(ii)* Second, users could deploy it as a "proxy" to a publicly available registry for which they do not have write access [15]; the proxy would typically tunnel queries to the public registry, but would hold locally any metadata information about registered services, and would therefore act as a personalised registry. *(iii)* Alternatively, the service could also federate entries from multiple registries. Such configurations, and many others, cannot be frozen at design time, but they need to be decided at deployment time according to the deployer's needs, or possibly at run time according to the users' needs. Therefore, the architecture of such a registry must be designed to allow easy configuration or re-configuration at deployment and run time, respectively.

Against this background, the myGrid registry implementation adopts a message-passing metaphore, in which messages are handled by components we refer to as *handlers* [18]. Handlers are typically designed to process messages of a same category, such as the messages of the UDDI publish interface, of the UDDI inquiry interface, or messages related to metadata; handlers in fact contain the business logic implementing some ports of the registry's WSDL interface.

To focus the discussion, we consider a typical distributed deployment of the registry, supporting some myGrid use cases. Figure 1 illustrates the scenario in which an expert scientist in an organisation has a personalised registry (Registry 1) that copies the service adverts published in one or more public remote registries. The expert then adds a trust value as metadata to each service advert, indicating how reliable they have found the service. A novice in the same organisation owns a personalised registry (Registry 2) that subscribes to notifications of changes in Registry 1, and copies new entries from Registry 1, if the trust value of such an entry is higher than a particular defined constant. The novice is the only user allowed to edit the metadata in Registry 2. As a result, all services discovered by the novice have been judged to be trustable by the expert.

The configurations of registries 1 and 2 are substantially different: Registry 1 sets up subscriptions from multiple public remote registries, has its trust metadata (among others) curated by an expert user, whereas Registry 2 subscribes to changes in metadata from Registry 1, and automatically updates its contents when trust values assigned by the expert are above a given threshold. We cannot expect the registry designers to anticipate all such possible configura-

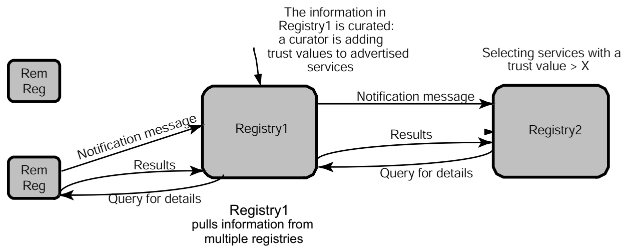
The information in Registry1 is curated: a curator is adding trust values to advertised services
Selecting services with a trust value > X
Rem Reg
Rem Reg
Registry1
Registry2
Notification message
Notification message
Results
Results
Query for details
Query for details
Registry1 pulls information from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

Figure 1. A Deployment Scenario

tions. Hence, we believe that a policy-based mechanism to specify configurations of such services would be very valuable. In Section 4, we discuss the different use of policies that we foresee in this context, but beforehand we introduce the KAoS policy language and its terminology.

3 KAoS Policy Language Overview

We refer the reader to previous KAoS publications such as [4, 22, 13] for a detailed presentation of the KAoS language and its associated graphical interface KPAT. In this section, we introduce the concepts of KAoS that we use in this paper.

KAoS key entity is the policy that can express authorization (i.e., constraints that permit or forbid some action) or obligation (i.e., constraints that require some action to be performed, or else serve to waive such a requirement) for some type of action performed by one or more actors in some situation. A policy is represented as an ontology subclass of one of the four types of policy classes: positive or negative authorization, and positive or negative obligation. An important property value is the name of a *controlled action* class, which is used to determine the actual meaning of the policy. Authorization policies use it to specify the action being authorized or forbidden, whereas obligation policies use it to specify the action being obliged or waived. Another property value is the *trigger* which identifes an action that will trigger the enactment of a policy. Finally, policies are bundled together into named *policy sets*.

Figure 2 illustrates the syntactic notation that we use in this paper and is shown by the KPAT editing interface. It contains the specification of a positive obligation policy p that mandates actors of class X to perform a **CommunicationAction**, which is a predefined concept in the KAoS ontology. The communication action is to send messages of class **Bar** to recipients of class Y; such a communication action is referred to as the “controlled action” of the policy, or control for short. In Figure 2, the trigger of the policy is itself another communication action that must take place to enable the obligation. The trigger requires an obligation’s incumbent of class X to be the recipient of a message of class **Foo**. Informally, the policy requires any X to send a message of class **Bar** to some Y, whenever it receives a message of class **Foo**. For instance, such an obligation policy could be instantiated to record incoming messages into a logging service. Concepts **Foo** and **Bar** must themselves be defined in an ontology; in our case study, they will denote messages that are used by the registry implementation.

Policy p:
X is obligated to perform
  CommunicationAction *with properties*
    hasDestination *is subset of* Y
    carriesMessage *is subset of* **Bar**
when X performs
  CommunicationAction *with properties*
    hasDestination *is subset of* X
    carriesMessage *is subset of* **Foo**

Figure 2. KAoS Obligation Policy

4 Multiple Levels of Policy Usage

In this Section, we analyse how policy languages can help specify the behaviour of services, and in particular grid registries. As an illustration, we consider the deployment scenario introduced in Section 2, and specifically, the configuration of Registry 2 for the novice. Figure 3 shows the expert registry and the contents of the novice registry. A notification message, in transit from the expert registry to the novice registry, is meant to indicate that some metadata has been updated in the expert registry. Such a notification message is received by the event handler in the novice registry, which needs to be configured so as to behave according to the scenario described previously. We now introduce different types of policy usage in the context of the registry, namely inside components, registry level and external to the registry.

Inside component Upon receiving a MetadataChanged notification message, the EventHandler in the novice registry needs to query the expert registry to obtain the details of the metadata that has changed. If such a metadata pertains to the trust given to a service, than the service detail (and all its metadata) should be retrieved from the expert registry and saved into the novice registry. Such a behaviour can be characterised by five obligation policies: *(i)* obtain trust metadata from expert registry and check it is greater than threshold; *(ii)* obtain service detail from expert registry; *(iii)* save service detail in novice registry using the UDDI publish interface [21]; *(iv)* obtain service metadata from expert registry; *(v)* save service metadata in novice registry using the metadata handler interface [17]. Such behaviour can be specified by obligation policies, which aggregated in a PolicySet, will define a component’s behaviour.

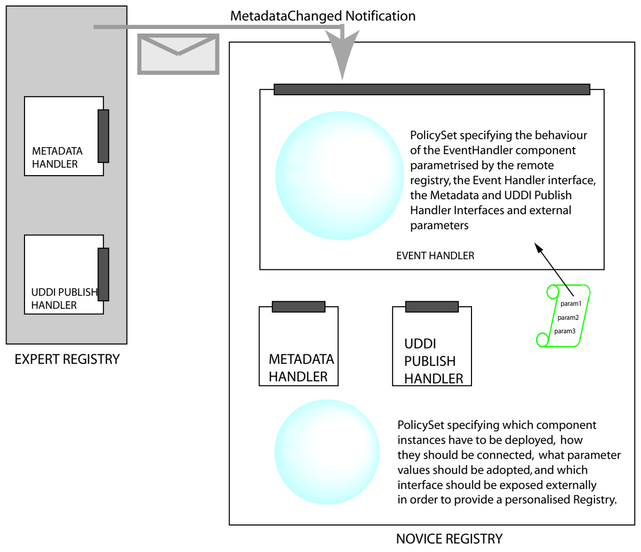
MetadataChanged Notification
METADATA HANDLER
UDDI PUBLISH HANDLER
EXPERT REGISTRY
PolicySet specifying the behaviour of the EventHandler component parametrised by the remote registry, the Event Handler interface, the Metadata and UDDI Publish Handler Interfaces and external parameters
EVENT HANDLER
param1
param2
param3
METADATA HANDLER
UDDI PUBLISH HANDLER
PolicySet specifying which component instances have to be deployed, how they should be connected, what parameter values should be adopted, and which interface should be exposed externally in order to provide a personalised Registry.
NOVICE REGISTRY

Figure 3. Policies in Novice Registry

Registry Level Given a set of components, either programmed directly or specified by policies, a given deployment of the registry will identify how such components need to be connected. Again, such configuration cannot be hard-coded by registry implementers but must be specifiable by deployers. In our case, the policy will identify three component instances (as depicted in Figure 3), a set of parameters for example read from a configuration file, and how each of these components will be connected.

Single and Distributed Registries Overarching Constraints Registries may have to conform to the rules set by the institution in which they are deployed. Examples of such rules are: *(i)* A registry may be authorised to replicate data only from a trusted remote registry, and it is at the institutional level that such a notion of trusted registry is specified. *(ii)* A collection of registries may be authorised to replicate data from each other, but a policy may force them to avoid circularity in data being replicated. *(iii)* An institution may set constraints on the set of resources a registry can use (such as processor or disk space). *(iv)* Likewise, an institution may require services in a given registry to be located within some administrative domain (e.g., Southampton services), or to be related to a given application domain (e.g., bioinformatics).

5 Policy Language Features

We now illustrate how specific application requirements can be expressed with KAoS policy language.

5.1 Action Must Refer to Trigger Properties

There is a type of constraints that is very common in practical policy definitions, as illustrated by the following examples: given a message from an entity X, the obligation is to send an acknowledgement back to X; or given a message with value v, the obligation is to send a message with value $v+1$. In such cases, the trigger of an obligation policy identifies a value, and the action of this policy is to perform an action that is parameterised by such a value.

Such constraints could naturally be expressed using a notion of variable: the trigger declares a variable, which is computationally bound to a value when the trigger of the policy is verified to be satisfied; the value the variable refers to is itself used when the variable is referenced in the action. KAoS policy language is based on OWL which does not offer variables directly; instead, KAoS relies on a mechanism, called role value maps [20], to express such constraints. Figure 4 illustrates how role value maps have been put into practice in an example obligation policy. Using the notation:

GetBusinessServiceMetadata → **hasMetadataKey**
equals
Trigger→**carriesMessage**→**hasMetadataKey**,

the set of values of the **hasMetadataKey** property of **GetBusinessServiceMetadata** messsages in the policy control is prescribed to be equal to the values of the property **hasMetadataKey** of the messages that are values of the property **carriesMessage**, for the obligation trigger. Likewise, the following notation:

MetadataChanged → **hasMetadataKey**
equals
Control→**carriesMessage**→**hasMetadataKey**.

requires the values of the property **hasMetadataKey** to be equal to values of the same property for messages in the Control. Within a policy specification, the reserverd words *Control* and *Trigger* refer to the terms respectively denoting the control and trigger of the current policy being defined.

5.2 Policy Set Context

A component's behavioural specification can be defined by a policy set; hence, within this set, a given policy may be parameterised by some parameters whose values are shared by its other policies. For a component specification, the deployment configuration of the registry may specify that different instances of the component should be deployed with different parameter values. Therefore, each policy instance in a given policy set instance can refer to the parameter values that were specified for this set.

```
EventHandler is obligated to perform
   CommunicationAction with properties
      hasDestination is subset of Registry
      carriesMessage is subset of
         GetBusinessServiceMetadata
            hasMetadataKey equals
               Trigger→carriesMessage
                            →hasMetadataKey

when EventHandler performs
   CommunicationAction with properties
      hasDestination is subset of EventHandler
      carriesMessage is subset of
         MetadataChanged
            hasMetadataKey equals
               Control→carriesMessage
                            →hasMetadataKey
```

Figure 4. Role Value Maps in Obligation Policy

Programming languages have a similar analogy. Consider that a component behavioural specification is implemented by a Java class, with a constructor initialising some private instance variables; each policy could be implemented by a method (or possibly inner class); policies would be entitled to refer to instance variables. Java scoping rules would ensure that private instance variables remain visible to the current block, i.e. the component. Components instances can be created by instantiating the class: instance variables would then be entitled to be bound to different values in different component instances. The role-value maps introduced in Section 5.1 can be used to express such a notion. Figure 5 provides us with an illustration.

Using role value maps and a keyword *PolicySet*, we can express contraints between concept properties and PolicySet properties. In Figure 5, we set a constraint to the concept **MetadataChanged** by requiring the values of its **hasTrust** property to be greater than the values of the **hasThreshold** property of the *PolicySet* the policy belongs to.

hasTrust *greaterThan PolicySet* → **hasThreshold**

In Figure 5, we show how a component behaviour can be specified by a set of policies, including p_1 we have just defined. Such a behaviour can be instantiated into two different concrete components c_1 and c_2, which are each given a specific threshold value (0.5 for c_1 and 0.8 for c_2). Within component c_1, policy p_1 will be instantiated: its constraint will refer to threshold 0.5. Likewise, the instance of p_1 in c_2 will refer to threshold value 0.8. We note that policy p_1 could also be used in a different policy set, which would be instantiated to form other component instances, referring to different threshold values.

```
Policy p1:
EventHandler is obligated to perform
   CommunicationAction with properties ...

when EventHandler performs
   CommunicationAction with properties
      hasDestination is subset of EventHandler
      carriesMessage is subset of
         MetadataChanged
            hasMetadataKey equals
               Control→carriesMessage
                            →hasMetadataKey
            hasTrust greaterThan
               PolicySet→hasThreshold

PolicySet ComponentBehaviour
   hasPolicy [p1, p2, ...]
   hasThreshold [Integer]
   hasOtherParameter [Class]

Instance c1 of ComponentBehaviour
   hasThreshold 0.5

Instance c2 of ComponentBehaviour
   hasThreshold 0.8
```

Figure 5. PolicySet Properties

5.3 Execution Context

An obligation policy such as p_1 may initiate an event, whose completion may trigger another obligation policy p_2. In other words, the succession of events and triggerings of policies result in the sequenced execution of obligation policies. In some cases, a given event may trigger several obligation policies, which potentially could be executed in parallel. Hence, the computing model underlying the KAoS policy language naturally supports sequentiality and parallelism. In a policy p_2 that is executed after p_1, it is frequent that we have to refer to property values that were extant in the trigger of p_1.

Hence, KAoS provides us with a notion of *execution context*, through which we can refer to properties that held during the enactment of a previous policy. Since parallel execution may take place, it is understood that multiple execution contexts may coexist, so that each "thread of execution" is entitled to its own context.

So, similarly to reserved words to refer to the trigger and the control of a policy, the reserved word *Context* is used to refer to the execution context of a policy. Its use is illustrated by policy p_2 in Figure 6 that mandates the saving of a pair, composed of the result returned by the action in the trigger and the threshold value accessible from the execution context. Policy p_1 is the obligation that performs the action, the completion of which triggers p_2. We have introduced in policy p_1 a new clause *with context*, which allows us to specify a new context value if p_1 is activated. In this example, the new context is formed by extending the

Policy p_1:
EventHandler *is obligated to perform*
CommunicationAction *with properties*
(*see Figure 5*)

with context: extends Context with properties
hasTrust *equals Trigger*→**carriesMessage**→**hasTrust**

when **EventHandler** *performs*
CommunicationAction *with properties*
hasDestination *is subset of* **EventHandler**
carriesMessage *is subset of*
MetadataChanged
hasMetadataKey *equals*
Control→**carriesMessage**→**hasMetadataKey**
hasTrust *greaterThan*
PolicySet→**hasThreshold**

Policy p_2:
EventHandler *is obligated to perform*
SaveAction *with properties*
hasValue Pair
hasLeft *equals Trigger* → **carriesMessage**
→ **returns**,
hasRight *equals Context* → **hasTrust**

when **EventHandler** *performs*
CommunicationAction *with properties*
hasDestination *is subset of* **EventHandler**
carriesMessage *is subset of*
GetBusinessServiceMetadata
returns *equals*
Control→**hasValue** → **hasLeft**

Figure 6. Execution Level Context

existing context with a property **hasTrust** that is equal to a property accessible from the trigger.

We note that the keywords *PolicySet* and *Context* have different purposes: the former refers to the static structure of policies, whereas the latter refers to execution. From a semantic viewpoint, we allow contexts to be "extended" by adding a new property to an existing context; if the property already existed, the new property overrides the previous one. Such a non-mutable semantics of contexts provides a precise behaviour when multiple policies can be triggered in parallel and they share a same execution context[1].

5.4 Overarching Constraint

Finally, we now show an example of an overarching constraint set by the institution in which a registry is deployed. In Figure 7, we present a negative authorization policy that prevents a Registry from communicating with remote registries located outside its administrative domain. Such a policy ensures that services advertised in the Registry are not propagated to other institutions, and that the Registry's content is not directly changed by updates coming from other domains.

Policy p:
Registry *is not authorized to perform*
CommunicationAction *with properties*
hasDestination *is subset of*
RemoteRegistry
hasDomain *not equal* **Registry** → **hasDomain**

Figure 7. Overarching Constraint

6 Implementation

The myGrid registry adopts a message passing metaphor, according to which a set of components only communicate by message passing [18]; each component is implemented as a handler of messages of a given type. From an implementation viewpoint, messages are specified in the registry's WSDL interface and converted into Java classes (using the Axis wsdl2java converter). In order to structure the code, messages implement the visitor pattern; handlers are expressed as visitors for sets of messages. Such a code design insures proper compile-time type checking and promotes compositionality of components.

In order to bind the KAoS run time system to the myGrid registry, we have defined new message visitors that are capable of converting messages and their contents into communication actions understood by KAoS. KAoS offers an API that allows the construction of "*Action Instance Descriptions*" as OWL concepts with their associated properties. A querying interface allows us to identify the obligations that are triggered by a given action; to this end, KAoS uses OWL-based reasoning to find the obligations, whose triggers match the current action, and obtain the resulting controls, instantiated for the given trigger. Such actions now need to be converted back to Java messages that can be handled by the myGrid registry; they are then dispatched to the appropriate handlers, as specified by the triggered obligation. As far as authorization policies are concerned, they can be enforced using a similar technique to [13], in which an access control modules decides, using OWL-based reasoning, whether a request can be appropriately delegated to a component, or whether it should be rejected.

7 Discussion and Related Work

Policy languages have been the focus of much attention lately in the Grid and Web Services communities. Policies have usefully been applied in the context of autonomous

[1]Specifically, the notion of context that we have introduced here corresponds to the notion of environment usually used in denotational semantics of programming language as opposed to the notion of store.

services and agents that cannot always be trusted to regulate their own behaviour appropriately, because poorly designed, buggy or malicious [4]. Specifically, policy languages, such as KAoS [4] or Ponder [6], allow system designers to externally adjust the bound of autonomous behaviour, in order to ensure safety and effectiveness of their system: policy languages and associated mechanisms to enforce them allow the dynamic regulation of the system components behaviour without changing their code, nor requiring the cooperation of the components being governed. Policy languages have also been adopted by the Web Services community, in particular, in the context of security: for instance, WS-Policy is a framework for indicating a service's requirements and policies [3].

WS-Policy introduces a simple grammar for expressing the capabilities, requirements, and general characteristics of Web Services and their clients. Policy expressions allow for domain-specific declarative and conditional assertions. WS-Policy defines a policy to be a collection of one or more policy assertions. WS-Policy, in comparison to KAoS, takes a different view on policy semantics since it defines policy from the perspective of capabilities or behaviour required to access particular service. Thus, the WS-Policy policies, in essence, provide lists of possible values for properties defining capabilities. As previously described, KAoS policy semantics is concentrated around authorizations and obligations, which can also be used to obtain lists of allowed values for a given service request. However, KAoS declarative definitions based on ontology allow for greater flexible computing of these lists based on the current context. In general, the fact that KAoS is using ontology both as a base of its policy vocabulary and as its policy constructs allows us to build much more complex policy definitions then WS-Policy and more importantly allows for reasoning of policy applicability and relations. Recent work by Parsia *et al.* [19] provides mappings of WS-Policy into OWL that allows them to reason about policy containment, i.e., whether the requirements for supporting one policy are a subset of the requirements for another. WS-SecurityPolicy defines extensions for security properties for Web Services [8]; SAML [12], the Security Assertion Markup Language, is a framework for exchanging authentication and authorization information.

Among the plethora of policy languages, KAoS adopts the original approach of expressing policy specifications in the ontology language OWL [1], which gives it a number of significant advantages. First, most policy specifications need to refer to domain specific concepts, which are typically formalised in an ontology; therefore, by expressing both policies and concepts in OWL, policy specifications are able to refer to concepts easily. Second, reasoning over policy specifications can decide if a set of policies subsumes another, or if a set of policies results in conflicts; thus, we naturally benefit from OWL's underlying reasoning mechanism by expressing polices in OWL. Third, the policy language is not constructed in an ad-hoc manner for specific application domains, but it draws its vocabulary from a well-understood, clearly specified set of OWL terms.

This paper introduces a new kind of use of the KAoS language since it applies KAoS to the behavioural specification of Grid registries. Miles *et al.* [16] is a first attempt of using policies for specifying the behaviour of Grid registries. Their approach differs from ours in two different ways. First, it adopts an ad-hoc policy language to describe the configuration of a system; such a language does not offer the systematism of OWL-based KAoS, nor its reasoning capababilities; on the other hand, since it was purposely designed for the task, it is more concise. Second, in order to enforce management policy in a registry, Miles *et al.* 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) [9]. As far as enforcement (i.e., execution) is concerned, both approaches have strong theoretical underpinning, respectively OWL-based reasoning and belief-desire-intention model. The KAoS approach offers the additional advantage that reasoning can be applied statically over policies, to detect conflicts or to perform subsumption reasoning, a facility that is not available in Miles' approach. From a performance viewpoint, reasoning is also used to decide if a policy applies to an incoming message; in the simplest case, pattern matching suffices to match an incoming message against applicable policies; in the most complex case, ontological reasoning is required which trigger is semmantically satisfied; our focus is on optimising such a process, by maximising the amount of offline reasoning, so as to reduce the amount of reasoning requied at run time.

8 Conclusion

By studying the configuration of distributed grid services using the policy language KAoS, we have elicited requirements that are typical of complex systems. We have introduced extensions to KAoS that address these requirements. The core extension is the notion of role-value maps that allows us to express constraints over OWL property values. This feature is extensively used in our other extensions for defining policies whose control refers to trigger value, for structuring policies into policy sets that are parameterised by run-time values, and for referring to past execution history. These extensions make KAoS a very powerful policy-based configuration language, while still providing benefits such as reasoning for detecting policy conflicts and enforcing polices.

9 Acknowledgements

This research is funded in part by the Overseas Travel Scheme of the Royal Academy of Engineering and EPSRC project (reference GR/S75758/01). The case study presented here is the myGrid registry (EPSRC Grant GR/R67743/01) which is being refactored as part of the OMII Grimoires project (EPSRC Grant GR/S90843/01).

References

[1] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL Web Ontology Language Reference. http://www.w3.org/TR/2004/REC-owl-ref-20040210/, Feb. 2004.

[2] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientifi c American*, 284(5):34–43, 2001.

[3] D. Box, F. Curbera, M. Hondo, C. Kaler, D. Langworthy, A. Nadalin, N. Nagaratnam, M. Nottingham, C. von Riegen, and J. Shewchuk. Web services policy framework (ws-policy). May 2003.

[4] J. M. Bradshaw, P. Beautement, M. Breedy, L. Bunch, S. V. Drakunov, P. J. Feltovich, R. R. Hoffman, R. Jeffers, M. Johnson, S. Kulkarni, J. Lott, A. Raj, N. Suri, and A. Uszok. Making agents acceptable to people. In N. Zhong and J. Liu, editors, *Intelligent Technologies for Information Analysis: Advances in Agents, Data Mining, and Statistical Learning*, pages 361–400. Springer, 2004.

[5] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specifi cation language. In *Workshop on Policies for Distributed Systems and Networks (POLICY 2001)*, volume 1995 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[6] N. Damianou, N. Dulay, E. C. Lupu, and M. Sloman. Ponder: a language for specifying security and management policies for distributed systems. *Imperial College Research Report DoC 2000/1*, 2000.

[7] DAML-S Coalition:, A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, D. Martin, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, and K. Sycara. DAML-S: Web Service Description for the Semantic Web. In *First International Semantic Web Conference (ISWC) Proceedings*, pages 348–363, 2002.

[8] G. Della-Libera, P. Hallam-Baker, M. Hondo, T. Janczuk, C. Kaler, H. Maruyama, N. Nagaratnam, A. Nash, R. Philpott, H. Prafullchandra, J. Shewchuk, E. Waingold, and R. Zolfonoon. Web services security policy (ws-securitypolicy). Dec. 2002.

[9] M. dInverno, M. Luck, M. Georgeff, D. Kinny, , and M. Wooldridge. The dmars architecture: A specifi cation of the distributed multi-agent reasoning system. *Autonomous Agents and Multi-Agent Systems*, 2004.

[10] I. Foster. What is the grid? a three point checklist. `http://www-fp.mcs.anl.gov/~foster/`, July 2002.

[11] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. The Physiology of the Grid — An Open Grid Services Architecture for Distributed Systems Integration. Technical report, Argonne National Laboratory, 2002.

[12] J. Hughes and E. Maler. Security assertion markup language (saml) 2.0, technical overview, working draft 03. Feb. 2005.

[13] M. Johnson, P. Chang, R. Jeffers, J. Bradshaw, M. Breedy, L. Bunch, S. Kulkarni, J. Lott, N. Suri, A. Uszok, and V.-W. Soo. Kaos semantic policy and domain services: An application of daml to web services-based grid architectures. In *AAMAS Workshop on Web-Services and Agent-based Engineering*, Merlbourne, Australia, 2003.

[14] 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, Oct. 2003.

[15] S. Miles, J. Papay, V. Dialani, M. Luck, K. Decker, T. Payne, and L. Moreau. Personalised grid service discovery. *IEE Proceedings Software: Special Issue on Performance Engineering*, 150(4):252–256, Aug. 2003.

[16] S. Miles, J. Papay, M. Luck, and L. Moreau. Implementing policy management through bdi. In *The Twenty-third SGAI International Conference on Innovative Techniques and Applications of Artifi cial Intelligence*, Dec. 2004.

[17] S. Miles, J. Papay, T. Payne, K. Decker, and L. Moreau. Towards a protocol for the attachment of semantic descriptions to grid services. In *The Second European across Grids Conference*, volume 3165 of *Lecture Notes in Computer Science*, pages 230–239, Nicosia, Cyprus, Jan. 2004.

[18] J. Papay, S. Miles, M. Luck, L. Moreau, and T. Payne. Principles of personalisation of service discovery. In *Proceedings of the UK OST e-Science second All Hands Meeting 2004 (AHM'04)*, Nottingham, UK, Sept. 2004.

[19] B. Parsia, V. Kolovski, and J. Hendler. Expressing ws-policies in owl. In *Submitted to Policy Management for the Web Workshop, 14th International World Wide Web Conference*, Chiba, Japan, May 2005. http://www.mindswap.org/papers/2005/WSPolicyInOWL.pdf.

[20] M. Schmidt-Schauss. Subsumption in kl-one is undecidable. In R. J. Brachman, H. J. Levesque, and R. Reiter, editors, *Proc. of the 1st Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR'89)*, pages 421–431, Los Altos, 1989. Morgan Kaufmann.

[21] Universal Description, Discovery and Integration of Business of the Web. `www.uddi.org`, 2001.

[22] A. Uszok, J. M. Bradshaw, and R. Jeffers. Kaos: A policy and domain services framework for grid computing and semantic web services. In C. Jensen, S. Poslad, and T. Dimitrakos, editors, *Trust Management: Second International Conference (iTrust 2004) Proceedings*, volume 2995 of *Lecture Notes in Computer Science*, pages 16–26, Oxford, UK, Mar. 2004. Springer.

[23] A. Uszok, J. M. Bradshaw, M. Johnson, R. Jeffers, A. Tate, J. Dalton, S. Aitken. KAoS policy management for semantic web services. IEEE Intelligent Systems, July/August, 19(4), pages 32-41, 2004.

[24] C. Wroe, R. Stevens, C. Goble, A. Roberts, and M. Greenwood. A suite of daml+oil ontologies to describe bioinformatics web services and data. *International Journal of Cooperative Information Systems*, 2003.