Extending OWL with Role-Value Maps to support the KAoS Policy Language for Policy-Based Component Specification of Grid Services

Luc Moreau\(^1\)  Jeff Bradshaw\(^2\)  Maggie Breedy\(^2\)  
Lary Bunch\(^2\)  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)

October 12, 2004

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 a Grid Registry capable of supporting annotations (semantic or not) of service descriptions by third-party users. Instead, services have to be conceived so that they can be configured at deployment and run times. We argue that policy languages — and KAoS in particular — are powerful and flexible languages 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. We then propose several extensions to the KAoS language that address these requirements. Namely, we introduce role-value maps to express constraints between property values; we introduce a notion of PolicySet with associated parameters that support parametric 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 our experience with introducing them in the KAoS policy language.
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 a Grid registry [18] capable of hosting service descriptions and third-party annotations such as semantic descriptions of services, trust or accuracy information about them, their usage policies or simply general statistics about their usage. 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.

Such a plethora of configurations implies that service designers cannot anticipate all possible configurations of services. It is therefore a requirement that such services be provided with the means to specify their configurations, hence giving them the flexibility required at deployment or run times. We argue that policy languages, such as Ponder [5], KAoS [25], or Rei [14] can be utilised for configuring complex grid services. In particular, the KAoS language, which has been used in multiple contexts such as grid security [13] or mobile agents, is an interesting candidate language 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.

We therefore undertook to use the policy language KAoS to specify configurations of a Grid registry [18]. The registry had been designed to be modularly organised around a set components [20]. Three different usages of KAoS policies have been adopted in our endeavour. First, KAoS is used to express components behavioural specification; second, components instantiation and composition are also defined in KAoS; finally, overarching constraints over registries are also characterised in KAoS.

Doing so, we have encountered an interesting set of new requirements, which we addressed by providing extensions to KAoS. These extensions constitute the key contributions of this paper: (i) We made extensive use of role-value maps [23] to add constraints to property values in concept definitions, which would otherwise be not expressible in owl. (ii) We support parametric constraints, i.e. constraints that can be parameterised by values that are only known at deployment or run times. (iii) PolicySets, defined as first-class owl concepts, can be extended with properties and can be referenced by policies belonging these sets. (iv) 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. The KAoS
extensions that we designed are in no way restricted to the current case study, but are applicable to policy specifications for a wide range of domains. They make KAoS a powerful and expressive language, while still preserving its capability of reasoning. Indeed, we have implemented some reasoning components capable of dealing with role-value maps, and the concepts of PolicySet and Context, and we have integrated them into the Java Theorem Prover JTP.

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 the KAoS policy language and the solutions we conceived (Section 5). 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, 26].

As part of the myGrid project (www.mygrid.org.uk), we have designed a service directory capable of containing semantic descriptions of services, including their functionality and their semantic inputs and outputs [15, 16, 19]. Semantic descriptions are currently expressed in an OWL ontology [26]. 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 [16]; such an approach supports a collaborative view of e-Science, since scientists can utilise
services because other found them useful or efficient to perform specific tasks.

We have identified different ways of deploying a Grid registry; the following examples illustrate configurations that are deemed desirable in different circumstances. (i) First, a Grid registry can be deployed as a standalone service, available to any client according to its security policy, and presenting the set of interfaces required by its deployer. (ii) Second, users could deploy it as a “proxy” to a publicly available service directory for which they do not have write access [16]; the proxy would typically tunnel queries to an existing service directory, but would hold any metadata information about registered services, and would therefore act as a personalised service directory. (iii) Alternatively, the service could also federate entries from multiple service directories. 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 runtime according to the users’ needs. Therefore, the architecture of such a registry must be designed to allow easy configuration at deployment time, and possibly re-configuration at runtime.

Hence, in the registry implementation, we have adopted a message-passing metaphor, in which messages are handled by components we refer to as handlers [20]. 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. 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 that 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 configurations. Hence, we believe that a mechanism to specify configurations of such services would be very valuable. To this end, policy languages can be used to specify the expected behaviour and configuration of services, both at deployment time and run time. 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, 25, 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. Obligation policies use it to specify the action being obliged or waived. Another property value is the trigger which identifies 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 \( X \) to perform a CommunicationAction, which is a predefined concept in the KAoS ontology. The communication action is to send a message Bar to a recipient \( Y \); such a communicatin 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 a communication action with recipient
X (the incumbent of the obligation) and with message Foo. Informally, the policy requires X to send a message Bar to Y, whenever it receives a message Foo. For instance, such an obligation policy can be used to record incoming messages into a logging service Y.

**Policy p:**

\[ X \text{ is obligated to perform} \]

\[ \text{CommunicationAction with properties} \]

\[ \text{hasDestination is subset of } Y \]

\[ \text{carriesMessage is subset of Bar} \]

\[ \text{when } X \text{ performs} \]

\[ \text{CommunicationAction with properties} \]

\[ \text{hasDestination is subset of } X \]

\[ \text{carriesMessage is subset of Foo} \]

Figure 2: Illustration of an Obligation Policy in KAoS

Concepts Foo and Bar must themselves be defined in an ontology; in our case study, they will be messages used internally by our registry implementation.

## 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 illustrates some policy uses in this context, which we discuss below.

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 will be 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 registry.

**Inside component**  Upon receiving a MetadataChanged notification message, the EventHandler 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, then the service detail (and all its metadata) should be retrieved from the expert registry and saved in 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 [24]; (iv) obtain service metadata from expert registry; (v) save service metadata in novice registry using the metadata handler interface [18]. In the context of this registry, a component is characterised by an interface composed of the messages that it can process, a set of external parameters (such as a trust threshold), a behavioural specification derived from a policy set, and a set of other components (in fact, their interfaces) it may interact with (such as the remote registry, the uddi and metadata publish handlers).

**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 New Requirements on Policy Language

This specific application puts a series of new requirements on the KaOS policy language, which we detail below. We shall also explain how we extend the language in order to address these requirements.

5.1 Action Must Refer to Trigger Properties

Figure 4 presents a concrete obligation policy that we would like to express in our registry application. Here, we have extended the KaOS notation with underlined terms, which we use to illustrate the kind of constraint we are attempting to express. The obligation specifies that an EventHandler is obligated to perform a CommunicationAction that carries a message of type GetBusinessServiceMetadata with a given MetadataKey, where the value is specified by the trigger of the obligation policy, i.e., that the EventHandler is the destination of a CommunicationAction, which carries a message MetadataChanged, with a MetadataKey. In other words, given a message that contains a key, the event handler must issue another message with an identical key. Constraints of this type are very common in practical policy definitions, as illustrate 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 all these cases, the trigger of an obligation policy identifies a value, which we have been referring to by names \( \kappa, v, X \), and the action of this policy is to perform an action that is parametrised by such a value.

Such constraints could naturally be expressed using a notion of variable: the trigger would declare a variable, which would computationally be bound to a value when the trigger of the policy is verified to be satisfied; the value the variable refers to would itself be used when the variable is referenced in the action. This is the solution adopted by rule-based programming languages, e.g., Prolog or Ponder [5]. KaOS policy language is based on owl which does not offer variables directly. Luckily, there exist some owl extensions that may help...
Policy $p_0$:
EventHandler is obligated to perform
CommunicationAction with properties
  hasDestination is subset of Registry
  carriesMessage is subset of GetBusinessServiceMetadata
  hasMetadataKey equals $\kappa$
when EventHandler performs
CommunicationAction with properties
  hasDestination is subset of EventHandler
  carriesMessage is subset of MetadataChanged
  hasMetadataKey equals $\kappa$

Figure 4: An Obligation Policy with Constraints

EventHandler is obligated to perform
CommunicationAction with properties
  hasDestination is subset of Registry
  carriesMessage is subset of GetBusinessServiceMetadata
  hasMetadataKey equals
    Trigger $\rightarrow$ carriesMessage $\rightarrow$ hasMetadataKey
when EventHandler performs
CommunicationAction with properties
  hasDestination is subset of EventHandler
  carriesMessage is subset of MetadataChanged
  hasMetadataKey equals
    Control $\rightarrow$ carriesMessage $\rightarrow$ hasMetadataKey

Figure 5: An Obligation Policy with Role Value Maps

us to meet our requirements: the Semantic Web Rule Language SWRL [12] adds Horn clauses to OWL, whereas role value maps [23] allow us to specify constraints between property values in OWL terms. For a number of reasons to be discussed below, we adopted the latter approach. Figure 5 illustrates how role value maps have been put into practice in our example obligation policy. The value of the hasMetadataKey property of the GetBusinessServiceMetadata message in the policy control is defined to be equal to:

$$\text{Trigger} \rightarrow \text{carriesMessage} \rightarrow \text{hasMetadataKey},$$

meaning to refer to the value of the property hasMetadataKey of the message that is value of the property carriesMessage, for the obligation trigger. Likewise, a symmetric constraint is defined for the value of the property of hasMetadataKey, set to be equal to:

$$\text{Control} \rightarrow \text{carriesMessage} \rightarrow \text{hasMetadataKey}.$$
Therefore, within a policy specification, the reserved words Control and Trigger refer to the terms respectively denoting the control and trigger of the current policy being defined.

In general, role value maps take the following form, summarised by the BNF grammar below. A role value map is a triple — property, operator, value — meant to denote that a given property has a value that is expected to satisfy the constrain set by the operator and value. The operator encountered so far is the equality “equals”, whereas reserved keywords that we introduced are Trigger and Control. A value is identified by a keyword followed by a sequence of properties. In the rest of the paper, we will introduce further operators and keywords.

\[
\text{(map)} ::= \text{(property)} \text{(operator)} \text{(value)} \\
\text{(property)} ::= \text{an owl property} \\
\text{(operator)} ::= \text{equals} | \text{greaterThan} | \ldots \\
\text{(value)} ::= \text{(keyword) \{\text{(property)}\}^+} \\
\text{(keyword)} ::= \text{Trigger} | \text{Control} | \text{PolicySet} | \text{Context}
\]

Andrzej, you might like to strengthen this paragraph. We opted for the role value map solution for a number of reasons:

1. it allows us to naturally express the constraints that are required by our application;

2. reasoning for such extensions can be implemented through a separate reasoner, which can easily be integrated in the JTP reasoner.

5.2 Parametric Constraints

Our use case introduces some constraints that are parameterised by external values. For instance, an obligation we encountered is the following: if \( x \) receives some trust metadata with value \( v \) greater than threshold \( t \), then \( x \) must perform a given action. In this example, threshold \( t \) is a value that would be defined at deployment in a configuration file loaded by the service. Therefore, we have to be able to deal with “parametric contraints”, since we want to refer to all values greater than a threshold \( t \).

Such a kind of parametric constraint presents some challenges to encode in owl. Indeed, owl relies on XML Schema Datatypes (XSD) to express numeric constraints. For instance, the set of integers less than 10 would be defined as a subclass of Integer, with maximum value 10. However, since this technique for defining constraints requires a maximum value to be a constant, it does not support parameters, such as the threshold \( t \) in our use case.

As a result, we had to define ourselves a set of numerical relationships, and we expect our reasoner to handle them. For instance, we defined the property
greaterThan with domain and range being floating point values. Such a property is meant to denote the numeric inequality “>”, and is an accepted constraint in the role value map definition introduced in the previous section.

5.3 Policy Set Context

Our use case again introduces a new requirement. Recall that behavioural specification of a component is defined by a policy set. Within this set, a given policy may be parameterised by some parameters whose values are shared by the other policies. Given 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 will refer to the parameter values that were specified for this set.

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 subclass); 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 KAoS policy language does not support such a notion of “block” that would give a scope to variables to be shared by a set of policies. However, the ontology language owl and the role-value maps introduced in Section 5.1 can be used to express such a notion. Figure 6 illustrates our solution to this problem.

Using role value maps and a newly introduced keyword PolicySet, we are able to express contraints between concept properties and PolicySet properties. In Figure 6, we set a constraint to the concept MetadataChanged by requiring its hasTrust property value to be greater than the hasThreshold property value of the PolicySet the policy belongs to.

\[ \text{hasTrust} \text{ greaterThan } \text{PolicySet} \rightarrow \text{hasThreshold} \]

In Figure 6, 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** $p_1$

**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`

---

$\text{PolicySet} \ ComponentBehaviour$

- `hasPolicy` [$p_1, p_2, ...$]
- `hasThreshold` [Integer]
- `hasOtherParameter` [Class]

---

**Instance** $c_1$ of **ComponentBehaviour**

- `hasThreshold` 0.5

**Instance** $c_2$ of **ComponentBehaviour**

- `hasThreshold` 0.8

---

**Figure 6**: PolicySet Entity and Associate Properties

---

### 5.4 Execution Context

Our use case presents us again with an interesting challenge. 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 $p_1$.

This requirement leads us to introduce a notion of *execution context*, through which we could refer to properties that held during the enactment of a previous policy. Given that parallel execution may take place, we should allow for multiple execution contexts, so that each “thread of execution” is entitled to its own context.

So, like we introduced reserved words to refer to the trigger and the control of a policy, we introduce the reserved word *Context* to refer to the execution
Policy \( p_1 \):

\textbf{EventHandler} is obligated to perform

\textbf{CommunicationAction} with properties

hasDestination is subset of Registry

\textbf{hasMetadataKey equals}

\textbf{carriesMessage is subset of GetBusinessServiceMetadata}

\textbf{hasMetadataKey}

\textbf{when} \textbf{EventHandler} performs \textbf{CommunicationAction} with properties

hasDestination is subset of \textbf{EventHandler}

\textbf{hasMetadataKey}

\textbf{hasTrust greaterThan}

\textbf{hasThreshold}

\textbf{Policy \( p_2 \)}:

\textbf{EventHandler} is obligated to perform

\textbf{SaveAction} with properties

hasValue Pair

\textbf{hasLeft equals} \textbf{Trigger} \rightarrow \textbf{carriesMessage} \rightarrow \textbf{returns,}

\textbf{hasRight equals} \textbf{Context} \rightarrow \textbf{hasTrust}

\textbf{when} \textbf{EventHandler} performs \textbf{CommunicationAction} with properties

\textbf{hasDestination} is subset of \textbf{EventHandler}

\textbf{hasMetadataKey}

\textbf{hasTrust}

\textbf{returns}

\textbf{when} \textbf{EventHandler} performs \textbf{CommunicationAction} with properties

\textbf{hasMetadataKey}

\textbf{hasValue}

\textbf{hasLeft}

Figure 7: Execution Level Context
context of a policy. Its use is illustrated by policy $p_2$ in Figure 7 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 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.

6 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 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-SecurityPolicy defines extensions for security properties for Web Services [8]; SAML [21], the Security Assertion Markup Language, is an XML 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

---

$^1$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 [22].
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 policies 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; we see this as a benefit when policy languages need to be “compiled” into other representations, e.g. to build enforcers.

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. [17] 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 capabilities; 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 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.

Andrzej, it would be nice if you could say something about how reasoning is implemented.

As far as enforcement is concerned, the KAoS machinery can still operate with the new extensions. Given an event, reasoning is used to determine which policies should be fired. We are currently working to bind the KAoS enforcer with the registry components, so that the actions that must be executed are performed in the registry; precisely, given that we have adopted a message passing metaphor in the registry, most of the actions that we have to perform consist of sending messages to components. Hence, from an implementation viewpoint, the carriesMessage property used in the paper has a value that is a message, which in our implementation has a well-defined Java structure that is homomorphic to the correspond OWL-concept. Role-value maps naturally translate to some constraint verification over the fields of such data structure. The challenge in this application is to be able to convert arbitrary OWL concepts into executable Java code; for the time being, we are adopting a pragmatic approach capable, aiming at dealing with the specific policies designed for our use case.
7 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 policies.

8 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 designed in part by Simon Miles and Juri Papay.

References


