Automated Generation of Enforcement Mechanisms for Semantically-rich Security Policies in Java-based Multi-Agent Systems

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

[1]*Dipartimento di Elettronica, Informatica e Sistemistica (DEIS)*
University of Bologna
Viale Risorgimento 2, 40136 Bologna - ITALY
{gtonti, rmontanari}@deis.unibo.it

[2] *Institute for Human and Machine Cognition (IHMC)*
40 S. Alcaniz Street, Pensacola, FL 32502 - USA
{jbradshaw, rjeffers, lbunch, nsuri , auszok}@ihmc.us

Abstract

Policies are being increasingly used for controlling the behavior of complex systems (including agent systems). The use of policies allows administrators to specify agent permissions and obligations without changing source code or requiring the consent or cooperation of the entities being governed. Past approaches to policy representation have been restrictive in many ways. By way of contrast, semantically-rich policy representations can reduce human error, simplify policy analysis, reduce policy conflicts, and facilitate interoperability. However, semantically-rich policies increase the complexity of fielding policy-governed multi-agent systems. This paper discusses some technical challenges to automatically enforce semantically-rich security policies in Java-based multi-agent systems and presents an engineering approach for addressing some of these challenges. We have developed a first implementation that allows to enforce OWL policies represented using the KAoS policy framework into multi-agent systems built on top of the JDK1.4. The proposed solution allows to control the behavior of agents at a high level of abstraction and exploits the security mechanisms provided by the Java Authentication and Authorization Service (JAAS) to enforce OWL policies.

1. Introduction

The multi-agent paradigm offers a promising software engineering approach for the development of applications in complex environments [3; 12]. By their ability to operate autonomously without constant human supervision, agents can perform tasks that would be impractical or impossible using traditional software techniques [22; 1]. On the other hand, this additional autonomy, if unchecked, also has the potential of causing severe damage if agents are poorly designed, buggy, or malicious. The technical challenge is to assure that agents will always operate within the bounds of any behavioral constraints currently in force while remaining responsive to human control [4].

Explicit policies can help in dynamically regulating the behavior of agents and in maintaining an adequate level of security, predictability, and responsiveness to human control. By changing policies, the levels of agent autonomy can be continuously adjusted to accommodate variations in externally imposed constraints and environmental conditions without modifying the agent code or requiring the cooperation of the agents being governed [6].

A policy-based approach calls for a policy model specifying how agent permissions and obligations can be expressed and for an enforcement model supporting dynamic control of agent behavior according to desired policies. A few research activities have emerged that propose semantically-rich policy-based approaches to the control of agent systems [8; 14]. Most proposals focus on the problem of policy definition by recognizing the need for the adoption of semantically-rich policy representations [24]. In contrast, relatively little attention has been paid to building general infrastructure-based mechanisms that can monitor and govern the behavior of agent systems.

The development of enforcement mechanisms for semantically-rich policy in agent systems raises several challenges. Semantically-rich policy specifications can be difficult to implement because their high-level descriptions can be far from the concrete implementation details required by policy enforcement components. The gap between specification and implementation of policies has to be resolved to a greater or lesser degree by human programmers, consistently with the capabilities and features of each platform. The mapping of specification to implementation usually requires ad-hoc platform-specific solutions developed each time from scratch and hardly reusable.

Novel engineering techniques and tools will be required to reduce the effort of integrating semantically-rich policies into multi-agent systems, especially into those that have not been specifically designed to make use of policies in their operation. This paper discusses some technical challenges that inhibit fully automatic integration of the enforcement mechanisms needed to maintain agent behavior in conformance to some set of semantically-rich policies (section 2). In particular, we describe an engineering approach to automatically integrate enforcement of semantically-rich policies within JAAS-based systems—whether agent-based or not. Our approach relies on the adoption of OWL ontologies to describe policy concepts and JAAS entities, and on the design of policy enforcement adaptors directly pluggable into the Java systems being governed (section 3). The paper describes the implementation of our approach within the KAoS policy and domain services framework, which supports both agent-based and traditional distributed applications (section 4). In the concluding section, future research steps are highlighted (section 5).

2. Semantically-rich Policies for controlling Agent Autonomy

Policies, which constrain the behavior of system components, are becoming an increasingly popular approach to dynamic adjustability of distributed applications in academia and industry. Policy-based network management has been the subject of extensive research over the last decade [27; 10]. Policies are typically applied to automate network administration tasks, such as configuration, security, recovery, or quality of service (QoS).

The scope of policy management is increasingly going beyond these traditional applications in significant ways. The management of multi-agent systems represents one of the most promising fields for the exploitation of policy-based approaches [2; 5; 14].

Controlling agent behavior is a complex task because agent behavior cannot be programmed a priori to face any operative run-time situation, but requires dynamic and continuous adjustments to allow agents to act in any execution context in the most suitable way. Policies provide the dynamic bounds within which an agent is permitted to function autonomously and limit the possibility of unwanted events occurring during operations. Policies can be exploited to control agent-to-resource and agent-to-agent interactions (*authorization policies*) and to impose upon agents to perform some action or waive some requirement (*obligation policies*). Obligation policies also allow agent system administrators to specify what actions must be specified when security violations occur and who must execute those actions; what auditing and logging activities must be performed, when, and by whom.

Note that control and autonomy are related in inverse fashion [11]: the larger the range of agent *permitted actions* and the smaller the set of agent *obligations*, the more freely the agent can act [6]. Elsewhere we have pointed out the many benefits of policy-based approaches, including reusability, efficiency, extensibility, context-sensitivity, verifiability, support for both simple and sophisticated components, protection from poorly-designed, buggy, or malicious components, and reasoning about agent behavior [4].

2.1. Policy Representation

Many approaches for policy representation have been proposed. These include formal policy languages that can be processed and interpreted easily and directly by a computer, rule-based policy notations that use an if-then-else format, representations of policies as entries in a table consisting of multiple attributes, and ontology-based policy representations. Each form of policy representation exhibits pros and cons, and thus the choice of an approach should be driven by the characteristics of the application domain and by the application requirements.

However, our experience to date seems to indicate quite clearly that the adoption of semantic representations provide several advantages for policy representation in multi-agent systems [24]. The use of ontologies allows the policy framework to be easily extended by simply adding new concepts to the ontology. In traditional languages this task is usually much trickier. In addition, the possibility to simultaneously model policy concepts at multiple levels of abstractions increases the control flexibility, by permitting users to choose the granularity of the control to apply depending on their expertise. For example, modeling policies at a high level of abstraction can allow users to focus their attention more on domain-related management requirements than on implementation details.

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

However, the exploitation of semantic languages for policy representation requires addressing several challenges. Ontology-based policy representations currently rely on a complex syntax, long declarative descriptions, and hyperlinks and references to external resources that make policy specifications very difficult to read in their native formats. This issue is typically addressed by defining graphical user interfaces that convert easy-to-read policy specifications into the syntax required for formal policy representation. A second challenge that has delayed widespread use of semantically-rich policy languages is the gap between policy specifications and enforcement mechanisms. Support for automated enforcement of semantically-rich policies into Java-based systems is the focus of this paper.

2.2. Policy Enforcement and Integration Issues

To better understand the complexity of automatically generating enforcement mechanisms from semantically-rich policy specifications, it is useful to review the set of policy services that have been generally considered necessary for comprehensive management and enforcement of these sorts of policies [8; 15; 21]. Our goal in this section is to give a notional idea of policy services architectures, and not to provide details on each service. Note that a given policy framework implementation may combine one of more of these services in a single component.

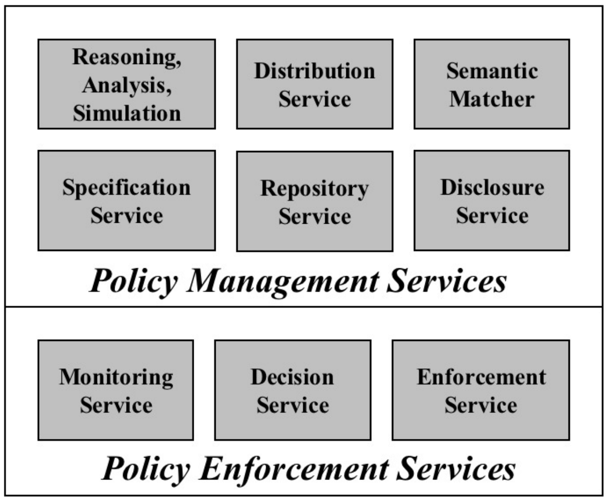


Figure 1. General policy framework architecture.

The upper part of figure 1 shows a set of policy management services supporting: policy specification (*Specification Service*), mapping between policy ontologies and concrete agent system entities (*Semantic Matcher)*, policy storage (*Repository Service)*, policy distribution to interested entities (*Distribution Service)*, policy disclosure to provide an interface for authorized entities to query and resolve questions about policies and controlled entities (*Disclosure Service*), and reasoning, analysis, and simulation to detect and resolve policy conflicts and otherwise provide reasoning support for the other services (*Reasoning, Analysis, Simulation)*.

The lower part of figure 1 shows the set of policy enforcement services for monitoring both application-level and environment-level conditions (*Monitoring Service*), for evaluating the run-time applicability of policies (e.g., checking pre-conditions or verifying policy constraints limiting the run-time applicability of the policy) (*Decision Service*), and to activate and carry out policy enforcement as required (*Enforcement Service)*.

The design of a policy-governed agent system can be complex because it requires not only developing the policy services but also extending the agent platform to take advantage of them. Policy-unaware agent platforms do not typically provide any means that allow developers to monitor and adapt their behavior without modifying the code of the agents themselves.

Whereas policy management services can be designed and implemented more or less independently from agent platforms, policy enforcement services require policy system programmers to know agent platform implementation details and to develop platform-specific adaptors. Policy enforcement consists of a chain of management tasks, i.e., the monitoring of the conditions causing policy activation, the policy evaluation and the policy execution. Each of these tasks has to be accomplished by specific enforcement adaptors.

The development of platform-specific adaptors, however, increases both the integration programming effort and time. The typical approach is to write and integrate a version of each adaptor for each agent platform.

We believe that the adoption of ontologies can play a key role to facilitate the automatic design and development of enforcement adaptors. Ontologies can provide a clear description of all the entities and relations involved in the policy lifecycle management including the policies, the policy services, and the agent system components to govern. Ontologies can be used as inputs to specific tools that are responsible for generating the adaptor pieces of code. By interpreting and reasoning about the contents of the ontologies the tools can produce the adaptor code more easily. The task of integrating the adaptor code into the specific platform to control can benefit from the adoption of weaving and reflective techniques [16; 20].

We have started to explore the adoption of ontologies and of the engineering schema shown in figure 2 in the context of JAAS-based systems. Section 3 describes our approach and highlights the main benefits and limitations we have observed in the adoption of semantically-rich security policies for governing JAAS-based multi-agent systems.

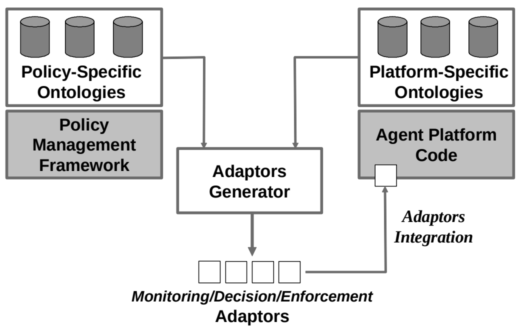

Policy-Specific Ontologies
Policy Management Framework
Platform-Specific Ontologies
Agent Platform Code
Adaptors Generator
Adaptors Integration
Monitoring/Decision/Enforcement Adaptors

Figure 2. Ontology-driven adaptor code generation

3. Enforcing Semantically-Rich Policies in a JAAS-Based Agent System

The Java Authentication and Authorization Service (JAAS) provides a framework and standard programming interface for authenticating users and for assigning privileges ([17;28]). JAAS augments the Java platform with both user-based authentication and access control capabilities. In particular, the JAAS Authentication service supports different forms of user authentication by means of pluggable authentication services, while the JAAS Authorization service can provide code-centric access control, user-centric access or a combination of both. Once authentication has successfully completed, JAAS provides the ability to enforce access controls upon the principals associated with the authenticated subject. JAAS uses the term *subject*, to refer to any user of a computing service. Either a user or a computing service, therefore, can play the role of subject. To identify the subjects with which it interacts, a computing service typically relies on names. The term, *principal*, represents a name associated with a subject.

Our approach extends JAAS security mechanisms by adding the capability to define, manage and enforce complex semantically-rich policies in place of or in addition to the simple default Java security policies. It relies on a set of ontologies to describe policy-related concepts and on a set of policy adaptors to ensure policy enforceability within the frameworks to control. In particular, the former set includes ontologies to describe policies, actors (human or computational), actions being governed by policy, the context of policy applicability, JAAS-related concepts regarding the authorization and authentication process, and the relations among all these concepts. The latter set includes adaptors designed to work in compliance with standard Java Virtual Machines (JVM) and be directly plugged into them without modifying the Java code of the agent platform or of the agents themselves.

3.1. Ontologies to Govern JAAS-Authenticated Entities

The adoption of semantically-rich policies for JAAS-based multi-agent systems requires the design of an appropriate set of ontologies to represent the main concepts involved in policy-based control of JAAS system. We propose one policy-specific ontology (the *policy ontology*) and two JAAS-specific ontologies to model JAAS-related concepts (the *Principal ontology* and the *JAAS action ontology*).

The *policy ontology* contains classes and instances of authorization policies and obligation policies. The policy ontology models the typical basic elements of authorization and obligation policies: the actor, the action and the action context. The *actor* of a policy refers to the entity or the set of entities attempting to perform some policy-governed actions. *Action*[1] classes represent the set of actions that policies can monitor and constrain within a system. The *action context* describes the properties and relations defining the execution context of the action including all the entities that are involved in the action (e.g., the entity being accessed in an access control policy) and the conditions (e.g., the time or location of the action) that determine the applicability of the policy. In addition, for obligation policies the policy ontology includes the description of the policy *trigger* element, i.e, of the causing event for a policy to take on actions.

A simplified diagram showing portions of the *Principal* ontology is shown in Figure 3A. The term *principal* refers to an identity that can be assigned to actors (e.g. humans, software or robotic agents, computational entities) after an authentication process. The ontology includes a taxonomy of controllable principals with each kind of principal represented by a specific class in the ontology. In addition to human-readable principal names that unequivocally identify specific people, the ontology includes Java principals to represent identities associated with requests originating from a Java program. Each Java Principal (e.g. the 'Sample Principal' frequently adopted as demo example in JAAS tutorials) should also be represented as a subclass of the Java Principal class. The property set of the Java Principal class includes the principal identifier and a reference to the Java class defining the principal.

Figure 3B shows a fragment of the *Java Action* Ontology. Each Java action is mapped to a subclass of the generic action class represented in the Policy ontology and describes the set of actions controllable by using the built-in Java security mechanisms. To keep the diagram simple, figure 3B shows only the most common controllable Java actions: the attempt to access a file by opening/reading/writing/closing it and the attempt to access a socket by accepting/connecting to/listening/ resolving a host connection. To simplify the policy specification task, high-level action concepts can be assembled from these atomic concepts.

[1] The action may be as abstract or fine-grained as desired; also, atomic actions may be configured into sets of composed actions [26]

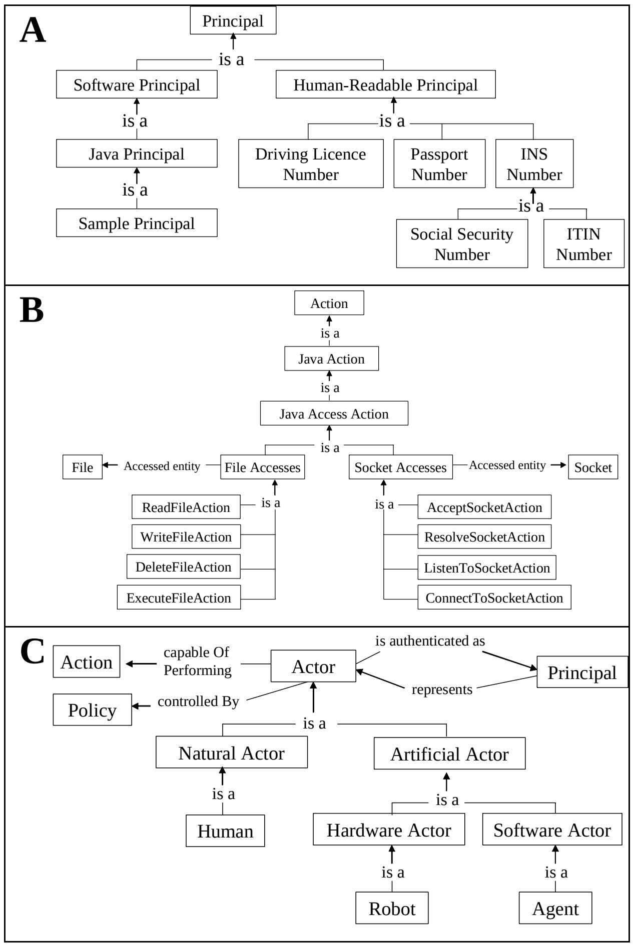


Figure 3. Simplified diagrams showing portions of the JAAS Ontologies[3]

Figure 3C provides an example of possible relations among the principal class and the other main classes of the Policy ontology. In particular, any agent can be represented as a specific kind of actor that can be authenticated and recognized with principals and then governed by policies.

3.2. Policy Enforcement Adaptors to Control JAAS-Authenticated Actors

To control JAAS-Authenticated Actors accordingly to policy specifications we have designed and developed two platform-specific adaptors directly pluggable into JVMs, one for performing the agent application and system monitoring tasks (the *JAAS Monitoring Adaptor*) and one for supporting access control decisions (the *JAAS Authorization Enforcement Adaptor*). In addition, we have designed two policy services that interoperate with the JAAS-specific adaptors in charge of performing policy evaluation (the *JAAS Decision Service*) and of supporting obligation policy enforcement (the *JAAS Obligation Enforcement Service*).

The JAAS Monitoring Adaptor relies on the default Java security mechanisms to monitor the tasks of JAAS-Authenticated Actors. In particular, the adaptor senses and notifies to interested entities, e.g., the decision, the authorization and obligation enforcement services, all the attempts of a JAAS-Authenticated Actors to invoke methods that contain a call to the Java `checkPermission()` method. For example, any attempt to open a file by calling the constructor of the `java.io.FileInputStream` class is detected because it triggers a permission check. We rely on this Java security mechanism to sense the execution of application methods.

The JAAS Authorization Enforcer Adaptor is implemented as a customized Java Security Manager that extends the default SecurityManager class to intercept any call to the `checkPermission` method and to interoperate with the JAAS Decision Service when a check permission is requested. In particular, any time the Authorization Enforcement Adaptor has to handle a check permission call, it creates a Java object instance that wraps the description of the permission to check in an ontology-compliant form and sends it to the JAAS Decision Service. When the decision process completes, the JAAS Authorization Enforcer Adaptor returns the control to the caller principal thread if the permission can be granted, otherwise it throws a security exception.

Figure 4 shows the main components and interactions involved during the enforcement of an authorization policy. When a JAAS Principal attempts to perform a method that can be controlled by the JAAS Monitoring Adaptor, such as an access to a file, the JAAS Monitoring Adaptor triggers the policy enforcement by calling the `checkPermission` method on the currently running Security Manager (i.e. the JAAS Authorization Enforcement Adaptor). For each permission check the JAAS Authorization Enforcement Adaptor queries the JAAS Decision Service and returns control to the Principal thread only if the policies in force authorize the reading of the file.

The JAAS Decision Service is designed as a stand-alone service that can reason over the set of semantically-rich policies in force. It relies on the reasoning capabilities provided by the external policy framework integrated with the JAAS system to determine whether a permission can be granted or whether an obligation action has to be enforced.

The JAAS Obligation Enforcer Service is designed as a service waiting for obligation action directives coming from the JAAS Decision Service. For each received directive the service has the capability to interpret and to enforce it. For this reason, the implementation of the JAAS Obligation Enforcement Service depends upon the kind of obligation policy to enforce and can include several specialized sub-services, one for each kind of enforceable policy action.

[3] Diagrams produced using IHMC's CMap Tools [http://cmap.ihmc.us/]

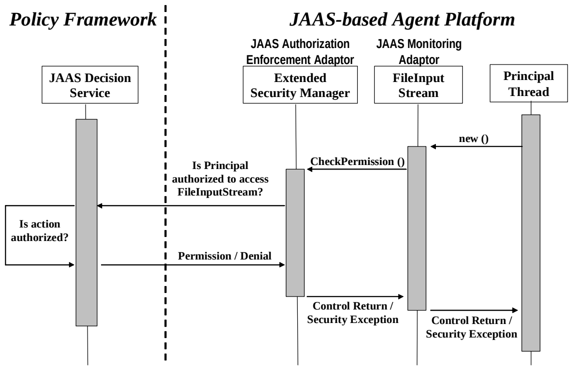

Policy Framework
JAAS-based Agent Platform
JAAS Authorization Enforcement Adaptor
JAAS Monitoring Adaptor
JAAS Decision Service
Extended Security Manager
FileInput Stream
Principal Thread
new ()
CheckPermission ()
Is Principal authorized to access FileInputStream?
Is action authorized?
Permission / Denial
Control Return / Security Exception
Control Return / Security Exception

Figure 4. Authorization Policy Enforcement

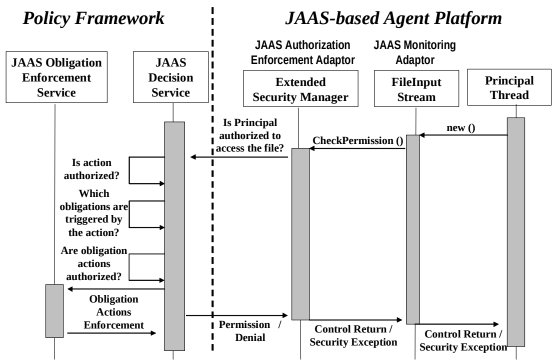

Policy Framework
JAAS-based Agent Platform
JAAS Authorization Enforcement Adaptor
JAAS Monitoring Adaptor
JAAS Obligation Enforcement Service
JAAS Decision Service
Extended Security Manager
FileInput Stream
Principal Thread
new ()
CheckPermission ()
Is Principal authorized to access the file?
Is action authorized?
Which obligations are triggered by the action?
Are obligation actions authorized?
Obligation Actions Enforcement
Permission / Denial
Control Return / Security Exception
Control Return / Security Exception

Figure 5. Obligation Policy Enforcement

Figure 5 shows the enforcement process for obligation policies. The JAAS Monitoring Adaptor works in similar fashion as for an authorization policy, while the JAAS Decision Service retrieves not only the authorization policies but also the obligation policies related to the attempted action. For any triggered obligation policy, authorization permissions to enforce the obliged actions are also checked. Then, for any permitted action it delegates the action enforcement to the JAAS Obligation Enforcement Service. The enforcement of obligation policy actions can be performed before, after or simultaneously with the enforcement of the authorization policies with the same triggering condition (the picture shows the case of simultaneous enforcement); the choice can be delegated to policy administrators or can be part of the policy specification itself.

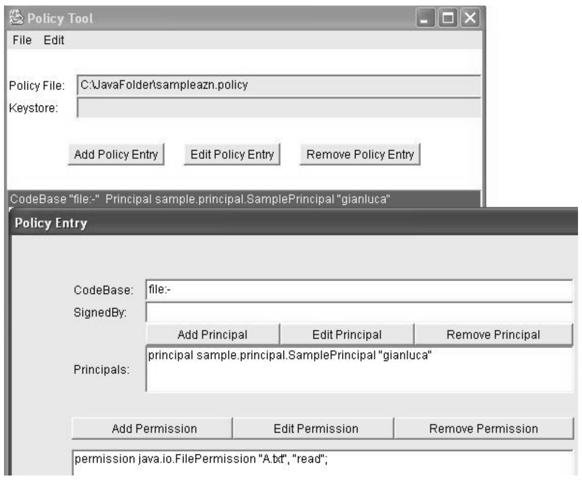
It is finally worth noting that we have chosen to delegate some policy enforcement tasks to services running outside the platform to control instead of using platform-specific adaptors to be as less intrusive as possible on the platform to control, avoiding the adaptation of the platform code that their integration would require. On the other hand, our approach relies on the customization facilities provided by the Java security architecture to plug-in the platform-specific adaptors, and thus can be applied to standard JVM without requiring adaptation of neither the code of the Java-based agent platform nor of the agents themselves.

In particular, the Authorization Enforcement Adaptor can be plugged into the JVM by installing it as a customized Security Manager interoperating with a customized Policy provider (the Java security architecture permits the customization of these components when launching the JVM). Thus the JAAS authentication process works in the standard way, while the authorization process distinguishes between principal-centric permission and code-centric permissions: for the former the authorization process is delegated to the policy frameworks, while the latter are enforced by using the standard Java Access Controller algorithm.

3.3. Main Benefits and Limits of Our Approach

In addition to the general benefits provided to the management of agent platforms described in Section 2.1, the automatic integration of semantically-rich policies within Java-based platforms can provide more context-specific benefits to their management of security.

In the first place, the policy specification task can be made easier and affordable for users without Java programming expertise. Semantically-rich policy representations permit the representation of entities and relations at the desired level of abstraction using ontology concepts. By way of contrast, the specification of policies in Java requires users to be familiar with low-level programming concepts, such as policy entries, principal packages and class names, as shown in Figure 6 where a Java policy assigning the right to read a file to a specific principal is specified with the assistance of the graphical *policytool* included in JDK distributions.



Policy File: C:\JavaFolder\sampleazn.policy
Keystore:

CodeBase "file:-" Principal sample.principal.SamplePrincipal "gianluca"

CodeBase: file:-
SignedBy:
Principals: principal sample.principal.SamplePrincipal "gianluca"

permission java.io.FilePermission "A.txt", "read";

Figure 6. Java policy specification with Java *policytool*

In addition, semantically-rich representations can enrich the policy expressiveness to fit a wider spectrum of needs and requirements. For example, they permit the specification not only of positive access authorization policies such as in Java, but also negative authorization policies, obligation policies, as well as arbitrary policies about any aspect of agent behavior.

Moreover, other benefits can be derived from the adoption of the policy management services provided by policy frameworks, as described in section 2.2. For example, the Specification Service can simplify the policy specification process through powerful and intuitive graphical interfaces. The Repository Service and Distribution Service can relieve users from the burden of spreading and manually linking policy files to distributed JVMs. In addition it can allow dynamic changing of policies at runtime without requiring a direct management of the policy refreshment in the Java code. The Disclosure Service can provide a centralized support for browsing policies and monitoring policy configurations spread among different agent platforms. Finally the Reasoning, Analysis, and Simulation Service can alert policy programmers about possible conflicts (both modality or application-specific conflicts [8; 19]) between newly edited policies and previously defined ones. Then, it can assist users in resolving the detected conflicts with appropriate mechanisms, thus minimizing the risk of policy conflicts among specifications from different platforms, administrators, and times.

Finally, while the Java Access Controller algorithm for checking Java permission always assumes a negative default authorization modality (no actions are authorized if not explicitly permitted), the adoption of a customized Decision Adaptor can make the default modality a context–dependent property and delegate the choice to local system administrators.

However, for assuring the capability to automatically enforce semantically-rich policies without adaptation of the Java code, our approach can support only a constrained set of enforceable policies. A first constraint regards the set of policy triggering conditions that can be monitored which is limited to the set of resource access controls performed by the JVM, such as accesses to files, to the network via socket, or to audio system resources. To extend this set, explicit calls to the Security Manager including the description of the attempted action should be included in the code of the resources to control, while corresponding ontologies describing the triggering condition should be loaded in the policy framework.

Another constraint regards the set of enforceable obligation actions. The Java framework doesn't provide the facilities necessary to generically force an active entity to perform a certain task from its outside. Thus, to avoid the adaptation of the Java code to include specific Obligation Enforcement Adaptors, the current set of applicable obligation actions is limited to those performing actions outside the framework to control or on its public interface, like the notification of warning messages to reachable humans controllable by Notification policies [9].

Finally, we note that to apply our automatic semantically-rich policy integration to running JAAS applications, their execution should be temporarily stopped and restarted after the plug-in of the enforcement adaptors.

4. Case Study: Automatic Enforcement of KAoS Policies in JAAS-based Frameworks

We have started to evaluate the feasibility of our proposed approach by implementing it within the KAoS policy framework.

4.1. KAoS Policy and Domain Services

KAoS is a collection of componentized policy and domain management services compatible with several agent frameworks, as well as some popular distributed computing platforms (e.g., Semantic Web Services, Grid Computing (Globus GT3), CORBA) [4; 8; 13; 25; 26]. KAoS has been deployed in a wide variety of applications, from coalition warfare [2], to robustness and survivability for distributed systems [18; 23], to human-agent teamwork in military and space applications [7], to cognitive prostheses [4].

KAoS domain services provide the capability for groups of software components, people, resources, and other entities to be organized into domains and subdomains to facilitate collaboration and external policy administration. *KAoS policy services* allow for the specification, management, conflict resolution, and enforcement of policies within domains. Policies are currently represented in OWL as ontologies.

The KAoS Policy Administration Tool (KPAT) is a graphical user interface that assists users in policy specification, revision, and application, in browsing and loading ontologies and in analyzing and deconflicting newly defined policies. As policies, domains, and application entities are defined using KPAT, the appropriate OWL representations are generated automatically in the background and asserted into or retracted from the system, insulating the user from having to know OWL or from coding directly in a policy language. Policy templates allow various classes of policy definitions to be defined as high-level domain-specific abstractions.

In addition, KAoS provides enforcement services as implemented by Directory Services, Guards, and Enforcers. The Directory Service is responsible for persistence, analysis, and distribution of policies, while Guards and Enforcers work together to ensure compliance with authorization and obligation policies. For instance, in the case of an authorization policy, the KAoS Enforcers create action descriptions when agents attempt policy-governed actions. These are passed to the Guard, which checks its store of local policies to determine whether the given action instance is in the range of permitted actions. If the Guard does not find any policy applicable to the action description, it answers the authorization question consistent with the default authorization modality of the appropriate domain for the context of the action. Defaults either correspond to a *laissez-faire* mode, where everything is permitted that is not explicitly forbidden, or a *tyrannical* mode, where everything is forbidden that is not explicitly permitted. Obligation policy enforcement works in a similar fashion. However rather than preemptively prohibiting actions, the enforcers either monitor the performance of the obligations, trigger the execution of actions by the agent intended to satisfy the obligations, or—in the case of special kinds of enforcers called enablers—fulfill the obligations themselves. Any necessary handling of sanctions for non-performance can also be performed by the enforcers.

4.2. Enforcement Adaptors for KAoS

The modular design of the KAoS policy framework has permitted us to easily integrate the enforcement adaptors required by our approach within its platform. As described in Section 3, the JAAS Monitoring Adaptor relies on the JVM capabilities to check several types of resource access thus not requiring it to be implemented as a KAoS component.

The KAoS-JAAS Authorization Enforcement Adaptor implements the interface of the KAoS enforcer from one side and extends the Java Security Manager class from the other side. As a KAoS Enforcer the adaptor is associated with a KAoS Guard running on a remote KAoS framework, while as a customized Security Manger it extends the behaviour of the default Java Security Manager, as shown in section 3.2.

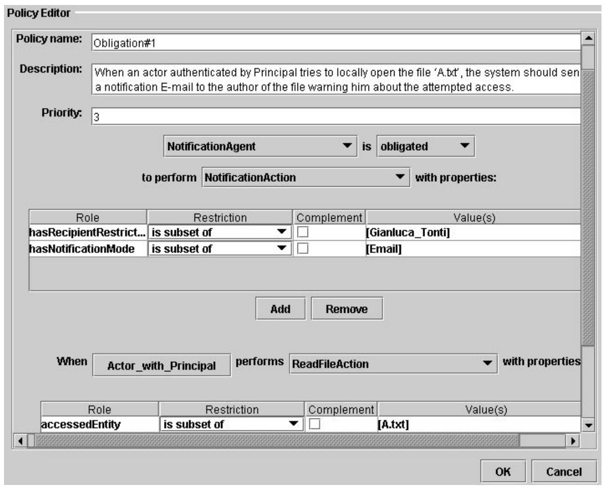
The KAoS Decision Service has been implemented as an instance of the KAoS Guard running as an agent on the KAoS platform. Any JVM instance to be policy-controlled has to be linked to a corresponding instance of the KAoS Decision Service on the KAoS Framework. The communication protocol between the KAoS Authorization Enforcer and the KAoS Decision Adaptor works as explained in section 3.2, and has been implemented on a TCP/IP Socket connection.

Finally, we have added the capability to enforce Obligation policies of type Notification within JAAS frameworks by developing the required KAoS Obligation Enforcer Service as a KAoS NotificationAgent agent running on the KAoS framework. The agent implements the KAoS Enforcer interface to enforce the Notification policies.

4.3. Automatic Enforcement of KAoS Notification Policy

We now show an example of automatic enforcement of a KAoS policy in a JAAS-based application. The policy example is an instance of a Notification policy of kind Obligation stating that "When an actor authenticated by Principal tries to locally open the file 'A.txt', the system should send a notification E-mail to the author of the file warning him or her about the attempted access".

After loading into KAoS both our Java Ontologies [30] and the Notification Policy Ontologies describing the notification process-related concepts and instances [31], the policy can be intuitively specified by using the KPAT graphical interface, as shown in Figure 7. After Specification and commitment into the KAoS Directory Service, the policy is ready to be enforced.



Policy name: Obligation#1

Description: When an actor authenticated by Principal tries to locally open the file 'A.txt', the system should sen a notification E-mail to the author of the file warning him about the attempted access.

Priority: 3

NotificationAgent is obligated

to perform NotificationAction with properties:

Role	Restriction	Complement	Value(s)
hasRecipientRestrict...	is subset of	☐	[Gianluca_Tonti]
hasNotificationMode	is subset of	☐	[Email]

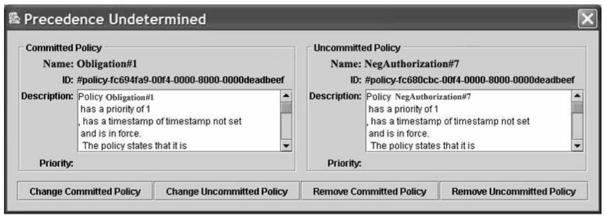
When Actor_with_Principal performs ReadFileAction with properties

Role	Restriction	Complement	Value(s)
accessedEntity	is subset of	☐	[A.txt]

Figure 7. Obligation Policy Specification in KPAT

On the Java-side, the platform/application to control has to be run after plugging in it the KAoS-JAAS Authorization Enforcement Adaptor provided by the KAoS library as a customized Java Security Manager and Java Policy provider. After authentication, any attempt to access the file "A.txt" by Java Principals prompts the KAoS policy framework to enforce both the authorization and the obligation policies triggered by this attempt. Independent of whether the authorization to access the file is granted or not, the KAoS Decision Service retrieves the obligation policy example from the KAoS Directory Service and enforces its action through the NotificationAgent, that sends an e-mail to the file author warning him about the attempted action and the Principal identity associated to the attempting actor.

Let us finally remark that KAoS provides also policy conflict detection capabilities. Figure 8 shows the graphical wizard assisting users in resolving policy conflicts. The wizard window is popped up on the user desktop any time he or she tries to commit to the KAoS Directory Service a policy in conflict with an existing one, such as when trying to add a negative authorization policy denying the permission to enforce the previously committed policy obligation action.



Committed Policy
Name: Obligation#1
ID: #policy-fc694fa9-00f4-0000-8000-0000deadbeef
Description: Policy Obligation#1
has a priority of 1
, has a timestamp of timestamp not set
and is in force.
The policy states that it is
Priority:

Uncommitted Policy
Name: NegAuthorization#7
ID: #policy-fc680cbc-00f4-0000-8000-0000deadbeef
Description: Policy NegAuthorization#7
has a priority of 1
, has a timestamp of timestamp not set
and is in force.
The policy states that it is
Priority:

Figure 8. Policy Conflict Detection and Resolution

4.4. Performance

We have tested the performances of the KAoS policy enforcement on a Pentium IV 1700 MHz, using 512 MB of memory and Sun JDK 1.4.2 running on Microsoft Windows XP operating system. For the test measurements we have adopted the code of the 'Sample' JAAS demo available at [29], modeling the access of Principals to files. Table 1 reports the enforcement times required to authorize the file access, in one case by applying the standard Java policy included in the demo and in the other case by applying a KAoS authorization policy with the same meaning.

The larger time required by the enforcement of the semantically-rich policy is mainly due to the time spent for the serialization of a Java object describing the attempted action to the KAoS Decision AdaptorService. In particular, the total enforcement time can be divided as follow:

- The time for building an action description intelligible for the KAoS framework from the Java permission is 30 ms;
- The time spent exchanging communications via Socket between KAoS and the JVM, including the serialization process is 90 ms ();
- The time spent by the KAoS framework to reason over the set of policies in force and to authorize the permission is 10 ms.

Table 1. Java and KAoS authorization policy enforcement time

	Enforcement Time (ms)
Java 'FilePermission' Policy	< 10
KAoS Semantic Authorization Policy	130

5. Conclusions and Future Work

An increasing number of approaches are adopting semantically-rich policy representations for expressing constraints on the behavior of multi-agent systems. Semantically-rich policy representations seem to provide several advantages in terms of increased expressiveness, analyzability and interoperability. However, enforcement code generation facilities and libraries of enforcement mechanisms adapted to specific platforms are among the major challenges limiting their widespread implementation. Our proposed approach represents a first step toward addressing this issue by providing the design of generic policy ontologies to control JAAS-based applications and frameworks and of enforcement adaptors directly pluggable in the JVM to control.

Our preliminary experiences in developing this approach within the context of the KAoS policy framework seem to indicate that the approach can simplify the programmer's task of controlling Java applications security by enriching the policy expressiveness with acceptable enforcement performances. The proposed approach, however, is currently restricted to the control of JAAS-based implementations and to the management of policies triggered by access control checking performed by standard JVMs. This is stimulating further research to enlarge the set of currently controllable policy triggers and also to export our approach to Java policy-unaware implementations while remaining rooted in semantic descriptions of systems for guiding the automatic generation and installation of the enforcement adaptors.

As a final remark, we note that although the examples in the paper focus on enforcement automation for agent systems, there is nothing that intrinsically limits our approach from being applied to more traditional distributed platforms or novel frameworks such as Grid Computing and Web Services.

References

[1] Allen, J. F., & Ferguson, G. (2002). Human-machine collaborative planning. *Proceedings of the NASA Planning and Scheduling Workshop*. Houston, TX,

[2] 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.

[3] Bradshaw, J. M. (Ed.). (1997). *Software Agents.* Cambridge, MA: The AAAI Press/The MIT Press.

[4] Bradshaw, J. M., Beautement, P., Raj, A., Johnson, M., Kulkarni, S., & Suri, N. (2003). Making agents acceptable to people. In N. Zhong & J. Liu (Ed.), *Intelligent Technologies for Information Analysis: Advances in Agents, Data Mining, and Statistical Learning.* (pp. in press). Berlin: Springer Verlag.

[5] Bradshaw, J. M., Boy, G., Durfee, E., Gruninger, M., Hexmoor, H., Suri, N., Tambe, M., Uschold, M., & Vitek, J. (Ed.). (2003). *Software Agents for the Warfighter. ITAC Consortium Report.* Cambridge, MA: AAAI Press/The MIT Press.

[6] Bradshaw, J. M., Jung, H., Kulkarni, S., & Taysom, W. (2004). Dimensions of adjustable autonomy and mixed-initiative interaction. In M. Klusch, G. Weiss, & M. Rovatsos (Ed.), *Computational Autonomy.* (pp. in press). Berlin, Germany: Springer-Verlag.

[7] Bradshaw, J. M., Sierhuis, M., Acquisti, A., Feltovich, P., Hoffman, R., Jeffers, R., Prescott, D., Suri, N., Uszok, A., & Van Hoof, R. (2003). Adjustable autonomy and human-agent teamwork in practice: An interim report on space applications. In H. Hexmoor, R. Falcone, & C. Castelfranchi (Ed.), *Agent Autonomy.* (pp. 243-280). Kluwer.

[8] Bradshaw, J. M., Uszok, A., Jeffers, R., Suri, N., Hayes, P., Burstein, M. H., Acquisti, A., Benyo, B., Breedy, M. R., Carvalho, M., Diller, D., Johnson, M., Kulkarni, S., Lott, J., Sierhuis, M., & Van Hoof, R.

(2003). Representation and reasoning for DAML-based policy and domain services in KAoS and Nomads. *Proceedings of the Autonomous Agents and Multi-Agent Systems Conference (AAMAS 2003)*. Melbourne, Australia, New York, NY: ACM Press,

[9] Bunch, L., Breedy, M. R., & Bradshaw, J. M. (2004). Software agents for process monitoring and notification. *Proceedings of AIMS 04*.

[10] 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.

[11] Falcone, R., & Castelfranchi, C. (2002). From automaticity to autonomy: The frontier of artificial agents. In H. Hexmoor, C. Castelfranchi, & R. Falcone (Ed.), *Agent Autonomy*. (pp. 79-103). Dordrecht, The Netherlands: Kluwer.

[12] Jennings, N. (2001). An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4), 35-41.

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

[14] Kagal, L., Finin, T., & Joshi, A. (2003). A policy language for pervasive systems. *Proceedings of the Fourth IEEE International Workshop on Policies for Distributed Systems and Networks,* (pp. http://umbc.edu/~finin/papers/policy03.pdf). Lake Como, Italy,

[15] Kagal, L., Finin, T., & Joshi, A. (2003). A policy-based approach to security for the Semantic Web. In D. Fensel, K. Sycara, & J. Mylopoulos (Ed.), *The Semantic Web—ISWC 2003. Proceedings of the Second International Semantic Web Conference, Sanibel Island, Florida, USA, October 2003, LNCS 2870*. (pp. 402-418). Berlin: Springer.

[16] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira-Lopes, C., Loingtier, J. M., & Irwin, J. (1997). Aspect-oriented programming. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP) (LNCS 1241)*. Finland, Springer-Verlag,

[17] Lai, C., Gong, L., Koved, L., Nadalin, A., & Schemers, R. (1999). User authentication and authorization in the Java platform. *Proceedings of the fifteenth Annual Computer Security Application Conference (ACSAC 1999),* (pp. 285-290).

[18] Lott, J., Bradshaw, J. M., Uszok, A., & Jeffers, R. (2004). KAoS policy management for control of security mechanisms in a large-scale distributed system. (pp. submitted for publication).

[19] Lupu, E. C., & Sloman, M. S. (1999). Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering—Special Issue on Inconsistency Management*.

[20] Maes, P. (1987). Concepts and experiments in computational reflection. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '87). ACM SIGPLAN Notices*, 22(10), 147-155.

[21] Montanari, R., Tonti, G., & Stefanelli, C. (2003). A policy-based mobile agent infrastructure. *Proceedings of the 2003 Symposium on Applications and the Internet (SAINT 2003),* (pp. 370-379). Orlando, FL, IEEE Press,

[22] Myers, K., & Morley, D. (2003). Directing agents. In H. Hexmoor, C. Castelfranchi, & R. Falcone (Ed.), *Agent Autonomy*. (pp. 143-162). Dordrecht, The Netherlands: Kluwer.

[23] Suri, N., Bradshaw, J. M., Carvalho, M., Cowin, T. B., Breedy, M. R., Groth, P. T., & Saavendra, R. (2003). Agile computing: Bridging the gap between grid computing and ad-hoc peer-to-peer resource sharing. O. F. Rana (Ed.), *Proceedings of the Third International Workshop on Agent-Based Cluster and Grid Computing*. Tokyo, Japan,

[24] Tonti, G., Bradshaw, J. M., Jeffers, R., Montanari, R., Suri, N., & 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 (Ed.), *The Semantic Web—ISWC 2003. Proceedings of the Second International Semantic Web Conference, Sanibel Island, Florida, USA, October 2003, LNCS 2870*. (pp. 419-437). Berlin: Springer.

[25] Uszok, A., Bradshaw, J. M., & Jeffers, R. (2004). KAoS: A policy and domain services framework for grid computing and semantic web services. *Proceedings of the Second International Conference on Trust Management*. Oxford, England,

[26] Uszok, A., Bradshaw, J. M., Jeffers, R., Johnson, M., Tate, A., Dalton, J., & Aitken, S. (2004). Policy and contract management for semantic web services. *AAAI 2004 Spring Symposium Workshop on Knowledge Representation and Ontology for Autonomous Systems*. Stanford University, CA, AAAI Press,

[27] Wright, S., Chadha, R., & Lapiotis, G. (2002). Special Issue on Policy-Based Networking. *IEEE Network*, 16(2), 8-56.

[28] Sun JAAS web site available at http://java.sun.com/products/jaas/

[29] Sun JAAS demo available at http://java.sun.com/j2se/1.4.2/docs/ guide/secrity/jaas/ JAASRefGuide.html# Sample

[30] KAoS OWL Java Ontologies available at http://ontology.ihmc.us/Java/JavaOntologies.owl

[31] KAoS OWL Notification Ontologies available at http://ontology.ihmc.us/Notification/NotificationOntologies.owl