New Developments in Ontology-Based Policy Management: Increasing the Practicality and Comprehensiveness of KAoS

Andrzej Uszok, Jeffrey M. Bradshaw, James Lott, Maggie Breedy, Larry Bunch,
Paul Feltovich, Matthew Johnson, and Hyuckchul Jung
Florida Institute for Human and Machine Cognition (IHMC), 40 S. Alcaniz, Pensacola, FL 32502
{auszok, jbradshaw, jlott, mbreedy, lbunch, pfeltovich, mjohnson, hjung}@ihmc.us

Abstract

The KAoS policy management framework pioneered the use of semantically-rich ontological representation and reasoning to specify, analyze, deconflict, and enforce policies [9, 10]. The framework has continued to evolve over the last five years, inspired by both technological advances and the practical needs of its varied applications. In this paper, we describe how these applications have motivated the partitioning of components into a well-defined three-layer policy management architecture that hides ontology complexity from the human user and from the policy-governed system. The power of semantic reasoning is embedded in the middle layer of the architecture where it can provide the most benefit. We also describe how the policy semantics of the core KAoS Policy Ontology has grown in its comprehensiveness. The flexible and mature architecture of KAoS enables straightforward integration with a variety of deployment platforms, ranging from highly distributed systems, such as the AFRL Information Management System, to human-robotic interaction, to dynamic management of quality-of-service and cross-domain information management of wireless networks in resource-constrained or security-sensitive environments.

Keywords: *policy, ontology, OWL, KAoS, policy management*

1. Introduction

Over the past five years, the use of W3C's Web Ontology Language (OWL) has expanded from its primary use in representing Semantic Web content and services to important roles in a variety of additional applications. Given the trend toward increasing complexity and dynamics in applications requiring policy services, and the high desirability of open and extensible standards for distributed systems, OWL occupies an attractive niche for policy representation.

The flexibility and power of a policy management framework is to a large degree determined by the expressivity of its policy representation, provided it is also computationally efficient. In our experience, OWL has not only proven to be remarkably expressive and efficient but is also straightforwardly extensible and adaptable, even at runtime when sophisticated real-time management and analysis of new or existing policies may be required [8, 10]. As an additional benefit, ontologies provide a natural means for supporting alternate sets of policy vocabulary for different applications.

Despite these advantages, our reliance on OWL for policy representation has provided its share of challenges. For example, thinking about policies in terms of sophisticated ontologies and reasoning mechanisms can be a daunting task for new users. Among the important things we have learned is that a general purpose ontology-based policy framework such as KAoS needs to make the sophistication, flexibility, and power of ontological representation and reasoning available to people in a simple and understandable manner. In response to this requirement, we have developed a three-layer policy management architecture that ensures consistency among these separate but interdependent components. Experience in a wide variety of applications and settings has helped us smooth many of the initial rough edges of the approach.

In this article, we describe many of the interesting new features of KAoS, including: details of the three-layer architecture, the hypertext policy editor and policy wizard, support for representing and reasoning about history, state, and spatial properties as they relate to policy, support for logical policy precedence, network-efficient policy distribution methods, a revamped Guard architecture, and additional mechanisms to support obligation policies.

2. Related Work

Ponder was the forerunner of modern policy languages and management systems, and significant work continues on its development ([1], www.ponder2.net). Recent years have witnessed the development of additional approaches—some based on XML (e.g., XACML [7]) and others, like KAoS, based on Semantic Web representations (e.g., Rei [4], PolicyTab [6]). There have also been attempts to recode XML-based policies into ontology-based policy representations [5]. Languages tailored for specific applications have also appeared (e.g., CoRaL [3] for radio spectrum control).

We believe that the increasingly demanding requirements of new applications (see e.g., [2]) will be difficult to meet without the advantages afforded by semantically-rich representations such as OWL, coupled with practical, usable, and efficient policy management systems that can make use of them. While special-purpose languages for narrow application deployment may sometimes outperform more general approaches in their particular niche, policy is being increasingly used in ways that cut across traditional domains (as described in section 8 and [2, 11]), making a more comprehensive approach to policy management extremely desirable.

3. KAoS Overview

The KAoS policy services framework [9, 10] has been adapted to run on a variety of agent, robotic, Web services, Grid computing (e.g., Globus), and traditional distributed computing platforms, and across a variety of industrial, military, and space applications [11]. In addition to services directly related to policy management, KAoS also provides the basic services for distributed computing, including message transport and directory services. Because the services are accessed through a well-defined Common Services Interface (CSI), application developers can use whatever subset of its capabilities (e.g., registration, transport, publish-subscribe, domain management, remote request forwarding, queries) are appropriate for a given situation.

3.1. Three-Layered Architecture

Two important requirements for the KAoS architecture are modularity and extensibility. These requirements are supported through a framework with well-defined interfaces that can be extended, if necessary, with the components required to support application-specific policies. The basic elements of the KAoS architecture are shown in Figure 1; its three layers of functionality correspond to three different policy representations:

- *Human interface layer:* This layer uses a hypertext-like graphical interface for policy specification in the form of natural English sentences. The vocabulary is automatically provided from the relevant ontologies, consisting of highly-reusable core concepts augmented by application-specific ones.
- *Policy Management layer:* Within this layer, OWL (http://www.w3.org/TR/owl-features) is used to encode and manage policy-related information. The Distributed Directory Service (DDS) encapsulates a set of OWL reasoning mechanisms.
- *Policy Monitoring and Enforcement layer:* KAoS automatically "compiles" OWL policies to an efficient format that can be used for monitoring and enforcement. This representation provides the grounding for abstract ontology terms, connecting them to the instances in the runtime environment and to other policy-related information.

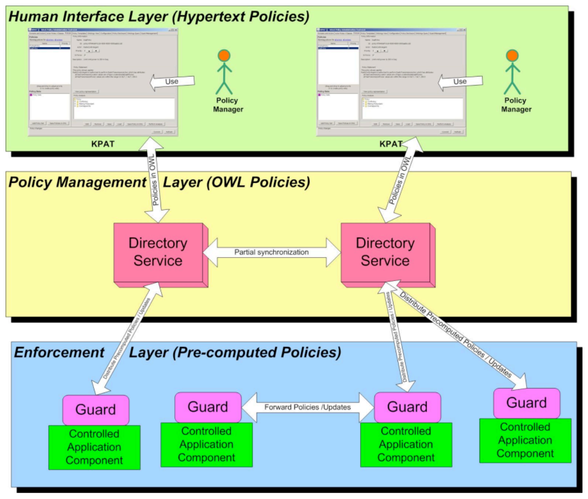

Human Interface Layer (Hypertext Policies)
Use
Policy Manager
KPAT
Use
Policy Manager
KPAT
Policies in OWL
Policies in OWL
Policy Management Layer (OWL Policies)
Directory Service
Partial synchronization
Directory Service
Distribute Precomputed Policies / Updates
Distribute Precomputed Policies / Updates
Distribute Precomputed Policies / Updates
Enforcement Layer (Pre-computed Policies)
Guard
Controlled Application Component
Guard
Controlled Application Component
Forward Policies /Updates
Guard
Controlled Application Component
Guard
Controlled Application Component

Figure 1: KAoS Policy Services Architecture

Maintaining consistency among these layers is handled automatically by KAoS, a task made more challenging because each layer implements its functionality in a distributed rather than a centralized manner. The three layers are described in more detail in sections 5, 6, and 7.

Within each of the layers, the end user may plug-in specialized extension components if needed, as described in more detail throughout the paper. Such components are typically developed as Java classes and described using ontology concepts in the configuration file. They can then be used by KAoS in policy specification, reasoning and enforcement processes.

Figure 1 shows one of many ways in which KAoS may be deployed. Developers can pick and choose whatever elements of KAoS they deem useful for a given application. For instance, some applications may only require a single guard while others may use several distributed guards. Some applications may not use guards at all, in this case using KAoS only as a policy specification and management tool.

3.2. Ontologies in KAoS Policy Definition

The KAoS core policy ontology consists of a set of independent OWL files (available at: http://ontology.ihmc.us/ontology.html). They define the root concepts for policy-governed actions, actors, places, states, history, situations, environmental properties related to actions (e.g., computing or network resources), groups (e.g., domains, roles, teams), and so forth. In addition, the ontology contains the concepts needed to define the policies themselves. There are currently more than 100 classes and properties defined in this basic ontology.

Application developers normally extend the core ontology with additional application-specific classes, properties, and individuals that can be used as vocabulary in policy definitions. Such ontologies define concepts that will enable information sharing between KAoS and the application itself. For example, in an application of KAoS to dynamic configuration of military radios, the application required runtime information about transmission parameters from the ontology. Using KAoS, policy developers are not only able to refer to application-specific concepts but also to link these concepts with more abstract ones in the core ontology.

The following types of ontologies are typically createdby application developers as extensions to the KAoS core ontology:

- *ApplicationAction.owl* – defines application action classes and their properties. New Action classes must be subclassed from the core class *http://ontology.ihmc.us/Action.owl#Action.* Examples of application-specific action classes might include radio transmission actions, actions describing the movement of robots, or actions relating to the issuance of weather reports.
- *ApplicationActor.owl* - defines application actor classes (or roles) and their properties. New Actor classes must be subclassed from *http://ontology.ihmc.us/Actor.owl#Actor.* Examples of application-specific actor classes might include radio operator, quadripedal robot, or producer of news report.
- *ApplicationEntity.owl* – defines application specific entities and their properties. These are used to define contexts for application actions. New Entity classes must be subclassed from *http://ontology.ihmc.us/Entity.owl#Entity*. Examples of application-specific entity classes might include transmission area, weather, or wind.

It is also possible to link application-defined concepts to any number of pre-existing ontologies. By subclassing new concepts as appropriate subclasses to existing concepts in the KAoS core ontology, the new concepts can be used in policy vocabulary, reasoning, and enforcement. New ontologies (and related policies) can be defined, imported, or modified at runtime as needed.

3.3. Grounding Ontologies in the World

Ontology-based policy services dynamically define mappings between class definitions and entities in the controlled environment and in the world. This can be accomplished in several ways.

First, static elements of the environment may be defined as individuals in application-specific ontologies. For instance, we might describe specific instances of robots, a range of existing radio spectrums, or a set of producers of weather reports for a given area. In addition, dynamic elements of the environment may be registered within KAoS through the Guard interface, or information about them can be provided to a guard at policy enforcement time. For instance, at runtime, new areas of robot operation can be defined, new weather reports can be produced, new radios can be introduced, and new domains, roles, or teams can be formed or removed.

Extension components added to KAoS can be used to collect the history of actions, sense the state of the environment, or access external databases to provide information needed for policy enforcement.

4. KAoS Policy Semantics

4.1. The Basic Form of KAoS Policies

Like Ponder and Rei, KAoS supports two main types of policy: authorization and obligation [8]. The set of permitted actions is determined by authorization policies that specify which actions an actor or set of actors is allowed (*positive authorization policies*) or not allowed *(negative authorization policies*) to perform in a given context. Obligation policies specify actions that an actor or set of actors is required to perform (*positive obligations*) or for which such a requirement is waived (*negative obligations*). All other kinds of policies (e.g., delegation, teamwork coordination) are built from these two primitive types, combined with other aspects of KAoS policy semantics (e.g., domains, history, state).

The basic form of KAoS policies is as follows:

[Actor] is [constrained] to perform [controlled action] which has [any attributes]

[Actor] is a variable that refers to the subject of the policy-controlled action. Any of the following can be defined as actors:

- A single actor instance (e.g. Robot32);
- An actor class or a role (as in role-based access control) using an actor class name (e.g. members of class Robot, Weather Producer, Team A; all Robots within 50 feet of my current location);
- The complement of some instance or set of instances (e.g. any Robot except Robot324);
- The complement of actor class or set of classes (e.g. any Robot that is not Pioneer).

[constrained] is a variable that refers to the basic type of the policy (i.e., positive or negative authorization, positive or negative obligation).

[controlled action] is a variable that refers to the action class that will be controlled by the policy (e.g. Radio Transmission, Movement).

[any attributes] is an optional variable referring to one or more attributes of the controlled action. For example, a Radio Transmission action may have attributes defining configuration parameters, the destination of the transmission, and so forth. These attributes will typically be used to describe aspects of context relating to the controlled action.

Attributes can be used either as part of simple value restrictions or to define a test that dynamically relates two or more separate attributes. A simple restriction typically has the form:

[all | some] [attribute] values are [within the set of enumerated instances | of a given type]

For example, such a restriction allows a policy to say that:

- A valid credential for a given user must be one of the set of recognized credentials.

- Receivers of a given radio transmission must all be holders of a particular security clearance.

Some policies require the definition of dynamic attributes whose values must be tested relative to the values of some other attribute. Support in KAoS for this feature allows users to define policies that relate to the local context of the action or actor. For example:

- A robot is authorized to request assistance only from current members of its team,
- Employees are forbidden from using printers belonging to departments other than their own.
- Users are authorized to share documents only if they share a common credential.

4.2. Role-Value Maps

OWL semantics do not allow the expression of the constraints on attributes described above. The KAoS role-value-map reasoner [10] solves this problem. The reasoner can handle any of the following forms:

- Attribute values must *equal* the values of another action attribute (e.g., user department membership must equal the ownership property of the printer used in the print action).
- Attribute values must *contain all* values of another action attribute (e.g., receivers of a radio transmission must all be members of the set of security clearance holders).
- *At least one* attribute value must equal the value of another action attribute (e.g., at least one of the credentials of the user initiating communication must be in the set of credentials of the receiver of the message).
- *None* of the attribute values is equal to the values of another action attribute.

4.3. Spatial Relations

Spatial relations (*http://ontology.ihmc.us/spatial/*) have been useful in policies such as: requiring a robot to stay to the right of an astronaut, restricting robot movement through a given area, or restricting radio transmission to authorized power levels within a given political zone.

For such purposes, we have implemented the KAoS Spatial Reasoning Component (KSPARC[1]). KSPARC can reason about the location and orientation of objects relative to an arbitrary coordinate system and any number of dynamically-definable regions described as polygons. KSPARC can reason with any mix of absolute and egocentric references, allowing the following sorts of policy reasoning:

- Queries for relative positions between objects (e.g., in front / behind, to the left / right, inside / outside);
- Translation between reference orientations (e.g., my right = your left);
- Listing spatial relations between two objects;
- Spatial relationships between two objects in relation to a reference object (e.g., further to the left or right of, higher than);
- Calculating whether or not an object can be seen from a given position and orientation;
- Calculating specific values associated with any spatial relations (distance, rotation).

4.5. Obligation Policy Triggers

Unlike authorization policies, obligation policies include triggers that specify the conditions under which the required action will be activated (e.g., When two hours have elapsed, the operator must terminate the transmission). Trigger actions are specified in a manner that is similar to controlled actions.

Relative attributes are typically used to relate the trigger to the obliged action (e.g., "If [some robot] fails, then [some robot] must notify its teammates; "If [some message] is of class 'secret' or greater, [some message] must be logged to the audit queue)."

4.4. History and Current State of a Situation

All policies are defined in the context of a *Situation*, which possesses a history and a set of variables describing its current state.

History is used to qualify the applicability of the policy relative to past events, i.e.:

This policy applies when [actor] has performed [action] which has [any attributes] _at least [#] times_ _within the last [x] [minutes|hours|days…]

For example, this feature could be used in a policy to forbid system access to anyone who has a history of two or more failed logon attempts in the last five days.

State information is used to qualify the applicability of a policy relative to values representing the current state of one or more variables, i.e.:

This policy applies when the [situation element] [has any state | is] [state] has [attributes]

For instance, a given policy governing the frequency of weather reports might apply only when the Weather is Bad Weather. Another policy might apply when the Weather has the following attributes: temperature is > 75 and sunlight is bright.

4.6. Policy Precedence

The ranking of policies by order of importance is used in two important phases of policy management. In the first case, when a policy is created or updated, the policy service must determine whether the new policy is consistent with the existing set of policies. If the new policy and an existing policy have the same ranking in importance, cover overlapping actor, action, and context classes, and have conflicting modalities (i.e., authorized/forbidden, required/forbidden, required/not required), the new policy is rejected and deconfliction recommendations are given to the user [9]. If the new

[1] Pronounced "KAY-spark."

policy overlaps with an existing policy and has a conflicting modality, but one of the two policies has a higher ranking than the other, no deconfliction is required.

The second case occurs during authorization policy decisions (i.e., determining whether or not an action is permitted). As part of this process, KAoS collects a set of policies with action classes positively classifying the action instance being tested. Policy ranking allows KAoS to group applicable policies into sets of decreasing importance. At policy creation time, the consistency checking mechanism has already assured that the sets are not in conflict. At policy decision time, within the policy set with the highest priority for a given action, a single positive or negative authorization policy will determine whether the action is permitted or forbidden.

KAoS originally relied on numeric policy priority assignments by users to determine how policies should be ranked. This mechanism has important advantages. For example, at policy commitment time it is immediately evident that only policies with the same numeric priorities must be tested for consistency with the new policy. When policy decisions are made, policy applicability is analyzed according to the partially-sorted priority sequence. Thus, the numeric policy ranking approach executes quickly. Unfortunately, a disadvantage of this approach has been that people may have difficulty assigning meaningful priorities and tracking how a given policy's priority relates to the priorities of other policies. For this reason, we are extending the priority mechanism in KAoS to use a logical precedence mechanism in addition to numeric priorities. The following relations are supported:

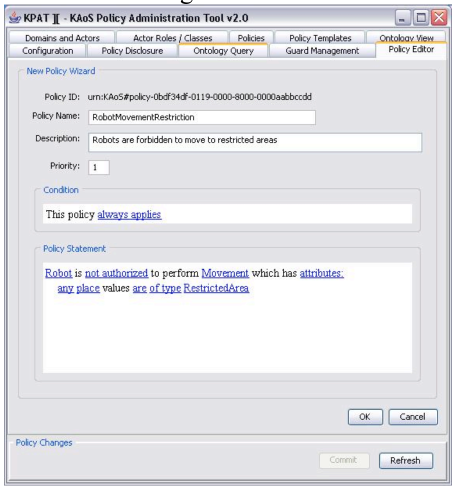
- Name or role of the person who authorized the policy (e.g., Jim Hanna's policies take precedence over anyone else's policies, policies of the domain administrator take precedence over user policies);
- Time when the policy was created (e.g., more recent policies take precedence over older policies);
- Relative scope of class of the policy subject (e.g., superdomain policies take precedence over subdomain policies; policies for Pioneer robots take precedence over policies for the general robot class);
- Relative scope of the class of policy action (e.g., policies about writing to a specific directory take precedence over policies about writing to the volume);
- Modality of the policy (e.g., Negative authorizations take precedence over Positive authorizations);
- Priority level of the policy (e.g., high-medium-low). These levels will allow users to define any number of arbitrarily-ordered priority categories and associate them with names of their own choice. These priority levels can be used in conjunction with any of the precedence relations described above.

The user is also allowed to manually change the location of any policy in the ranked list.

When the policy service does not find any policy applicable to the current situation, it must still provide an answer to the authorization question. For this reason, users can specify a default authorization mode on a per domain basis. A domain is considered *tyrannical* if it is configured such that nothing is permitted unless explicitly authorized, and *laissez-faire* in the opposite case.

5. KAoS Human Interface Layer

The KAoS Policy Administration Tool (KPAT[2]) implements a graphical user interface for policy management functionality. Besides its use in policy specification and analysis, it is used for administration tasks such as browsing and loading ontologies, and domain and Guard management.



Policy ID: urn:KAoS#policy-0bdf34df-0119-0000-8000-0000aabbccdd
Policy Name: RobotMovementRestriction
Description: Robots are forbidden to move to restricted areas
Priority: 1
This policy always applies
Robot is not authorized to perform Movement which has attributes:
any place values are of type RestrictedArea

Figure 2: Authorization Policy in Hypertext Mode

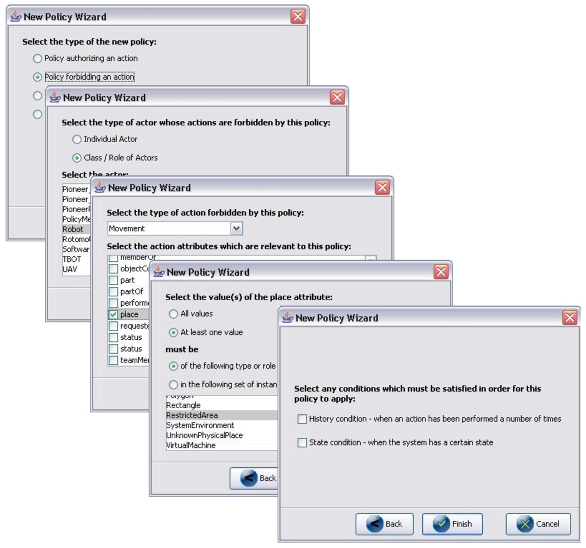
KPAT's generic Policy Editor (Fig. 2) presents an administrator with a starting point for policy construction—essentially, a very generic policy statement shown as hypertext (see the generic policy statement in section 4.1). Clicking on a specific link that represents a variable provides the user with choices allowing him to make a more specific policy statement. During use, KPAT accesses the ontology loaded into the DDS and always provides the user with the list of choices narrowed to the current context of the policy construction. New classes and instances can also be created from KPAT.

To further simplify policy construction, KPAT provides two additional policy creation interfaces:

- The *Policy Wizard* (Fig. 3) takes a user step-by-step through the policy creation process. Information selected for presentation is conditioned on whatever has been selected previously, making the experience as simple and foolproof as possible.
- The *Policy Template Editor* allows custom policy editors for a given kind of policies to be created by point-and-click methods. For instance, if an application will require the definition of several

[2] Pronounced "KAY-pat."

policies governing publish/subscribe actions, a custom policy editor can be quickly created by limiting choices to just what is needed, thus eliminating the requirement for repetitive selections.


New Policy Wizard

Select the type of the new policy:
- ( ) Policy authorizing an action
- (●) Policy forbidding an action

New Policy Wizard

Select the type of actor whose actions are forbidden by this policy:
- ( ) Individual Actor
- (●) Class / Role of Actors

Select the actor:

Pioneer_
Pioneer_
PioneerF
PolicyMe
Robot
Rotomol
Softwar
TBOT
UAV

New Policy Wizard

Select the type of action forbidden by this policy: Movement

Select the action attributes which are relevant to this policy:
- [ ] memberOf
- [ ] objectC
- [ ] part
- [ ] partOf
- [ ] perform
- [✓] place
- [ ] requeste
- [ ] status
- [ ] status
- [ ] teamMe

New Policy Wizard

Select the value(s) of the place attribute:
- ( ) All values
- (●) At least one value

must be
- (●) of the following type or role
- ( ) in the following set of instan

Polygon
Rectangle
RestrictedArea
SystemEnvironment
UnknownPhysicalPlace
VirtualMachine

New Policy Wizard

Select any conditions which must be satisfied in order for this policy to apply:
- [ ] History condition - when an action has been performed a number of times
- [ ] State condition - when the system has a certain state

Figure 3: Authorization Policy Using KPAT Wizard.

KPAT is also used to change policy precedence, activate and deactivate policies, and to provide for a variety of analysis and test options not discussed here. Once a policy is deconflicted and committed, the Jena framework (*http://jena.sourceforge.net/*) is used to dynamically build the OWL policy. A demonstration version of KPAT is available as a Java Web Start application from *http://ontology.ihmc.us/kaos.html*.

6. KAoS Policy Management Layer

This layer mediates between the human interface layer and the monitoring and enforcement layer. Though other kinds of reasoning take place in the top and bottom layers, the middle layer is where virtually all the ontological reasoning and representation takes place. The higher computational cost of reasoning for policy deconfliction and analysis is paid up front so that policy monitoring and enforcement in the lowest layer can be performed in a highly efficient manner.

6.1. Bootstrapping and Basic Policy Reasoning

An entire KAoS configuration, including application-specific ontologies and policies, can be captured declaratively as OWL and reused at a later time. During bootstrap, the core policy ontology (section 3.1) is loaded into the ontology reasoner integrated with KAoS. We have used KAoS with Java Theorem Prover (JTP[3]) and Pellet,[2] but there is no reason why other reasoners could not be used, since the DDS interacts with them through a well-defined abstract interface. After bootstrap, application-specific ontologies may be loaded. The reasoner maintains information about domain structures, registered actors and other entities pertinent to the situation. as described in section 3.3. With respect to policy management, the reasoner supports the creation of policies by supplying KPAT with lists of vocabulary terms (e.g., all of the action classes which can be performed by a given class of actor). Policies can also be created, of course, through a programmatic interface. During policy analysis the reasoner finds relations between action classes controlled by policies through subsumption reasoning [9]. Description logic, however, does not recognize role-value map semantics. So when the subsumption reasoner finds a relation between actions and subsequently policies it is still up to the manager to determine whether potential instances of role-value maps separate the actions and nullify the policy relation.

As policies are distributed to guards (see below), the reasoner classifies existing instances (e.g., the list of actors) so that relevant information of other kinds can be sent to the guards at the same time.

6.2. Policy Distribution

When policies are added and modified, or when a guard connects or reconnects to the Distributed Directory Service (DDS), the DDS assembles the appropriate update information relevant to a particular guard. The DDS maintains a registry about the interests of each guard (see below) and a history of what has already been sent to it.

During the policy update process, the OWL policy representations are converted to a form that enables the guards to make complex enforcement decisions very efficiently. The algorithm traverses the OWL policy structure and "compiles" it into a hash-table-like structure where entries for each policy describe policy action attributes, a range of acceptable values, a list of super-attributes and sub-attributes defined in the ontology, and, if available, the encoded definition of the range class. Abstract actions not explicitly included in the business code are changed into the enforceable action with a definition of restrictions that distinguish the abstract class from a base class encoded as an attribute. For instance, if a given policy talks about the publication of weather reports and the business code just has a method of publication with a parameter specifying the information, then the range of values for the corresponding attribute of the published action used in the enforceable representation of policy is set to the weather report type. Since the DDS has a record of information sent to guards it can recognize that some of the information cached in the pre-computed guard policy representation has changed when entities register or deregister, and can send updates as appropriate.

[3] http://www.ksl.stanford.edu/software/JTP/

[2] http://pellet.owldl.com/

We are near completion of a new mechanism to allow direct communication among a group of guards for the exchange of policy and cache updates. We have designed this capability for a mobile ad-hoc network environment where continuous direct communication between the DDS and the guards is not always possible.

7. Policy Monitoring and Enforcement Layer

The guard is where KAoS meets the application. Its primary role is as a policy decision point. Guards register to receive policies about particular entities, classes of entities, and/or for a given set of action classes. Because guards can save their policies and reload them directly from a snapshot, they can be bootstrapped in a standalone mode without a need to connect to the DDS. This functionality allows policies to govern the actions of standalone sensors or similar components.

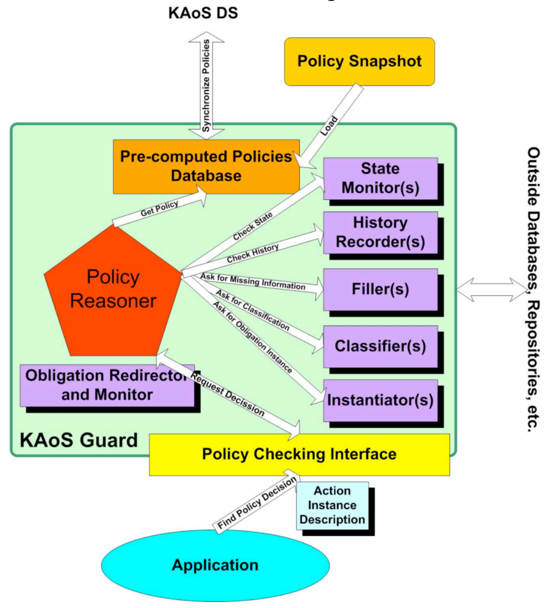


Figure 4: Architecture of the KAoS Guard

Guards not only receive information about policies, but also about the state of the system and the entities being managed. Guards do not by themselves provide monitoring functions, but they do provide interfaces to plug in outside monitors or databases providing access to external state or event-related information (Fig 4).

The Policy Checking Interface provides a set of methods that allow a given action instance to be checked for:

- **Authorization**. If an action is not authorized, an exception is thrown with information about the policy that prevented it. In some secure applications, however, it would not be desirable to release information about the cause of the policy exception.
- **Obligations**. A list of obligations for a given actor is returned, sorted in rank order of importance. In addition, if there are obligations for other actors that are triggered by an external event, then KAoS will try to locate them and forward the obligations to them.
- **Configuration options**. If a partial description of the action is sent to KAoS, a range of allowed values for properties of a given action is returned. For instance, if an application were to query the guard about a planned radio transmission, information about the maximum power and range of frequencies allowed to be used in the given geographical area would be returned to it. Disclosure policies would be used to filter out unauthorized information in the results.

The data structure exchanged between an application and KAoS in these methods is an Action Instance Description (AID), easily constructed with a Java class provided by KAoS. For those not wanting to use Java for this purpose, KAoS also accepts a string with an OWL definition of an AID. Applications that check authorization policies must be able to create AIDs, and applications that handle obligations must be able to interpret them when received. AIDs can contain complex values in the form of additional structures.

OWL requires references to the URLs of ontology concepts defining actions. For this purpose, KAoS provides a simple tool to create Java constants for ontological concepts. The values of these constants are URLs. This approach allows URLs to be easily referenced and used in application code.

In order to cope with the OWL open-world assumption,[4] KAoS uses intersection to define classes of policy controlled actions[5] for deterministic policy decisions. When the policy checking method is executed, it traverses the policy database in priority order and checks to see whether the AID is in the range of actions controlled by any policy—the range of actions is derived from an analysis of the policy's controlled action class. To make this determination, each attribute of the AID's action must be checked to see whether its value is in the range of its corresponding attribute in the policy control action class. The role-value map relations, defining aspects of policy context, are checked as well. This is just a brief sketch of the full guard policy-checking algorithm which, because it does not require the power of a complete OWL reasoner, executes in a few milliseconds.

In order to support the semantics of complex application-specific policies, guards accommodate a variety of extensions. These can be activated on demand, as specified in each guard's configuration information. Examples include:

- **Obligation Policies**
  Obligation Action Instantiator: helps customize the creation of obligation actions.
  Obligation Monitor: monitors fulfillment of obligations and notifies responsible parties about perceived violations.

[4]http://en.wikipedia.org/wiki/Open_World_Assumption
[5] http://ontology.ihmc.us/Policy

Obligation Redirector: delegates obligations to some other actor.

- **History and State**
  History Monitor: tracks the history of specific actions and verifies whether a given history is present.
  State Monitor: a sensor that provides information about some aspect of the environment or situation.
- **Spatial Reasoner**: provides spatial reasoning support to the guard.

8. Application Case Studies

Below is a brief summary of some current KAoS applications illustrating the comprehensiveness and maturity of the system.

The **AFRL Information Management System**[6] is a large-scale highly-distributed publish-subscribe system in which KAoS has been integrated to deal with security and increasingly challenging quality-of-service policies (http://ontology.ihmc.us/Raytheon/index.html). In a related project, we are implementing a KAoS-enabled infosphere federation service that includes support for policy negotiation among federation partners. We have also built a prototype for the support of communities of interest across their life cycle (http://ontology.ihmc.us/COI/index.html).[7]

In a set of ARL-supported projects supporting the goal of **Cross-Domain Information Exchange (CDIX)**, we have developed a system that represents and reasons about domain-specific policies to help recognize what otherwise sensitive documents a soldier is allowed to receive given the current mission context. The system also relies on policies to help recognize when appropriate human approval can be obtained or a specific transformation of the information can be performed to allow the information to be released [13].

As part of a series of ONR-sponsored studies of **Human–Agent-Robot Teamwork**, we have exercised KAoS in an application involving a variety of application domains and enforcement at different levels of control, from low level network resource control to high level organizational constraints, spatial reasoning, and coordination management. A recent phase of the study culminated in an outdoor field exercise that required policy-based coordination in real time of mixed subteams, involving a combination of two people and five robots communicating over wireless 802.11b, while performing a hide-and-seek style search and apprehension of an intruder on a Navy pier [12].

In partnership with Raytheon, we have developed prototypes of **Radio Spectrum and Transmission Control** applications, where settings governing power and other configuration parameters of the transmission in given geographical areas are approximated from reasoning based on spatial knowledge and the current type and usage of a radio (http://ontology.ihmc.us/Raytheon/index.html).

[6] http://www.infospherics.org/
[7] http://ontology.ihmc.us/coi

9. Conclusions and Future Work

In this article we have described how the architecture and capabilities of the KAoS policy services framework have evolved, inspired by both technological advances and the practical needs of its varied applications. Areas of future research include: automated policy refinement, visualization of policy relations and applicability, enhancement of probabilistic techniques for automated policy adjustment (adjustable autonomy), uncertainties in information used in policy checking, and the handling of time—as in, for example, performing progress appraisal.

Acknowledgments

We are grateful for the contributions of Niranjan Suri, Marco Carvalho, Pat Hayes, Tom Eskridge and Ken Ford. The authors gratefully acknowledge the sponsorship by Air Force Research Lab, Army Research Lab, Raytheon, and the Office of Naval Research.

References

[1] Damianou, N., Dulay, N., Lupu, E. C., & Sloman, M. S. (2000). Ponder: A Language for Specifying Security and Management Policies for Distributed Systems. Version 2.3, Imperial College.

[2] Dini, P., Clemm A., Gray, T., Fuchun, J., Logrippo, L., and Reiff-Marganiec, S. (2004). Policy-enabled mechanisms for feature interactions: reality, expectations and challenges. *Computer Networks*, Vol. 45,

[3] Elenius, D., Denker, G., Stehr, M., Senanayake, R., Talcott, C. and Wilkins, D. (2007). CoRaL--Policy Language and Reasoning Techniques for Spectrum Policies. *Proc. 8th Workshop on Policies*.

[4] Kagal, L., Finin, T. and Joshi, A. (2003). A Policy Language for a Pervasive Computing Environment. *Proc. 4th Workshop on Policies*.

[5] Kolovski, V., Parsia, B., Katz, Y., Hendler, J. (2005). Representing Web Service Policies in OWL-DL. *Proc. of the Semantic Web Conference*, LNCS 3729.

[6] Nejdl, W., Olmedilla, D., Winslett, M. and Zhang, C. (2005). Ontology-Based Policy Specification and Management. *Proc. of 2nd European Semantic Web Conference*, LNCS 3532.

[7] Yague, M. (2006). Survey on XML-Based Policy Languages for Open Environments. *Journal of Info. Assurance and Security*, Vol. 1.

[8] Tonti, G., Bradshaw, J., Jeffers, R., Montanari, R., Suri, N. and Uszok, A. (2003). Semantic Web Languages for Policy Representation and Reasoning: A Comparison of KAoS, Rei, and Ponder. In D. Fensel, K. Sycara & J. Mylopoulos (Eds.), *The Semantic Web—ISWC 2003*, LNCS 2870. Berlin, Germany: Springer, pp. 419-437.

[9] Uszok, A., Bradshaw, J., Jeffers, R., Suri, N., Hayes, P., Breedy, M., Bunch, L., Johnson, M., Kulkarni, S. and Lott, J. (2003). KAoS Policy and Domain Services: Toward a Description-Logic Approach to Policy Representation, Deconfliction and Enforcement. *Proceedings of the IEEE Fourth International Workshop on Policy (Policy 2003).* Los Alamitos, CA: IEEE Computer Society, pp. 93-98.

[10] Uszok, A., Bradshaw, J., Jeffers, R., Johnson, M., Tate A., Dalton, J., Aitken, S. (2004). KAoS Policy Management for Semantic Web Services. *IEEE Intelligent Systems*, July/August, 19(4), pp. 32-41.

[11] Johnson, M., Bradshaw, J. M., Jung, H., Suri, N. & Carvalho, M. (2008). Policy management across multiple platforms and application domains. Proc. of the 2008 IEEE Conference on Policy.

[12] Bradshaw, J. M., Feltovich, P. J., Johnson, M., Bunch, L., Breedy, M., Jung, H., Lott, J. & Uszok, A. (2008). Coordination in human-agent-robot teamwork. Proceedings of the CTS 2008, Special Session on Collaborative Robots and Human Robot Interaction, Irvine, CA.

[13] Bunch, L., Bradshaw, J. M. & Young, C. O. (2008). Policy-governed information exchange in a US Army operational scenario. Demonstration track. 2008 IEEE Conference on Policy, Palisades, NY, 2-4 June.