Semantic Web Languages for Policy Representation and Reasoning: A Comparison of KAoS, Rei, and Ponder

Gianluca Tonti[1,2], Jeffrey M. Bradshaw[1], Renia Jeffers[1], Rebecca Montanari[2], Niranjan Suri[1], and Andrzej Uszok[1]

[1] Institute for Human and Machine Cognition (IHMC)
University of West Florida
40 S. Alcaniz Street, Pensacola, FL 32501 – USA
{jbradshaw,rjeffers,nsuri,gtonti,auszok}@ihmc.us
[2] Dipartimento di Elettronica, Informatica e Sistemistica
University of Bologna
Viale Risorgimento, 2 - 40136 Bologna - ITALY
{rmontanari,gtonti}@deis.unibo.it

Abstract. Policies are being increasingly used for automated system management and controlling the behavior of complex systems. The use of policies allows administrators to modify system behavior without changing source code or requiring the consent or cooperation of the components being governed. Early approaches to policy representation have been restrictive in many ways. However semantically-rich policy representations can reduce human error, simplify policy analysis, reduce policy conflicts, and facilitate interoperability. In this paper, we compare three approaches to policy representation, reasoning, and enforcement. We highlight similarities and differences between Ponder, KAoS, and Rei, and sketch out some general criteria and properties for more adequate approaches to policy semantics in the future.

1 Introduction

Until recently, the use of Semantic Web languages has been limited primarily to representing Web content and services. However the capabilities of these languages, coupled with the increasing availability of mature environments and tools to facilitate their use, make them well suited for many other kinds of applications. In this paper, we evaluate ongoing efforts to use Semantic Web languages to represent and reason about policies for multi-agent and distributed systems. In particular, we intend to show the strengths and limitations of such languages for policy representation and reasoning by comparing and contrasting three approaches: KAoS, Rei and Ponder.

Policies are a means to dynamically regulate the behavior of system components without changing code and without requiring the consent or cooperation of the components being governed [1, 2]. By changing policies, a system can be continuously adjusted to accommodate variations in externally imposed constraints and environmental conditions. The adoption of a policy based-approach for controlling a system requires an appropriate policy representation and the design and

development of a policy management framework. Policies will become increasingly more important to the real world implementation of Semantic Web Services [3, 4].

KAoS uses DAML[1] (and soon will use OWL[2]) as the basis for representing and reasoning about policies within Web Services, Grid Computing, and multi-agent system platforms [2, 3, 4, 5]. KAoS also exploits ontologies for representing and reasoning about *domains* describing organizations of human, agent, and other computational actors. Rei is a new deontic concept-based policy language that is currently implemented in Prolog with a semantic representation of policies in RDF-S. In the future, it is expected that Rei will support OWL and Notation3[3] ontologies [6, 7]. Ponder is an object-oriented policy language for the management of distributed systems and networks [1, 8]. The developers of Ponder pioneered many of the policy management concepts used in KAoS and Rei, though its implementation differs in important ways [9].

The paper is organized as follows. Section 2 describes the motivation and background of policy-based approaches for the management of software systems, beginning with the extensive research over the last decade about the management of network and quality of service and continuing to the present day in new application areas such as the control of multi-agent systems. Section 3 briefly presents the three different approaches for policy representation, reasoning, and enforcement. These are compared in Section 4 through the implementation of a common example of a communication policy. This is followed by a discussion of the approaches and of some of the open issues that must be resolved as a prerequisite to widespread adoption. Finally, in Section 5, we present some conclusions.

2   Motivation and Background for Policy-Based Management

Policies, which constrain the behavior of system components, are becoming an increasingly popular approach to dynamic adjustability of applications in academia and industry. Elsewhere [2], we have pointed out the many benefits of policy-based approaches, including reusability, efficiency, extensibility, context-sensitivity, verifiability, support for both simple and sophisticated components, protection from poorly-designed, buggy, or malicious components, and reasoning about component behavior. Policies have important analogues in animal societies and human cultures [10].

Policy-based network management has been the subject of extensive research over the last decade [11]. Policies are often applied to automate network administration tasks, such as configuration, security, recovery, or quality of service (QoS). In the network management field, policies are expressed as sets of rules governing choices in the behavior of the network. Multiple approaches for policy specification have been proposed that range from formal policy languages that can be processed and interpreted easily and directly by a computer, to rule-based policy notation using an if-then-else format, to the representation of policies as entries in a table consisting of multiple attributes [12, 13, 14]. There are also ongoing standardization efforts toward

---

[1] DARPA Agent Markup Language (http://www.daml.org)

[2] Web Ontology Language (http://www.w3.org/2001/sw/WebOnt/)

[3] http://www.w3.org/DesignIssues/Notation3.html

common policy information models and frameworks. The Internet Engineering Task Force, for instance, has been investigating policies as a means for managing IP-multiservice networks by focusing on the specification of protocols and object-oriented models for representing policies [15].

The scope of policy management is increasingly going beyond these traditional applications in significant ways. New challenges for policy management include:

- sources and methods protection, digital rights management, information filtering and transformation, capability-based access;
- active networks, agile computing, pervasive and mobile systems;
- organizational modeling, coalition formation, formalizing cross-organizational agreements;
- trust models, trust management, information pedigrees;
- effective human-machine interaction: interruption and notification management, presence management, adjustable autonomy, teamwork facilitation, safety; and
- intelligent retrieval of all policies relevant to some situation.

The management of multi-agent systems represents one of the most promising fields for the exploitation of policy-based approaches [2, 6, 16]. By their ability to operate independently without constant human supervision, agents can perform tasks that would be impractical or impossible using traditional software applications. On the other hand, this additional autonomy, if unchecked, also has the potential of effecting severe damage if agents are poorly designed, buggy, or malicious.

Controlling agent behavior is a complex task because agent behavior cannot be a-priori programmed to face any operative run-time situation, but requires dynamic and continuous adjustments to allow agents to act in any execution context in the most suitable way [2]. From a technical perspective, we want to be able to help ensure the protection of agent state, the viability of agent communities, and the reliability of the resources on which they depend. To accomplish this, we must guarantee, insofar as is possible, that the autonomy of agents can always be bounded by explicit enforceable policy that can be continually adjusted to maximize the agents' effectiveness and safety in both human and computational environments. From a social perspective, we want agents to be designed to fit well with how people actually work together. Explicit policies governing human-agent interaction, based on careful observation of work practice and an understanding of current social science research, can help assure that effective and natural coordination, appropriate levels and modalities of feedback, and adequate predictability and responsiveness to human control are maintained [17]. In short, interaction among humans and agents must be graceful and should enhance rather than hinder human work. All these factors are key to providing the reassurance and trust that are the prerequisites to the widespread acceptance of agent technology for non-trivial applications.

3 Policy Languages and Frameworks

Policies can be specified in many different ways and multiple approaches have been proposed in different application domains [11]. There are, however, some general requirements that any policy representation should satisfy regardless of its field of applicability: *expressiveness* to handle the wide range of policy requirements arising

in the system being managed, *simplicity* to ease the policy definition tasks for administrators with different degrees of expertise, *enforceability* to ensure a mapping of policy specifications into implementable policies for various platforms, *scalabilty* to ensure adequate performance, and *analyzability* to allow reasoning about policies. The challenge is to achieve a suitable balance among the objectives of expressiveness, computational tractability, and ease of use.

The aim of this section is not to provide a general survey of the state-of-the-art in policy representation, but to describe selected technical aspects of a few policy approaches that have been specifically designed and extensively tested for management of multi-agent and distributed systems. We first present KAoS followed by Rei, both of which were originally designed for governing agent behavior, and finally Ponder which is one of the most well-known policy-based systems for network management and is being evaluated in several universities and industrial organizations.

3.1 KAoS

KAoS a collection of componentized policy and domain management services compatible with several popular agent frameworks, including Nomads [18], the DARPA CoABS Grid [19], the DARPA ALP/Ultra*Log Cougaar framework (http://www.cougaar.net),   CORBA   (http://www.omg.org),   Voyager (http://www.recursionsw.com/osi.asp), and Brahms (www.agentisolutions.com). While initially oriented to the dynamic and complex requirements of software agent applications, KAoS services are also being adapted to general-purpose grid computing (http://www.gridforum.org) and Web Services (http://www.w3.org/2002/ws/) environments as well [3, 4]. KAoS has been deployed in a wide variety of applications, from coalition warfare [5, 16], to robustness and survivability for distributed systems [http://www.ultralog.net], to human-agent teamwork in space applications [17], to cognitive prostheses [2]. See [2] for a summary of these and other applications.

KAoS domain services provide the capability for groups of software components, people, resources, and other entities to be organized into domains and subdomains to facilitate agent-agent collaboration and external policy administration.

KAoS policy services allow for the specification, management, conflict resolution, and enforcement of policies within domains. Policies are currently represented in DAML+OIL as ontologies. The KAoS Policy Ontologies (KPO) distinguish between *authorizations* (i.e., constraints that permit or forbid some action) and *obligations* (i.e., constraints that require some action to be performed, or else waive such a requirement).

The following important features of KAoS are worth noting. First, the approach does not assume that the policy-governed system is comprised of a homogeneous set of components that have been designed in advance to work with KAoS services. Rather the goal is to be able to have KAoS services work with arbitrarily written components after the fact through support being added transparently at the platform level. Second, insofar as possible, the KAoS framework supports dynamic runtime policy changes and not merely static configurations determined in advance. Third, the framework is extensible to a variety of execution platforms that might be simultaneously running with different enforcement mechanisms—in principle any

platform for which policy enforcement mechanisms may be written. Fourth, the KAoS framework is intended to be robust and adaptable in continuing to manage and enforce policy in the face of attack or failure of any combination of components.

Policy representation. Within the KAoS Policy Ontologies (KPO), a policy is an instance of the appropriate policy type (i.e., positive or negative authorization; positive or negative obligation) that defines the associated values for its properties, such as site of enforcement, priority, and update time stamp. Complex policy constructs such as delegation are built directly out of the four basic policy types rather than through special-purpose mechanisms (e.g., metapolicies). Through various property restrictions in the action type, the specific context for the action is defined (e.g., relations to target and other situation entities). A given policy can also be variously scoped, for example, either to individual components, to components of a given class, to components belonging to a particular group, or to components running in a given physical place or computational environment (e.g., host, VM). The current version of the core ontologies in KPO consists of more than 100 classes and related properties that together define basic ontologies for actions, actors, groups, places, various entities related to actions (e.g., computing resources), and policies. Application developers typically extend KPO with additional domain specific ontological elements. As the application runs, classes and individuals corresponding to new policies and application entities are also transparently added and deleted as needed. Figure 1 shows a fragment of a communication policy example stating that the members of a domain called A are permitted to communicate to members outside the domain using encrypted communication:

```
<daml:Class rdf:ID="ExampleAction">
   <rdfs:subClassOf rdf:resource="#EncryptedCommunicationAction" />
   <rdfs:subClassOf>
        <daml:Restriction>
               <daml:onProperty rdf:resource="#performedBy" />
               <daml:toClass rdf:resource="#MembersOfDomainA" />
           </daml:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <daml:Restriction>
               <daml:onProperty rdf:resource="#hasDestination" />
               <daml:toClass rdf:resource="#notMembersOfDomainA " />
           </daml:Restriction>
       </rdfs:subClassOf>
</daml:Class>
<policy:PosAuthorizationPolicy rdf:ID="Example">
  <policy:controls rdf:resource="#ExampleAction" />
  <policy:hasSiteOfEnforcement rdf:resource="#ActorSite" />
  <policy:hasPriority>10</policy:hasPriority>

<policy:hasUpdateTimeStamp>4237445645589</policy:hasUpdateTimeStamp>
```

Fig. 1. Example of DAML policy representation in KAoS. Note that the use of the KPAT GUI shields users from having to know and use DAML notations (see Figure 2).

Policy management features. KAoS provides the KAoS Policy Administration Tool (KPAT)[4] as a graphical interface (Figure 2). KPAT assists users in policy specification, revision, and application. In addition, KPAT can be used to browse and load ontologies and to analyze and deconflict newly defined policies. As policies, domains, and application entities are defined using the KPAT GUI, the appropriate DAML representations are generated automatically in the background and asserted into or retracted from the system, insulating the user from having to know DAML or from coding directly in a policy language as in Rei and Ponder. Policy templates allow various classes of policy definitions to be defined as high-level domain-specific abstractions. A rich set of queries is also available through KPAT or through programmatic interfaces.

KAoS detects potential conflicts between policies at specification time, whenever a user tries to add a new policy to the Directory Service. The KAoS algorithm for policy conflict detection is able to detect conflicts between policies by relying on algorithms implemented as extensions to Stanford's Java Theorem Prover (JTP) (http://www.ksl.stanford.edu/software/JTP/), which is integrated into KAoS. The engine identifies policy conflicts by using the subsumption mechanisms among relevant classes. KAoS tries to resolve conflicts by ordering policies according to their precedence and, where necessary and desired, generating new harmonized policies [4, 5].

In addition, KAoS provides a complete deployment model that consists of some basic services and components such as the Domain Manager, the Directory Service, and Guards. The KAoS framework is responsible for the management of domains of agents and assures policy consistency at all the levels of the domain hierarchy. The Directory Service is responsible for overall policy management, while Guards interpret policies and pass them on to Enforcers, which are platform-specific components that handle the actual policy enforcement. Before performing its function to allow or forbid a given action, Enforcers for authorization policies must obtain an answer to the question, "is a given action permitted or not?" or reasons of performance and robustness. KAoS typically provides the answer locally through intelligent lookup rather than inference. An action description is created by the Enforcer and passed to the Guard, which traverses policy storage and checks to see if the given action instance is in the range of actions controlled by any of its policies. If the authorization mechanism does not find any policy applicable to the action description passed to it, it answers the question consistent with whatever default authorization modality it has been given. Domain defaults either correspond to a *democracy,* where everything is permitted that is not explicitly forbidden, or a *tyranny,* where everything is forbidden that is not explicitly permitted. Enforcement mechanisms for obligation policies work in a similar fashion, however rather than preemptively prohibiting actions, either they monitor and respond as necessary to the performance or non-performance of obligation actions, or else they actively facilitate performance through types of enforcers called enablers.
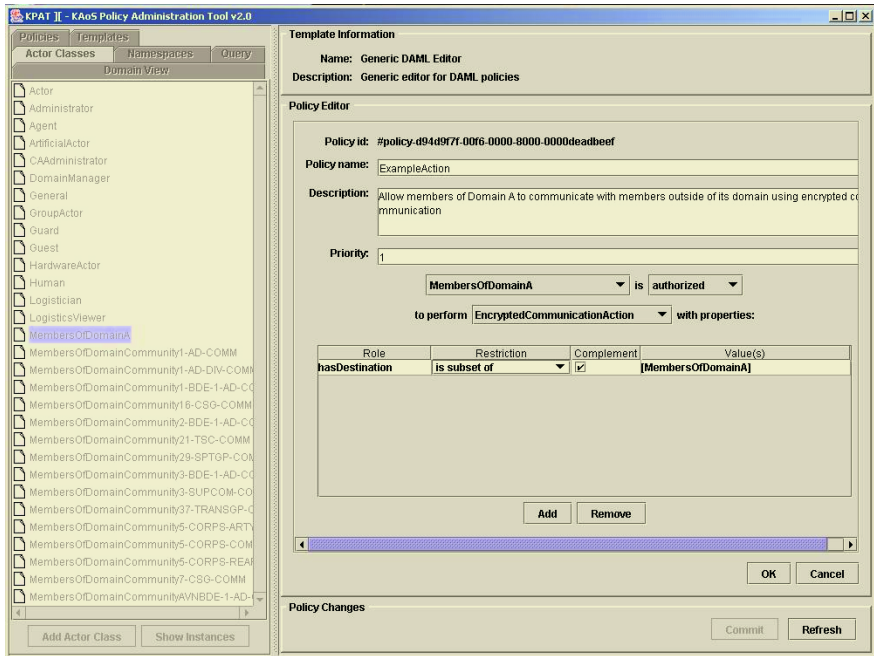
---

[4] pronounced "KAY-pat"

Fig. 2. KPAT hides the complexity of DAML through the use of a graphical user interface and a generic DAML template. Additional language-specific templates or domain-specific templates for common classes of policies can be easily defined.

3.2 Rei

Rei is a policy framework that integrates support for policy specification, analysis and reasoning in pervasive computing applications [6, 7]. Rei has been used in conjunction with the Vigil security framework [6], Fujitsu's Task Computing project, and with the Groove workspace [http://www.groove.net] within the DARPA Genoa II program [http://www.darpa.mil/iao/GenoaII.htm].

The Rei deontic concept-based policy language allows users to express and represent the concepts of *rights*, *prohibitions*, *obligations,* and *dispensations*. These concepts correspond, respectively, to the conditions of positive and negative authorization, and positive and negative obligation in KAoS and Ponder. In addition, Rei permits users to specify policies that are defined as rules associating an entity of a managed domain with its set of rights, prohibitions, obligations, and dispensations.

Rei relies on an application-independent ontology to represent the concepts of rights, prohibitions, obligations, dispensations, and policy rules. This allows different elements of a pervasive environment to understand and interpret Rei policies in the correct way. The basic ontology also includes the description of actions. A general action is described by its unique action identifier, the target objects on which the action can be performed, a set of pre-conditions that must be true before the action can be performed, and the effects that result from the action when it is performed.

Like KAoS, Rei allows users to extend the basic ontology with additional domain dependent ontologies to express concepts and resources that are peculiar to certain domains. For instance, if there is a need to model the specific action of printing a file on a local printer, the general action class of the Rei basic ontology can be customized to include more contextual information about specific printing options.

Unlike KAoS, which relies on the native APIs or message-passing facilities of the underlying platform in which its services reside, the Rei language relies on a rich set of speech act primitives to dynamically exchange rights and obligations between entities, such as the delegation, request, cancellation, or revocation of policies. Rei uses meta-policies to resolve policy conflicts that are automatically detected by the Rei policy engine.

Policy specification. Rei's concepts of rights, permissions, obligations, dispensations, and policy rules are represented as Prolog predicates. No GUI equivalent of KPAT is provided in the current version of Rei.[5] Table 1 reports some examples of Rei rights and policy specifications.

Table 1. Example of Rei policy specification.

	Rei Policy Constructs	**Rei Policy Instance (example)**
A	*PolicyObject (Action , Conditions)*	right (examineStudent , professor (Var))
B	*has (Subject , Policy Object)*	has (Var , right (examineStudent, professor (Var))
C	*action (ActionName, TargetObjects, Pre-Conditions, Effects)*	action (examineStudent, Y, [student(Y)] , [])

Row A describes a *Policy Object* that is used to model rights, permissions, obligations, and dispensations to perform a specific action under certain conditions. The right column of Row A shows a Policy Object instance that defines the right to perform the *examineStudent* action when the condition on the variable *Var* is true— that is, when Var is a professor. Row B illustrates the concept of a *Policy Rule*, which is used to define the association between the subject of a policy and a Policy Object via the *has* construct. In the example on Row B, the right defined in row A is now associated with a set of entities that are represented by the variable *Var*. These entities can examine students only if they satisfy the *professor(Var)* condition, i.e., the entities need to be professors. Finally, row C shows the definition of an action. In the right column, the action *examineStudent* is performed on all the entities (*Y* variable) that are students *(student(Y))* without producing any effect on the system *([])*. For additional detail, see [6].

Rei also permits users to specify role-based access control policies or policies relating not only to individuals but also to groups of entities. For instance, in row B the variable *Var* (all variables are denoted by a beginning capital letter) can be resolved by a role or group based set of entities that resolve the condition of the Policy Object, i.e. by all the members of the *professor* role or group. Roles and groups are not represented by specific constructs of the language and are not described in the basic Rei ontology. In making this design decision, the developers of Rei argue that roles do not receive a consistent interpretation in different applications, and may not

---

[5] A new GUI for Rei is under development in the form of a plug-in for Protégé.

always be required. Policy programmers deal with a smaller set of constructs and code domain-specific implementations of roles and groups when required.

Policy deployment model. The Rei framework provides a policy engine that reasons about the policy specifications. The engine accepts policy specification in both the Rei language and in RDF-S, consistent with the Rei ontology. Specifically, the engine automatically translates the RDF specification into triples of the form (subject, predicate, object). The engine also accepts additional domain-dependent information in any semantic language that can then be converted into this recognizable form of triple. The engine is consistent and complete [6] and allows queries according to the Prolog language about any policies, meta-policies, and domain dependent knowledge that have been loaded in its knowledge base.

The Rei policy engine can detect modality conflicts among policies. In particular, the engine marks two policies as conflicting if they are of conflicting modalities and if there is an overlap in subject, target, and action. In order to resolve the detected conflicts, Rei provides the possibility to specify meta-policies that define priorities on policies, and/or set precedence relations between policy modalities.

Rei decentralizes the policy control to system entities. Some constructs allow specification of speech acts to exchange policies between entities and comprehensive delegation management allows the rights of entities to be configured dynamically.

The Rei framework does not provide an enforcement model. In fact, the policy engine has not been designed to enforce the policies but only to reason about them and reply to queries. For example, the engine can say if an entity has the right or obligation to perform a certain action, but then it does not include mechanisms to ensure enforcement of the policy by, for example, forbidding the policy subject from performing unauthorized actions or by forcing the entity to execute required actions. Thus it provides limited or no protection from malicious or non-compliant components or agents.

3.3 Ponder

Ponder is a declarative, object-oriented language that supports the specification of several types of management policies for distributed object systems and provides structuring techniques for policies to cater for the complexity of policy administration in large enterprise information systems [1, 9, 14]. It has been widely deployed in many applications.

Ponder distinguishes between basic and composite policies. A basic policy is considered a rule governing the choices in system behaviour and is specified by a declaration between a set of *subjects* and a set of *targets*. These sets are used to define the managed objects that the policy operates over. Ponder uses the term subject to refer to users, principals, or automated manager components, which have management responsibility (i.e., have the authority to initiate a management decision). A subject can operate on target objects (resources or service providers) by invoking methods visible in the target interface. The fundamental policy types in Ponder are obligations and authorizations. For example, *obligation policies* define "the actions that policy subjects must perform on target entities when specific relevant

events occur" while *authorization policies* define "what operations a subject is authorized to do on target objects" [14].

Composite policies allow the basic policies relating to organizational units to be grouped. Examples of composite policies are *role* and *relationships* policies. In Ponder, roles are groups of policies governing the behavior of the same subject by specifying its rights and duties while relationships group the policies defining the rights and duties of roles towards each other.

Ponder policies rely on the key concept of management domains. Domains provide a means of grouping objects on which policies apply and can be used to partition the objects in a large system as desired. Ponder policy subjects and targets are defined in terms of domain scope expressions that allow the combination of domains to form composite set of objects, but also to identify a single named object within a single domain.

Policy specification. Figure 3 illustrates an example of a Ponder authorization policy. The policy specifies that the professor principals have read access to all the exercise files of their students only during the opening hours of the school, i.e. from 7 am to 7 pm and from Monday to Friday.

```
type auth+ FileAccess (subject professor,
                        target   exerciseFiles )
  {
    action   read;
     when
          Time.between(0700, 1900) and
          Time.between('mon', 'fri');
  }
inst auth+ P1 = FileAccess ("professor/Green",
                  "NodeServer/StudentFiles");
```

Fig. 3. Ponder authorization policy specification

A *type* policy definition introduces a new user-defined policy type, from which one or more policy instances of that type can be created. Policy types can also be parameterized and the instances created with context-specific parameters, for example, the subject and target sets can be passed as actual parameters. A policy instance declaration creates an instance of a user-defined policy type. P1 is a policy instance that authorizes the professor principal called Green to access the student exercise files hosted on the NodeServer. The read action is a method of the target interface and can be neither refined nor specialized. The `when` clause is used to deny read access to the files during school closing hours.

Note that Ponder does not currently support the specification of default rules for policies. For example, if an authorization policy is not specified for an action, the default constraint on behavior (i.e., whether to permit or forbid the action) is implementation dependent.

Policy deployment model. Ponder provides various graphical tools for editing, updating, removing, and browsing Ponder policies [20]. There are also tools for syntactic and semantic analysis of policy specifications and for transforming Ponder

language specifications directly into XML or Java code that can be interpreted at runtime.

In addition, the developers of Ponder have implemented a prototype conflict detection tool to detect overlaps and conflicts between policies. The tool distinguished between modality conflicts and application-specific conflicts [8]. Similar to KAoS and Rei, the former are inconsistencies in the policy specification that may arise among policies with modalities of opposite signs that refer to the same subjects, targets, and actions (e.g., conflicts between permissions and prohibitions or between obligations and prohibitions). However, the lack of an ontology limits its ability to deal with subjects, actions, and targets at varying levels of abstraction. The latter are inconsistencies between the policy content and external criteria (e.g. conflict between an obligation to access a resource and a limitation on the resource availability). More recent work on policy analysis in Ponder has focused on using abductive reasoning together with an event calculus representation of the policies to identify and detect the policy conflicts [21].

Ponder proposes a general-purpose deployment model [22]. At the end of the policy specification the policy is compiled by the Ponder compiler into a Java class and then represented at runtime by a Java object. The downside is that runtime changes to policy are not possible. The distribution and enforcement model distinguishes between authorization and obligation policies. Authorization policy objects are distributed closer to the target objects of the policy where an Access Controller Agent provides their runtime interpretation and enforcement by allowing or rejecting access requests to controlled target resources. Obligation policy objects are distributed closer to the subject objects of the policy where a Policy Management Agent provides their run-time interpretation and enforcement at policy-relevant event occurrence. Ponder provides the specification of the interfaces for the enforcement agents, i.e., for the Access Controller and Policy Management Agent, but no implementation is provided. There are some systems that implement the Ponder policy deployment model in different application domains [23].

4   Case-Study: Controlling Communication

This section compares the policy frameworks described in Section 3 with the main goal of highlighting their differences and of outlining some general key requirements for a comprehensive semantic approach to policies. The case of a communication policy among agents is considered for purposes of comparison. Communication in multi-agent systems has proven to be a very common application of policy. Let us consider the case of a multi-agent system modeling an academic institute where there is the need to rule the communication between professors and their students, both represented as agents. In the following sections we will use a policy that states that "*professors are permitted to communicate the final examination grade to their students using an encrypted communication only after the approval of the institute's director*" as a running policy example to show how the three approaches represent, analyze and deploy it.

4.1 Policy Representation

Table 2 shows the different representations of the policy example in KAoS, Rei and Ponder.

KAoS exploits a DAML-based representation of the policy example. In particular it is modeled as an instance of a positive authorization policy type with associated properties set to the desired values. For example, the *performedBy* property represents the sender and its value is restricted to the "AgentProfessors", while the *siteOfEnforcement* value is set to the "SubjectSite" meaning that the enforcement mechanisms will be associated with the professor agents that are the subject of the policy.

In Rei, the policy specification is based on a Prolog-like representation. The policy is modeled by means of three basic constructs. In particular, the *action* construct models the "gradeCommunication" action that involve "Professors" and "Students", the *right* construct models the right of executing this action when the set of policy conditions are verified, and finally the *has* construct assigns this right to the "Professors".

The exploitation of a Ponder authorization policy type for mapping the communication policy requires the identification of the resources that need protection and to know in advance their specific names and interfaces. For example, in the specification we assume that the resource to be protected is the communication channel (the "CommunicationChannel" target resource) and that the action that a policy subject can exploit to send data is the "communication" method of the "CommunicationChannel" object interface.

A major advantage of using KAoS for representing the policy is that any policy element (groups, actors, actions, action properties, conditions of applicability) is described by an appropriate ontology at the desired level of abstraction. KAoS policy specification is split into two parts. The first part defines the aspects of the controlled action. This part, represented within a single DAML class, is self-contained and includes information about the type of action, the class of subjects performing this action and additional context information. In comparison, other policy specification approaches separate information about the action type from the class of subjects, making analysis of the policy more complex. KAoS' approach allows users to analyze and compare action classes independently of policies and to apply the policy management information (e.g., policy modality, precedence), which is included in the second part of the policy specification, at the end of the decision process. Finally, KAoS provides a rich set of predefined ontologies that can be extended to accommodate specific application requirements and hides the potential complexity of policy representation and languages behind the KPAT graphical user interface.

In Rei the specification is more readable and compact than a semantic web language such as DAML. On the other hand, the Prolog-like syntax policy specification is not so easy for users with limited expertise in logic languages.

Finally, the Ponder policy specification is easy to understand thanks to its declarative specification. However, the low level of abstraction provided in Ponder by method calls may be a limitation to its use in multi-agent systems, where support for dynamic runtime changes and the need for simultaneously varying levels of description are usually important requirements.

Table 2. Policy Specification in KAoS, Rei, and Ponder

<table>
<tr>
<td>KAoS</td>
<td>

```
<?xml version='1.0'?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
         xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
         xmlns:xsd ="http://www.w3.org/2000/10/XMLSchema#"
         xmlns:policy="http://ontology.coginst.uwf.edu/Policy.daml#"
         xmlns="http://ontology.coginst.uwf.edu/ExamplePolicies/PolicyExample.daml#">
<daml:Ontology rdf:about="">
    …..
<daml:Class rdf:ID="ExaminationGradePolicyAction">
    <daml:intersectionOf rdf:parseType="daml:collection">
     <daml:Class rdf:about="http://ontology.coginst.uwf.edu/Action.daml#EncryptedCommunicationAction"/>
      <daml:Restriction>
        <daml:onProperty rdf:resource="http://ontology.coginst.uwf.edu/Action.daml#performedBy"/>
        <daml:toClass rdf:resource="http://ontology.coginst.uwf.edu/ActorClasses.daml#AgentProfessors"/>
      </daml:Restriction>
      <daml:Restriction>
        <daml:onProperty rdf:resource="http://ontology.coginst.uwf.edu/Action.daml#hasDestination"/>
        <daml:toClass rdf:resource="http://ontology.coginst.uwf.edu/ActorClasses.daml#AgentStudents"/>
      </daml:Restriction>
      <daml:Restriction>
        <daml:onProperty rdf:resource="http://ontology.coginst.uwf.edu/ Action.daml#hasApproval"/>
        <daml:toClass
             rdf:resource="http://ontology.coginst.uwf.edu/ActorClasses.daml#AgentInstituteDirector"/>
      </daml:Restriction>
    </daml:intersectionOf>
</daml:Class>
<policy:PosAuthorizationPolicy rdf:ID="ExaminationGradePolicy">
    <policy:controls rdf:resource="#ExaminationGradePolicyAction"/>
    <policy:hasSiteOfEnforcement rdf:resource="http://ontology.coginst.uwf.edu/Policy.daml#SubjectSite"/>
    <policy:hasPriority>10</policy:hasPriority>
     <policy:hasUpdateTimeStamp>446744445544</policy:hasUpdateTimeStamp>
</policy:PosAuthorizationPolicy >
```

</td>
</tr>
<tr>
<td>Rei</td>
<td>

</td>
</tr>
<tr>
<td>Ponder</td>
<td>

```
domain prof = /SysEntities/Agents/ProfessorAgents;
domain stud = /SysEntities/Agents/StudentsAgents;

inst auth+ ExamGradeComm {
    subject  s= prof;
    target  t = /SysEntities/SysServices/CommunicationChannel;
    action t.communication ("Encrypted", data, destination) ;
    when data.getType = "Grade"
     && destination == (stud -> select (st | st.professor == s))
     && s.receivedApproval(s.getInstituteDirector()) == 'true' ;
}
```

</td>
</tr>
</table>

4.2 Policy Reasoning

In KAoS, various inferences can be performed for a variety of purposes, such as detecting and harmonizing conflicting policies (e.g., is any negative authorization

policy denying the same action or a subclass of the same action?), disclosing related policies (e.g. which policies deal the approval of the institute director?), or reasoning about future actions based on knowledge of policies in force (e.g. which entity can read from or write to the database of examination grades?). The power and efficiency of description logics can be leveraged to their full extent, and the use of JTP allows going beyond description logic where necessary. Queries can also be written in Rei through its expressive query language. Rei relies on the Prolog reasoning engine to analyze the policies and can reason over domain-dependent ontologies by automatically translating the RDF policies into triples. In KAoS, this conversion is hidden from the users by Stanford's Java Theorem Prover that converts the knowledge base into triples before reasoning over them and providing only the end results. In Rei, the users have the ability to directly manage the intermediate form. For example, it is possible to create a policy by using the conversion to triple of any instance of a domain dependent class. Moreover, users have the capability to access the policy instances in this form by submitting some Prolog queries to the policy engine.

The increased representational power of DAML/OWL over RDF-S allows more powerful forms of policy inference to be used by developers and users. Moreover, because all aspects of KAoS representation are encoded purely in DAML/OWL, any third-party tool or environment supporting DAML/OWL can perform specialized analyses of the full knowledge base completely independent of KAoS itself, thus easing integration with an increasingly sophisticated range of new tools and language enhancements in the future.

Finally, Ponder can detect modality and application specific conflicts in policies, such as conflicts of duty. Ponder policy rules are translated into an event calculus representation that describes the semantics of the policy language. Abductive reasoning techniques are exploited to analyze the policy specifications to identify existing conflicts and provide explanations on how they might arise [21].

4.3   Policy Deployment

In KAoS, the abstract representation of actions (in contrast to the method calls in Ponder) is both a boon and a bane. On the one hand, this abstraction allows KAoS to be adapted for just about any platform that has the requisite enforcement hooks. On the other hand, deployment in any given platform requires the availability of a specific enforcer for the policy Action Class, i.e., in our example for the EncryptedCommunicationAction class. If such an enforcer does not already exist, it must be implemented. Its design is usually a non trivial task and requires programmers to have a deep understanding of the action class and its mapping to the platform. As libraries of enforcers for common actions across platforms are developed, this will become less and less of an issue.

Rei, as explained in Section 3.2, does not provide any enforcement support and this is one of the main drawbacks of exploiting this language for practical applications. In addition, even if the engine accepts ontology-based policy specification, the main policy management is over the Rei constructs. This is a constraint when compared to KAoS that manages the policy in its original form, leaving to the implementing platforms the possibility to choose the most appropriate run-time representation.

In Ponder, the main advantage of modeling policy actions at a low level of abstraction is that they can be directly implemented in Java with little additional effort. This is the reason for its wide adoption in many existing systems.[6] For example, the communication policy example can be simply implemented by a proxy object with the same interface of the CommunicationChannel object. When the 'communication' method is invoked by the policy subject, the proxy grants/denies the communication permission according to the caller identity and to the specified constraint.

For the sake of clarity, Table 3 reports and synthesizes some of the features of the three approaches discussed in the previous Sections.

Table 3. Features of KAoS, Rei, and Ponder

	KAoS	Rei	Ponder
Ontology-based	Yes	Yes	No
Specification language	DAML/OWL	Rei: (Prolog-like syntax + RDF-S)	Ponder (declarative specification)
Tools for policy specification	KPAT – Graphical editor for ontology and policy management	No** ** a GUI is being developed for the next Rei version	Graphical editor and compiler
Reasoning support	Java Theorem Prover	Prolog engine	Event calculus representation
Enforcement mechanisms	Need to write the code of appropriate enforcers and to insert them in entities to control ** ** Policy automation being explored for the next version	Action execution is outside the Rei engine	Java interfaces for enforcement agents are provided
Flexibility	Ontology can be extended with domain dependent descriptions of local entities	Ontology can be extended with domain dependent descriptions of local entities	Management domain as a structuring technique for partitioning complex object systems

4.4 Discussion

As the previous sections made clear, each form of policy representation exhibits pros and cons and thus the choice of an approach should be driven by the characteristics of the application domain and by the application requirements.

However, our experience to date seems to indicate quite clearly that, apart from the specific considerations of the representation employed, the adoption of Semantic Web representations provide more advantages than drawbacks. Table 4 compares generic Semantic Web approaches to policy specification (KAoS, Rei) to the Ponder approach that is used as a representative of the non-ontology-based languages. The comparison takes into consideration the basic language requirements pointed out in Section 3.

 Ontologies simplify the task of governing the behavior of complex environments. The possibility of representing entities and behaviors at multiple levels of abstraction improves the global *expressiveness,* allowing flexibility in dealing with several kinds of contexts at different level of specifications. The use of ontologies permits the

---

[6] The automatic generation of native Java VM policies directly from KPAT is under development in KAoS.

policy framework to be easily extended simply adding new concepts to the ontology. In traditional languages this task is usually much trickier. For example in Ponder the specification of the communication policy example requires the non-trivial effort to extend the policy language or to convert the policy into a resource control policy. In addition, the possibility of modeling policies at a high level of abstraction allows users to focus their attention more on high-level management requirements than on implementation details, such as the method names in Ponder.

An ontology-based description of the policy enables the system to use concepts to describe the environments and the entities being controlled, thus simplifying their description and improving the *analyzability* of the system. Policy frameworks can take advantage of this powerful feature, such as policy conflict detection and harmonization. In addition, ontology-based approaches simplify access to policy information, with the possibility of dynamically calculating relations between policies and the environment, entities, or other policies based on ontology relations rather than fixing them in advance. Like databases, it is possible to access the information provided by querying the ontology according to the ontology schema. This is an advantage in comparison to traditional languages that provide only pre-defined queries to access information and static representations of policy. How to design the ontology is an application dependent problem. As in database design, ontologies should be designed and extended consistent with the application context and optimized for the most common queries. Finally, ontologies can also simplify the sharing of policy knowledge among different organizations and applications, thus increasing the possibility for entities to negotiate policies and to agree on a common set of policies.

However, the adoption of ontologies for policy specification requires addressing some technical difficulties.

Semantic web languages used for ontology representation still present a complex syntax, long declarative description, and hyperlinks and references to external resources that make the code very difficult to read (e.g., compare the readability of a Ponder policy with a DAML policy). To improve the *ease of use* of these languages, graphical interfaces or other tools for policy specification should be adopted. For example, KPAT provides a graphical interface for policy specification and management but such tool always exacts a price in continued development to keep up with significant changes in foundational ontologies; while Rei provides a simplified language for policy specification that hides the complexity of the semantic representation even if it burdens the user with the task of learning an additional language.

Finally, *enforceability* is a critical aspect for semantic web languages. Ontology-based policy specification can be difficult to implement in comparison to other policy specification such as Ponder, because of the high level specification of ontology-based policies can be far removed from the concrete implementation of the policy enforcement on the systems. Usually the gap between the specification and the implementation of policies cannot be completely overcome in an automated manner, but has to be resolved to a greater or lesser degree by human programmers consistent with the capabilities and features of each platform. For example, independent of the semantic representation adopted, the implementation of an enforcer for the policy example requires the enforcers to locate the Professor agents in the system to control, to intercept any attempt to send data to Student agents (this task might also require the

addition of code in the agent), and to query the policy repository with an appropriate description of the action. Enforcement code generation facilities and libraries of enforcement mechanisms adapted to specific platforms are among the most important features for policy management frameworks to provide to enable their widespread implementation.

Table 4. Comparison between semantic web and traditional languages

	Semantic web languages for policy specification	Ponder ** ** used as example of non-semantic web language
Expressiveness	Capable of representing concepts and behavior of any complex environment	Capable of controlling specific sorts of behavior within object-oriented systems
	Multiple levels of abstraction	Low level of abstraction: object level
	Easy to extend policy ontology at runtime with new concepts	Extensibility supported by object-oriented inheritance at compile-time
Analyzability	Ontology representation simplifies and directly supports policy reasoning, conflict detection and harmonization	Conflict detection requires transforming policy specification into an event calculus representation
	Simplified access to policy information by querying the ontology	Access to single policy object by API – Access to policy repository to be designed
Ease-of-use	Need of specialized tools to assist unskilled users with policy specification and interpretation	Language specifically designed for simple policy specification and direct readability
Enforceability	High-level specification requires skilled programmers or sophisticated policy automation mechanisms for enforcement	Detailed specifications can be directly mapped into policy enforcement mechanisms
	Policy sharing among heterogeneous systems requires an agreement on a common ontology	Policy sharing among heterogeneous systems requires agreement on interfaces

5   Conclusion

Policy based approaches to systems management are of particular importance because they allow the separation of the rules that govern the behavior of a system from the functionality provided by that system [11]. Research into policy based systems management has focused on languages for specifying policies and architectures for managing and deploying policies in distributed environments and complex systems, such as multi-agent systems. This paper has described similarities and differences among three important policy management approaches: KAoS, Rei and Ponder.

Existing approaches have generated divergent solutions that tend to be best suited for particular ranges of applications and discourage a common approach for all situations. Ideally, a common approach to specifying and deploying policy for the management aspects of all application domains is desirable to simplify policy analysis and reduce policy inconsistencies and conflicts and to facilitate policy reuse across various systems. Notwithstanding this ideal, the concept of a universal programming language has never been successful and it is not clear why a common approach should succeed for policy specification and deployment. Although we hope to conduct more formal and thorough analyses in the future, our examination of the issues to date indicate promising advantages in the adoption of Semantic Web languages for policy representation and reasoning.

References

1. N. Damianou, N. Dulay, E. Lupu, M. Sloman: The Ponder Policy Specification Language. In proceedings of Workshop on Policies for Distributed Systems and Networks (POLICY 2001). Springer-Verlag, LNCS 1995, Bristol, UK, (2001)

2. J. M. Bradshaw, P. Beautement, L. Bunch, S. V. Drakunov, P. Feltovich, R.R. Hoffman, R. Jeffers, M. Johnson, S. Kulkarni, A.K. Raj, N. Suri, A. Uszok: Making Agents Acceptable to People. In N. Zhong and J. Liu (Eds.), Intelligent Technologies for Information Analysis: Advances in Agents, Data Mining, and Statistical Learning, Berlin: Springer Verlag, in press, (2003)

3. M. Johnson, P. Chang, R. Jeffers, J. Bradshaw, V-W. Soo, M. Breedy, L. Bunch, S. Kulkarni, J. Lott, N. Suri, A. Uszok: KAoS semantic policy and domain services: An application of DAML to Web-Services-based grid architectures. Proceedings of the AAMAS 03 Workshop on Web Services and Agent-Based Engineering, Melbourne, Australia, July, (2003)

4. A. Uszok, J. Bradshaw, R. Jeffers, N. Suri, P. Hayes, M. Breedy, L. Bunch, M. Johnson, S. Kulkarni, J. Lott: KAoS policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement. In Proceedings of IEEE Fourth International Workshop on Policy (Policy 2003). Lake Como, Italy, 4–6 June, Los Alamitos, CA: IEEE Computer Society, pp. 93–98, (2003)

5. J. M. Bradshaw, A. Uszok, R. Jeffers, N. Suri, M. Burstein: Representation and reasoning for DAML-based policy and domain services in KAoS and Nomads. Proceedings of the Autonomous Agents and Multi-Agent Systems Conference (AAMAS 2003). Melbourne, Australia, New York, NY: ACM Press, pp. 835–842, (2003)

6. L. Kagal, T. Finin, A. Johshi: A Policy Language for Pervasive Computing Environment. In Proceedings of IEEE Fourth International Workshop on Policy (Policy 2003). Lake Como, Italy, 4–6 June, Los Alamitos, CA: IEEE Computer Society, pp. 63–76, (2003).

7. L. Kagal: Rei: A Policy Language for the Me-Centric Project. HP Labs Technical Report, HPL-2002-270, (2002)

8. E. C. Lupu, M. Sloman: Conflicts in policy-based distributed system management. IEEE Transaction on Software Engineering, Vol. 25, No. 6, (1999)

9. M. Sloman: Policy Driven Management for distributed Systems. Plenum Press Journal of Network and Systems Management, Vol. 2, No. 4, (1994), 333–360

10. P. Feltovich, J. M. Bradshaw, R. Jeffers, A. Uszok: Order and KAoS: Using policy to represent agent cultures. In Proceedings of the AAMAS 03 Workshop on Humans and Multi-Agent Systems. July, Melbourne, Australia

11. S. Wright, R. Chadha, G. Lapiotis (eds.): Special Issue on Policy Based Networking. IEEE Network, Vol. 16, No. 2, March, (2002), 8–56

12. J. Fritz Barnes, R. Pandey: CacheL: Language Support for Customizable Caching Policies. In Proceedings of 4th Interantional Web Caching Workshop, San Diego, CA, (1999)

13. J. Hoagland: Specifying and Implementing Security Policies Using LaSCO, the Language for Security Constraints on Objects. Ph.D. dissertation, UC Davis, (2000)

14. N. Damianou, N. Dulay, E. C. Lupu, M. Sloman: Ponder: A Language for Specifying Security and Management Policies for Distributed Systems. Imperial College, UK, Research Report Department of Computing 2001, (2000)

15. The IETF Policy Framework Working Group: Charter available at http://www.ietf.org/html.charters/policy-charter.html

16. D. N. Allsopp, P. Beautement, J. M. Bradshaw, E. H. Durfee, M. Kirton, C.A. Knoblock, N. Suri, A. Tate, C. W. Tompson: Coalition Agents Experiment: Multiagent Cooperation in International Coalitions. IEEE Intelligent Systems, Vol. 17, No. 3, (2002), 26–35

17. J. M. Bradshaw, M. Sierhuis, A. Acquisti, P. Feltovich, R. Hoffman, R. Jeffers, D. Prescott, N. Suri, A. Uszok, R. Van Hoof: Adjustable autonomy and human-agent teamwork in practice: An interim report on space applications. In H. Hexmoor, R. Falcone, & C. Castelfranchi (Ed.), Agent Autonomy. Dordrecht, The Netherlands: Kluwer, (2003), pp. 243–280
18. N. Suri, J.M. Bradshaw, M.R. Breedy, P.T. Groth, G.A. Hill, R. Jeffers: Strong Mobility and Fine-Grained Resource Control in NOMADS. In Proceedings of the 2nd International Symposium on Agents Systems and Applications and the 4th International Symposium on Mobile Agents (ASA/MA2000). Springer-Verlag, (2000)
19. M. Kahn, C. Cicalese: CoABS Grid Scalability Experiments. O.F. Rana (Ed.), 2nd International Workshop on Infrastructure for Scalable Multi-Agent Systems at the 5th International Conference on Autonomous Agents. Montreal, CA, New York: ACM Press, (2001)
20. N. Damianou, N. Dulay, E. Lupu, M. Sloman, T. Tonouchi: Tools for Domain-based Policy Management of Distributed Systems. In Proceedings of Network Operations and Management Symposium (NOMS'02). IEEE Press, Florence, Italy, (2002), 203–217
21. A. K. Bandara, E. Lupu, A. Russo: Using Event Calculus to Formalise Policy Specification and Analysis. In Proceedings of 4th IEEE workshop on Policies for Distributed Systems and Networks (POLICY 2003), Lake Como, Italy, (2003)
22. N. Dulay, E. Lupu, M Sloman, N. Damianou: A Policy Deployment Model for the Ponder Language. In Proceedings of IEEE/IFIP International Symposium on Integrated Network Management (IM'2001). IEEE Press, Seattle, May 2001
23. R. Montanari, G. Tonti, C. Stefanelli: A Policy-based Mobile Agent Infrastructure. In Proceedings of 2003 Symposium on Applications and the Internet (SAINT 2003). IEEE Press, Orlando, Florida, US, (2003), 370–379