KAoS Policy and Domain Services: Toward a Description-Logic Approach to Policy Representation, Deconfliction, and Enforcement

A. Uszok, J. Bradshaw, R. Jeffers, N. Suri, P. Hayes, M. Breedy, L. Bunch, M. Johnson, S. Kulkarni, J. Lott
Institute for Human and Machine Cognition (IHMC), Univ. West Florida, 40 S. Alcaniz, Pensacola, FL 32501
{auszok, jbradshaw, rjeffers, nsuri, phayes, mbreedy, lbunch, mjohnson, skulkarni, jlott}@ai.uwf.edu

Abstract

In this paper, we describe our initial implementation of the KAoS policy and domain services. While initially oriented to the dynamic and complex requirements of software agent applications, the services are also being adapted to general-purpose grid computing and web services environments as well. The KAoS services rely on a DAML description-logic-based ontology of the computational environment, application context, and the policies themselves that enables runtime extensibility and adaptability of the system, as well as the ability to analyze policies relating to entities described at different levels of abstraction. An online theorem-prover is used for policy disclosure, conflict detection, and harmonization, and for reasoning about domain structure and concepts. In future versions, DAML will be replaced by OWL, and description-logic will be supplemented with the possibility of rule-based representation.

Keywords: *policy, agent, ontology, DAML, domains, KAoS, description logic, policy conflict resolution.*

1. Introduction

The increased intelligence afforded by software agents is both a boon and a danger. By their ability to operate independently without constant human supervision, they 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 in the case of buggy or malicious agents. Techniques and tools must be developed to assure that agents will always operate within the bounds of established behavioral constraints and will be continually responsive to human control [3, 7]. Moreover, the policies that regulate the behavior of agents should be continually adjusted so as to maximize their effectiveness in both human and computational environments [6].

Under DARPA and NASA sponsorship, we have been developing the KAoS [4, 5, 7] policy and domain services to increase the assurance with which agents can be deployed in a wide variety of operational settings. In conjunction with Nomads [18, 19] strong mobility and safe execution features, KAoS services and tools allow for the specification, management, conflict resolution, and enforcement of policies within the specific contexts established by complex organizational structures. While initially oriented to the dynamic and complex requirements of software agent applications, the 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.

Following a discussion of the background and motivation for our approach (section 2), we will provide an overview of the KAoS Policy Ontologies (KPO), which represent policies, the relevant computational environment, and application context declaratively using the DARPA Agent Markup Language (DAML) (section 3). Then we present our approach to policy conflict detection and resolution (section 4). As a next step, we show details of the current implementation of the Policy and Domain Services in KAoS including policy distribution, disclosure, and enforcement mechanisms (section 5). Finally, we describe current applications of KAoS and Nomads (section 6) along with its limitations and plans for future work (section 7).

2. Background and motivation

Interest in policy-based management approaches has grown considerably in popularity over the last years (*http://www.policy-workshop.org*). The scope of our policy-based agent management approach includes the typical security concerns such as authorization, encryption, access and resource control policies, but also goes beyond these in significant ways. For example, KAoS pioneered the concept of agent conversation

policies [4, 5, 10]. In addition to conversation policies, we are in the process of developing representations and enforcement mechanisms for mobility policies [15], domain registration policies, and various forms of obligation policies (see below).

There are some important differences, however, between the objectives of our approach and that of other more typical approaches. First, the approach does not assume that we are dealing with a homogeneous, static set of components. Our approach seeks to enable policy uniformity in domains that might be simultaneously distributed across multiple platforms and execution environments, as long as semantically equivalent monitoring and enforcement mechanisms are available. Second, insofar as possible the framework needs to support dynamic runtime policy changes, and not merely static configurations determined in advance. Third, the framework needs to be extensible to a variety of execution platforms with different enforcement mechanisms—initially Java and Aroma [18, 19]—but in principle any platform for which policy enforcement mechanisms may be written. Fourth, the framework must be robust and adaptable in continuing to manage and enforce policy in the face of attack or failure of any combination of components. Finally, we recognize the need for easy-to-use policy-based administration tools capable of containing domain knowledge and conceptual abstractions that let application designers focus their attention more on high-level policy intent than on implementation details. Such tools require sophisticated graphical user interfaces for monitoring, visualizing, and dynamically modifying policies at runtime.

In short, the policy management framework must ensure maximum freedom and heterogeneity of the components and non-intrusiveness of the enforcement mechanisms, while respecting the bounds of human-determined constraints designed to ensure selective conformity of behavior.

3. KAoS policy ontologies

The representation chosen to describe the policies and their context largely determines the flexibility, extensibility, and amenability to analysis of a given implementation. KAoS services rely on a DAML description-logic-based ontology of the computational environment, application context, and the policies themselves that enables runtime extensibility and adaptability of the system, as well as the ability to analyze policies relating to entities described at different levels of abstraction. The representation facilitates careful reasoning about policy disclosure, conflict detection, and harmonization, and about domain structure and concepts.

3.1. Short overview of DAML

The KAoS Policy Ontologies (KPO) are expressed in DAML (http://www.daml.org). Designed to support the emerging "Semantic Web," DAML is the latest in a succession of Web markup languages [2]. HTML, the first Web markup language, allowed users to markup documents with a fixed set of formatting tags for human use and readability. XML allows users to add arbitrary structures to their documents but expresses very little directly about what the structures mean. RDF (Resource Description Format) encodes meaning in sets of subject-verb-object triples, where elements of these triples may each be identified by a URI (typically a URL). OWL (Ontology Web Language), a W3C-approved evolution of DAML, is nearing final release (http://www.w3.org/2001/sw/).

DAML extends RDF to allow users to specify ontologies composed of taxonomies of classes and properties, as well inference rules. These ontologies can be used by people for a variety of purposes, such as enabling more accurate or complex Web searches. Agents can also use semantic markup languages to understand and manipulate Web content in significant ways; to discover, communicate, and cooperate with other agents and services; or, as we outline in this paper, to interact with policy-based management services and control mechanisms.

3.2. Core ontologies for policy definition

The current version of KPO defines basic ontologies for actions, actors, groups, places, various entities related to actions (e.g., computing resources), and policies. There are currently 79 classes and 41 properties defined in the basic ontologies. It is expected that for a given application, the ontologies will be further extended with additional classes, individuals, and rules.

The actor ontology distinguishes between people and various classes of software agents or components that can be the subject of policy. Most agents can only perform ordinary actions, however various agents that are part of the infrastructure as well as authorized human user may variously be permitted or obligated to perform certain policy actions, such as policy approval and enforcement. Groups of actors or other entities may be distinguished according to whether the set of members is defined extensionally (i.e., through explicit enumeration in some kind of registry) or intentionally (i.e., by virtue of some common property such as a joint goal that all actors possess or a given place where various entities may be currently located).

3.3. Representing policies in DAML

A policy is a statement enabling or constraining execution of some type of action by one or more actors in relation to various aspects of some situation. Our current policy ontology distinguishes 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 serve to waive such a requirement) [8].

enforcement. The most imported property value is, however, the name of a controlled action class. Usually, a new action class is built automatically when a policy is defined. Through various property restrictions, a given policy can be variously scoped, for example, either to individual agents, to agents of a given class, to agents belonging to a particular group, or to agents running in a given physical place or computational environment (e.g., host, VM). Additionally, action context can be precisely described by restricting values of its properties. Figure 1

```
<?xml version="1.0" ?>
<!DOCTYPE P1 [
    <!ENTITY policy  "http://ontology.coginst.uwf.edu/Policy.daml#" >
    <!ENTITY action  "http://ontology.coginst.uwf.edu/Action.daml#" >
    <!ENTITY domains  "http://ontology.coginst.uwf.edu/ExamplePolicy/Domains.daml#" >
]>
<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:policy="http://ontology.coginst.uwf.edu/Policy.daml#"
>
<daml:Ontology rdf:about="">
    <daml:versionInfo>$ http://ontology.coginst.uwf.edu/ExamplePolicy/P1.daml $</daml:versionInfo>
    <daml:imports rdf:resource="http://www.daml.org/2001/03/daml+oil" />
    <daml:imports rdf:resource="http://ontology.coginst.uwf.edu/Policy.daml" />
    <daml:imports rdf:resource="http://ontology.coginst.uwf.edu/Action.daml" />
    <daml:imports rdf:resource=" http://ontology.coginst.uwf.edu/ExamplePolicy/Domains.daml" />
</daml:Ontology>

<daml:Class rdf:ID="P1Action">
    <rdfs:subClassOf rdf:resource="&action;NonEncryptedCommunicationAction" />
    <rdfs:subClassOf>
        <daml:Restriction>
            <daml:onProperty rdf:resource="&action;#performedBy" />
            <daml:toClass rdf:resource="&domains;MembersOfDomainArabello-HQ" />
        </daml:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <daml:Restriction>
            <daml:onProperty rdf:resource="&action;#hasDestination" />
            <daml:toClass rdf:resource="&domains;notMembersOfDomainArabello-HQ" />
        </daml:Restriction>
    </rdfs:subClassOf>
</daml:Class>

<policy:NegAuthorizationPolicy rdf:ID="P1">
    <policy:controls rdf:resource="#P1Action" />
    <policy:hasSiteOfEnforcement rdf:resource="&policy;ActorSite" />
    <policy:hasPriority>1</policy:hasPriority>
    <policy:hasUpdateTimeStamp>446744445544</policy:hasUpdateTimeStamp>
</policy:NegAuthorizationPolicy>
```

Figure 1. Example policy in DAML

In DAML, a policy is represented as an instance of the appropriate policy type with associated values for properties: priority, update time stamp and a site of

shows an example of the policy written in DAML stating that the members of some domain called Arabello-HQ are forbidden to communicate with the outside of this

domain using unencrypted communication. The syntax of this example may seem very complex, however, the DAML policy is not meant to be written or analyzed by a human but by the computer. A graphical interface hides the complexity of this representation from the user.

4. Policy conflict resolution

The KAoS Policy Ontologies are intended for a variety of purposes. One obvious application is during inference relating to various forms of online or offline analysis. They can be used for a variety of purposes, including policy disclosure management, reasoning about future actions based on knowledge of policies in force, and in assisting users of policy specification tools to understand the implications of defining new policies given the current context and the set of policies already in force.

In the current version of KAoS, changes or additions to policies in force or a change in status of an actor (e.g., a human administrator being given new permissions; software agent joining a new domain or moving to a new host) requires logical inference to determine first of all which policies are in conflict and second how to resolve these conflicts [16]. We have implemented a general-purpose algorithm for policy conflict detection and harmonization whose initial results promise a high degree of efficiency and scalability.

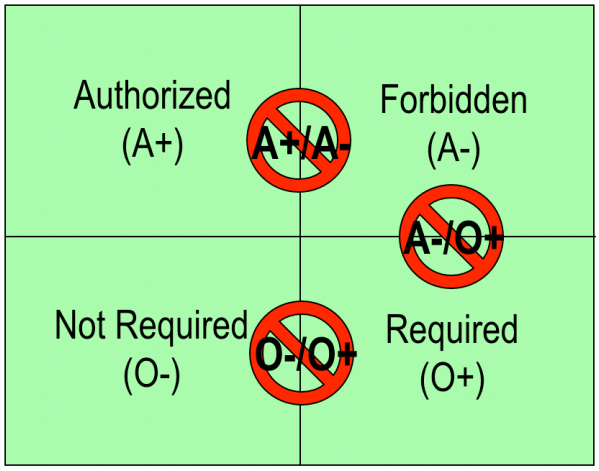

Figure 2. Types of policy conflicts

Figure 2 shows the three types of conflict that can currently be handled: positive vs. negative authorization (i.e., being simultaneously permitted and forbidden from performing some action), positive vs. negative obligation (i.e., being both required and not required to perform some action), and positive obligation vs. negative authorization (i.e., being required to perform a forbidden action). We have developed policy deconfliction and harmonization algorithms within KAoS to allow policy conflicts to be detected and resolved even when the actors, actions, or targets of the policies are specified at very different levels of abstraction. These algorithms rely in part on a version of Stanford's Java Theorem Prover (JTP; 12) that we have integrated with KAoS.

4.1. Policy precedence conditions

Policy precedence conditions are needed to properly execute the automatic conflict resolution algorithm. When policy conflicts occur, these conditions are used to determine which of the two policies being compared is most important. The conflict can then be resolved automatically in favor of the most important policy. Alternatively, the conflicts can be brought to the attention of a human administrator who can make the decision manually.

We currently rely exclusively on the combination of numeric policy priorities[1] and update times to determine precedence—the larger the integer and the more recent the update the greater the priority. This is consistent with the natural intuition that more recent and higher priority policies should trump older and lower priority ones.

In the future we intend to allow people additional flexibility in designing the nature and scope of precedence conditions. For example, it would be possible to define precedence based on the relative authorities of the individual who defined or imposed the policies in conflict, which policy was defined first, which has the largest or smallest scope, whether negative or positive authorization trumps by default, whether subdomains takes precedence over superdomains or vice versa, and so forth.

4.2. Steps in policy conflict resolution.

The steps in order to resolve policy conflicts in the set of policies are as follow:

1. DAML policy ontologies are loaded into JTP along with the set of DAML policies to be deconflicted.
2. A list of all policies in constructed and sorted according to user-defined criteria for policy precedence.
3. For each policy in the sorted list, iterate through all the elements with a lower priority and check to see if there is a policy conflict. A policy conflict occurs if the two policies are instances of conflicting types and if a subsumption

[1] In the absence of an explicitly rated priority, the priority value of a policy may be "inherited" from the person who defined the policy or the priority of the controlled action class.

algorithm determines that the action classes that the two policies control are not disjoint.

4. The lower priority policy from the conflicting pair of policies is removed from the list and the policy harmonization algorithm is invoked. It attempts to modify the policy with the lower precedence to the minimum degree necessary to resolve the conflict (if the policies are of equal precedence, a user may be required to specify which policy will take precedence). The harmonization algorithm may generate zero, one or several new policies to replace the removed policy.
5. The newly constructed harmonized policies inherit the precedence and the time of last update from the removed policy, and a pointer to the original policy is maintained so that it can be recovered if necessary as policies continue to be added or deleted in the future.

4.3. Details of policy harmonization

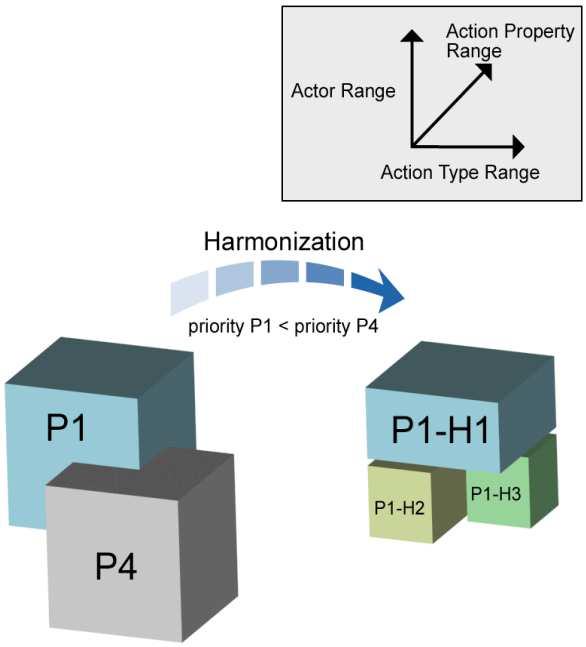

Figure 3. Graphical representation of policy harmonization

The derivation of the newly-generated set of harmonized policies can be understood by imagining an intersection of two N-dimensional Cartesian products:
If *P1* and *P4* are two Cartesian products defined as:

$$P1 = D11 \times D12 \times .... \times D1n$$
$$P4 = D21 \times D22 \times .... \times D2n$$

then

$$P1 \backslash P2 = subP1 + subP2 + ... + subPn$$

where

$$subPk = (D11 \cap D21) \times ... \times (D1(k-1) \cap D2(k-1)) \times (D1k \backslash D2k) \times D1(k+1) \times .. \times D1n$$

Figure 3 shows a 3-D graphical representation of policy harmonization. Mapping the mathematical definition above to the generation of harmonized policies we get the following:

1. The first harmonized policy has a range of actors that corresponds to the difference between the ranges of the two original policies and a controlled action and range of values on the action properties that correspond to those of the lower-precedence policy.
2. The second harmonized policy has a range of actors that corresponds to the intersection of the ranges of the two original policies, a controlled action that corresponds to the differences between those of the two policies, and a range of values on the action properties that correspond to that of the lower-precedence policy.
3. Additional harmonized policies are built to correspond to each action properties in the two original policies. The range of actors corresponds to the intersection of the ranges of the two original policies and the controlled action corresponds to the intersection between those of the two policies.

The results of computing any of the above policies may be empty.

5. KAoS Policy and Domain Services

KAoS is a collection of componentized services compatible with several popular agent frameworks, including Nomads [18, 19], the DARPA CoABS Grid [14], the DARPA ALP/Ultra*Log Cougaar framework (http://www.cougaar.net) and CORBA (http://www.omg.org). While initially oriented to the dynamic and complex requirements of software agent applications, the 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. The adaptability of KAoS is due in large part to its pluggable infrastructure based on Sun's Java Agent Services (JAS) (http://www.java-agent.org). For a full description of KAoS, the reader is referred to [4; 5; 6; 7].

Groups of people and computational entities are logically structured into domains and subdomains to facilitate policy administration. Domains may represent any sort of group imaginable, from potentially complex organizational structures to administrative units to

dynamic task-oriented teams with continually changing membership. Membership in a given domain can extend across host boundaries and, conversely, multiple domains can exist concurrently on the same host. Domains may be nested indefinitely and, depending on whether policy allows, membership in more than one domain at a time is possible.

KAoS domain services are not fully described in this paper, however the following subsections describe KAoS policy services in more detail.

5.1. Initialization and Querying

During the bootstrapping process, the core policy ontologies (KPO) are loaded into JTP. Afterwards, additional ontologies specific to a particular environment or application can be loaded dynamically from the KAoS Policy Administration Tool (KPAT; see section 5.2). In addition to a base set of policies, these additional ontologies usually include descriptions of domain structures and the entities associated with them.

The policy service is configured to handle a variety of queries at different stages of the policy management process. Some of the more common ones are listed below.

Defining a new policy (or editing an existing one):

1. Obtain all the action classes that can be performed by either a given class of actors or a given actor instance.
2. Obtain all the properties of a given action class.
3. Obtain the range of targets for the property applicable to the given action class.
4. Obtain all the known subclasses of the Target class within a given range.
5. Obtain all the known instances of the Target class within a given range.

Detecting policy conflicts:

6. Check if two subclasses of the action class controlled by two selected polices are disjoint.
7. Check if the subclass of the action class controlled by the policy with the lower priority is a subclass of the subclasses of the action class controlled by the policy with the higher priority.

Harmonizing policies:

8. Obtain the explicit superclasses of the action classes controlled by the policies.
9. Obtain the range of a property in the action classes controlled by the policies.
10. Check if the property in one controlled action class is a subproperty of some properties in the second controlled action class.

Policies distribution and disclosure:

11. Obtain all the policies whose controlled actions can be performed by either a given class of actors or a given actor instance.
12. Check if the given action instance is an instance of some action class controlled by existing policies.

5.2. KAoS Policy Administration Tool (KPAT)

The KAoS Policy Administration Tool (KPAT[2]) implements a graphical user interface to domain and policy management functionality of KAoS. It has been developed to make policy specification, revision, and application easier for administrators without extensive training (Figure 4).

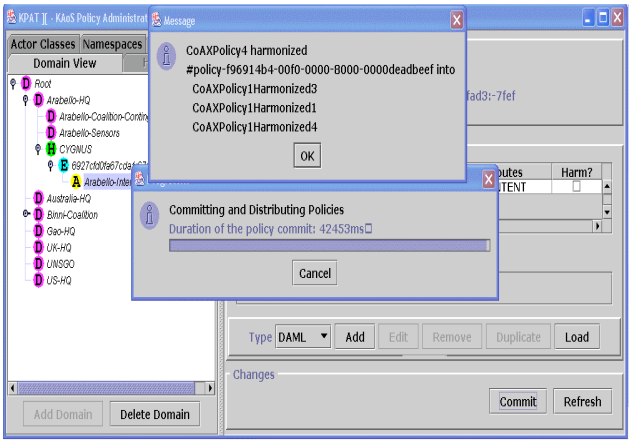

Figure 4. KPAT notifies the user of the results of policy harmonization

KPAT can be used to browse and load ontologies, to define, deconflict, and commit new policies, and to modify or delete old ones. The generic DAML Policy Editor (Figure 5) is driven by the ontologies loaded into JTP and always provides the user with the list of choices narrowed to the current context using the queries enumerated in the previous subsection. Custom editors tailored to particular kinds of policies may also be added to KPAT and triggered if a policy about the action class associated with the given custom editor is selected.

When a user commits a change to an ontology (e.g., a new or edited policy, changes to domain structure) the Jena framework [11] is used to dynamically build DAML representation based on the values selected by the user.

[2] Pronounced KAY-pat.

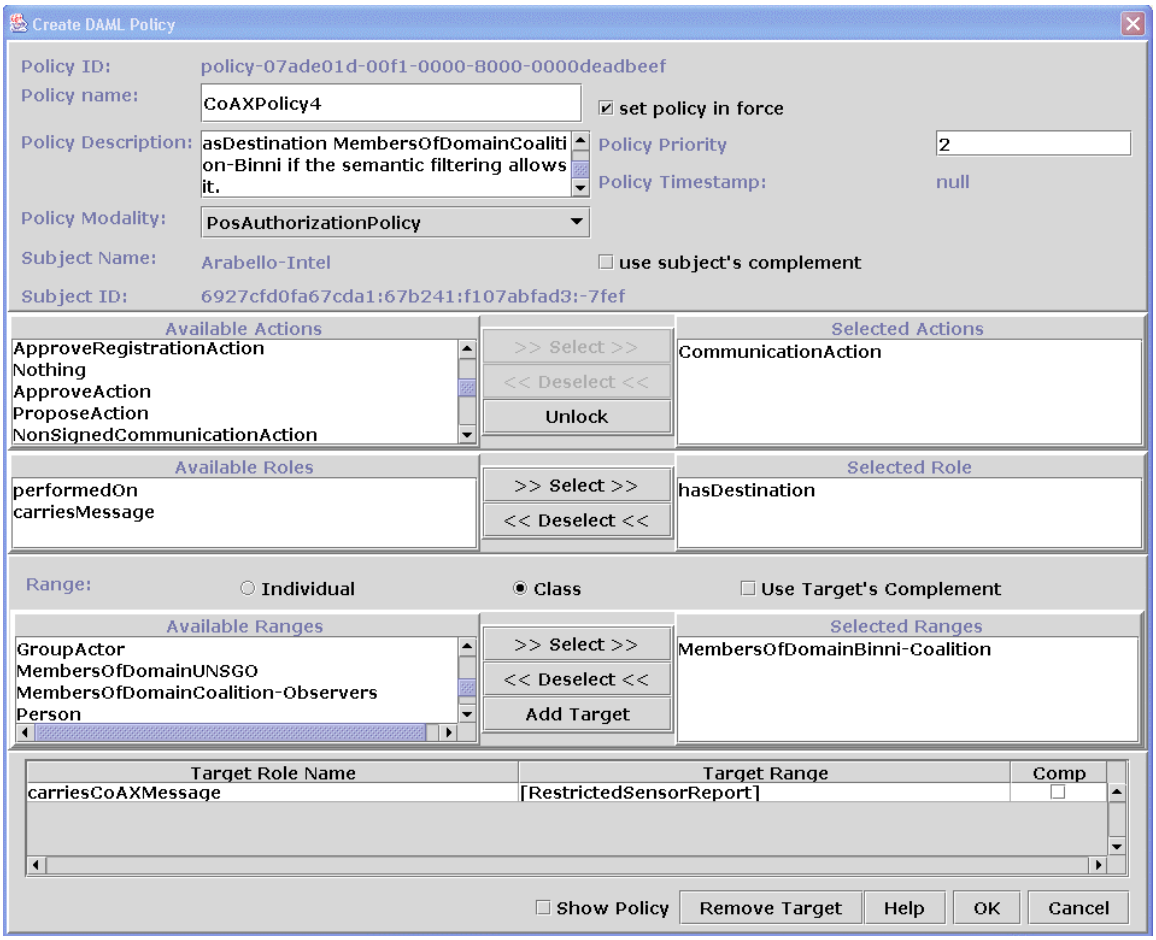

Figure 5. Generic DAML Policy Editor. Custom editors tailored to specific classes of policy can also be easily developed.

5.3. Policy distribution and disclosure

Figure 6 shows the major components of the KAoS policy and domain services architecture. KAoS Domain Managers (DM) act in the role of policy decision points to determine whether agents can join their domain and for policy conflict resolution. The DM is responsible for ensuring policy consistency at all levels of a domain hierarchy, for notifying Guards about changes in policy or other aspects of system state that may affect their operation, and for storing state in the directory service.

Because DM's are stateless, one DM instance may serve multiple domains or conversely, a single large domain may require several instances of the DM to achieve scalable performance.

Policies are stored within ontologies in the directory service (DS). Although DM's normally provide the limited public interface to the DS, private interfaces may allow the DS to be accessed by other authorized entities in accordance with policies on disclosure.[3] For example, trusted components could be allowed to perform queries concerning domain policies in advance of submitting a registration request to a new domain. Because the policies in the directory service are expressed declaratively, some forms of analysis and verification can be performed in advance and offline, permitting execution mechanisms to be as efficient as possible.

Guards are responsible for policy enforcement within the bounds of a specified computational environment. They interpret policies that have been approved by the DM and enforce them with appropriate native enforcement mechanisms. While KPAT and the DM, and the Guards are intended to work identically across different agent platforms (e.g., DARPA CoABS Grid, Cougaar) and execution environments (e.g., Java VM, Aroma VM), enforcement mechanisms are necessarily designed for a specific platform and execution environment.

[3] We are investigating the work of Seamons [17] on incremental policy disclosure strategies for future use in KAoS.

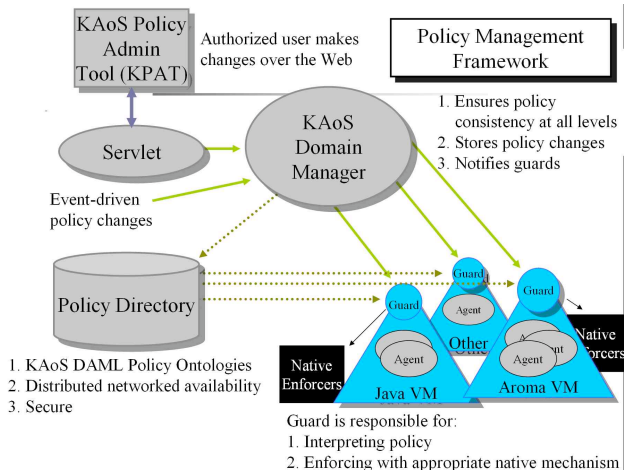

Figure 6. KAoS Policy Service architecture

Whereas KAoS can currently represent and deconflict both authorization and obligation policies, the current implementation only supports enforcement for authorization policies. In the near future, we will extend KAoS policy services with support for obligation policies through the use of monitors (monitoring compliance with obligations) and enablers (facilitating the performance of selected obligations).

5.3.1 Enforcers Enforcers are the mechanism by which Guards ensure compliance with authorization policies. The enforcer cannot be made fully generic, as it is inherently specific to the type or types of actions being enforced by it and to the environment in which these actions are being performed. What can be made generic however is the interface to the system answering the question, "is a given action authorized or not?" By defining standard interfaces for answering this question, we aim to help enable developers to concentrate on the application-specific portions of the enforcers:

- intercepting the action being attempted by the controlled actors;
- creating an object with a description of the action and calling the method with it asking for authorization on the generic part of the enforcer;
- blocking or allowing the action based on the obtained result.

All of the capabilities of the policy services are made fully available to the developer

In order to implement a new enforcer the programmer extends the provided class *GenericEnforcer* and implement its abstract methods:

- *getControlledActionClasses* - provides names of the action classes on which policies can be enforced.
- *initEnforcer* - initializes the specific enforcer capabilities by inserting the triggering or monitoring functionality into the computing environment (e.g., installing filters or resource usage monitors).

The capability for interception action inserted by the *initEnforcer* method uses the method *isActionAuthorized*, which is provided by the *GenericEnforcer* superclass. The method takes as the parameter a description of the action, which includes:

- unique ID of the agent performing the action;
- names of the Action classes describing the type of the intercepted action instance consistent with the ontologies;
- list of action property descriptions comprising the context of the action.

Each action property description requires the following information:

- name of the property, consistent with the ontology;
- indication of the form of the target description, which can be either the unique ID of the target (*String*), the *DAMLModel* (from Jena [11]) with target instance description, or opaque application-specific data containing the instance description;
- actual data containing the target description in one of the forms indicated above.

It is the responsibility of the developer to understand the definition of the action class(es) in the ontology that the enforcer is being designed to enforce. An action class can be represented as a list of properties whose names are in the ontology. By referencing a property using this name the developer can determine the value of the property. We have developed a mapping tool, which analyzes a given ontology and generates the ontology names as Java constants.

5.3.2 Enforcer repository Enforcer class names are recorded in the Enforcer Registry. The registry associates an enforcer class with the names of the action classes on which it can enforce policy.

The registry is either stored in a local Jar file or can be made available on the network. When a particular enforcer is needed, a lookup is performed on the registry. If an enforcer is found, it can then be created through the Java Reflection mechanism.

Guards are responsible for creating instances of the enforcers they need to control the actions specified by policies it receives from Domain Managers. Of course this presumes that an appropriate enforcer is available in the registry. The use of a registry allow for a dynamic extension of the system. Even when agents are running it is possibly to dynamically insert a new enforcer class into a network-based Enforcer Registry so that new

policies about a class of actions controlled by this enforcer can be defined. The list of actions shown in KPAT is limited to these for which there are enforcers in the Enforcer Registry. In this way the system can prevent attempts to put a policy in force for which there is no available enforcer.

5.3.3. Authorization mechanism As mentioned previously, the enforcer is concerned with getting the answer to the question, "is a given action authorized or not?" It is not so much concerned with how this answer is obtained. In the current KAoS implementation, it is possible to answer the authorization question in one of three ways: through the use of the Directory Service, some intermediary module, or the enforcer itself.

For performance reasons, it is preferable to answer the authorization question locally in the enforcer or an intermediary module whenever possible. In this way enforcement can also continue even if the connection to the Directory Service is interrupted. In such cases, the worst that could happen is that the DS will be temporarily unable to notify enforcers about policy-relevant changes to the ontology.

Every actor in the system is associated with a Guard that is responsible for the computational environment in which it is running. When the guard receives a new policy, it obtains the types of actions controlled by this policy and checks to see if it already has an enforcer for these action types. If not, it will consult the Enforcer Registry and create the enforcer. When the appropriate enforcer is found, the policy is pushed to it. The generic enforcer's policy storage object appropriately adds, removes or updates policies in its storage.

These policies are used by the *isActionAuthorized* method. When the method is executed, it traverses the enforcer policy storage and checks to see if the given action instance is in the range of actions controlled by any policy (the range of actions is defined by the action class associated with a policy). This can be checked by investigating all of the action properties. Each property has to be checked to assure that the value assigned to this property in the action instance description is in the range of this property defined in the action instance description by the appropriate restriction class. The enforcer can get different forms of the description of the instance.

When the DS already knows about the instance, its unique id can be provided. This is the case of any concepts defined in the preloaded ontologies or explicitly registered in the directory (e.g., actors , domains). In this situation, the enforcer has either already cached the set of instances from its range (which it obtained from the DS), or can obtain this set directly and then easily check to see if the instance is within the set. There are ways to tailor the performance of this mechanism, depending whether this set is dynamic or static.

Sometimes the instance is new. For example, it may have just been created in the context of the action, as is often the case in message content filtering, where the content can be very diverse and the description complex. In these cases, the description of the target instance has to be built dynamically, and two options of passing it to the authorization mechanism exist. We can use either *DAMLModel* object (from Jena) if it is possible to analyze the value of the property or if it is already encoded in DAML, or else we can use opaque application-specific data containing the instance description.

5.3.4. Semantic matching Semantic matching is performed by the authorization mechanism by its looking into the Semantic Matcher Repository to determine if there is a matcher associated with the given action property name. Each matcher in the repository implements the same interface containing methods to:

- initialize the specific semantic matcher for instance loading necessary ontologies, and so forth;
- check to see if the instance from the provided description is of the type indicated by the name of the provided ontology class.

5.3.5. Handling partial action description Situations sometimes arise when an enforcer is able to provide only partial information about the intercepted action. In such cases, the following approach is applied. If the action has a given property and the description of the action is missing the value for this property then the algorithm assumes that it matches the property range in the case of negative authorization policies or does not match in the case of positive authorization policies.

The rationale for doing this is that, in the case of negative authorization, since we do not know the value we cannot exclude the possibility that it was in the given range. To be on the safe side we assume, therefore, that it was in the range. Similarly, in the case of positive authorization, since we cannot exclude the possibility that the value was out of range, we assume that it was indeed out of range. Further experience will tell us whether these assumptions should be made to hold in all applications.

Additionally, the algorithm can investigate subproperties relations. For instance, an action description may lack the value for some property *P*. If, however, a value is defined for some subproperty or super-property of the property *P* then this value maybe taken into account. This aspect requires further study and experimentation.

5.3.6 Default authorization When the authorization mechanism does not find any policy applicable to the provided action instance, it still must answer the authorization question. Hence, the need for a default authorization modality for enforcers that either permits all actions not explicitly forbidden or else forbids all actions not explicitly permitted.

The current way of configuring the default authorization modality is on a per domain basis. Each domain can have one default authorization: negative or positive. In order to resolve potential conflicts, when an agent is registered in more than one domain, a domain can have an associated priority. When a guard registers itself in various domains, it receives the default authorizations from them. If they are in conflict, the guard selects the default coming from the domain with the highest priority. Then it passes this default to its enforcers.

We will investigate more flexible means of configuring default authorization modalities once more sophisticated precedence mechanisms that go beyond current numeric priority schemes are in place.

5.4. Policy enforcement mechanisms

In applications to date, we have relied on several different kinds of enforcement mechanisms. Enforcement mechanisms built into the execution environment (e.g., OS or Virtual Machine level protection) are the most powerful sort, as they can generally be used to assure policy compliance for any agent or program running in that environment, regardless of how that agent or program was written. For example, the Java Authentication and Authorization Service (JAAS) provide methods that tie access control to authentication. In KAoS, we have in the past developed methods based on JAAS that allow policies to be scoped to individual agent instances rather than just to Java classes. Currently, JAAS can be used with Java VMs; in the future it should be possible to use JAAS with the Aroma VM as well. As described above, the Aroma VM provides, in addition to Java VM protections, a comprehensive set of resource controls for CPU, disk and network. The resource control mechanisms allow limits to be placed on both the rate and the quantity of resources used by Java threads. Guards running on the Aroma VM can use the resource control mechanisms to provide enhanced security (e.g., prevent or disable denial-of-service attacks), maintain quality of service for given agents, or give priority to important tasks.

A second kind of enforcement mechanism takes the form of extensions to particular agent platform capabilities. Agents that participate in that platform are generally given more permission to the degree they are able to make small adaptations in their agents to comply with policy requirements. For example, in applications using the DARPA CoABS Grid, we have defined a *KAoSAgentRegistrationHelper* to replace the default GridAgentRegistrationHelper. Grid agent developers need only replace the class reference in their code to participate in agent domains and be transparently and reliably governed by policies currently in force. On the other hand, agents that use the default *GridAgentRegistrationHelper* do not participate in domains and as a result they are typically granted very limited permissions in their interactions with domain-enabled agents.

Finally, a third type of enforcement mechanism is necessary for obligation policies. Because obligations cannot be enforced through preventive mechanisms, enforcers can only monitor agent behavior and determine after-the-fact whether a policy has been followed. For example, if an agent is required by policy to report its status every five minutes, an enforcer might be deployed to watch whether this is in fact happens, and if not to either try to diagnose and fix the problem, or alternatively take appropriate sanctions against the agent (e.g., reduce permissions or publish the observed instance of noncompliance to an agent reputation service). In the near future, we plan to develop monitors and enablers to allow the full use of obligation policies.

Each policy has a property that defines the site of policy enforcement. For example, access control policies are typically enforced by a mechanism directly associated with the resource to be protected (i.e., the target). However in some cases, administrators may not have control over this resource and instead may require the policy to be enforced by a mechanism associated with the actor (i.e., the subject) or some other entity under their purview.

5.5. Performance of the current system

We have tested the performance of KAoS policy conflict resolution algorithms on a machine with Pentium III 1.2 GHz and 640 MB RAM using JDK 1.3.1. In the limited non-optimized tests we have made to date, policy commitment, conflict resolution, and harmonization is consistently performed in a fraction of a second. For reasons that are not yet fully understood, however, assertion of each new policy into the JTP database typically takes an order of magnitude longer than that. Stanford JTP developers are currently working on performance improvements that should significantly affect these results.

6. Example applications

KAoS policy and domain services are being extended and evaluated in the context of several applications.

The first application is the DARPA CoABS-sponsored Coalition Operations Experiment (CoAX) (*http:// www.aiai.ed.ac.uk /project/ coax/*) [1; 20]. CoAX models military coalition operations and implement agent-based systems to mirror coalition structures, policies, and doctrines. The project aims to show that the agent-based computing paradigm offers a promising new approach to dealing with issues such as the interoperability of new and legacy systems, the implicit nature of coalition policies, security, and recovery from attack, system failure, or service withdrawal. KAoS provides mechanisms for overall management of coalition organizational structures represented as domains and policies, while Nomads provides strong mobility, resource management, and protection from denial-of-service attacks for untrusted agents that run in its environment.

Within the DARPA Ultra*Log program (*http://www.ultralog.net*) we are developing agent policy and domain services to assure the robustness and survivability of logistics functionality in the face of information warfare attacks or severely constrained or compromised computing and network resources.

Another application is within the NASA Cross-Enterprise and Intelligent Systems Programs, where we are investigating the use of policy-based models to drive human-robotic teamwork and adjustable autonomy for highly-interactive autonomous systems such as the Personal Satellite Assistant (PSA), a softball-sized flying robot that is being designed to operate onboard spacecraft in pressurized micro-gravity environments [6]. The same approach is also being generalized for use in other testbeds, such as unmanned vehicles and other highly interactive autonomous systems.

Under funding from DARPA's Augmented Cognition Program, we are taking this approach one step further as we investigate whether a general policy-based approach to the development of cognitive prostheses can be formulated, where human-agent teaming could be so natural and transparent that robotic and software agents could appear to function as direct extensions of human cognitive, kinetic, and sensory capabilities [3; 9].

7. Current limitations and future work

We are continuing research and development to address various limitations of the current implementation. The ontology and reasoning does not yet adequately support reasoning about composite actions or processes and relations between fine-grain actions and composed by them high-level actions. We plan to add foundational support for composition of abstract, high-level policies from simple policies through the concept of policies sets. Under funding from the DARPA DAML program, we are also working in collaboration with Austin Tate to investigate how DAML-S (*http://www.daml.org/services/*) may address some of these issues.

Another limitation is our current ad hoc approach to conditional policies through the use of a policy condition monitor. In the coming year, we plan to investigate the use of DAML-R/RuleML for this purpose. We also plan to deal more adequately with the temporal aspects of policy through incorporation of the DAML-Time ontologies.

Additional future work will include full support for obligation policies, performance enhancements to reasoning mechanisms, simplification and streamlining of the KPAT user interface, transition from DAML to OWL, and policy implementation constraint resolution to deal with policies involving contention for finite resources. We will also continue in our efforts to develop versions of KAoS suitable for deployment in Web Services and Grid Computing environments.

Acknowledgments

The authors gratefully acknowledge the sponsorship of this research by the NASA Cross-Enterprise and Intelligent Systems Programs, and a joint NASA-DARPA ITAC grant. Additional support was provided by DARPA's CoABS, Ultra*Log, DAML, and Augmented Cognition programs, and by the Army Research Lab's Advanced Decision Architectures program (ARLADA). We are also grateful for the contributions of Patrick Beautement, Guy Boy, Marshall Brinn, Murray Burke, Mark Burstein, Marco Cavalho, Bill Clancey, Tom Cowin, Rob Cranfill, Naranker Dulay, Paul Feltovich, Rich Feiertag, Richard Fikes, Ken Ford, Mark Greaves, Jack Hansen, Robert Hoffman, Wayne Jansen, Jessica Jenkins, Mike Kerstetter, Mike Kirton, Emil Lupu, Deborah McGuinness, Sheila McIlraith, Nicola Muscettola, Anil Raj, Timothy Redmond, Sue Rho, Dylan Schmorrow, Mike Shafto, Morris Sloman, Austin Tate, and Tim Wright.

References

[1] Allsopp, D., Beautement, P., Bradshaw, J. M., Durfee, E., Kirton, M., Knoblock, C., Suri, N., Tate, A., & Thompson, C. (2002). Coalition Agents eXperiement (CoAX): Multi-agent cooperation in an international coalition setting. A. Tate, J. Bradshaw, and M. Pechoucek (Eds.), Special issue of *IEEE Intelligent Systems*, 17(3), 26-35.

[2] Berners-Lee, T., Hendler, J., & Lassila, O. (2001). The semantic Web. *Scientific American*, 284: 5 (May), 34-43.
[3] Bradshaw, J. M., Beautement, P., Raj, A., Johnson, M., Kulkarni, S., & Suri, N. (2002). Making agents acceptable to people. In N. Zhong & J. Liu (Ed.), *Handbook of Intelligent Information Technology*. (in preparation). Amsterdam, The Netherlands: IOS Press.
[4] Bradshaw, J. M., Dutfield, S., Benoit, P., & Woolley, J. D. (1997). KAoS: Toward an industrial-strength generic agent architecture. In J. M. Bradshaw (Ed.), *Software Agents*. (pp. 375-418). Cambridge, MA: AAAI Press/The MIT Press.
[5] Bradshaw, J. M., Greaves, M., Holmback, H., Jansen, W., Karygiannis, T., Silverman, B., Suri, N., & Wong, A. (1999). Agents for the masses: Is it possible to make development of sophisticated agents simple enough to be practical? *IEEE Intelligent Systems* (March-April), 53-63.
[6] Bradshaw, J. M., Sierhuis, M., Acquisti, A., Feltovich, P., Hoffman, R., Jeffers, R., Prescott, D., Suri, N., Uszok, A., & Van Hoof, R. (2002). Adjustable autonomy and human-agent teamwork in practice: An interim report on space applications. In H. Hexmoor, R. Falcone, & C. Castelfranchi (Ed.), *Agent Autonomy*. (in press). Kluwer.
[7] Bradshaw, J. M., Suri, N., Breedy, M. R., Canas, A., Davis, R., Ford, K. M., Hoffman, R., Jeffers, R., Kulkarni, S., Lott, J., Reichherzer, T., & Uszok, A. (2002). Terraforming cyberspace. In D. C. Marinescu & C. Lee (Ed.), *Process Coordination and Ubiquitous Computing*. (pp. 165-185). Boca Raton, FL: CRC Press. Expanded version of an article originally published in *IEEE Intelligent Systems*, July 2001, pp. 49-56.
[8] 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 of Science, Technology and Medicine, Department of Computing, 20 October 2000.
[9] Ford, K. M., Glymour, C., & Hayes, P. (1997). Cognitive prostheses. *AI Magazine*, 18(3), 104.
[10] Greaves, M., Holmback, H., & Bradshaw, J. M. (2001). Agent conversation policies. In J. M. Bradshaw (Ed.), *Handbook of Agent Technology*. (in preparation). Cambridge, MA: AAAI Press/The MIT Press.
[11] Jena, http://www.hpl.hp.com/semweb/
[12] JTP – Java Theorem Prover, http://www.ksl.stanford.edu/software/JTP/
[13] Kagal, L., Finin, T. & Joshi, A. (2001). A Delegation-based Distributed Trust Model for Multi Agent Systems, under review, http://www.csee.umbc.edu/~finin/papers/aa02/
[14] Kahn, M., & Cicalese, C. (2001). CoABS Grid Scalability Experiments. O. F. Rana (Ed.), *Second International Workshop on Infrastructure for Scalable Multi-Agent Systems at the Fifth International Conference on Autonomous Agents*. Montreal, CA, New York: ACM Press,
[15] Knoll, G., Suri, N., & Bradshaw, J. M. (2001). Path-based security for mobile agents. *Proceedings of the First International Workshop on the Security of Mobile Multi-Agent Systems (SEMAS-2001) at the Fifth International Conference on Autonomous Agents (Agents 2001)*, (pp. 54-60). Montreal, CA, New York: ACM Press,
[16] Lupu, E. C., & Sloman, M. S. (1999). Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*,—Special Issue on Inconsistency Management, 25(6), November/ December, 852-869.
[17] Seamons, K. E., Winslet, M., & Yu, T. (2001). Limiting the disclosure of access control policies during automated trust negotiation. *Proceedings of the Network and Distributed Systems Symposium.*
[18] Suri, N., Bradshaw, J. M., Breedy, M. R., Groth, P. T., Hill, G. A., & Jeffers, R. (2000). Strong Mobility and Fine-Grained Resource Control in NOMADS. *Proceedings of the 2nd International Symposium on Agents Systems and Applications and the 4th International Symposium on Mobile Agents (ASA/MA 2000)*. Zurich, Switzerland, Berlin: Springer-Verlag,
[19] Suri, N., Bradshaw, J. M., Breedy, M. R., Groth, P. T., Hill, G. A., Jeffers, R., Mitrovich, T. R., Pouliot, B. R., & Smith, D. S. (2000). NOMADS: Toward an environment for strong and safe agent mobility. *Proceedings of Autonomous Agents 2000*. Barcelona, Spain, New York: ACM Press,
[20] Suri, N., Bradshaw, J. M., Burstein, M. H., Uszok, A., Benyo, B., Breedy, M. R., Carvalho, M., Diller, D., Groth, P. T., Jeffers, R., Johnson, M., Kulkarni, S., & Lott, J. (2002). DAML-based policy enforcement for semantic data transformation and filtering in multi-agent systems. (submitted for publication).