KAoS Policy Services Framework: User Guide

Date: 21 January 2013

Contents

1 KAoS Overview

In the mid-1990's, we began to develop KAoS. KAoS was originally designed as an agent platform and still provides many features essential to distributed computing, but has now become a set of platform-independent services that enable people to define policies for the security, predictability, and controllability of both agents and traditional distributed systems. KAoS Domain Services provide the capability for groups of software components, people, resources, and other entities to be semantically described and structured into organizations of domains and subdomains to facilitate collaboration and external policy administration. KAoS Policy Services allow for the specification, management, conflict resolution, and enforcement of policies.

1.1 KAoS Objectives

- Provide policy, domain and other services for a wide variety of agent, robotic, and distributed computing platforms
- Be compatible with semantic technology standards (e.g., OWL)
- Provide persistent policy and actor registration storage and sophisticated query and analysis mechanisms
- Support easy extension and customization of service framework elements
- Provide policy dissemination and decision making infrastructure that is distributed, highly efficient and transparently pluggable

1.2 What is Policy?

In agent and distributed computing contexts, policy can be defined as *an enforceable, well-specified constraint on the performance of a machine-executable action by a subject in a given situation.*

- *enforceable:* In principle, an action controlled by policy must be of the sort that it can be prevented, monitored, or enabled by the system infrastructure;

- *well-specified:* Policies are well-defined declarative descriptions;

- *constraint on the performance:* The objective of policy is to ensure, with or without the knowledge or cooperation of the entity being governed, that the policy administrator's intent is carried out with respect to whether or not the specified policy governed action takes place;

- *machine-executable action:* In addition to purely machine-executable actions, we include situations where a person is responsible for completing an action and then somehow signaling that fact to the machine;

- *subject:* The subject is either a human being or a hardware or software component, or a group of such entities;

- *situation:* Policy applicability may be determined by a variety of preconditions and contextual factors.

Policies constrain or amend user or system activity or state. They include a description (*class)* of the controlled *situation.* This constitutes a test (*template)* for the applicability of the policy. They also contain a definition of action Subject; an extension of traditional policy Role. KAoS supports two main types of policies. The set of permitted actions is determined by *authorization policies* that specify which actions an actor or set of actors is allowed (*positive authorizations* policies) or not allowed (*negative authorizations* policies) to perform in a given context. *Obligation policies* specify actions that an actor or set of actors is required to perform (*positive obligations*) or for which such a requirement is waived (*negative obligations*). All other kinds of policies (e.g., delegation, teamwork coordination) are constructed from these two primitive types, combined with other aspects of KAoS policy semantics (e.g., domains, history, or state).

1.3 Policy Management vs. Planning

Policy management should not be confused with planning or workflow management, which are related but separate functions. Planning mechanisms are generally *deliberative* (i.e., they reason deeply and actively about activities in support of complex goals) whereas policy mechanisms tend to be *reactive* (i.e., concerned with actions triggered by some environmental event). Plans are a unified roadmap for accomplishing some coherent set of objectives. However, bodies of policy collected to govern some sphere of activity are made up of diverse constraints imposed by multiple potentially-disjoint stakeholders and enforced by mechanisms that are more or less independent from the ones directly involved in planning. Plans tend to be strategic and comprehensive, while policies, in our sense, are by nature tactical and piecemeal. In short, we might say that while policies constitute the “rules of the road” providing the stop signs, speed limits, and lane markers that serve to coordinate traffic and minimize mishaps, they are not sufficient to address the problem of route planning.

1.4 The Importance of Semantics

The use of XML as a standard for policy expression has both advantages and disadvantages. The major advantage of using XML is its straightforward extensibility (a feature shared with Semantic Web languages such as RDF and OWL, which are built using XML as a foundation). The problem with using XML alone is that its semantics are mostly implicit (meaning is conveyed based on a shared understanding derived from human consensus), which has the potential for ambiguity, promotes fragmentation into incompatible representations, and requires extra effort that could be saved by a richer representation.

OWL was developed under the DARPA Agent Markup Language (DAML) program and adopted as a standard by the W3C. OWL can be easily mapped to lower level XML-based representations if required – mapping from more expressive to less expressive representations is relatively straightforward.

A few policy approaches based on Semantic Web representations (e.g., Rei, PolicyTab) have previously been attempted, but we have found that none have the generality or wide range of capabilities of needed for policy management frameworks like KAoS.

As a means of providing a formal semantics for policies and increasing their expressivity, many specialized logics have been used and extended (e.g., modal logics, event calculus). In contrast, by adopting OWL, a policy representation based on a widely-used formalism with well-understood and highly desirable properties (i.e., description logic), we automatically harness many years of previous development, and the momentum of the W3C standards process that has led to a proliferation of widely-available tools.

KAoS polices are expressed in OWL2 (Web Ontology Language: http://www.w3.org/ 2004/OWL), the current version of the W3C standard, optionally augmented with other constructs (e.g., role-value maps) for greater expressivity. This allows us to provide descriptions of actors, actions and situations at different levels of abstraction. It enables the possibility to dynamically calculate relations among policy, platform entities, and other policies based on concepts ontology relations. We can create a dynamic extension of the service framework by specifying domain specific extensions to the ontology and linking them with the generic KAoS ontology. OWL vocabularies allows for declarative definition of policy applicability.

1.5 The Advantages of Using Policies

Some of the characteristics of KAoS policies that make them useful are their powerful expressiveness, external nature, transparency, and flexibility.

1.5.1 Expressive Power

The choice of policy language directly impacts the expressiveness available in policies. We use OWL to provide declarative specification of policies at a broad range of levels. By combining this with reasoning capabilities, we are able to reason about relationships and produce complex context sensitive policies. They can address the entire system, groups within the system or individual instances within the system. They can refer to actions at any level of abstraction and translate between levels. Most importantly, policies allow for a context to be explicitly defined, which helps to prevent over (or under) restricting the autonomy of the system. Policies provide a mechanism to explicitly define the "work-around" solution based on context. Context can be any information, including things that the robot was never programmed to consider, such as time of day or outside temperature.

1.5.2 External nature of policy

Policies can be used to separate the behavioral constraints and preferences of operators from the underlying functionality. This is an idea that has been successful in many other areas such as database and web design. Because these different aspects of knowledge are decoupled, KAoS policies can be easily reused across different robots and in different situations. By putting the burden for policy analysis and enforcement on the infrastructure, rather than having to build such knowledge into each component themselves, we minimize the implementation burden on developers and ensure that all components operate within the bounds of policy constraints.

1.5.3 Transparency

The use of KAoS policies can also help to make the component behavior more transparent. Again, constraints are made explicit, instead of being scattered and buried in the code. The benefits of transparency are not restricted to humans. Deliberative systems are also free to take advantage of the information available through policy disclosure mechanisms. Such information can be used to reason about the implication of policies and generate a more accurate model of the system. The transparency of policies can be used for planning purposes, resulting in more efficient plans by considering constraints. This can both reduce the search space and prevent futile actions from being attempted. Finally, policies are viewable and verifiable. As systems grow, multiple constraints in complex systems can lead to unexpected (and possibly undetected) conflict. Often these oversights surface at very inopportune moments. Polices can be screened for conflicts prior to activation and in some cases can be automatically harmonized. More importantly, the policy creators can be informed of the problem, so they may take the best course of action.

1.5.4 Flexibility

One of the main advantages of using KAoS policies is that they are a means to dynamically regulate the behavior of a system without changing code. New constraints can be imposed at runtime and can be dynamically changed and updated as the environment or domain changes. Policy flexibility can also be used to suit the system to the human, instead of solely training the human to the system. Through policy, people can precisely express bounds on autonomous behavior in a way that is consistent with their appraisal of an agent's competence in a given context. This provides a broad range of controllability, as well as allowing individuals to tailor the system to their needs. As trust increases, policy can be altered to allow greater autonomy.

2 KAoS Policy Services Architecture

KAoS Services are provided by a few core components. Figure 1 presents basic elements of the KAoS framework. Framework[1] functionality can be divided into two categories: generic and application/platform-specific. The generic functionality includes reusable capabilities for:

- •Creating and managing the set of core ontologies;
- •Storing, deconflicting and querying;
- •Distributing and enforcing policies;
- •Disclosing policies.

For specific applications and platforms, the KAoS framework can be extended and specialized by:

- •Defining new ontologies describing application-specific and platform-specific entities and relevant action types;
- •Creating extension plug-ins specific for a given application environment such as:
- •Policy Template and Custom Action Property editors;
- •Enforcers controlling, monitoring, or facilitating subclasses of actions;
- •Classifiers to determine if a given instance of an entity is in the scope of a given class-defining range.

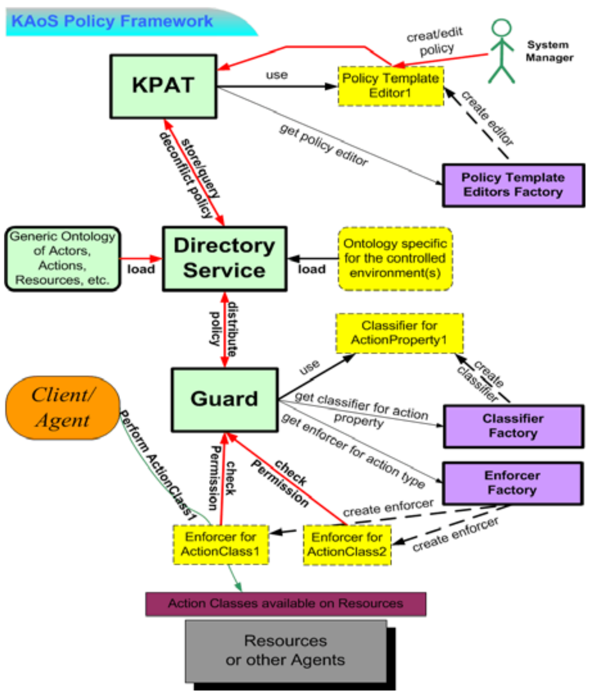

KAoS Policy Framework
KPAT
creat/edit policy
System Manager
use
Policy Template Editor1
create editor
get policy editor
Policy Template Editors Factory
store/query deconflict policy
Generic Ontology of Actors, Actions, Resources, etc.
load
Directory Service
load
Ontology specific for the controlled environment(s)
distribute policy
Classifier for ActionProperty1
create classifier
use
Client/ Agent
Guard
get classifier for action property
Classifier Factory
get enforcer for action type
Enforcer Factory
Perform ActionClass1
check Permission
check Permission
create enforcer
create enforcer
Enforcer for ActionClass1
Enforcer for ActionClass2
Action Classes available on Resources
Resources or other Agents

Figure 1 Selected elements of the KAoS policy and domain services framework

2.1 General Operation

A brief informal description of the general operation of the KAoS Framework starts with launching the Directory Service. The Directory Service is the hub for all activity. The Directory service will use the Internet to access various ontologies. KAoS provides a proxy service for applications that do not have Internet access. Next the managed components are launched. These components can be agents, robots, web services, or just plain applications. Basically they are bits of software that you would like to enforce policies on. For historical reasons we will use the term agent to refer to these managed components. Once launched, these agents register with the Directory Service, potentially providing additional information about their capabilities or other properties. When agents register, a guard is create on the local platform to provide a local policy decision point, enabling policy checking even with intermittent network connectivity. Enforcement is handled by a domain specific application. The Enforcer must be capable of intercepting actions, querying the guard for policy decisions and then enforcing those decisions on the native application. The enforcer does not need to reason about policies as this is the role of the guard, it only needs to enforce the decision. Policies are generally created graphically through KPAT, although they can also be created programmatically through the Common Services Interface (CSI). Once created, KPAT sends the policy to the Directory Service where it is stored and distributed to the relevant guards. Guards only receive policies that are relevant to the agents they are guarding. When an agent attempts to perform an action, the local enforcer intercepts the action, passes it to the guard, the guard informs the enforcer of the policy decision (authorized or not and any obligations) and the enforcer imposes the result on the agents action (e.g. preventing execution if not authorized).

2.2 Major components

2.2.1 Directory Service

The Directory Service is the main component of the system. It is the central location for storage and distribution of information. It keeps information about the domain structure of the environment and contains ontological definitions of the platform and active applications. It allows actors to register their name and identities, membership in domains and ontologically specified types and capabilities. It keeps the state of policies and the ontological description of the current situation by collecting the history of events and monitoring states. It stores policy information, handles policy deconfliction and handles policy distribution. The Directory can be run as a single component or can be distributed (see section ** below). The Directory Service needs access to the Internet (or the KAoS ontology proxy) in order to obtain the required ontologies.

2.2.2 Reasoning Services

Reasoning services are used in KAoS to perform inferences about ontological relations and policies. We currently use Stanford **JTP (**Java Theory Prover). It provides the following features:

- First-order logic reasoning:
  - With support for description logic reasoning over OWL defined Knowledge Bases
- Support for non-monotonic reasoning:
  - Untell operation
- Framework architecture allowing for adding new specialized sub-reasoners

2.2.3 Guard

The Guard is the policy-decision-point, typically running local to the application being governed (e.g., per JVM), but can be deployed remotely as well. Guards can be connected to the DS by a network to enable automatic policy updates, or they can be run in standalone mode, with policies loaded manually. There are usually many Guards in a system, each controlling part of environment activities. It is also integrated with the Directory Service to allow controlling DS actions by policies. The Guard stores precompiled (with cached ontology relations) policies applicable to its area of interest. It does not use external reasoning services directly. The Guard was designed to be easily integrated with legacy system via a Java API. It provides a framework allowing to plug enforcement and classification modules specific for the current application into the reasoning. The Guard is generic. It is not application specific, although its extensions can be.

2.2.4 Enforcer

Enforcement is always platform specific. There are several ways to provide enforcement. We describe a few in chapter 10. The key point to understand is that while the Guard provides a generic way to determine the policy decision, the enforcer is the application specific enforcement mechanism that makes that decision effective.

2.2.5 KAoS Policy Administration Tool (KPAT)

KPAT (KAoS Policy Administration Tool, pronounced KAY-pat) is a graphical user interface that allows people to specify, analyze, modify, and test authorization and obligation policies during development or at runtime. It can also be used to manage sets of ontologies, to configure and inspect Guards, and to perform a variety of other administrative tasks. While KPAT is a sophisticated user interface tool in its own right, most of its functionality is implemented within the DS and the Guards themselves. KPAT provides a means to access that functionality and to view results and state interactively.

KPAT hides the complexity of the OWL representation from users. The reasoning and representation capabilities of OWL are used to full advantage to make the process as simple as possible. Whenever users are required to provide an input, they are presented with a complete set of context-driven values from which to select.

KPAT’s generic Policy Editor (See Figure 2) presents an administrator with a starting point for policy construction – essentially, a very generic policy statement shown as hypertext. Clicking on a specific link that represents a variable provides the user with choices allowing him to make a more specific policy statement. During use, KPAT accesses the loaded ontologies and provides the user with the list of choices, narrowed to the current context of the policy construction. New classes and instances can also be created from KPAT. To further simplify policy construction, KPAT provides two additional policy creation interfaces: A *Policy Wizard* to guide users step-by-step, and a *Policy Template Editor* that allows custom policy editors for a given kind of policy to be created by point-and-click methods.

KAoS includes a series of views, within the KPAT environment, that permit the policy generator the ability to review the policies being generated. They are: Domain View (hierarchy of registered domains), Actor Class View (list of actor classes defined in the loaded ontologies, Policies (shows the entire list of policies and policy hierarchy sets in the system), Policy Templates (list of available policy templates from which the user can create new policies, Policy Disclosure (list of policy disclosure queries), Namespaces (list of loaded ontologies and information about selected ontology, Configuration (current configuration of the Directory Service), Ontology Query (allows the user to query the ontology), Guard Manager (hierarchy of registered Guards and information about the selected Guard)

2.2.6 Ontology Proxy

The ontology proxy is an optional tool that allows KAoS to run without access to the Internet. See chapter 13.3 for more details.

2.3 Policy Distribution

Every actor in the system is associated with a Guard. Each Guard receives policy updates from the Directory Service based on what it guards; actors ids, roles/classes of actors, and actions classes. Before a policy leaves Directory Service it is transformed from OWL to semi-table format. Information about instances in the classes, relevant class and properties relations are cached. The policy is stored in the Guard Policy Information database, according to its priority in order to facilitate efficient policy queries.

3 Getting Started

To begin using KAoS, you must first download and install it. Next you create some actors, enable them with some actions, and provide an enforcement mechanism. Then you can create and apply policies to your actors. We will describe each step of this process in the following sections.

The sections of this chapter are a basic introduction to KAoS. There is a corresponding code example (*kaos/core/tutorial*) that is a physical implementation of this chapter. The example is composed of four classes:

1) SimpleAgent – the main agent example that extends *KAoSActorImpl,* which a KAoS helper class to take care of some of the configuration and registration process.
2) SimpleAgentHuman – an extension for demonstrating ontological types
3) SimpleAgentRobot – an extension for demonstrating ontological types
4) TutorialDemo – a graphical interface to allow for easy creation of agents and domains. It allows for testing out some of the features such as querying the Directory Service, sending messages, and performing a simple action that we can check against policies. This application is simply the scenario driver for the tutorial.

This example is designed to help get you going quickly using helper classes provided by KAoS, but as with everything, there is always more than one way to things. As such, we will occasionally mention some alternative methods, such as using CSI directly. KAoS tries to impose as little as possible on the applications that make use of its services.

It is important to remember that KAoS is *not* an agent environment. It does provide a communication mechanism and a simple method for registering with KAoS, but you must create the actors, their abilities, and provide planning mechanisms as desired. Similarly, KPAT is *not* and centralized controller for an agent system. It is a policy administration tool that allows you to view registered agents and create policies. Lastly, enforcement is always platform specific. It will be up to you to determine the best enforcement strategy for your system. We provide several example techniques that show how enforcement can be achieved. If you keep in mind that the main goal of KAoS is provide policy and domain services to variety of systems you can avoid a lot of confusion.

3.1 Installation

The KAoS distribution is currently offered to users after training. They have to be able to come to our facility in Pensacola and get the proper software training. We will give them the version of the distribution you desire and how to download it to your computer. For installation, simply you unzip the distribution file to the desired location (*your_kaos_root*) on your computer. KAoS requires:

- Java 1.5 or higher (http://www.java.com)
- Ant 1.7 or higher (http://ant.apache.org)

Once the distribution is unzipped you will see several directories under *your_kaos_root*. These directories are: config, lib, scripts and Servlets. The config directory provides several subdirectories that can be configured according to your system needs (Section 3.2). The lib directory contains the required jar files to run KAoS. The Servlets directory contained jar files and a default configuration to run KAoS as a servlet and the scripts directory which you can configure to run your own agent applications (Section 3.2). No specific configuration or additional packages are required for basic execution which uses the KAoS native raw TCP transport protocol and ant scripts. It also provides ant scripts for running the various tools and components. The available targets and descriptions of each of the build.xml files can be obtained by executing "ant -p" (ant version 1.6 and up) in a given subdirectory.

KAoS in general needs access to the Internet to load required ontology files. It is however possible to use KAoS without the Internet access. KAoS includes a tool called "ontology proxy", which can simulate web servers providing ontology files. This is covered in chapter 13.

3.2 KAoS Configuration Files

The directory *kaos.config* has subdirectories *tcp, state and metrics* and several configuration and properties files that can be configurable: logging*.properties*, *kpat.cfg*, *kpat_allTabs.cfg*, *log4j.properties*, and *guardConfiguration.cfg*.

The tcp directory has a subdirectory: *default*. The *default* directory has two files *DS.cfg* and *Guard.cfg.* The *Guard.cfg* contains information about the guard. You can specify the Locators which use UDP to automatically discover the Directory Service. If there are multiple directory services you may specify a preferred one in the locator preferred host parameter setting it as a localhost, (default) or as a hostname (symbolic name) or as the entity IP (numerical address in quotes). For TCP, you can set the network interface when the system has multiple network cards. This applies enabling *preferred-network-interface eth0* to the TCP communication as well as UDP. Also, you can set the source IP address to use for UDP advertisement/discovery (numerical address in quotes) for systems with multiple IP addresses bound to a single interface. By default it is set to localhost: *discovery-source-address localhost.* You can configure the time period for sending keep-alive advertisement via UDP in milliseconds (1000 ms in the example) and enable or disable the compression of the message over the wire by setting the parameter *transport-compress-msg* to either true or false or by comment it out. For the guard locator, you can enable or disable (commenting out the lines) the UDP discovery for the local guard. If it is enabled you can specify the discovery advertise, discovery group, and entity type. An example of the Guard.cfg file is as follow:

```
(java-agent-services
  (key-prefix javax.agent.service)
  (agent-directory-service
    (service-factory kaos.core.service.directory.tcp.TCPAgentDirectoryServiceFactory)
    # specifies the locator to use for the Directory Service
    (directory-service-locator DiscoveryDS)
  )
  (agent-naming-service
    (service-factory kaos.core.service.naming.tcp.TCPAgentNamingServiceFactory)
  )
  (message-transport-system
  # set to false, or comment out to disable message compression
    (transport-compress-msg true)
    (service-factory kaos.core.service.transport.KAoSTransportSystemFactory)
    (message-transport-service

      (TCP
        (transport-factory-class kaos.core.service.transport.BufferedMessageTransportFactory)
        (transport-service-class kaos.core.service.transport.tcp.TCPMessageTransportService)
        (preferred-network-interface eth0)
        (discovery-source-address localhost)

       # set the period (in ms) for sending keep-alive advertisement via UDP
       (discovery-advertisement-period 1000)

        (locators
       # DirectoryService locator: automatically discovered by UDP
          (DiscoveryDS
            (discoveryEnabled true)
        # unique discoveryGroup allows multiple KAoS systems to run on the same LAN
            (discoveryGroup kaos)
            (entityType DirectoryService)
            (preferredHost localhost)
            (entityId KAoSDirectoryService)
          )

        # alternate DirectoryService locator: manually configured hostname/IP address.
          (HostBasedDS
            (host localhost)
            (port 2002)
            (name KAoSDirectoryService)
          )

          # Guard locator: enables UDP discovery. Comment out lines below to disable discovery
          (LocalGuard
            (discoveryAdvertise true)
            (discoveryGroup kaos)
            (entityType Guard)    ))))))
```

The *DS.cfg* file describes the Directory Service configuration. You can enable or disable the compression of the message over the wire by setting the parameter *transport-compress-msg* to either true or false or by comment it out. For TCP, you can enable *preferred-network-interface eth0* when the system has multiple network cards. This applies to the TCP communication as well as UDP. You can set the source IP address to use for UDP advertisement/discovery (put numerical address in quotes) for systems with multiple IP addresses bound to a single interface. You can configure the time period for sending keep-alive advertisement via UDP in milliseconds (1000 ms in the example). For the locators you can comment out *discoveryGroup* to disable UDP discovery/advertisement. When it is enable, a guard can find a directory service by broadcasting its information with the discovery group of interest and the directory service that belongs to that group will respond accordingly. Here is an example of the DS.cfg file:

```
(java-agent-services
 (key-prefix javax.agent.service)
 (agent-directory-service
  (service-factory kaos.core.service.directory.tcp.TCPAgentDirectoryServiceFactory)
 )
 (agent-naming-service
  (service-factory kaos.core.service.naming.tcp.TCPAgentNamingServiceFactory)
 )
 (message-transport-system
 # set to false, or comment out
  (transport-compress-msg true)
  (service-factory kaos.core.service.transport.KAoSTransportSystemFactory)
  (message-transport-service

   (TCP
    (transport-factory-class kaos.core.service.transport.BufferedMessageTransportFactory)
    (transport-service-class kaos.core.service.transport.tcp.TCPMessageTransportService)
    # set the network interface, e.g. for systems with multiple network cards
    (preferred-network-interface eth0)
    # set the source IP address to use for UDP advertisement/discovery
    (discovery-source-address localhost)

    # set the period (in ms) for sending keep-alive advertisement via UDP
    (discovery-advertisement-period 1000)

    (locators
     (KAoSDirectoryService
      (host localhost)
      (port 2002)
      (name KAoSDirectoryService)
      # comment out the line below to disable UDP discovery/advertisement
      (discoveryGroup kaos)     ))))
```

The state directory contains the state.properties file that allows you to instantiate the list of state sensors you are monitoring, their classes and parameters. Upon startup, the StateManager reads this configuration file, instantiates each of the StateSensor, and registers the StateSensor with itself as a listener for policy state conditions, based on the interested types of states. (Details in Section 10.5).For example:

```
#State sensors to instantiate
```

```
#state              class                                            params
Weather             kaos.core.csi.extension.state.WeatherMonitor     PNS MOB
FlightConditions    kaos.core.csi.extension.state.FlightConditionMonitor   PNS MOB
```

The metrics directory has the *MetricManager.properties* file to allow you to see the KAoS metrics which by default is disabled or commented out. You can enable it to see the metrics by removing the # on the parameter: #enabled = true.

In the looging.properties and log4j.properties files you can specify the level of debug information you want while running your application, for example:

```
level=WARNING
level=SEVERE
level=INFO
level=FINE
```

The kpat_allTabs.cfg contains all the tabs available in KPAT. In the kpat.cfg you can configure which tabs you want to be display by default when you are running KPAT.

The guardConfiguration.cfg allows specifying information about your agent as domain name, transport and policy interest. For instance:

```
(java-agent-services
        (key-prefix javax.agent.service)
        (guard-service
                (service-factory kaos.policy.guard.GuardRetriever)
                (Transport
                        (transportName tcp)
                )
                (DomainNames
                        (TSOA 1)
                )
                (PolicyInterests
                        (ActorClasses
                                (http://ontology.ihmc.us/Actor.owl#Actor 1)
                        )
                        (ActionClasses
                                (http://ontology.ihmc.us/Action.owl#Action 1)
                        )
                )
        )
)
```

The directory *kaos.scripts* contains two subdirectories: *kaos-core* and *kaos-tools* with build.xml files for running the KAoS application. You can change parameters in the configuration files according to your system specifications. For instance in the kaos.scripts.kaos-core.build.xml has the following targets for running KAoS and KPAT: run-kaos that runs Directory Service and KPAT. You can also run them separately using the targets: run-ds, and run-kpat. The kaos-tools directory has a build.xml file that allows you to run the KAoS Ontology Proxy described in section 13.3. You can also build a script that starts all KAoS applications after you have created a snapshot of your agents (Section 13.2) as follows:

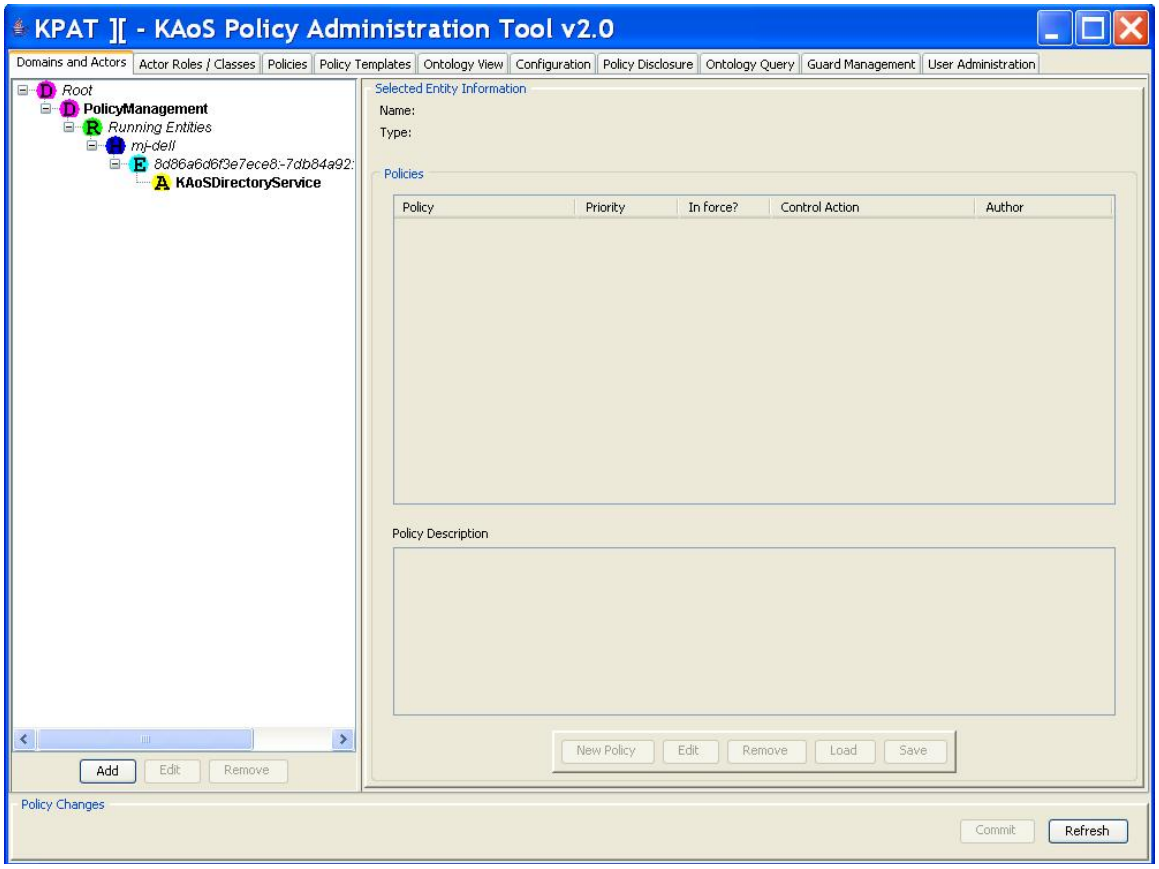
```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project name="Start KAoS with Delegation Management policy and ontolgy snapshot"
        default="runKAoSforDM" basedir="..">
        <import file="${basedir}/config/commonConfig.xml"/>
        <property name="ontology.file" value="${basedir}/config/ontologySnapshots/DM.ont" />
        <property name="directory.snapshot" value=
                        "${basedir}/config/policyConfigurationSnapshots/DMpolicy.cfg"/>
        <property name="bootPath" value="${basedir}/config/tcp/default" />
        <target name="runKAoSforDM"
                description="Start KAoS with Delegation Management demo policy and ontolgy snapshot">
                <fail unless="KAOS_HOME" message="Please set the variable KAOS_HOME" />
                <echo message = "Starting ontology proxy, KAoS DS, Servlet and KPAT"/>
                <parallel threadCount="4">
                    <ant inheritAll="true" antfile="scripts/kaos-tools/build.xml"
                                target="ontology-proxy-autostart" dir="${KAOS_HOME}"/>
                        <sequential>
                                <sleep seconds="8" />
                                <ant inheritAll="true" antfile="scripts/kaos-core/build.xml"
                                        target="run-kaos" dir="${KAOS_HOME}"/>
                        </sequential>
                </parallel>
        </target>
</project>
```

3.3 Running KAoS

Using KAoS ant scripts provided on section 3.2 we can run various tools and components. They can all be started individually, but we provide a default ant target the usually starts everything you need. To run basic KAoS services (DS, servlet and KPAT) the default target of the kaos-core subdirectory build.xml file can be used. Simple:

1) open a terminal and go to the *your_kaos_root* \kaos\scripts\kaos-core directory
2) type “ant”

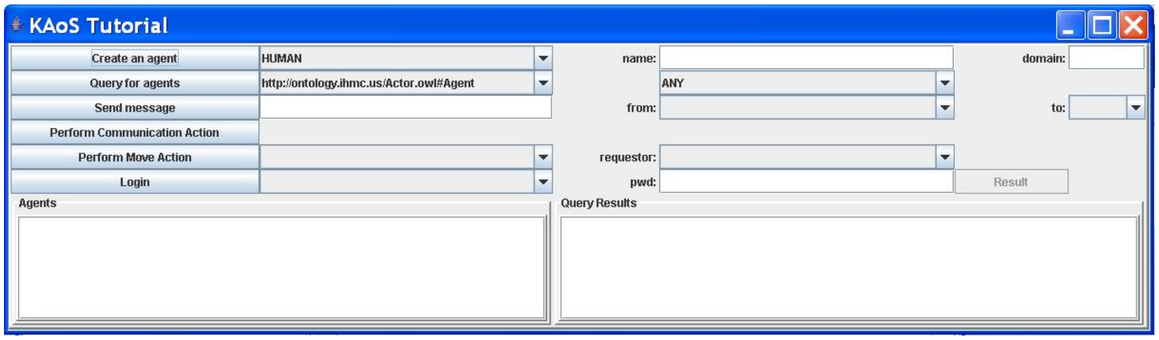
KPAT should appear within one minute with the Directory Service being the only registered agent, as in Figure 2. In general, you can usually start the Directory Service and KPAT once, and then start and stop agents without restarting either the Directory Service or KPAT. You will have to expand the Policy Management domain to see the Directory Service.



D Root
D PolicyManagement
R Running Entities
H mj-dell
E 8d86a6d6f3e7ece8:-7db84a92:
A KAoSDirectoryService

Figure 2 Initial KPAT with only Directory Service running

The ant script for our example also starts KAoS for you, so you can skip this step and go directly to the tutorial script in *your_kaos_root* \kaos\scripts\tutorial and type "ant". You should see KPAT start as in Figure 2, followed by our simple tutorial interface shown in Figure 3. This interface will demonstrate all the items discussed in this chapter.



Create an agent: HUMAN
Query for agents: http://ontology.ihmc.us/Actor.owl#Agent
ANY

Figure 3 Tutorial GUI

3.4 Creating Agents

In order to make use of policies, most developers want actors upon which the policies can be applied. Typically these actors are referred to as agents in this document, but they could be robots, web services, grid services, or anything on which you would like to apply policies.

The simplest method to create an agent is to extend *KAoSActorImpl*. This class provides all the basics for getting an agent started. You create one with a unique identifier (guid), name, list of domains to register in, and desired transport. The guid can be automatically generated by KAoS to ensure uniqueness, although for most projects using the name is sufficient. The transport for this tutorial is just the sting “tcp”. You can specify these parameters on the command line as well and pass the arguments to KAoSActorImpl for parsing. The command line arguments are:

> –guid *agentIdentifie*r {optional: if not supplied, the name will be used}
> –name *agentName*
> –domain *domainName*
> – transport *tcp*

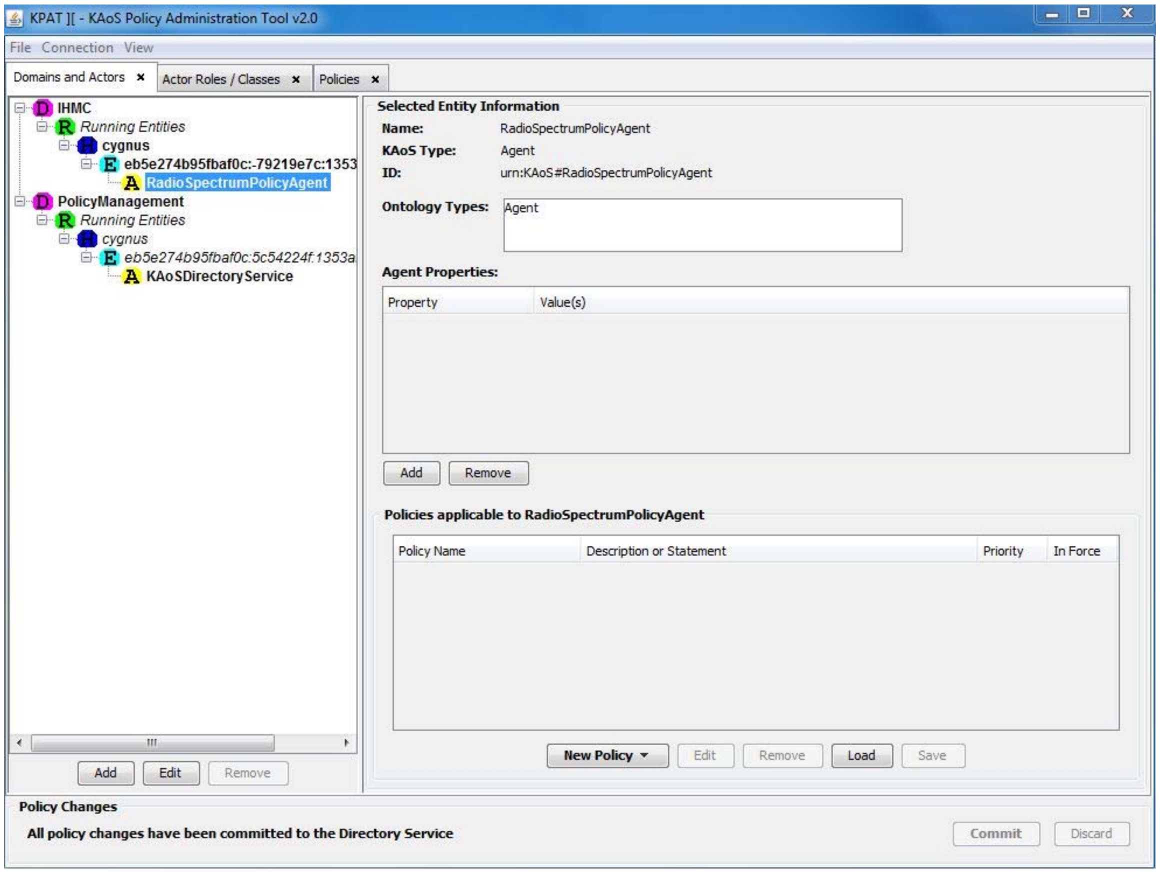
Here is an example of an agent named RadioSpectrumPolicyAgent which only registers with the Directory Service and its arguments are:

> –name: RadioSpectrumPolicyAgent
> –domain: IHMC

```
public class RadioSpectrumPolicyAgent extends KAoSActorImpl
{
   public RadioSpectrumPolicyAgent  (String names, domainNames) throws Exception
   {
      super (name, domainNames);
      initialize();
   }

   private void initialize() throws Exception
   {
      //Get the transport
      String transport = CSIFactory.TCP_TRANSPORT;
      Transoprt trans = new TransportImpl();
       trans.setName (transport);


      //register the agent with Directory Service
      super.registerWithKAoS();
   }

   public static void main(String[] args) throws Exception
   {
      // create the agent
     String domain = “IHMC”;
     Vector<String> domains = new Vector<String>();
      domains.add (domain);

      RadioSpectrumPolicyAgent  agent = new RadioSpectrumPolicyAgent  (“RadioSpectrumPolicyAgent”, domains);

      // wait around until terminated
      synchronized (Thread.currentThread())
      {
         Thread.currentThread().wait();
      }
   }
```

The wait call on the thread prevents the agent from terminating, which would deregister the agent from KAoS.

After the code is compiled, we can run the agent, so the RadioSpectrumPolicyAgent will be registered and be visible in KPAT as in Figure 4.



D IHMC
R Running Entities
cygnus
E eb5e274b95fbaf0c:-79219e7c:1353
A RadioSpectrumPolicyAgent
D PolicyManagement
R Running Entities
cygnus
E eb5e274b95fbaf0c:5c54224f:1353a
A KAoSDirectoryService

Selected Entity Information

Name: RadioSpectrumPolicyAgent
KAoS Type: Agent
ID: urn:KAoS#RadioSpectrumPolicyAgent
Ontology Types: Agent

Agent Properties:

Property	Value(s)

Policies applicable to RadioSpectrumPolicyAgent

Policy Name	Description or Statement	Priority	In Force

Policy Changes

All policy changes have been committed to the Directory Service

Figure 4. RadioSpectrumPolicyAgent is registered in the IHMC domain in KPAT

3.5 Creating Domains

The easiest way to create domains is through the KAoSActorImpl arguments as described in the previous section. This class will automatically register the domains if needed. The domain must exist prior to trying to register in it or you will get an exception. You can register domains directly by using the Common Services Interface (see chapter 0). You can also create them through KPAT manually by going to the “Domains and Actors” tab and pressing the Add button. The domain will be added to the currently highlighted domain or the root domain if nothing is highlighted. Note that added domains will not be available next time you start KPAT unless you save and load the configuration (see chapter 14).

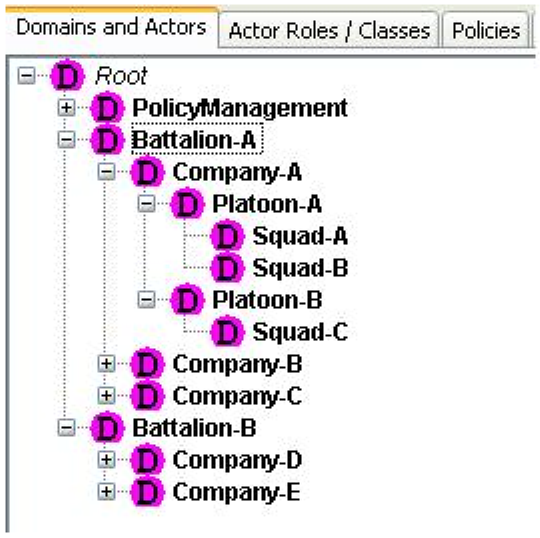


Figure 4 Example of Nested Domains

3.6 Creating Agent Descriptions

Agent descriptions are one way to make use of the richness of using a semantic representation. By describing what group or groups an agent belongs to (domains), what class the agent belongs to (ontological type) and any properties the agent might have (such as capabilities). We will describe how this information is included in the agent description in the following sections and then show how we reason over them in chapter 4.

3.6.1 Specifying Domains

Agents must be registered into a domain and that domain must exist prior to the agent trying to register into it. KAoSActorImpl handles this for you. Domains are a nice way to group entities allowing you to write policies about groups instead of individuals. For example, you could write a policy that members of team A are not allow to send messages. They can represent organizational structures like teams, countries, or chains of command. They can also be used to represent roles such as scout, sentry, duty officer, or plant manager. You can assign an agent to more then one domain at a time and they can be changed at run time as necessary.

3.6.2 Specifying Ontological Type

By default agents register as agents (http://ontology.ihmc.us/Actor.owl#Agent). There are several other types specified in the KAoS onotology, such as Human and Robot. You can also extend these types to include your own types as described in section 12. Similar to domains, ontological types allow you to specify policies about groups instead of individuals. To set the ontological type simply:

agentDescription.setEntityOntologicalType(*url_of_type*);

You can assign more than one type to an agent and they can be changed a runtime as necessary. You can confirm that you have set the type properly by viewing the agent in KPAT under the *Domains and Actors* tab. On the right side, under *Selected Entity Information* you will see the default type of *Agent* and the additional type, *Human*, that was added by the tutorial GUI as shown in Figure .

3.6.3 Specifying Properties

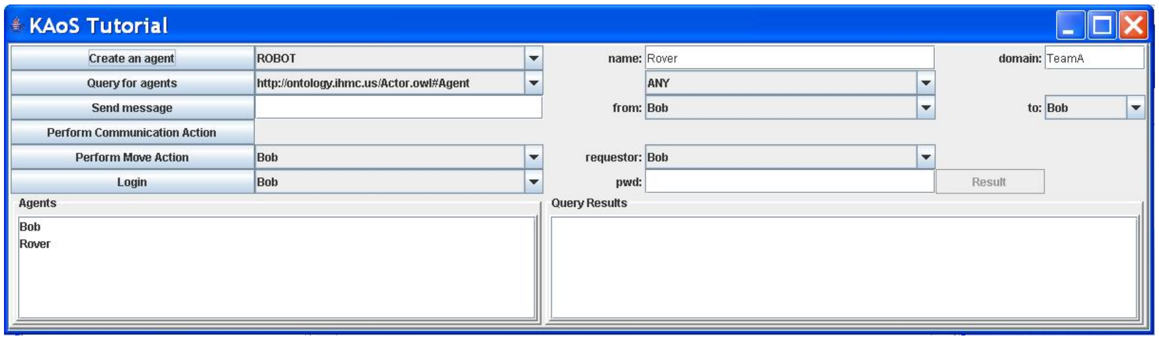
Properties can be arbitrarily added to agent descriptions. You provide property-value pairs for each property you wish to define. You can assign multiple properties to an agent and they can be updated at runtime. Properties can be anything you would like such as location, clearance, or capabilities. We will demonstrate the property of "capabilities" which is a default property available with agent descriptions. As in section 3.3 we can create a new agent named *SimpleAgentRobot.* For this new class we override the *getCapabilities* method to define a capability of *Target* as follows:

```
List<AgentCapability> capabilities = new ArrayList<AgentCapability>();
AgentCapability agentCapability = new AgentCapabilityImpl();
agentCapability.setName(ActionConcepts.Target());
capabilities.add(agentCapability);
```

Target is not a very meaning full property for a robot, but it is just for demonstration purposes and this tutorial is only using the core KAoS ontology and does not include the robot ontology extensions. The property is added to the agent's description in the initialize method of *SimpleAgent*:

```
List<AgentCapability> capabilites = getAgentCapabilities();
_actorDesc.setCapabilities(capabilites);
```

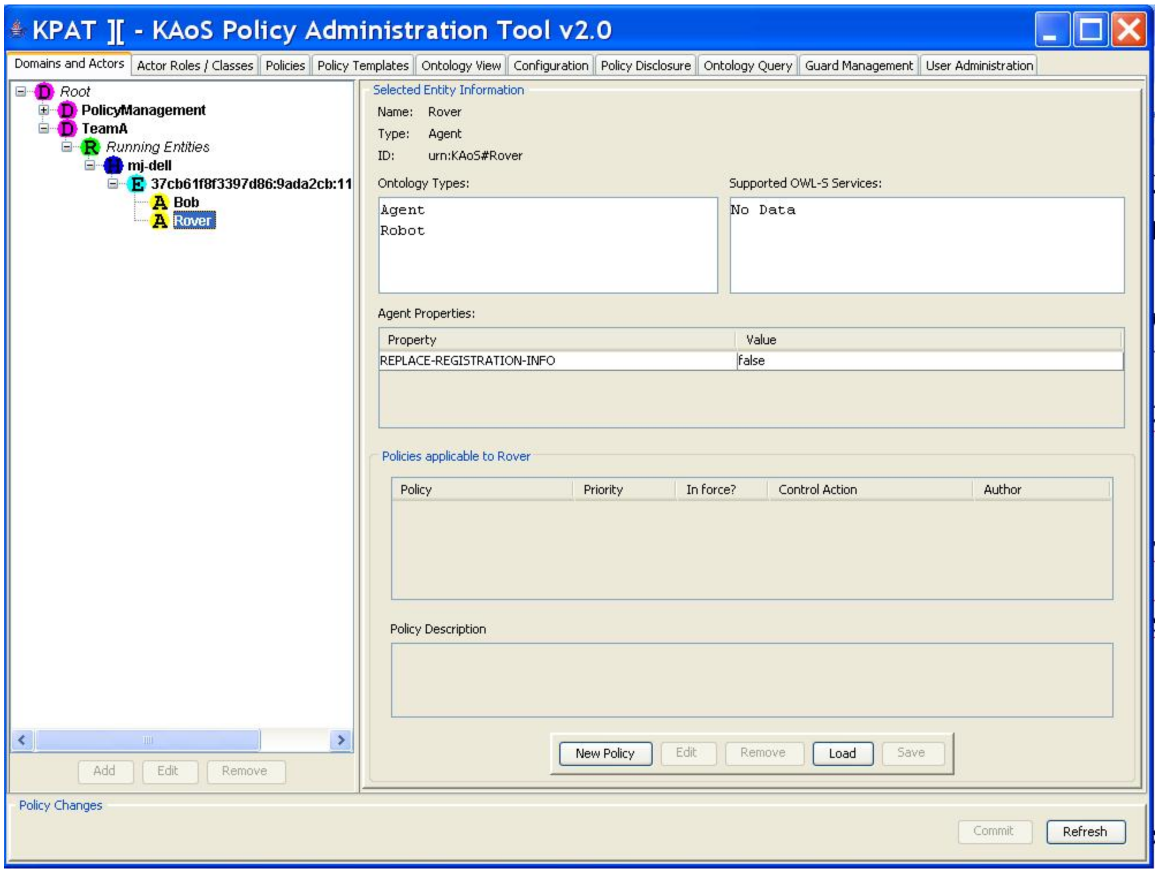
Now we will create a robot by selecting "Robot" as the type, "Rover" as the name and leaving "TeamA" as the domain.



Create an agent: ROBOT — name: Rover — domain: TeamA
Query for agents: http://ontology.ihmc.us/Actor.owl#Agent — ANY
Send message — from: Bob — to: Bob
Perform Move Action: Bob — requestor: Bob
Login: Bob — pwd:
Agents: Bob, Rover

Figure 5 Rover Registered as a Robot

Again, Rover will show up as an agent in the tutorial GUI (Figure 5) and in KPAT as in Figure 6. Notice that the ontological type is different in Figure 5.



KPAT][- KAoS Policy Administration Tool v2.0

```
D Root
  D PolicyManagement
  D TeamA
    R Running Entities
      mj-dell
        E 37cb61f8f3397d86:9ada2cb:11
          A Bob
          A Rover
```

Selected Entity Information

Name: Rover
Type: Agent
ID: urn:KAoS#Rover

Ontology Types:

```
Agent
Robot
```

Supported OWL-S Services:

```
No Data
```

Agent Properties:

Property	Value
REPLACE-REGISTRATION-INFO	false

Policies applicable to Rover

Policy	Priority	In force?	Control Action	Author

Policy Description

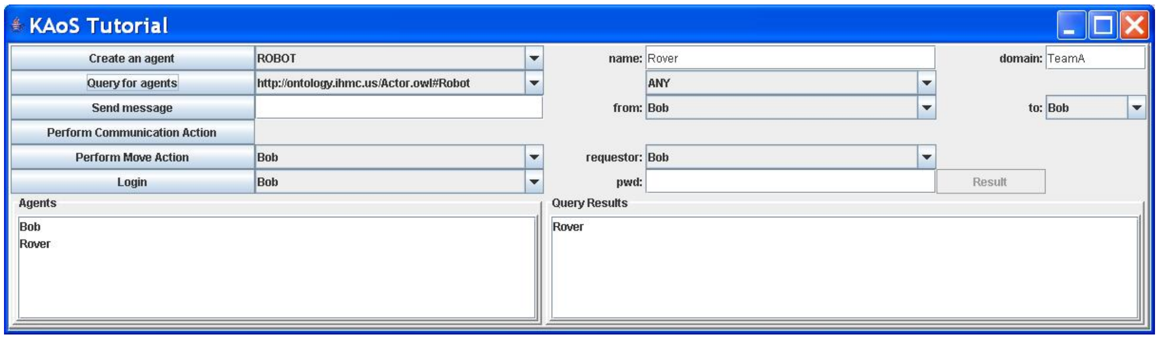
Figure 6 Rover Registered in KPAT

3.7 Registering Agents

Once you have created the description you would like, simply call *registerWithKAoS()* if you have extended *kaos.core.csi.KAoSActorImpl*. You can also register through CSI as described in chapter 0. Registrations can be updated or replaced at runtime as necessary.

3.8 Finding Agents

Agents can be found by name, domain, ontological type, or capability. The details are provided in the chapter 0 under Query. Our tutorial GUI allows you to query by ontological type or capability and displays the results in the right hand side as shown in Figure 7.



KAoS Tutorial

Create an agent	ROBOT	name:	Rover	domain:	TeamA
Query for agents	http://ontology.ihmc.us/Actor.owl#Robot		ANY		
Send message		from:	Bob	to:	Bob
Perform Communication Action					
Perform Move Action	Bob	requestor:	Bob		
Login	Bob	pwd:		Result	

Agents

```
Bob
Rover
```

Query Results

```
Rover
```

Figure 7 Query Results for Robots

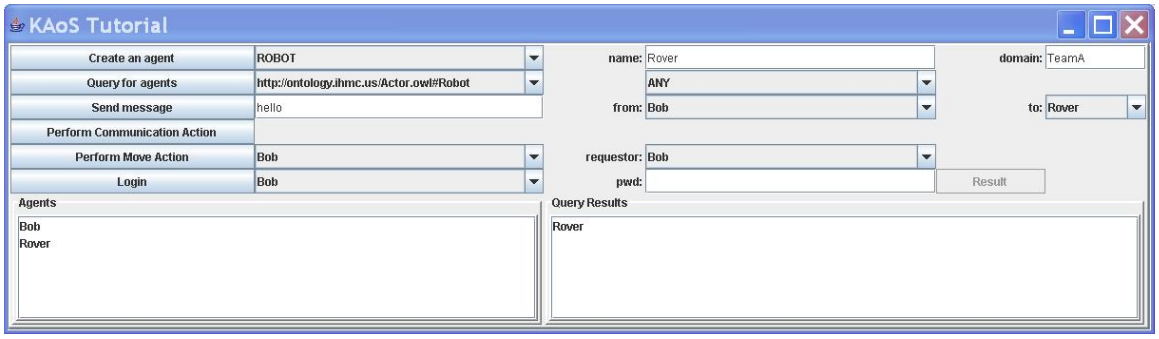
3.9 Sending Messages

Although there are many ways to send messages this section will discuss using *KAoSActorImpl* to send messages. *KAoSActorImpl* provides a simple method that uses the transport layer of CSI (refer to chapter 0):

```
sendMessage(message, receiverName);
```

Our tutorial GUI (Figure 8) allows you to type your text message next to the “Send message” button and select the sender and receiver. Then press the “Send message” button and you should see:

[java] *SenderName* sent message to *ReceiverName*: *message*
[java] *RecieverName* received message from *SenderName*: *Message*



ROBOT
http://ontology.ihmc.us/Actor.owl#Robot
hello
Bob
Bob
name: Rover
domain: TeamA
ANY
from: Bob
to: Rover
requestor: Bob
pwd:
Agents: Bob, Rover
Query Results: Rover

Figure 8 Sending a Message

3.10 Receiving Messages

To receive messages using the KAoS transport your agent must implement the MessageListener interface. The KAoSActorImpl class already does this, so if you are extending it you only need to override the receive message method as shown in SimpleAgent:

```
public void receiveMessage(Serializable messageContent, KAoSActor sender)
{
    System.out.println(_actorDesc.getAgentNickname() + " received message from " +
    sender.getName() + ": " + messageContent);
}
```

3.11 Creating a Simple Action

Action descriptions are the main datatype exchanged with KAoS:

- kaos.core.csi.ActionInstanceDescription(Impl)
- kaos.core.csi.OntPropertyDescription(Impl);

Applications that check authorization policies must be able to create of action descriptions, and applications that handle obligations must be able to interpret them when received. Action descriptions can contain a complex value in the form of:

- kaos.core.csi.OntInstanceDescription(Impl);

Names used as types and properties in these data structures are from the Java vocabulary files generated using the mapping tool (chapter 12.4).

We will use a simple *ActionInstanceDescription(AID)* to represent an action. You can create an AID in several ways including using OWL or just Strings from the ontology. Our example uses simple Strings from the ontology:

```
HashMap properties = new HashMap();
  properties.put(ActionConcepts.hasDestination(), NamespaceValidator.validateURI(receiverName));
  properties.put(ActionConcepts.carriesMessage(), message);
  aid = new ActionInstanceDescriptionImpl(null,
                                           ActionConcepts.CommunicationAction(),
                                           NamespaceValidator.validateURI(this.getName()),
                                           properties);
```

ActionConcepts is the Java class that contains the corresponding strings from the KAoS OWL ontology. It is automatically generated using tools described in chapter 12.3. We are going to add two properties to the action. The first is *hasDestination* and the value is the intended receipiant. The second is *carriesMessage* and message is the value. When agents register with KAoS, their names are not valid URIs, so KAoS appends *urn:KAoS#* to make them valid. The *NamespaceValidator.validateURI* method appends the appropriate prefix. You could append it yourself, but using this method ensures you will not have problems if the prefix changes in the future. **Forgetting to add this prefix when using names is the most frequent cause of problems people encounter**. Please ensure whenever you use a name in a query or in an AID that you qualify as shown above. It will save you a lot of headaches.

3.12 Adding a Simple Enforcer for Policy Checking

Adding a simple enforcer is easy using our *KAoSActorImpl*. You just call the *enforcePolicies* method which does the following:

```
PolicyChecking policyChecking = CSIFactory.getPolicyChecking();
Vector<ActionInstanceDescription> obligations = policyChecking.findPolicyDecision(aid, null);
```

Basically this uses CSI (chapter 0) to get a Policy Checking interface and then uses the interface to get the policy decision. Ignore the returned obligations for now (see chapter0). The important feature for this section is that this method throws a *KAoSSecurityException* if the action is not authorized. There are many ways to provide enforcement, which we discuss in chapter 10. In this example we are providing our own enforcement, by checking the action and only allowing the action if it is authorized. Inside t

3.13 Creating a Simple Policy without Using KPAT

We can build policies using KPAT or adding some specifications in the RadioSpectrumPolicyAgent to build the policy and then using KPAT to validate the policy. We are going to start by using CSI to get the ontology service and then build a “Channel” class using the KAoSOntClassBuilder from the KAoS classes and load it.

```
OntologyService os = CSIFactory.getOntologyService();
//Create a class and an ontology builder for Channel
KAoSOntClassBuilder channelBuilder =
  new KAoSOntClassBuilderImpl (http://ontology.ihmc.us/Radio/RadioEntity.owl#” + policygui +“Channel”);

//Use the Radio ontology to load the new class
channelBuilder.setMainSuperClass (RadioEntityConcepts.Channel);
```

We can set a Control Action class “RadioTransmissionAction” as follow:

```
String action = RadioActionConcepts.RadioTransmissionAction;

//Get a builder for the control action class
KAoSOntClassBuilder controlActionBuilder = new KAoSOntClassBuilderImpl ("ControlActionForPolicy" +
                                                                         Policyguid);
```

```
    //Use the Radio ontology to load the new class “Channel”
    controlActionBuilder.setMainSuperClass (action);
```

Then, we set the property range “hasChannel” using the control action builder already defined and load the control action builder into DirectoryService using the SerializableOntModel class in KAoS:

```
controlActionBuilder.setPropertyRangeClass (RadioActionConcepts.hasChannel(),
                                            channelBuilder.getClassName());
```

We can set another property range to a class value already defined on existing KAoS ontologies like the Actor class.

```
controlActionBuilder.setPropertyRangeClass (ActionConcepts.performedBy(), ActorConcepts.Actor);
```

Finally, load the model:

```
SerializableOntModelImpl model = controlActionBuilder.getOntModel();
os.loadOntology (model, false);
```

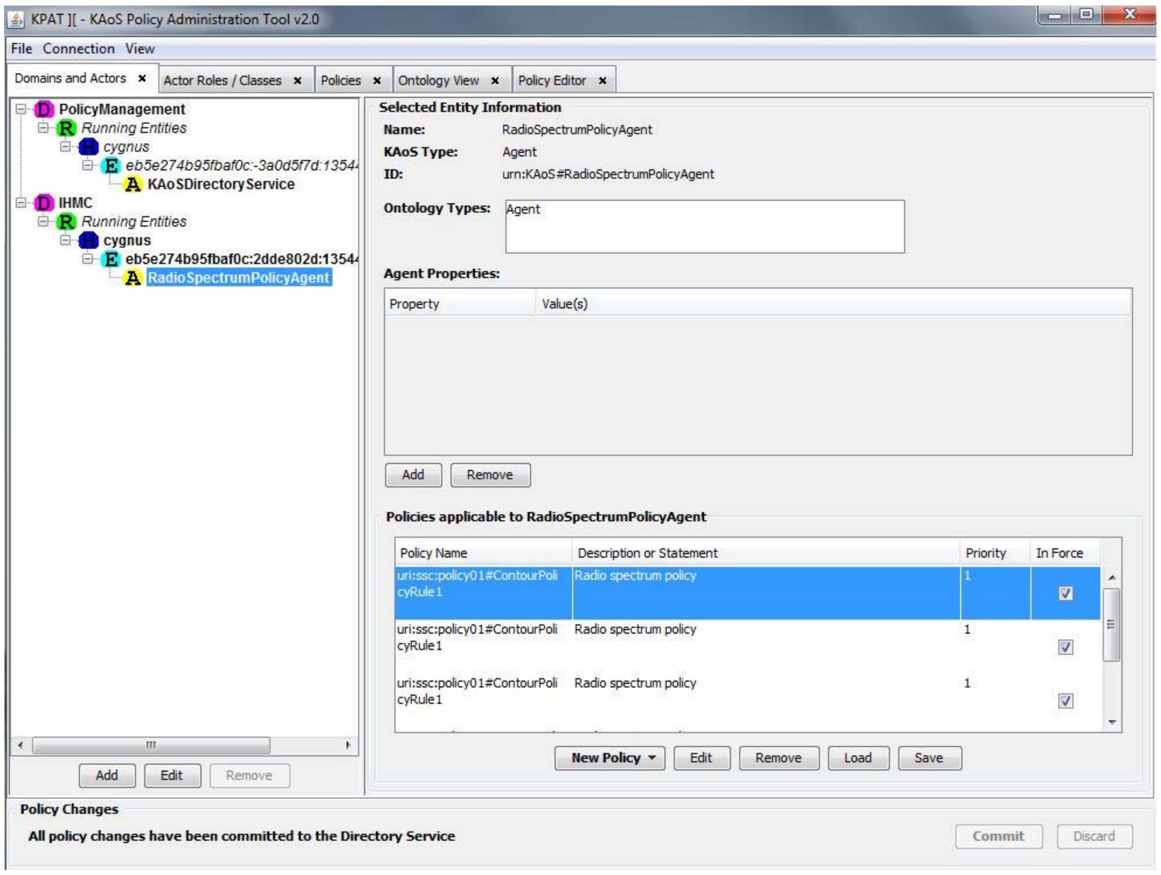
After this requirements are set we are ready to build the policy using the KAoS policy builder class and adding the policy id, name, modality, description and priority:

```
KAoSPolicyBuilder owlPolicyBuilder = new KAoSPolicyBuilderImpl();
owlPolicyBuilder.setPolicyName (“RadioSpectrumPolicy”);
owlPolicyBuilder.setPolicyAuthor (“KAoSTeam”);
owlPolicyBuilder.setPolicyDescription ("This is the Radio spectrum policy");
owlPolicyBuilder.setPriority (1);

//Set the control action to the owl policy builder
owlPolicyBuilder.setControlActionClass (controlActionBuilder);

//Get the policy message from the owl policy builder and add it to a list of policies
PolicyMsg polMsg = owlPolicyBuilder.getPolicyMsg();
List<PolicyMsg> policyList = new ArrayList();
policyList.add (polMsg);
```

We get the policy management from CSI and update it with the new created policy so we can run the RadioSpectrumPolicy agent and see the policy display on KPAT.

```
PolicyManagement pm = CSIFactory.getPolicyManagement();
pm.updatePolicies (policyList, new ArrayList(), new ArrayList());
```

We can run KAoS and KPAT so we can validate the policy. Figure 11 a. shows the RadioSpectrumPolicy that we created without using KPAT.



KPAT][- KAoS Policy Administration Tool v2.0

File Connection View

Domains and Actors | Actor Roles / Classes | Policies | Ontology View | Policy Editor

- PolicyManagement
  - Running Entities
    - cygnus
      - eb5e274b95fbaf0c:-3a0d5f7d:1354
        - KAoSDirectoryService
- IHMC
  - Running Entities
    - cygnus
      - eb5e274b95fbaf0c:2dde802d:1354
        - RadioSpectrumPolicyAgent

Selected Entity Information

Name: RadioSpectrumPolicyAgent

KAoS Type: Agent

ID: urn:KAoS#RadioSpectrumPolicyAgent

Ontology Types: Agent

Agent Properties:

Property	Value(s)

Policies applicable to RadioSpectrumPolicyAgent

Policy Name	Description or Statement	Priority	In Force
uri:ssc:policy01#ContourPolicyRule1	Radio spectrum policy	1	✓
uri:ssc:policy01#ContourPolicyRule1	Radio spectrum policy	1	✓
uri:ssc:policy01#ContourPolicyRule1	Radio spectrum policy	1	✓

Policy Changes

All policy changes have been committed to the Directory Service

Figure 9a RadioSectrumPolicydisplyed by KPAT

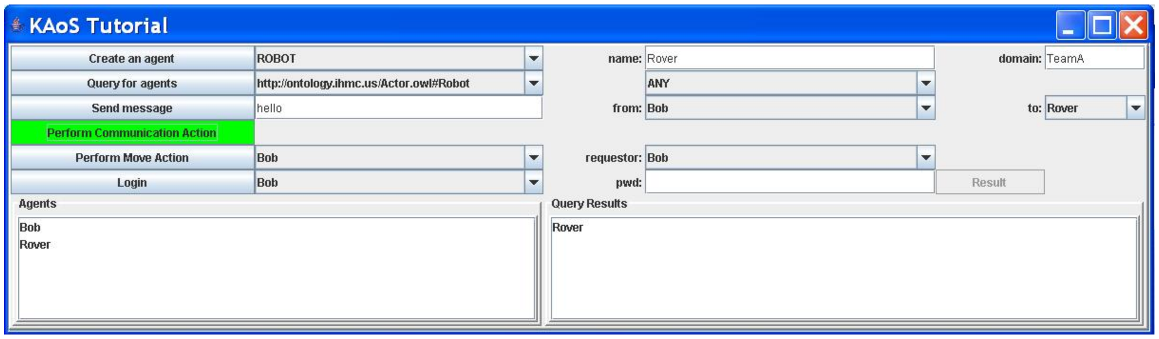
3.14 Creating a Simple Policy Using KPAT

We now have the basic requirements to demonstrate a simple policy. Using the KAoS tutorial, create two agents as we have done in the above sections. The policy we will create is a simple authorization policy. Authorization policies permit or deny actions. The basic format is an actor, modality and action. For example:

Bob	is not authorized	to send messages	to Sam
actor	modality	action	[context]

In this policy, Bob is the actor, negative authorization is the modality, and sending messages is the action. We also include the context of “to Sam”. Context is optional. It is usually some property of the action, but can be complex. Context enables very rich policy specification. For our first policy we will not use any context, but will add some in future examples.

Enter a message as in the sending message section and select Bob as the sender and Rover as the receiver. Instead of pressing “Send message” use the “Perform Communication Action” button. This will build the AID and check policies on the action. Since we have not created the policy yet, the result should see the button turn green indicating authorized as in Figure 10b.



Create an agent	ROBOT	name:	Rover	domain:	TeamA
Query for agents	http://ontology.ihmc.us/Actor.owl#Robot		ANY		
Send message	hello	from:	Bob	to:	Rover
Perform Move Action	Bob	requestor:	Bob		
Login	Bob	pwd:			

Agents

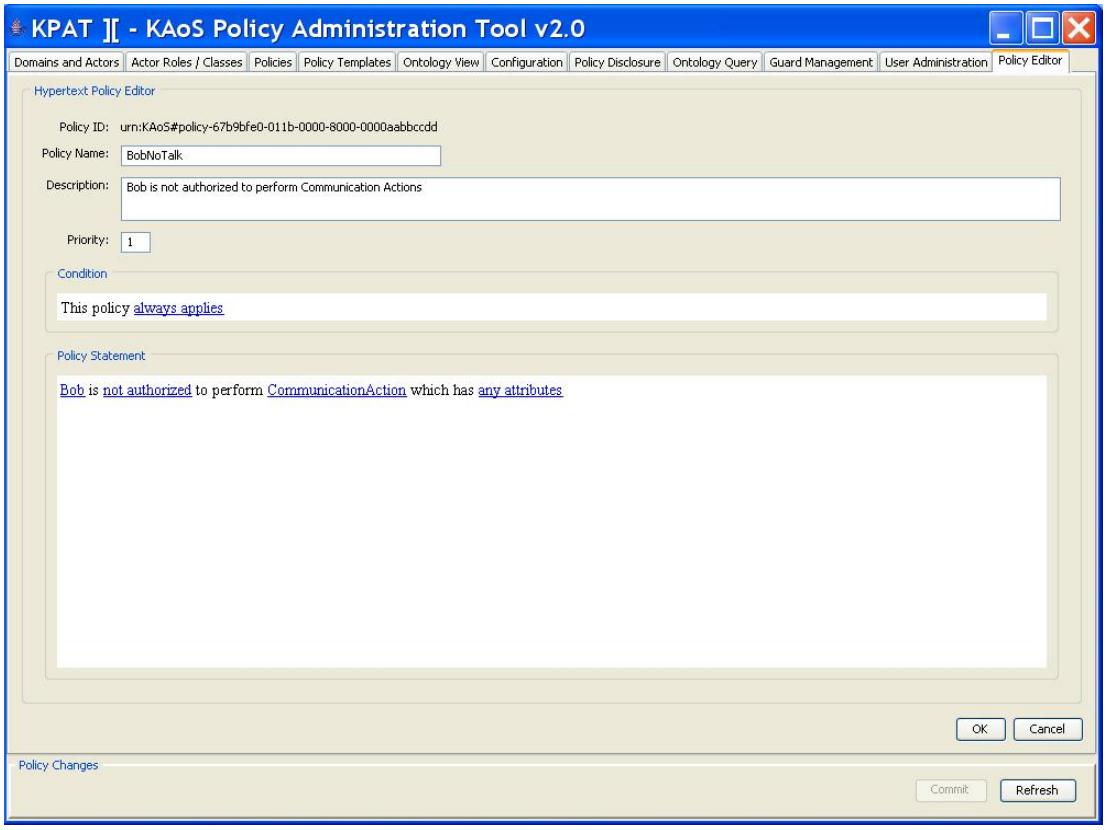
Bob
Rover

Query Results

Rover

Figure 10b Authorized Action

Now let's create a policy. We will create an authorization policy that says Bob is not authorized to perform Communication Actions. This means Bob can not perform any Communication Actions no matter what the properties. We will relax this in later policies, but for now go to KPAT and select Bob. Press the "New Policy" button and select "Use hypertext editor". An editor will appear. Give the policy a name, a description, and a priority. The original priority mechanism was a simple integer ranking, so a value of one is fine. We provide other priority mechanisms, but they are covered elsewhere. Ignore the condition for now and move to the Policy Statement. Bob will be the default, since you selected Bob before starting the policy. Click on *constrained* and choose *not authorized*. Click on *action* and select *CommunicationAction*. Your policy should look like Figure 11. Now press OK. KPAT will remind you to commit your changes. Press commit and KPAT will tell you that it was successful. Congratulations! You just made your first policy.



Policy ID: urn:KAoS#policy-67b9bfe0-011b-0000-8000-0000aabbccdd

Policy Name: BobNoTalk

Description: Bob is not authorized to perform Communication Actions

Priority: 1

Condition

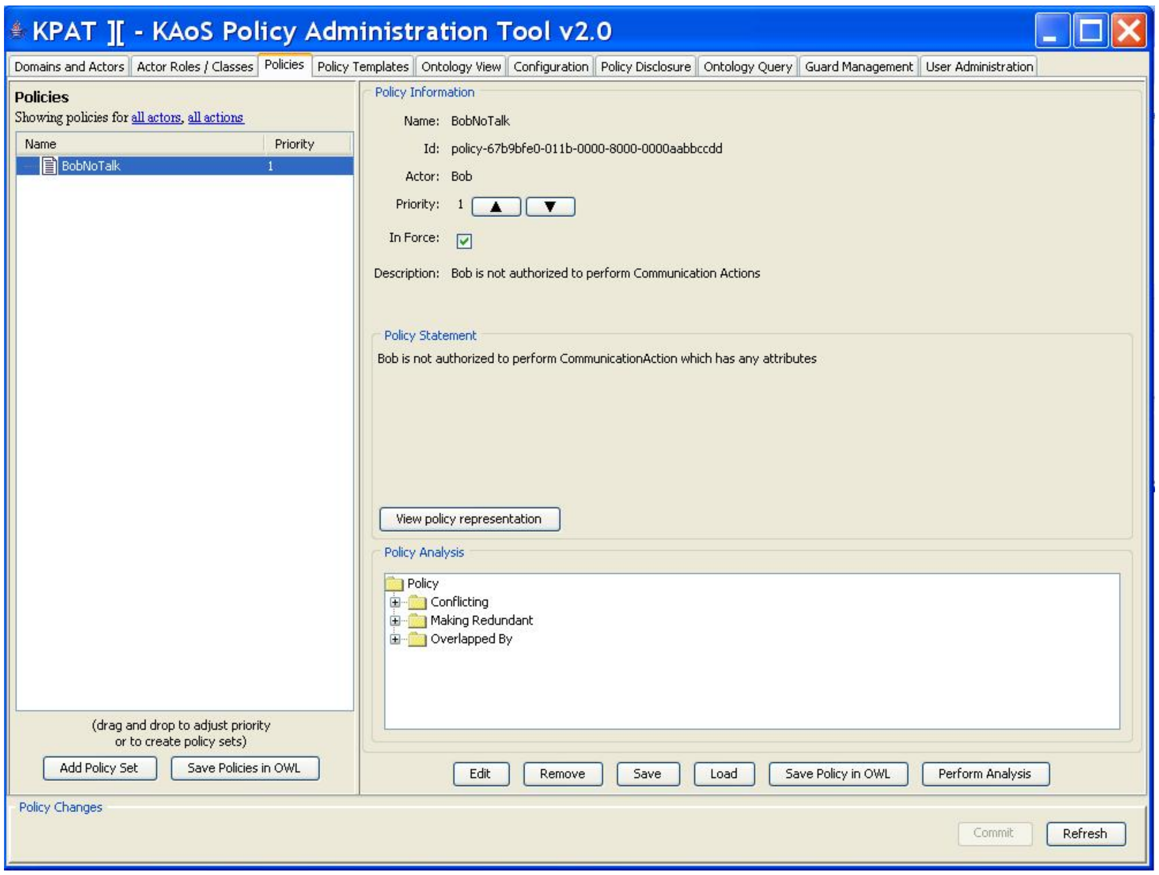
This policy always applies

Policy Statement

Bob is not authorized to perform CommunicationAction which has any attributes

Figure 11 Policy not Authorizing Bob to Perform Communication Actions

The policy is now viewable in KPAT by selecting Bob or going to the Policy tab as shown in Figure 12. KAoS provides a traditional form based version (classic editor) of the policy editor if you are uncomfortable with the hypertext version.



Policies

Showing policies for all actors, all actions

Name	Priority
BobNoTalk	1

(drag and drop to adjust priority
or to create policy sets)

Policy Information

Name: BobNoTalk
Id: policy-67b9bfe0-011b-0000-8000-0000aabbccdd
Actor: Bob
Priority: 1
In Force: ✓
Description: Bob is not authorized to perform Communication Actions

Policy Statement

Bob is not authorized to perform CommunicationAction which has any attributes

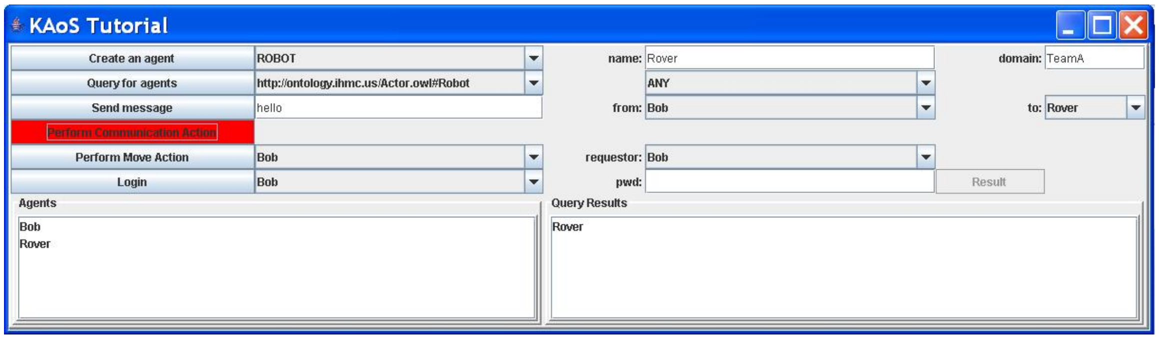
Policy Analysis

- Policy
  - Conflicting
  - Making Redundant
  - Overlapped By

Policy Changes

Figure 12 Policy Tab Showing BobNoTalk Policy

Now using the tutorial GUI and the same parameters, press “Perform Communication Action”. You should see the button turn red () indicating that the action was not authorized. When a policy check determines an action was not authorized, it throws a *KAoS Security Exception*. Our tutorial GUI catches this exception and turns the button red accordingly. The security exception can provide a lot of useful information about the violation, but the details are deferred until the CSI Policy discussion in chapter 0.



Create an agent	ROBOT	name:	Rover	domain:	TeamA
Query for agents	http://ontology.ihmc.us/Actor.owl#Robot		ANY		
Send message	hello	from:	Bob	to:	Rover
Perform Communication Action					
Perform Move Action	Bob	requestor:	Bob		
Login	Bob	pwd:			

Agents

Bob
Rover

Query Results

Rover

Figure 13 Action Not Authorized

4 Policy Reasoning Examples

Now that you have a working policy system, we will build on the example from chapter 3 to create increasingly more complex policies. The goal of this section is to demonstrate the type of reasoning our system can do and highlight classes of policies with functional demonstrations.

We will start by going to the tutorial script in *your_kaos_root* \kaos\scripts\tutorial and type "ant". After KPAT and the tutorial GUI start, create three agents:

1) Bob – a human on TeamA
2) RoverA – a robot on TeamA
3) RoverB – a robot on TeamB

KPAT should look like Figure 14 Three actors with Two Teams in KPAT. This will give us sufficient structure to demonstrate several policy ideas.

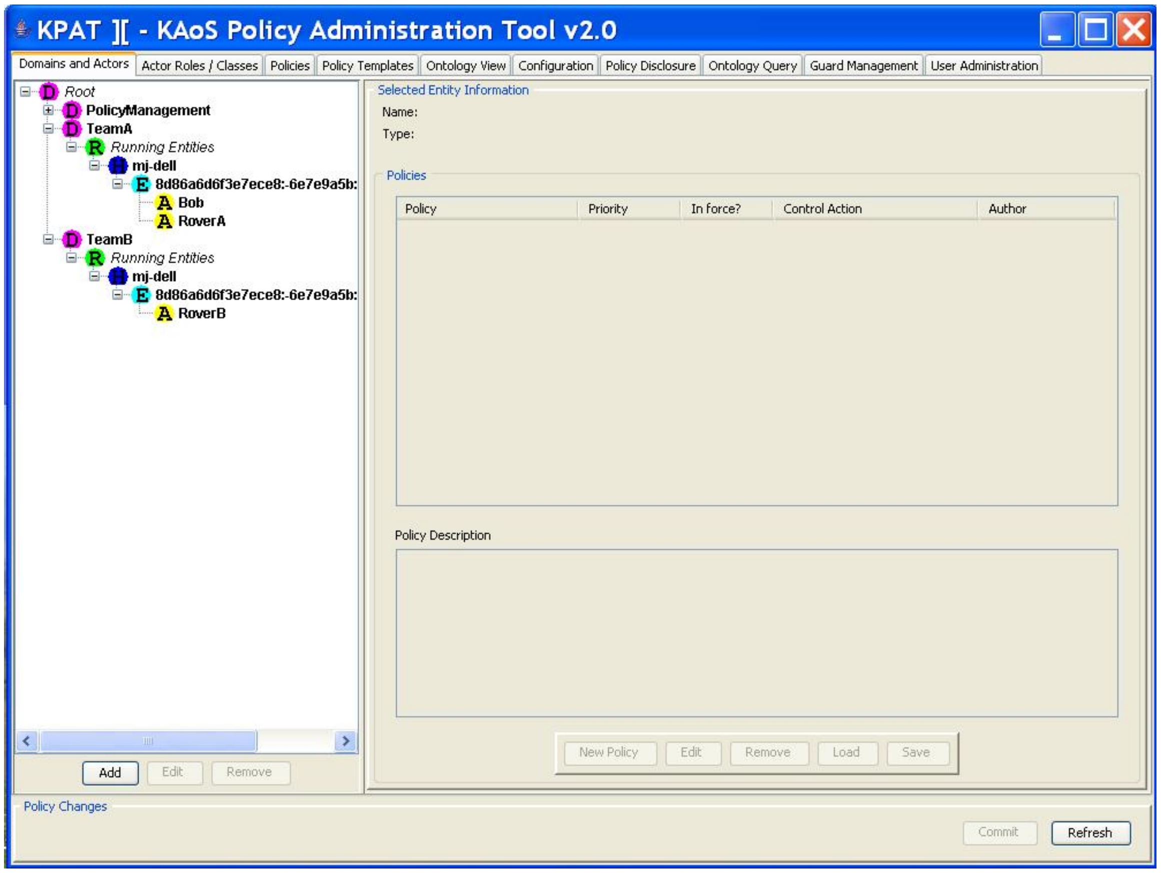
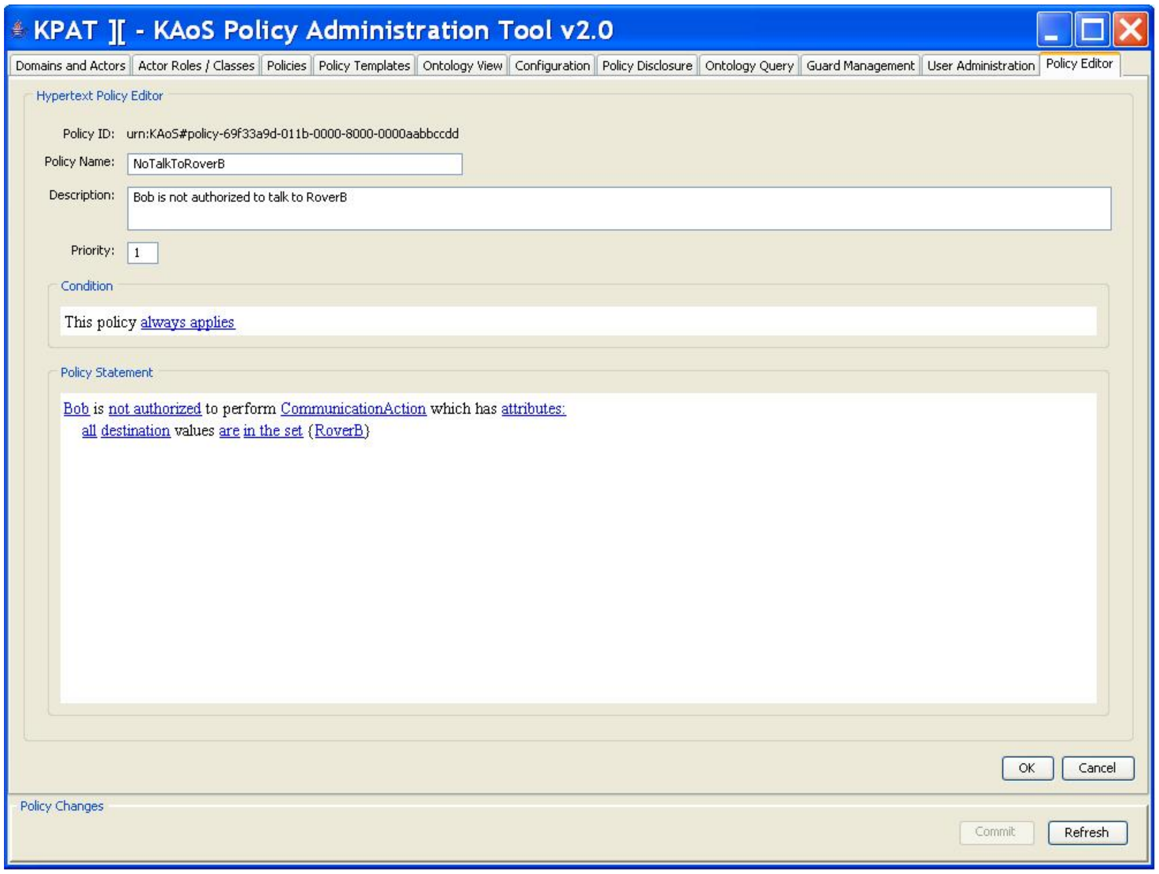


Figure 14 Three actors with Two Teams in KPAT

4.1 Reasoning over Action Properties

The first example is adding properties to the action. When you add properties to an action, you are basically adding context on which you can reason about. Taking our first example policy, we stated that Bob was not authorized to perform Communication Actions. We may have wanted to restrict Bob completely, but we may only have wanted to restrict Bob from communicating with RoverB. We will create a policy in the same way as in section 3.14, except we will add an attribute. Click on “any attribute” and select “Add attribute”. Now you should see a property add below the original text. Click on the property and select “destination”. This is the hasDestination property we described in section 0. You should no see values are [Select…]. Click on “Select” and choose “in the set”. This allows you to view instances. Select “RoverB” from the list. The policy now reads that Bob is not authorized to perform a Communication Action which has the attribute of all destination values in the set of RoverB. This sounds a little strange, but the awkwardness comes from the vagueness inherent in our language. The policy in fact states our intent which is that Bob is not authorized to perform a Communication Action that has the destination of RoverB. You can add multiple properties to an action and the properties can themselves have properties, but we will keep it simple to start. You can test this policy using the tutorial GUI. Bob should not be allowed to perform a Communication Action with RoverB as the receiver. All other communication, including RoverB performing a Communication Action with Bob as the receiver, should be authorized.



Policy ID: urn:KAoS#policy-69f33a9d-011b-0000-8000-0000aabbccdd

Policy Name: NoTalkToRoverB

Description: Bob is not authorized to talk to RoverB

Priority: 1

This policy always applies

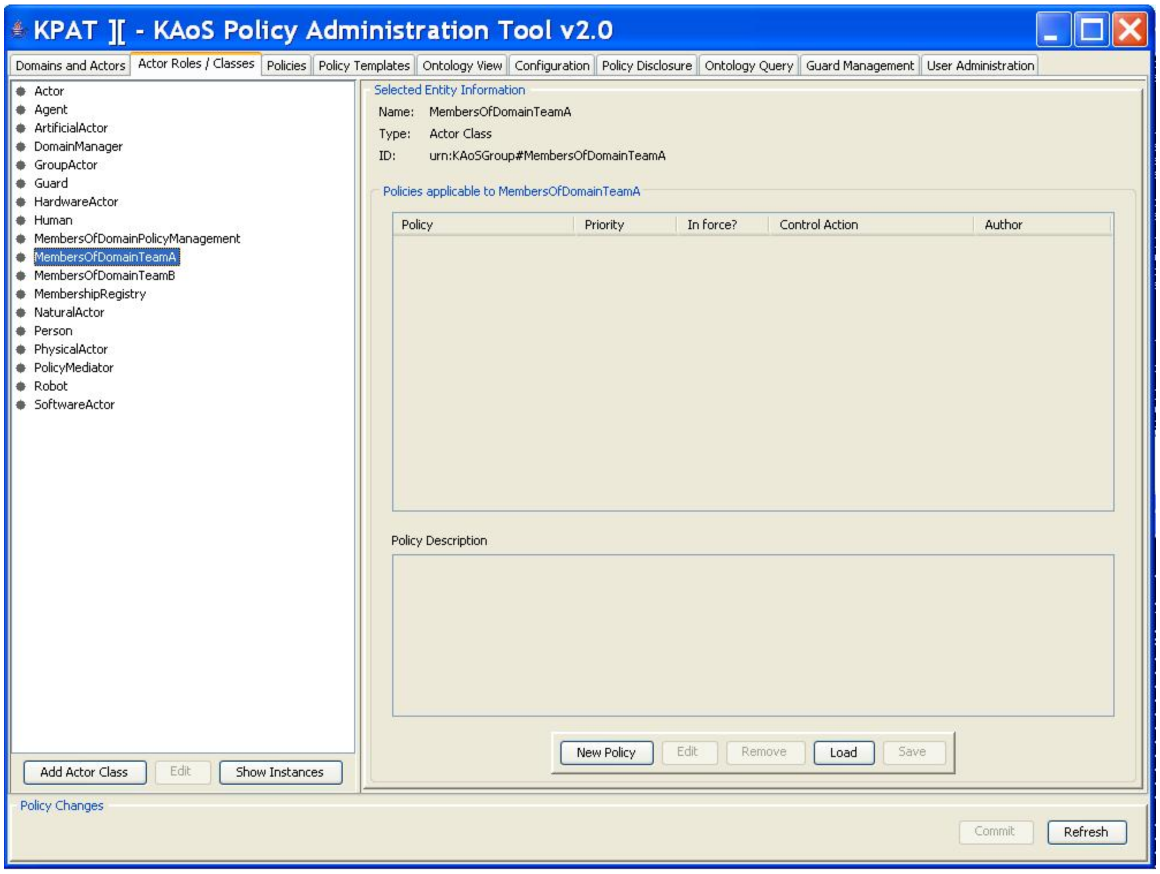
Bob is not authorized to perform CommunicationAction which has attributes:
all destination values are in the set {RoverB}

Figure 15 Bob is not authorized to Communicate with RoverB

4.2 Reasoning over Domains

In the previous policy we used instances (Bob and RoverB). While it is nice to be able to address instances specifically, it is also handy to reference groups (or domains) for efficiency and generality. Let’s say that we wanted to restrict all members on TeamA from communicating with all members on TeamB.

Start by removing the previous policy if it exists. This can be done by going to the “Policy” tab, selecting the policy and pressing the remove button. Don’t forget to commit the change. Now go to the “Actor Roles/Classes” tab and select “MembersOfDomainTeamA” as in Figure 16.



Selected Entity Information

Name: MembersOfDomainTeamA
Type: Actor Class
ID: urn:KAoSGroup#MembersOfDomainTeamA

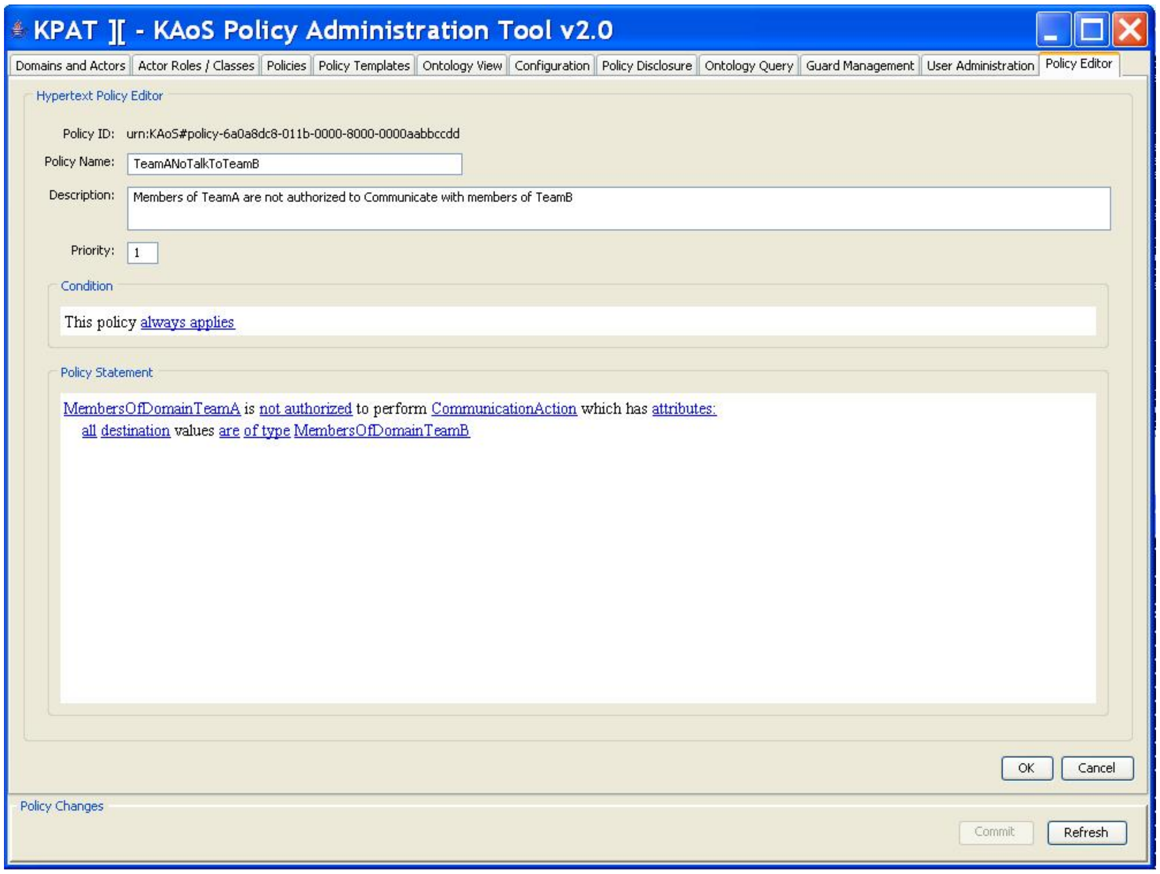
Policies applicable to MembersOfDomainTeamA

Policy	Priority	In force?	Control Action	Author

Policy Description

Figure 16 Actor Roles/Classes Tab in KPAT

We create the policy just like the previous one, except for values we select “of type” to view the classes and then select “MembersOfDomainTeamB”. When done the policy should look like Figure 17. After committing the policy, you can test it with the tutorial GUI. You should not be able to perform a Communication action between members of TeamA and members of TeamB. You will be able to send message from Bob to RoverA because they are both members of TeamA.



Policy ID: urn:KAoS#policy-6a0a8dc8-011b-0000-8000-0000aabbccdd

Policy Name: TeamANoTalkToTeamB

Description: Members of TeamA are not authorized to Communicate with members of TeamB

Priority: 1

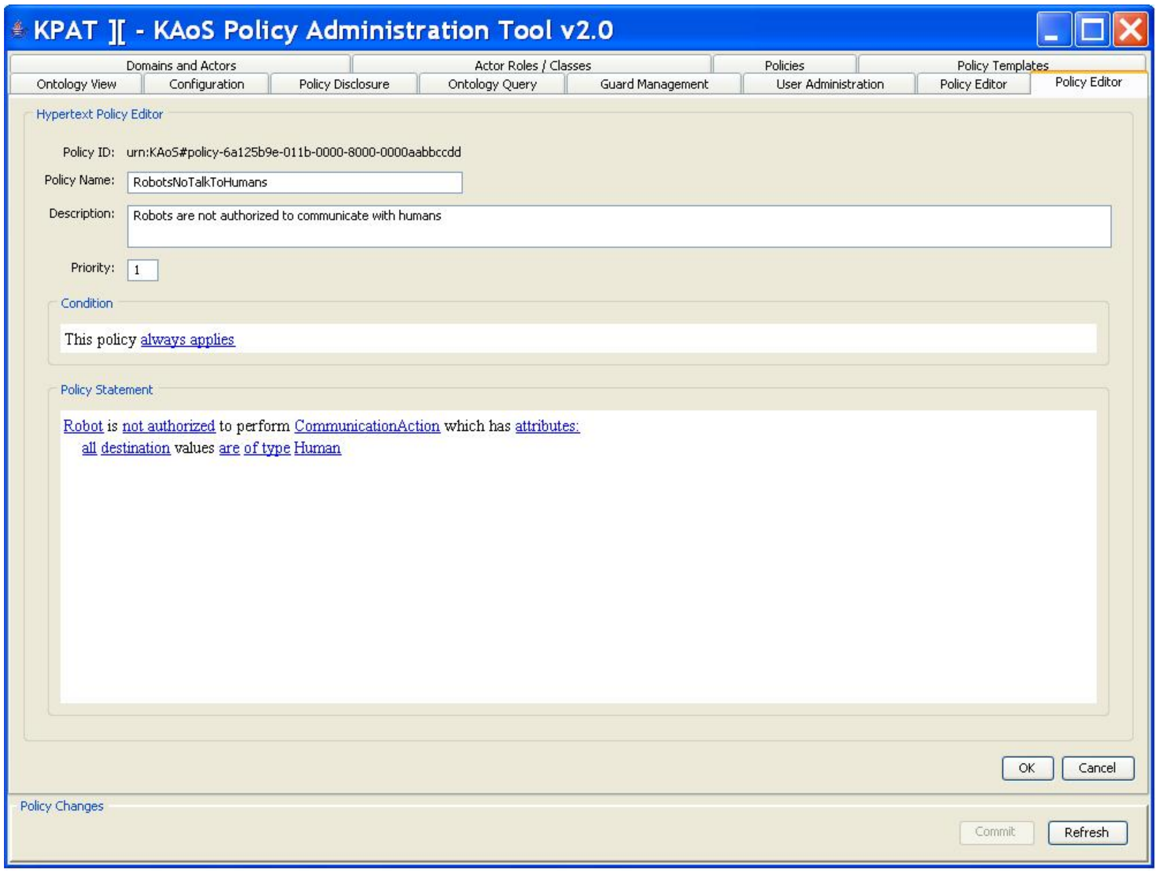
This policy always applies

MembersOfDomainTeamA is not authorized to perform CommunicationAction which has attributes:
all destination values are of type MembersOfDomainTeamB

Figure 17 TeamA is not Authorized to Communicate with TeamB

4.3 Reasoning over Ontological Types

Similar to domains, ontological types allow use to generalize our policies. For this example we will state that Robots are not authorized to communicate with humans. Again, remember to remove any old policies and commit the change. Go to the “Actor Roles/Classes” tab and select “Robot”. Create a new policy similar to the last, except select “Human” instead of “MembersOfDomainTeamB”. The policy should look like Figure 18. You can test this policy with the tutorial GUI. Neither RoverA nor RoverB should be able to perform a Communication Action with Bob as the destination.



Hypertext Policy Editor

Policy ID: urn:KAoS#policy-6a125b9e-011b-0000-8000-0000aabbccdd

Policy Name: RobotsNoTalkToHumans

Description: Robots are not authorized to communicate with humans

Priority: 1

Condition

This policy always applies

Policy Statement

Robot is not authorized to perform CommunicationAction which has attributes:
all destination values are of type Human

Figure 18 Robots are not Authorized to Communicate with Humans

4.4 Additional Thoughts on Reasoning

The most difficult part of creating policies is determining how to model the policy itself. For example, in section 4.1 we had a policy that restricted Bob from communicating with RoverB, but we probably also want the reciprocal to be true. This could be modeled with a second policy. Figuring out what to model as a domain, what to model as an ontological type and what to model as a property can be challenging. Like all things in life, the best way to learn is by doing; so dive in and try a few things out!

5 Obligation Policies

This section will demonstrate how to build the second major type of policy; the obligation. The previous examples were all authorization policies that prohibited certain actions. Obligations require an actor or group of actors to perform some action based on an associated condition, which we call the trigger. The basic format is similar to the authorization policy except that it includes a trigger action. Let’s say we wanted our robot to beep before it moves, to warn people in the area. For example:

Robot	is	obligated	to	beep	before it moves
actor		modality		action	trigger condition

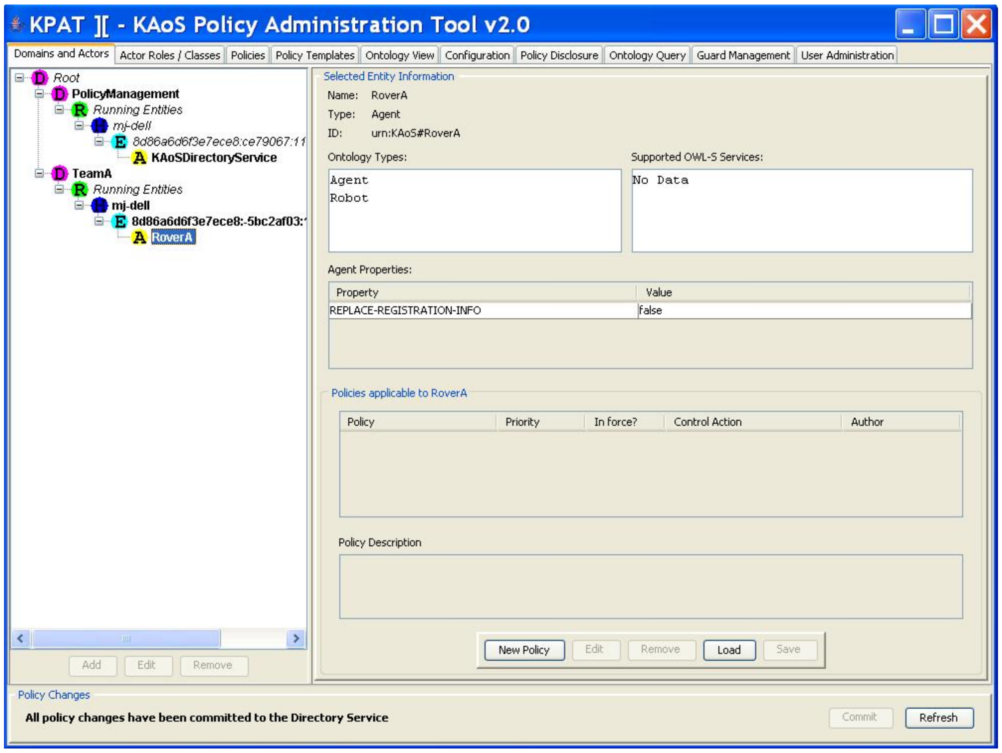
The main portion of this policy is very similar to that of an authorization policy. It has an actor (Robot) which can be an individual or group. It also has a modality (obligated) and an action (beep). In this case the action has no properties (context), but you can apply properties as we did for authorization policies. The last portion of the policy is the main difference. The trigger condition determines when this policy applies. Triggers typically have a temporal relation to the obligated action and we have provided terms to describe that relation. You can select whether you want the obligation to occur before or after the trigger action. Since actions are finite in time, you can also specify whether the before or after refers to the beginning or end of the trigger action. You can also specify whether the obligation must start or be completed by the specified time. This allows for several possibilities:

1) Start obligation before trigger starts
2) Start obligation after trigger starts
3) Start obligation before trigger finishes
4) Start obligation after trigger finishes
5) Finish obligation before trigger starts
6) Finish obligation after trigger starts
7) Finish obligation before trigger finishes
8) Finish obligation after trigger finishes

These timing mechanisms can be used to help clarify the precise meaning of the obligation. For example, it would not make sense to beep after the movement has been completed, since the purpose of the obligation is to warn people of the movement. To make the policy clear we would specify to finish the beep action before starting the move action. This produces the desired result. The last thing to note about the trigger condition is that the trigger action can also have properties that help define context, just like the obligated action and authorization policy action.

We will now walk through the steps of creating and testing an obligation policy. We will use the same tutorial code as before, but add a few pieces to make the obligation demonstration complete. We will start by going to the tutorial script in *your_kaos_root* \kaos\scripts\tutorial and type “ant”. After KPAT and the tutorial GUI start, create one agent (Figure 19):

RoverA – a robot on TeamA



Name: RoverA
Type: Agent
ID: urn:KAoS#RoverA

Ontology Types:

```
Agent
Robot
```

Supported OWL-S Services:

```
No Data
```

Agent Properties:

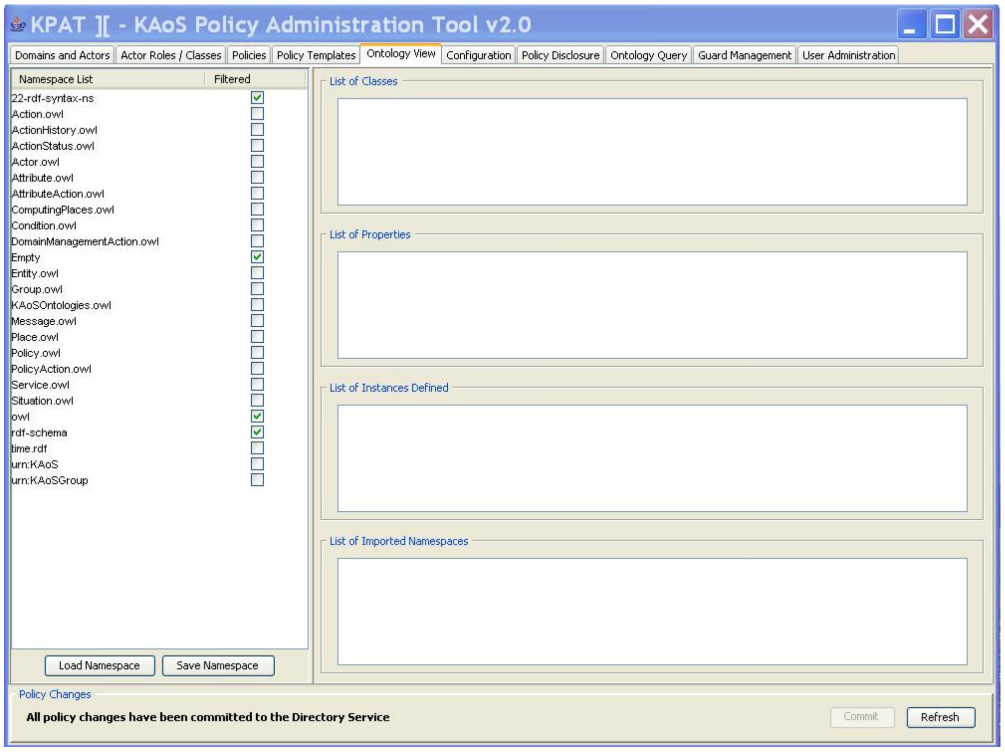
Property	Value
REPLACE-REGISTRATION-INFO	false

All policy changes have been committed to the Directory Service

Figure 19 KPAT with Just one Agent

5.1 Extending the KAoS Ontology

KAoS provides a core ontology that provides the basic terms and relations necessary to build policies. To make an interesting and simple obligation, we will extend the KAoS Core Ontology with the KAoS Robot Ontology (http://ontology.ihmc.us/Robot/index.php). This ontology defines some simple robot actions (like beep and move) among other things. To load the ontology, make sure you are connected to the Internet and go to the “Ontology View” tab in KPAT (Figure 20).

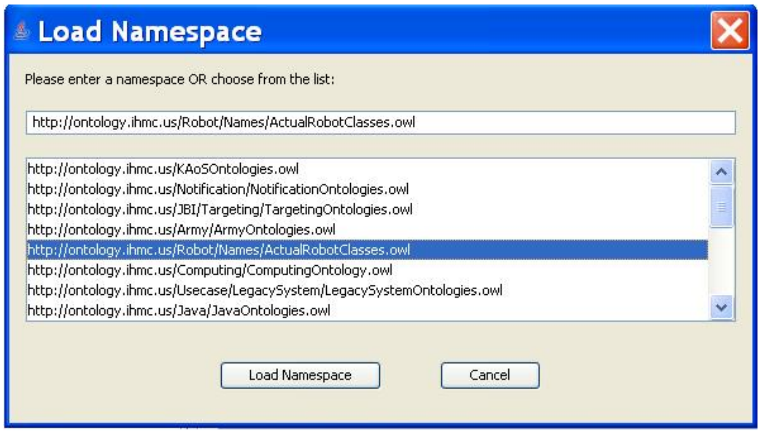


Namespace List	Filtered
22-rdf-syntax-ns	✓
Action.owl	
ActionHistory.owl	
ActionStatus.owl	
Actor.owl	
Attribute.owl	
AttributeAction.owl	
ComputingPlaces.owl	
Condition.owl	
DomainManagementAction.owl	
Empty	✓
Entity.owl	
Group.owl	
KAoSOntologies.owl	
Message.owl	
Place.owl	
Policy.owl	
PolicyAction.owl	
Service.owl	
Situation.owl	
owl	✓
rdf-schema	✓
time.rdf	
urn:KAoS	
urn:KAoSGroup	

All policy changes have been committed to the Directory Service

Figure 20 Ontology View Tab

Press the "Load Namespace" button. You will see a small dialog box. Select (Figure 21).



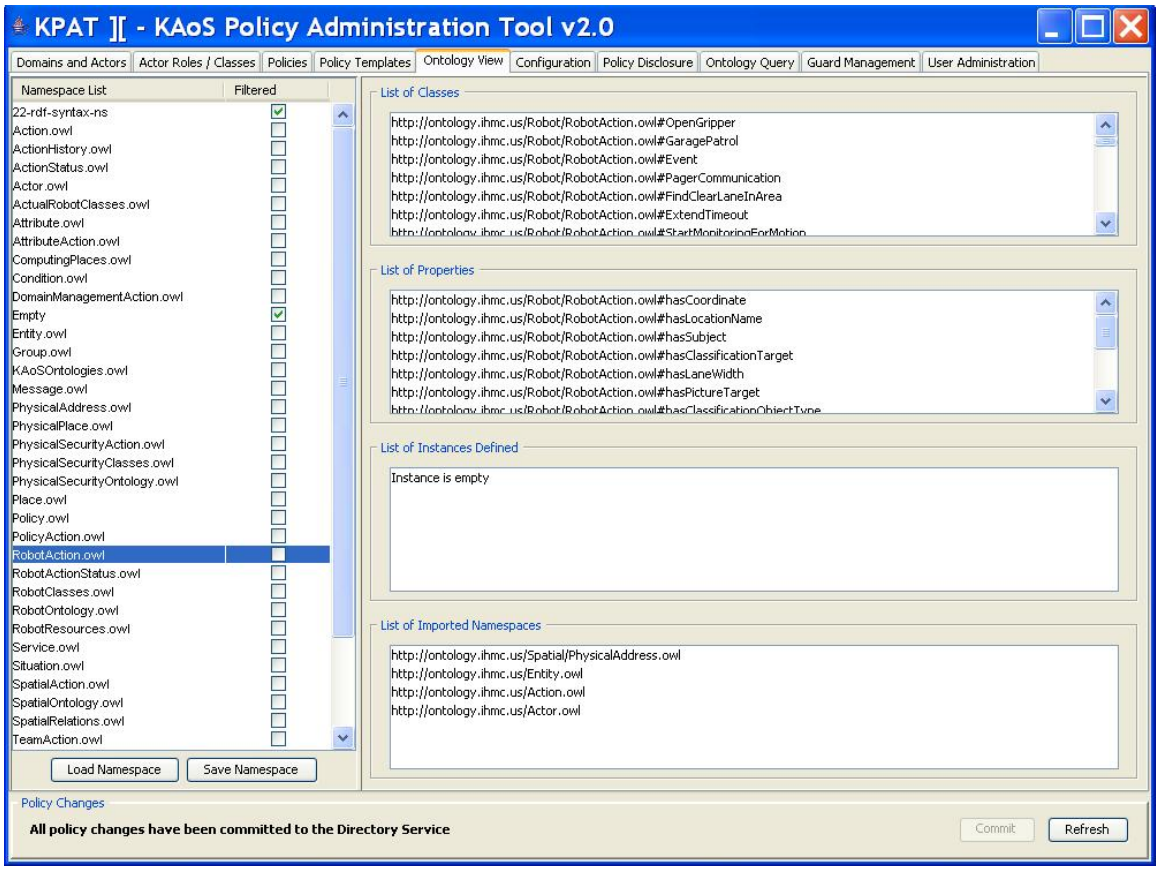
Please enter a namespace OR choose from the list:

http://ontology.ihmc.us/Robot/Names/ActualRobotClasses.owl

http://ontology.ihmc.us/KAoSOntologies.owl
http://ontology.ihmc.us/Notification/NotificationOntologies.owl
http://ontology.ihmc.us/JBI/Targeting/TargetingOntologies.owl
http://ontology.ihmc.us/Army/ArmyOntologies.owl
http://ontology.ihmc.us/Robot/Names/ActualRobotClasses.owl
http://ontology.ihmc.us/Computing/ComputingOntology.owl
http://ontology.ihmc.us/Usecase/LegacySystem/LegacySystemOntologies.owl
http://ontology.ihmc.us/Java/JavaOntologies.owl

Figure 21 Load Namespace Dialog

After a few seconds the KAoS Robot Ontology should be loaded. You will see additional classes in the view. You should see RobotAction.owl and if you select it you will see the various actions on the right side of KPAT(Figure 22).



KPAT][- KAoS Policy Administration Tool v2.0

Domains and Actors | Actor Roles / Classes | Policies | Policy Templates | Ontology View | Configuration | Policy Disclosure | Ontology Query | Guard Management | User Administration

Namespace List	Filtered
22-rdf-syntax-ns	✓
Action.owl	
ActionHistory.owl	
ActionStatus.owl	
Actor.owl	
ActualRobotClasses.owl	
Attribute.owl	
AttributeAction.owl	
ComputingPlaces.owl	
Condition.owl	
DomainManagementAction.owl	
Empty	✓
Entity.owl	
Group.owl	
KAoSOntologies.owl	
Message.owl	
PhysicalAddress.owl	
PhysicalPlace.owl	
PhysicalSecurityAction.owl	
PhysicalSecurityClasses.owl	
PhysicalSecurityOntology.owl	
Place.owl	
Policy.owl	
PolicyAction.owl	
RobotAction.owl	
RobotActionStatus.owl	
RobotClasses.owl	
RobotOntology.owl	
RobotResources.owl	
Service.owl	
Situation.owl	
SpatialAction.owl	
SpatialOntology.owl	
SpatialRelations.owl	
TeamAction.owl	

Load Namespace | Save Namespace

List of Classes

http://ontology.ihmc.us/Robot/RobotAction.owl#OpenGripper
http://ontology.ihmc.us/Robot/RobotAction.owl#GaragePatrol
http://ontology.ihmc.us/Robot/RobotAction.owl#Event
http://ontology.ihmc.us/Robot/RobotAction.owl#PagerCommunication
http://ontology.ihmc.us/Robot/RobotAction.owl#FindClearLaneInArea
http://ontology.ihmc.us/Robot/RobotAction.owl#ExtendTimeout
http://ontology.ihmc.us/Robot/RobotAction.owl#StartMonitoringForMotion

List of Properties

http://ontology.ihmc.us/Robot/RobotAction.owl#hasCoordinate
http://ontology.ihmc.us/Robot/RobotAction.owl#hasLocationName
http://ontology.ihmc.us/Robot/RobotAction.owl#hasSubject
http://ontology.ihmc.us/Robot/RobotAction.owl#hasClassificationTarget
http://ontology.ihmc.us/Robot/RobotAction.owl#hasLaneWidth
http://ontology.ihmc.us/Robot/RobotAction.owl#hasPictureTarget
http://ontology.ihmc.us/Robot/RobotAction.owl#hasClassificationObjectType

List of Instances Defined

Instance is empty

List of Imported Namespaces

http://ontology.ihmc.us/Spatial/PhysicalAddress.owl
http://ontology.ihmc.us/Entity.owl
http://ontology.ihmc.us/Action.owl
http://ontology.ihmc.us/Actor.owl

Policy Changes

All policy changes have been committed to the Directory Service

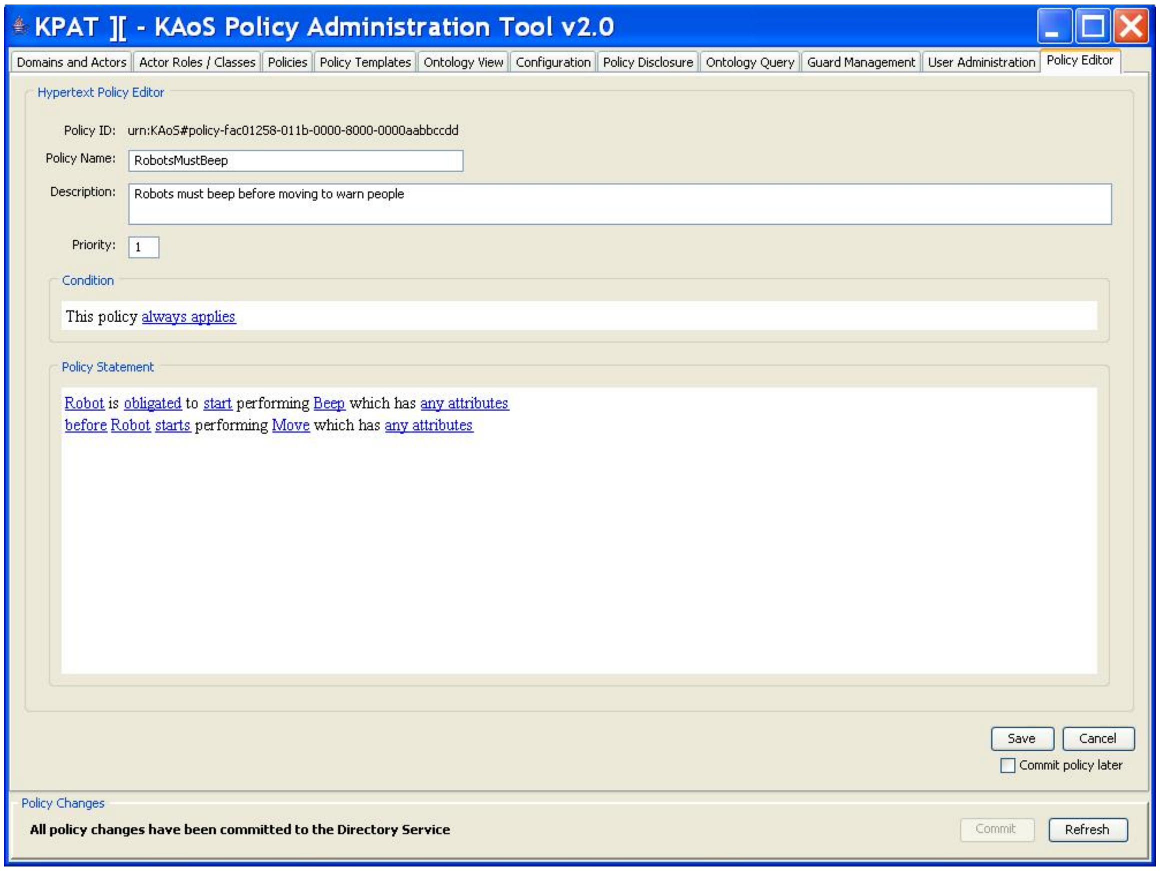
Commit | Refresh

Figure 22 Ontology View Tab after KAoS Robot Ontology is loaded

This was just a brief description of how to extend the KAoS Core Ontology. A full description can be found in chapter 12.

5.2 Creating an Obligation Policy

Now that we have the necessary terms to create our first obligation lets get started. You should still have your one agent running and the KAoS Robot Ontology loaded from the previous sections in this chapter. Go to the “Actor Roles/Domains” tab and select “Robot”, since we want this to apply to all robots. Now press the “New Policy” button and chose hypertext editor. Give the policy a name, a description and a priority of 1. Select the modality of “obligated.” You will see the trigger condition template added on the line below. Select the action of “Beep.” You will notice that more actions are available since we loaded an additional ontology. All of these actions are applicable to things that are from the robot class. Now in the trigger action select the actor to be the class of “Robot” and the action to be “Move.” We will not use any action properties for this simple example. When you are done, the policy should look like the one shown in Figure 23.



Policy ID: urn:KAoS#policy-fac01258-011b-0000-8000-0000aabbccdd

Policy Name: RobotsMustBeep

Description: Robots must beep before moving to warn people

Priority: 1

Condition

This policy always applies

Policy Statement

Robot is obligated to start performing Beep which has any attributes
before Robot starts performing Move which has any attributes

☐ Commit policy later

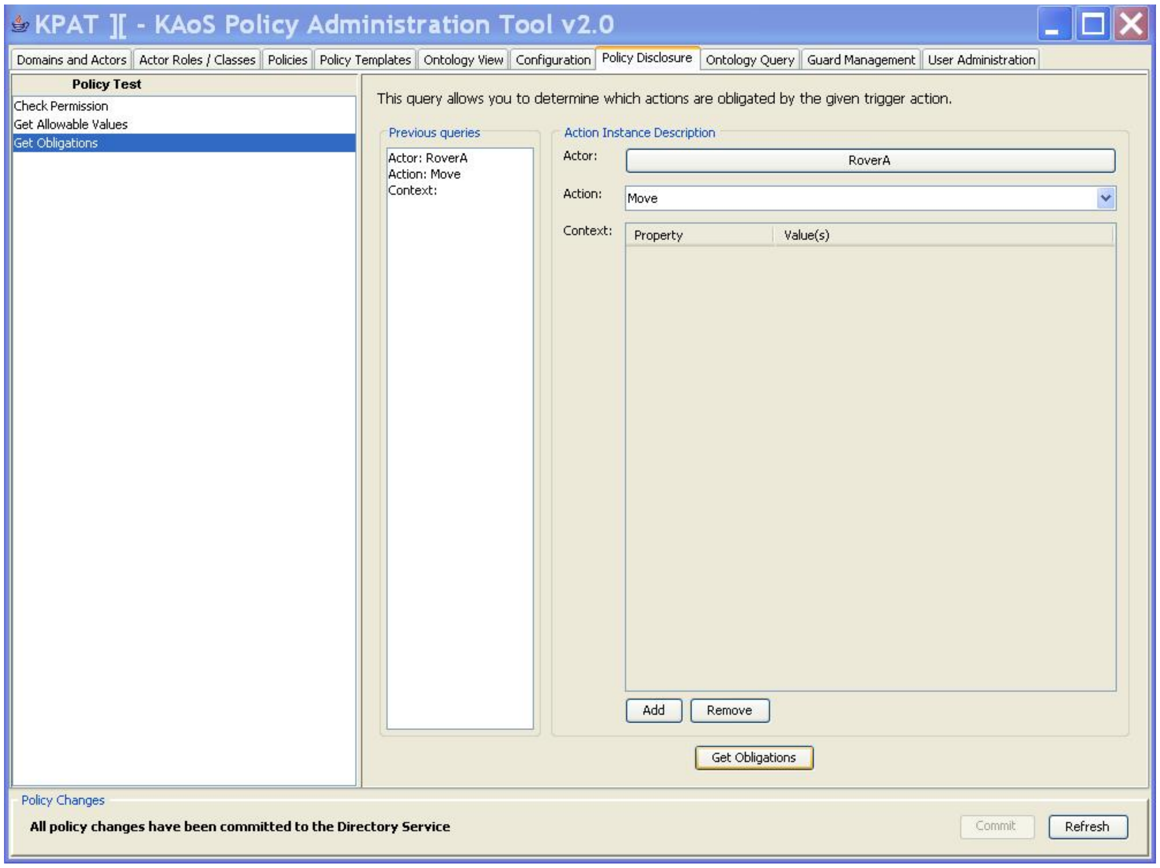
Policy Changes

All policy changes have been committed to the Directory Service

Figure 23 Obligation Policy Example

5.3 Testing Obligations

A simple way to test your obligation is using KPAT’s Policy Disclosure Tab. Go to the tab, select “Get Obligations” from the list of the left side, select “RoverA” as the actor on the right side, and select “Move” as the action on the right side. KPAT should now look like Figure 24.



This query allows you to determine which actions are obligated by the given trigger action.

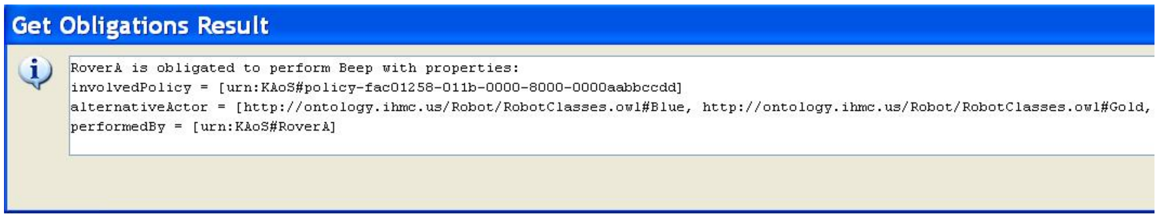
Actor: RoverA
Action: Move
Context:

Actor: RoverA
Action: Move
Context: Property | Value(s)

All policy changes have been committed to the Directory Service

Figure 24 Policy Disclosure for Beep before Moving Obligation

Now press “Get Obligations” and you should see the obligation to beep as shown in Figure 25.


Get Obligations Result

```
RoverA is obligated to perform Beep with properties:
involvedPolicy = [urn:KAoS#policy-fac01258-011b-0000-8000-0000aabbccdd]
alternativeActor = [http://ontology.ihmc.us/Robot/RobotClasses.owl#Blue, http://ontology.ihmc.us/Robot/RobotClasses.owl#Gold,
performedBy = [urn:KAoS#RoverA]
```

Figure 25 Obligation Disclosure Results for Beep before Moving Obligation

5.4 Implementing Enforcement for Obligations

Use the same method used for checking authorizations to check for obligations. You just call the *enforcePolicies* method which does the following:

```
PolicyChecking policyChecking = CSIFactory.getPolicyChecking();
Vector<ActionInstanceDescription> obligations = policyChecking.findPolicyDecision(aid, null);
for(ActionInstanceDescription action: obligations)
{
   performAction(action);
}
```

We learned in chapter 3.12 that this method throws a *KAoSSecurityException* if the action is not authorized. It also executes any obligated actions using Java reflection. There are many ways to provide enforcement, which we discuss in chapter 10. In this example we are providing our own enforcement by checking the action and executing any obligations automatically. The main issue with obligations is how to execute obligations. Authorizations only require the enforcer to be able to prevent actions. Obligations must map down to concrete implementation in the end. For our example, we want to be able to make a robot beep, or more specifically execute a *Beep* method on our *SimpleAgentRobot* class. We use Java reflection to accomplish this task. The code to perform the reflection is in the *performAction* method. It requires the method name to match the ontological name. It also would require the action properties to be consistent with the method parameters, but we will not be using any properties for this first example.

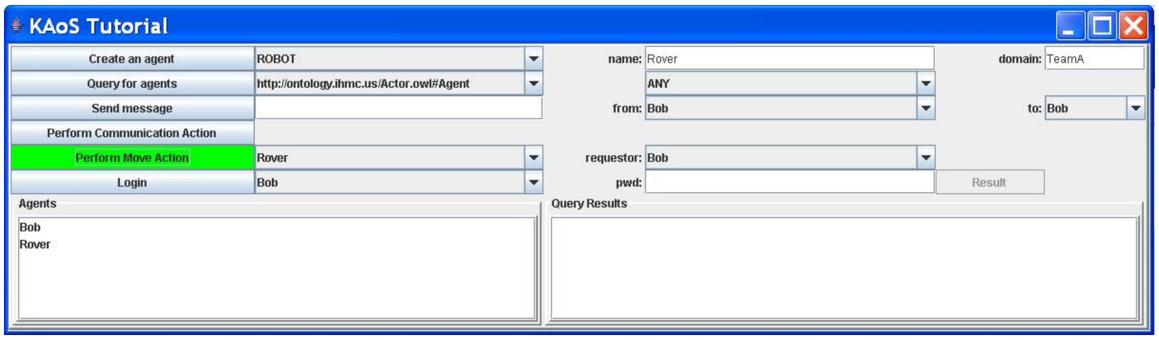
Now if you select rover in the drop down menu and press the "Perform Move Action" button is press (Figure 26), you will see a notification about the action in the command window like this:

```
[java] RoverA: moving
```

If the obligation policy from section 5.2 is in force, you will see the beep action occur before the move action like this:

```
[java] RoverA: BEEP!
[java] RoverA: moving
```

You now have a working obligation! It is important to note that this enforcement example is very primitive and just designed to get you started. It did not take into account the information about timing and sequencing, even though it was available. The Beep came before the move simply because it was coded to perform obligations before executing the trigger action. There are more advanced ways to perform enforcement, but they are not covered by this introductory example.



ROBOT
http://ontology.ihmc.us/Actor.owl#Agent
Rover
Bob

name: Rover
ANY
from: Bob
requestor: Bob
pwd:

domain: TeamA
to: Bob

Bob
Rover

Figure 26 Perform Move Action

6 Policy Prioritization

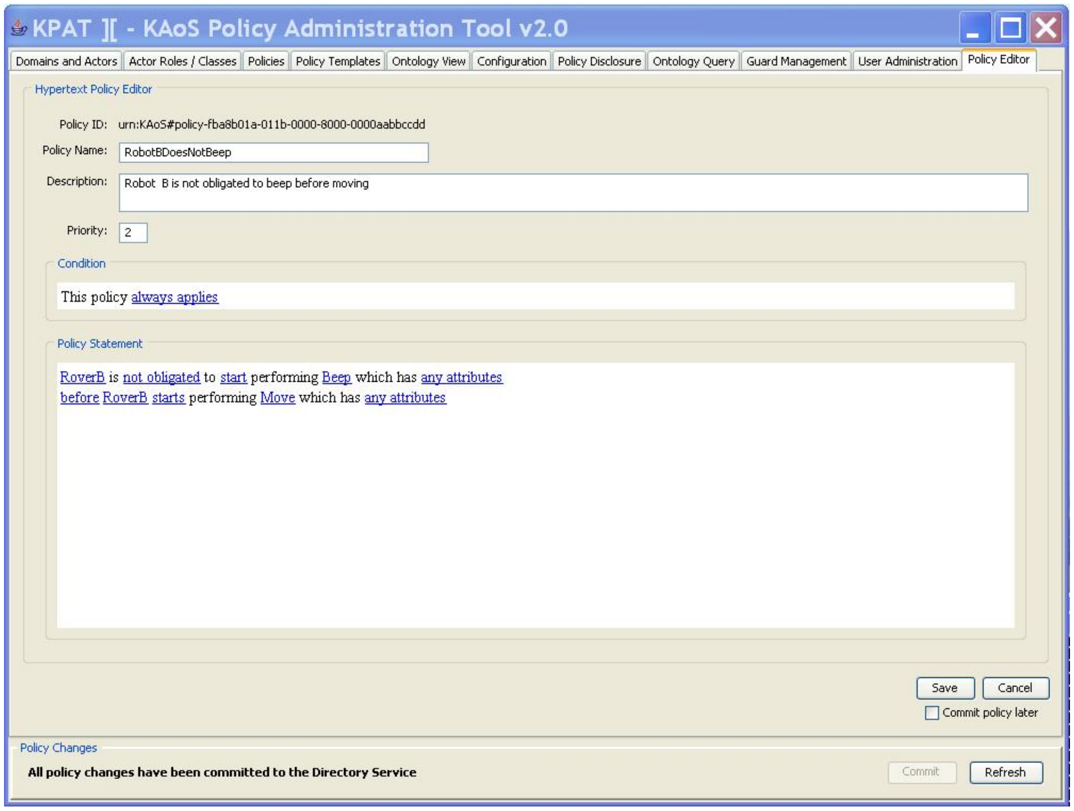
It is very likely that if you make enough policies, they will eventually conflict in some ways. It is also very common, to desire to add an exception to a general rule. When policies conflict in some way, there must be a mechanism to determine which policy has precedence. KAoS provides a few ways to handle policy prioritization.

6.1 Priority integer

The simplest way is to use the priority integer associated with each policy. Although this way is not very elegant, it is very simple and is often sufficient. The higher integer value policy takes precedence over the lower integer value policy. Let's do an example using the basic setup in chapter 4 with two teams each having one robot. We will then load the ontology and add the general obligation for robots to beep before moving as in chapter 0. Now when each robot is tasked to move, you should see the beep message:

```
[java] RoverA: BEEP!
[java] RoverA: moving
[java] RoverB: BEEP!
[java] RoverB: moving
```

Now consider that RoverB is far away from any people and we would like to conserve power by not forcing it to beep unnecessarily. Let's create a second policy that states that RoverB is not obligated to beep before moving. First go to the "Domains and Actors" tab and select "RoverB." Now press the "New Policy" button and create a similar obligation as the first. The new policy should refer to RoverB instead of all Robots and it should be a negative obligation (not obligated) instead of a positive one. Also make sure the priority is 2 instead of the priority of 1 used on the first policy. The final policy should look like Figure 27.



Policy ID: urn:KAoS#policy-fba8b01a-011b-0000-8000-0000aabbccdd
Policy Name: RobotBDoesNotBeep
Description: Robot B is not obligated to beep before moving
Priority: 2

This policy always applies

RoverB is not obligated to start performing Beep which has any attributes
before RoverB starts performing Move which has any attributes

All policy changes have been committed to the Directory Service

Figure 27 Negative Obligation Policy

Now if you go to the Policy tab you should see both policies. You can view there details by selecting either one. When you execute the “Perform Move Action” for each robot you should not get a Beep message from RoverB, because the higher priority policy waived the obligation:

```
[java] RoverA: BEEP!
[java] RoverA: moving
[java] RoverB: moving
```

*** NOTE: The policy priority adjustment arrow buttons on the policy tab do not work reliably. Edit the policy directly to change the priority for now

*** NOTE: The automatic policy conflict detection and deconfliction is disabled until KAoS has been converted to use Pellet.

*** NOTE: We are currently working on mechanisms to allow logical policy precedence constraints to be used as a method of policy prioritization as an alternative to the exclusive use of numeric priorities. This is a much more powerful and scalable approach than using numeric priorities alone.

7 Role-Value Maps

Originally KAoS used only OWL-DL (initially DAML), which had difficulty dealing with situations where it was needed to define policies in which one element of an action's context depended on the value of another part of the current context. Some examples include:

- Loop Communication Action
- Relation to the current location, time, other aspect of the current action instance context
- Relation between Trigger Action and Obliged Action
- Relation between a condition, state or history and the current action

These requirements can be fulfilled by role-value-map semantics (see page 94 in The Description Logic Handbook). Maps allow policy to express equality or containment of values that has been reached through two chains of instance properties. KAoS was equipped with role-value-map semantics to defined policy actions when necessary.

It is often useful to refer to aspects of a policy from within the policy itself. This sort of runtime binding enables the creation of general policies with specific context based application. The example we will use is providing feedback. The idea is that if somebody asks you to do something, you should let them know when it is done. Specifically, our policy will state that if a robot is tasked by a requestor, the robot should notify the requestor when the task is finished. We will connect this with our previous "Beep before you Move" obligation that we applied to robots.

7.1 Creating a Policy Using Role-Value-Maps

Start the tutorial up as in chapter 0, loading in the Robot ontology. Next create two agents on TeamA; Bob a human and Rover a robot. KPAT should now show both agents as in Figure 6. We will now create our policy by going to the *Actor Roles/Classes* tab and selecting *Robot*. Then press the *New Policy* button and select the *hypertext editor*. Name the policy and give it a priority of 1. Now fill in:

```
Robot is obligated to start performing CommunicationAction which has any attributes
```

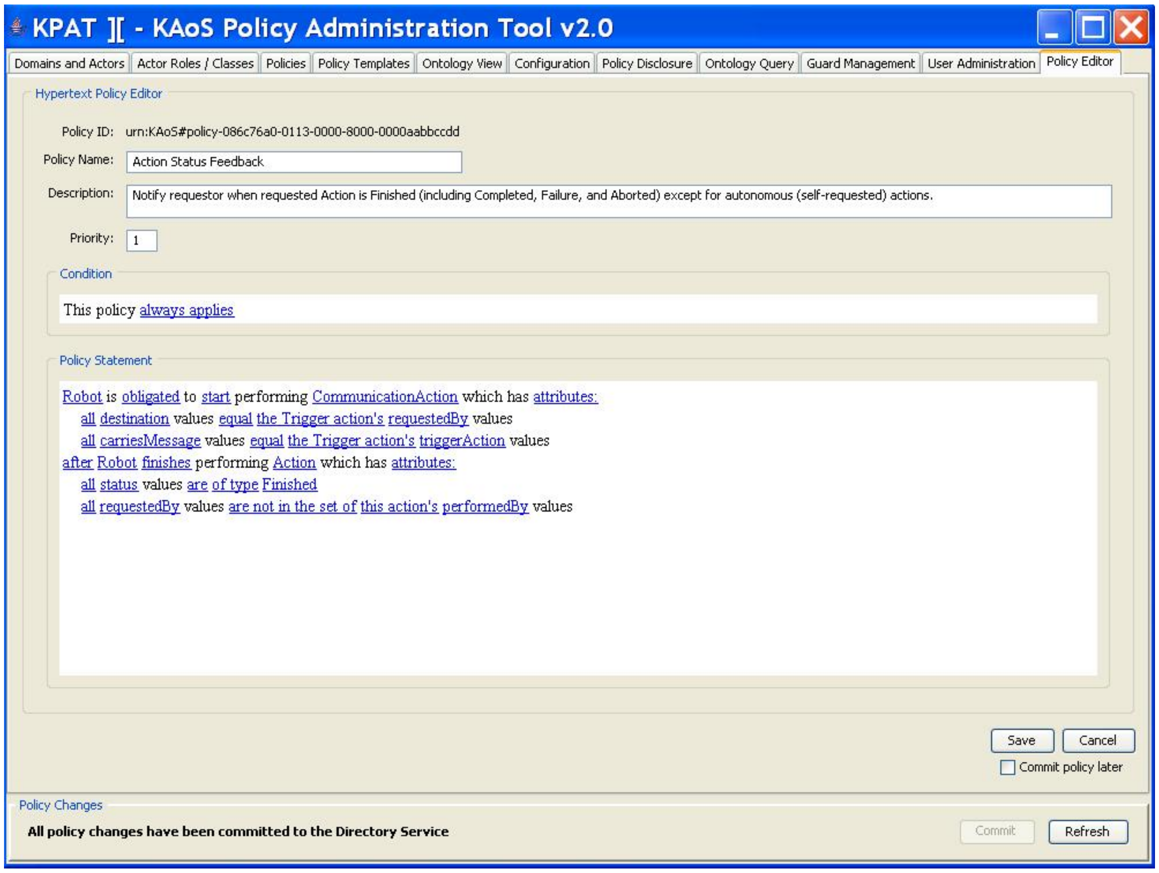
This will create a line for the trigger action. Before we provide attributes for the obligated action (CommunicationAction) we will first add the trigger action as follows:

```
after Robot finishes performing Action which has attributes
all status values are of type Finished
all requestedBy values are not in the set of this action's performedBy values
```

So the trigger action for this obligation is any action, or specifically any action that extends Action from the KAoS ontology. The trigger action must have a status of Finished (which includes Completed and Aborted). Our first direct use of the role-value-map is in the last line above, where we reference the current action's properties. The performedBy property indicates which actor is actually performing the action. The requestedBy property indicates who has requested this action to be performed. What the last line is saying is that requestor should not be the same as the performer. The goal of this policy is to provide feedback to another actor. If the requestor is the performer (i.e. a self generated plan) then there is no reason to provide feedback to oneself. Now that we have the trigger action defined, let's back up to the obligation again and add:

```
Robot is obligated to start performing CommunicationAction which has any attributes
All destination values equal the Trigger action's requestedBy values
All carriesMessage values equal the Trigger action's triggerAction values
```

Here we can see some role-value-mapping again as both properties reference the trigger action. The first says that the destination should be the requestor (i.e. send this feedback to the person who asked you to do the task). The second property says that the message should be the trigger action itself, including the current status. The completed policy should look like Figure 28.



Policy ID: urn:KAoS#policy-086c76a0-0113-0000-8000-0000aabbccdd
Policy Name: Action Status Feedback
Description: Notify requestor when requested Action is Finished (including Completed, Failure, and Aborted) except for autonomous (self-requested) actions.
Priority: 1

This policy always applies

Robot is obligated to start performing CommunicationAction which has attributes:
  all destination values equal the Trigger action's requestedBy values
  all carriesMessage values equal the Trigger action's triggerAction values
after Robot finishes performing Action which has attributes:
  all status values are of type Finished
  all requestedBy values are not in the set of this action's performedBy values

All policy changes have been committed to the Directory Service

Figure 28 Role-Value Maps - Action Status Feedback Policy in KPAT

These six lines of hypertext capture a very powerful concept of feedback in a very general way. It is external to any agent or robot code and visible and accessible to any human operators.

7.2 Role-Value-Map Implementation Changes

There are no general implementation changes to deal with role-value-maps, but our example required two specific modifications. To enforce this new policy, we need add status to an action after it completes and check policies after an action as well. Looking at the *Move* method of *SimpleAgent*:

```
public void Move(String requestor) throws Exception
   {
      // build aid
      HashMap properties = new HashMap();
      properties.put(ActionConcepts.requestedBy(), NamespaceValidator.validateURI(requestor));
      ActionInstanceDescription aid =
              buildActionInstanceDescription("http://ontology.ihmc.us/Robot/Teleoperation.owl#Move", properties);

      // perform enforcement
      enforcePolicies(aid);

      // perform move
```

```
                System.out.println(_actorDesc.getAgentNickname() + ": moving as requested by " + requestor);

                // move is now finished, so check policies again
                OntPropertyDescription status = new OntPropertyDescriptionImpl(ActionStatusConcepts.hasStatus());
                status.setValue(ActionStatusConcepts.DefaultFinishedActionStatus());
                aid.addProperty(status);

                // perform enforcement
                enforcePolicies(aid);
   }
```

What we have added is the section after performing the move where we add a status property that has a value of *DefaultFinishedActionStatus* to the original *Move* action. This strange value is because OWL-DL does not allow classes to be values of property, so we simply define default instances and they have the same effect. Additionally we added another policy check. There are more general ways to architect the policy checking mechanism so that it is not part of every method, but we have done it this way in the tutorial to make it easier to follow along.

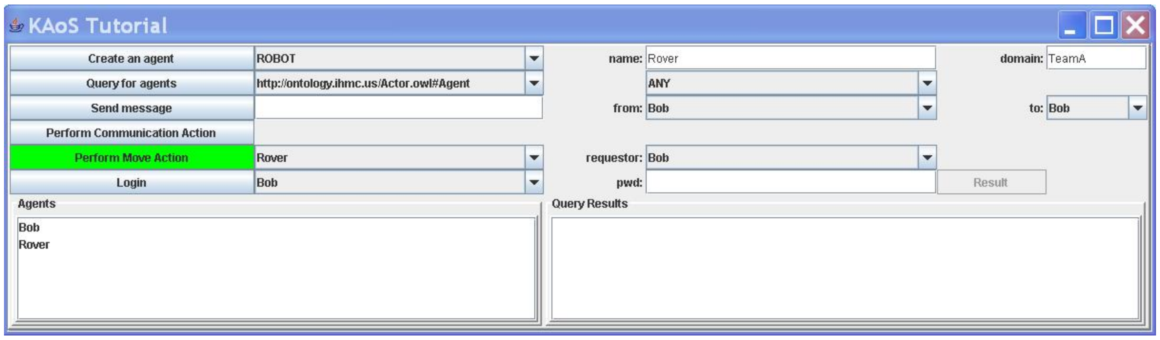
While ontological classes allow for general reasoning over classes and very general policies, they do not provide the mechanics necessary for runtime binding of instances. Role-value-maps extend the ontological class functionality by allowing general policies to have runtime binding.

7.3 Testing the Role-Value-Map Example

With our actors registered and our new policy in place we are ready to test it. In the tutorial GUI, select *Rover* from the dropdown menu next to the *Perform Move* button. Make sure *Bob* is selected in the *requestor* dropdown menu as in Figure 29. Now press the *Perform Move* button. The output should be:

```
[java] Rover: moving as requested by Bob
[java] Rover sent message to urn:KAoS#Bob: Move DefaultFinishedActionStatus
[java] Bob received message from Rover: Move DefaultFinishedActionStatus
```

The first line indicates that the move is being performed and who requested it. The second is the obligation being fulfilled by the robot to provide feedback when done. The third line is the requestor receiving the feedback.


KAoS Tutorial

Create an agent: ROBOT — name: Rover — domain: TeamA
Query for agents: http://ontology.ihmc.us/Actor.owl#Agent — ANY
Send message — from: Bob — to: Bob
Perform Communication Action
Perform Move Action: Rover — requestor: Bob
Login: Bob — pwd: — Result

Agents
Bob
Rover

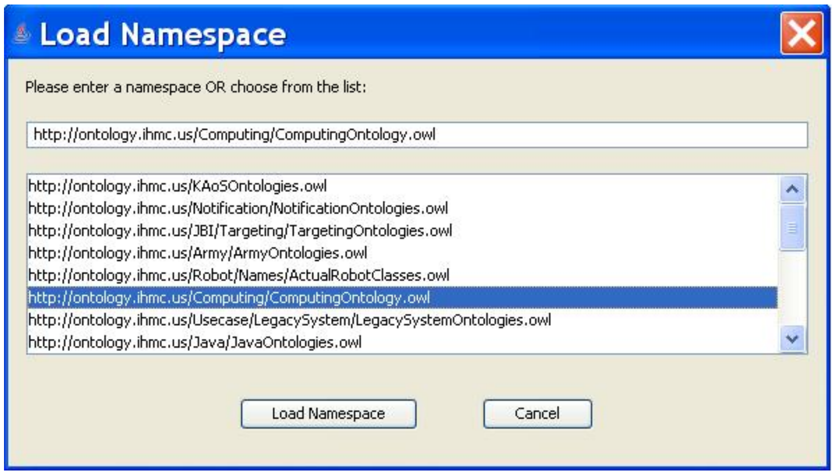
Query Results

Figure 29 Perform Move with Requestor (Tutorial GUI)

8 Reasoning about History and Time

The richness of context is what makes a policy system useful. One important aspect of context that can be very valuable is referring to history or past events. For example, you may want to limit the number of login attempts on a system to be three. This can be represented by a negative authorization policy that has context specifying three failed login attempts. The trouble with representing this is that the action being checked should not be responsible (or trusted) to provide its own history. This means that the fourth attempt would be just another “Login Action” with some credentials. Somewhere, the system must maintain a history of events so that we can reason about the current action in the context of previous ones.

Our example is a simple authorization policy to prevent a third attempt to login if two attempts have failed in the previous thirty seconds. This will demonstrate both the event precedence (two previous attempts) and the temporal reasoning (within a specified time period). The first step is to run the tutorial script in *your_kaos_root* \kaos\scripts\tutorial by typing “ant” as in section 3.3. Next we will load an extension ontology containing computing terms as in section 5.1. The ontology to select is the *ComptingOntology.owl* as shown in Figure 30. This ontology contains the “login action” which we will be using.



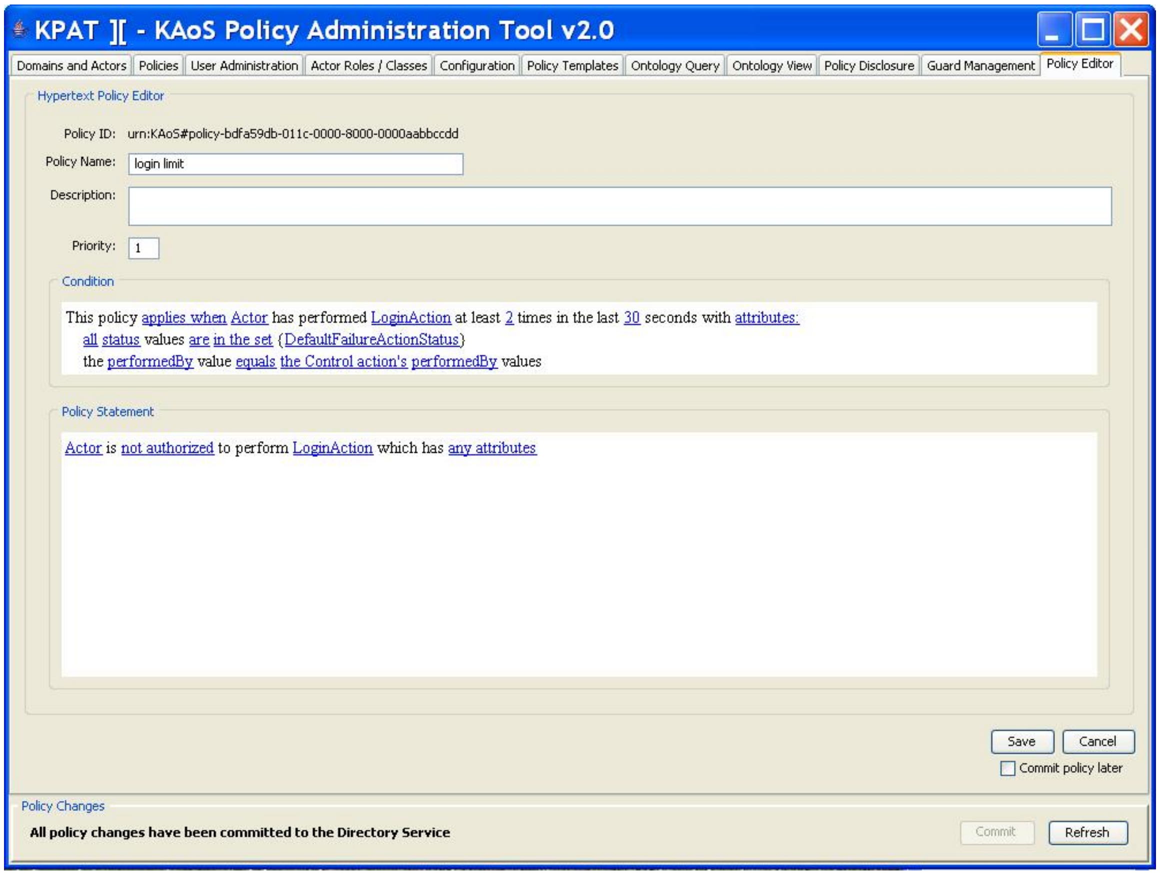
Please enter a namespace OR choose from the list:

http://ontology.ihmc.us/Computing/ComputingOntology.owl

http://ontology.ihmc.us/KAoSOntologies.owl
http://ontology.ihmc.us/Notification/NotificationOntologies.owl
http://ontology.ihmc.us/JBI/Targeting/TargetingOntologies.owl
http://ontology.ihmc.us/Army/ArmyOntologies.owl
http://ontology.ihmc.us/Robot/Names/ActualRobotClasses.owl
http://ontology.ihmc.us/Computing/ComputingOntology.owl
http://ontology.ihmc.us/Usecase/LegacySystem/LegacySystemOntologies.owl
http://ontology.ihmc.us/Java/JavaOntologies.owl

Figure 30 Loading Computing Ontology

Next we create an authorization policy as in section 3.14 except this time we will add historical context. Start by going to the *Actor Roles/Classes* tab and selecting *Actor*. Now press the *New Policy* button and select the hypertext editor. Now we will create a policy that says “Actors are not authorized to login if they have failed to login three times previously.” Our first version will look like Figure 31.



Hypertext Policy Editor

Policy ID: urn:KAoS#policy-bdfa59db-011c-0000-8000-0000aabbccdd

Policy Name: login limit

Description:

Priority: 1

Condition

This policy applies when Actor has performed LoginAction at least 2 times in the last 30 seconds with attributes:
all status values are in the set {DefaultFailureActionStatus}
the performedBy value equals the Control action's performedBy values

Policy Statement

Actor is not authorized to perform LoginAction which has any attributes

☐ Commit policy later

Policy Changes

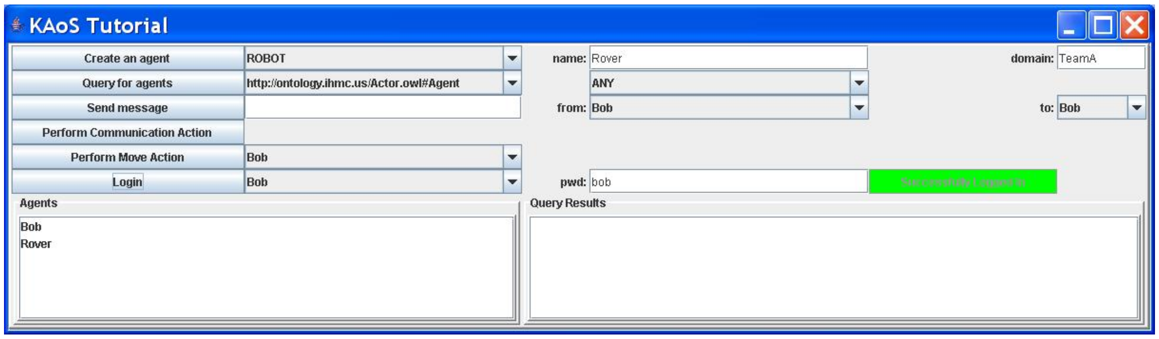
All policy changes have been committed to the Directory Service

Figure 31 Login Limit Policy

You will notice the *Policy Statement* section looks like a normal negative authorization policy. What we have added is the *Condition* section. This section typically says that “This policy always applies.” However, we have added historical context that refers to the status of previous login attempts. Specifically, we refer to when *Actors* have performed *LoginActions* that have failed (bad password) at least two times. We will include the temporal portion that limits the context to the last thirty seconds. Setting the time to zero results in an unlimited (all recorded history) context. The *LoginActions* have an action property of *status* with a value of *DefaultFailureActionStatus* to indicate that the previous login attempts failed. Additionally, they include the requirement that the historical records considered where performed by the same actor as the action currently under consideration. If this attribute is omitted, a failure of any agent will block all agents from logging in.

*** Note: there are currently two status concepts in the ontology; one in ActionStatus.owl and one in Entity.owl. These have different meanings and will not match each other. We are working to disambiguate them in KPAT. For now you can view the OWL representation to ensure you have selected the correct one. ***

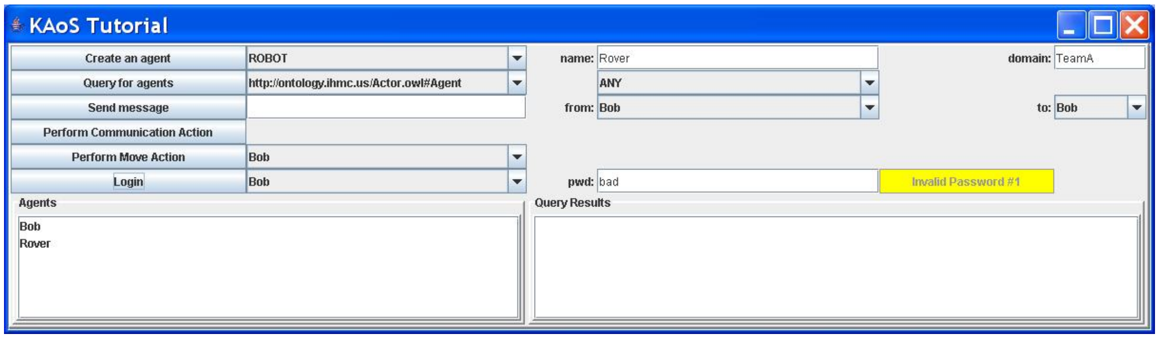
Save the policy and you are ready to test it. Create two agents as in Chapter 3. By default the password for any agent will be its name in lower case. Select an Agent from the dropdown menu next to the Login button. Enter the agents name in all lower case in the password text field. Press the Login button and the results indicator should turn green as in Figure 32.



ROBOT
http://ontology.ihmc.us/Actor.owl#Agent
Bob
Bob
name: Rover
ANY
from: Bob
pwd: bob
Successfully Logged In
domain: TeamA
to: Bob
Agents
Bob
Rover
Query Results

Figure 32 Login Success

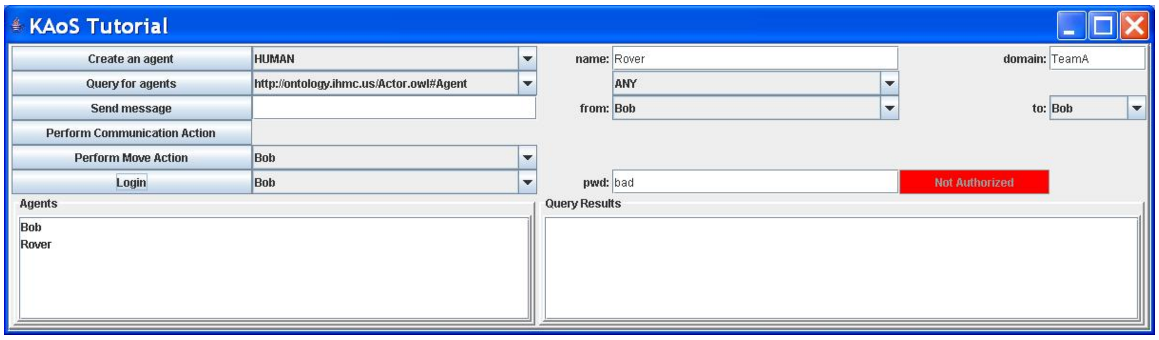
Now change the password to something else and press the Login button again. You should see the results indicator turn yellow and display a count of the number of failed login attempts, as in Figure 33.



ROBOT
http://ontology.ihmc.us/Actor.owl#Agent
Bob
Bob
name: Rover
ANY
from: Bob
pwd: bad
Invalid Password #1
domain: TeamA
to: Bob
Agents
Bob
Rover
Query Results

Figure 33 Login Invalid

After two failed attempts, as specified in the policy, the agent will no longer be authorized to login and the results indicator will turn red as in Figure 34.



HUMAN
http://ontology.ihmc.us/Actor.owl#Agent
Bob
Bob
name: Rover
ANY
from: Bob
pwd: bad
Not Authorized
domain: TeamA
to: Bob
Agents
Bob
Rover
Query Results

Figure 34 Login Not Authorized

Try logging in with the correct password and you will also be denied. After thirty seconds, login attempts will be authorized again and if the correct password is used, they will login successfully.

Now we will take a look at the code behind this example. *SimpleAgent* has a method called *LoginAction* that contains all the code used for this example. As in the previous examples, we will be demonstrating how to perform self enforcement. For history based policies this means tracking the actions in a way consistent with history monitoring and policy evaluation. KAoS provides a history monitor through CSI as shown in the *LoginAction*:

```
HistoryMonitor historyMonitor = CSIFactory.getHistoryMonitor();
```

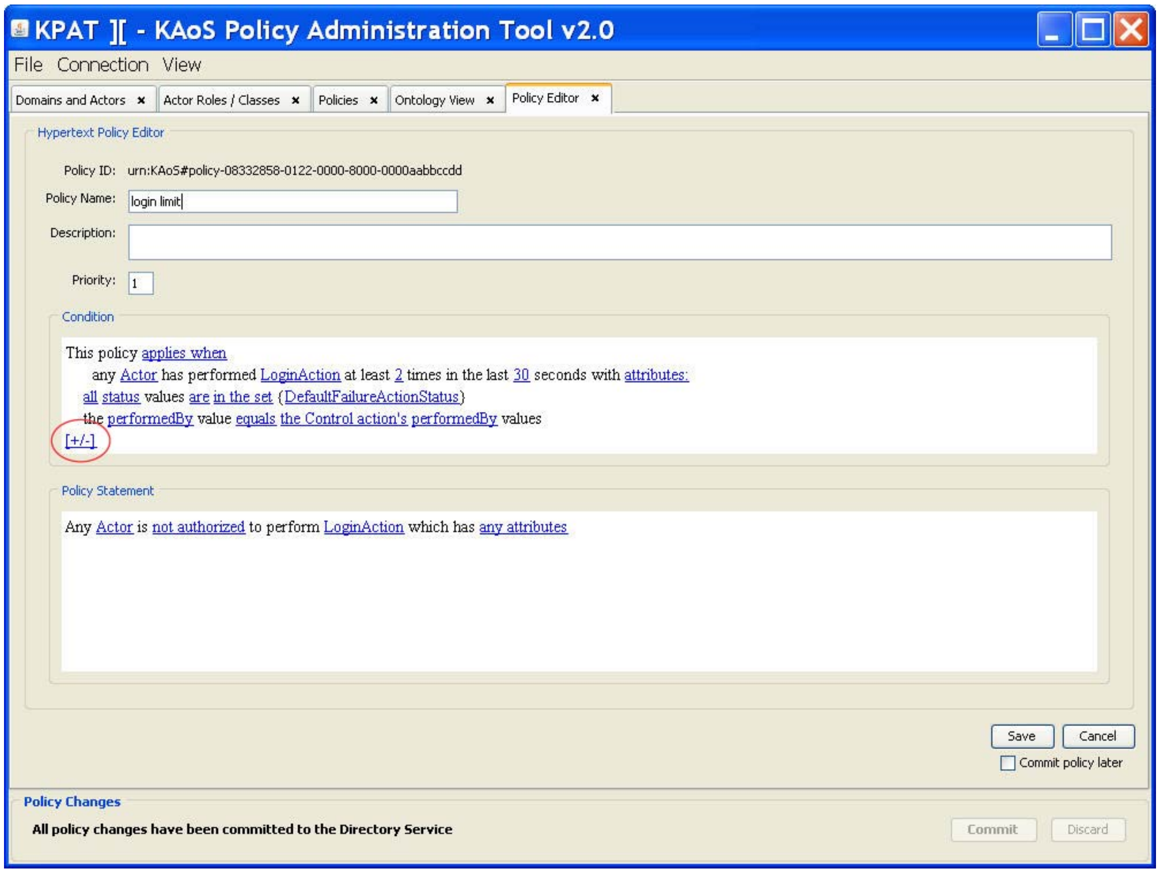
Now we need to log any failed login attempts. For the sake of this tutorial, we do this by checking the password and if it is not valid, we add a failure status and tell the history monitor about the attempt:

```
OntPropertyDescriptionImpl hasStatusProperty = new OntPropertyDescriptionImpl(ActionStatusConcepts.hasStatus());
hasStatusProperty.addValue(ActionStatusConcepts.DefaultFailureActionStatus);
logginAID.addProperty(hasStatusProperty);

historyMonitor.logEvent(logginAID);
```

The KAoS Guard also has access to the history monitor and performs the necessary checks for policy decisions. Note that the action being checked is always just a normal login action with username and password properties and makes no reference to history.

8.1 Multiple History Conditions

To add additional conditions simple click on the "[+/-]" at the end of the first condition. This will allow you to add an AND condition or remove a condition. To test this we will run the same login example as in the previous section. Once the policy is in ready, click on the "[+/-]", circled in Figure 35, to add an additional history condition. Then add the history condition that any actor has performed a Communication Action 1 time in the last 60 seconds. This condition is strictly to demonstrate the multiple condition policy and is not particularly meaningful. The policy should now look like Figure 36. Now have Bob login with the incorrect password multiple times, as in the last section, and it will not be blocked because of the AND condition. Press the "Perform Communication Action" button and then try to login and it will be blocked as shown in Figure 37, assuming the communication action occurred with 30 seconds of the two login failures.



Policy ID: urn:KAoS#policy-08332858-0122-0000-8000-0000aabbccdd

Policy Name: login limit

Description:

Priority: 1

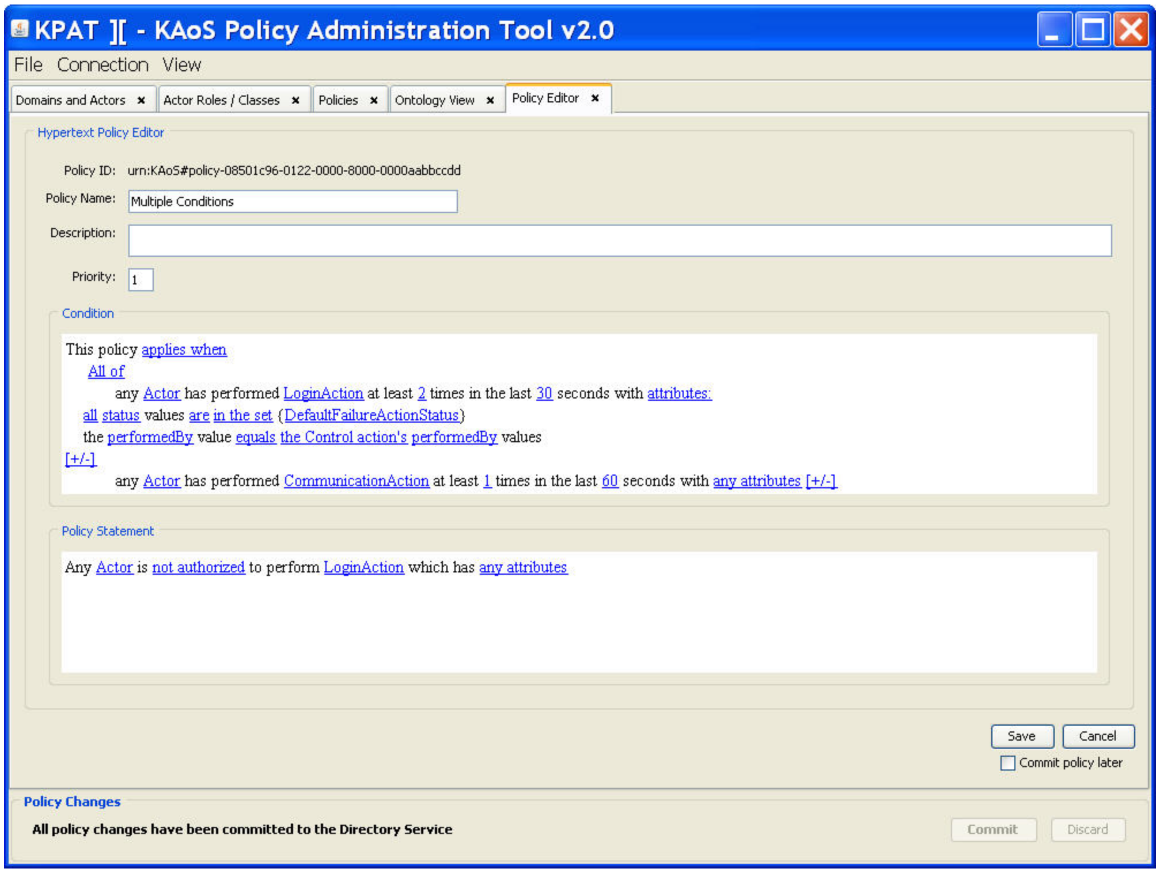
This policy applies when
any Actor has performed LoginAction at least 2 times in the last 30 seconds with attributes:
all status values are in the set {DefaultFailureActionStatus}
the performedBy value equals the Control action's performedBy values
[+/-]

Any Actor is not authorized to perform LoginAction which has any attributes

Commit policy later

All policy changes have been committed to the Directory Service

Figure 35 Adding multiple conditions



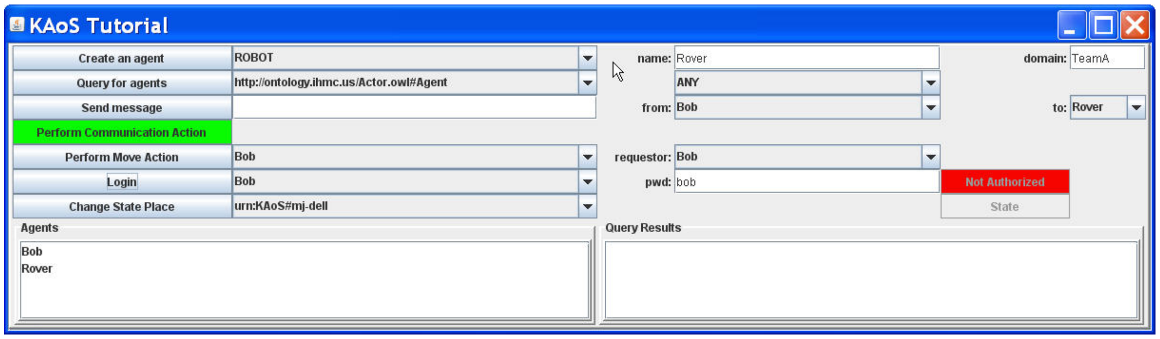
Policy ID: urn:KAoS#policy-08501c96-0122-0000-8000-0000aabbccdd
Policy Name: Multiple Conditions
Description:
Priority: 1

This policy applies when
All of
any Actor has performed LoginAction at least 2 times in the last 30 seconds with attributes:
all status values are in the set {DefaultFailureActionStatus}
the performedBy value equals the Control action's performedBy values
[+/-]
any Actor has performed CommunicationAction at least 1 times in the last 60 seconds with any attributes [+/-]

Any Actor is not authorized to perform LoginAction which has any attributes

All policy changes have been committed to the Directory Service

Figure 36 Multiple History Conditions



Create an agent: ROBOT
Query for agents: http://ontology.ihmc.us/Actor.owl#Agent
Perform Move Action: Bob
Login: Bob
Change State Place: urn:KAoS#mj-dell
name: Rover
ANY
from: Bob
requestor: Bob
pwd: bob
domain: TeamA
to: Rover

Agents:
Bob
Rover

Figure 37 Login Not Authorized after two failed attempts AND a Communication Action

9 The API - Common Services Interface (CSI)

KAoS provides the basic services for distributed computing, including message transport and directory services. Because the services are accessed through a well-defined Common Services Interface (CSI), application developers can selectively use subsets of its capabilities (e.g., registration, transport, publish-subscribe, domain management, remote request forwarding, queries) as appropriate. Documentation can be found at http://ontology.ihmc.us/WorkArea/KAoS/doc/csi-api/index.html.

There are several key services provided by CSI. They include:

- Transport
- Registration
- Request
- Query
- Publish/Subscribe
- Policy

9.1 Transport

Transport provides an abstraction to the underlying message passing mechanism, a simple way to bind to a given transport and send messages allowing applications to tailor their own communication protocol. Transport gives you low level access to the message transport, including binging to the transport and sending messages. To access the message transport simply:

```
transportSupport = CSIFactory.getTransportSupport();
CSIFactory.setCurrentTransport(dsTransportName);
```

Then you can send messages by:

```
transportSupport.sendMessageTo(_actorDesc, receiverDesc, message);
```

To receive messages, you must implement MessageListener and bind to the transport:

```
transportSupport.bindMsgListenerToTransport(this, name, _actorDesc);
```

9.2 Registration

Registration provides the ability to publish an entity, its capabilities and status and update both the capabilities and status. Registration allows you to register an entity, assign properties to it, create groupings (domains). There is also a *QueryRegistration* for querying to get information about actors and there properties. The basic registration looks something like this:

```
registration = CSIFactory.getRegistration();
registration.registerEntity(_actorDesc, true, true);
```

To query for agents use a *QueryRegistration* as follows:

```
query = CSIFactory.getQueryRegistration();
result = query.getAllAgents();
```

9.3 Request

Request allows one entity to send a request to another typically to execute some action. The actions and properties of the action are specified using terms from the ontology. It is built on top of the registration and transport layers.

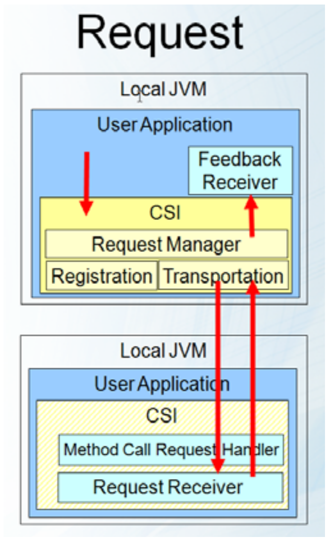


Figure 38 Request architecture

9.4 Query

Query allows an entity to retrieve information about another entity. The allowable queries are specified in the ontology, as well as the properties associated with each query. This is a "pull" method for getting information from an another actor.

9.5 Publish/Subscribe

Subscribe defines the operations to register, deregister and notify observers when the state of this observable changes or an event occurs that is associated with the observable. This is a "push" method.

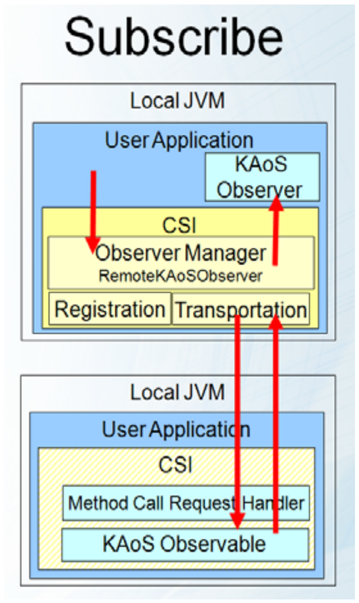


Figure 39 Subscribe architecture

9.6 Policy

Policies allow constraints to be applied to an entity. They can be dynamically modified to adjust the bounds on a particular entity based on the current context. The policy interface allows you to generate and modify policies, but also query them to understand the applicable policies and allowable ranges.

10 Enforcement

10.1 Action Instance Description

As we discuss earlier an *ActionInstanceDescription(AID)* represents an action. For example, let's define an AID for the RadioTransmissionAction with point of origin specified by latitude and longitude for the previous created RadioSpectrumPolicyAgent :

```
ActionInstanceDescription radioTransmAID = new
ActionInstanceDescriptionImpl (aidprefix + transimttedAIDnum,
RadioActionConcepts.ChangeFrequancyAction,
radioSpectrumPolAgent.getGUID());
```

`where` *`aidprefix`* is a string that we can define as *`radioTransmitionAID`*, `t`*`ransimttedAIDnum`* is the number of constructed AIDs and the `RadioActionConcepts` is the control action class that defined the *ChangeFrequency* action.

10.2 Checking Authorization

To check authorization we are going to define a property called *hasTransmissionPower* and add a value to it, for instance a double *usedPower*:

```
OntPropertyDescription hasTransmissionPower = new
OntPropertyDescriptionImpl(RadioEntityConcepts.hasPower());

hasTransmissionPower.addValue ((new Double(usedPower)).toString());
radioTransmAID.addProperty(hasTransmissionPower);
```

Then we define the OntPropertyDescription for the latitude and longitude and add the property to the radioTransmAID from section 10.1:

```
OntPropertyDescription hasTransmissionLocation =
createXmsnLocationProperty(transimttedAIDnum, latitude, longitude);
radioTransmAID.addProperty(hasTransmissionLocation);
```

Then check authorization catching a KAoSSecurityException statement. If an exception is thrown means that is not authorized, otherwise it is granted:

```
PolicyChecking policyChecking = CSIFactory.getPolicyChecking();
policyChecking.checkPermission(radioTransmitAID, null);
```

The KAoSSecurityException will give the user a statement indicating why the policy was not granted. For instance "Original action not permitted. Obligations not attempted.", "Original action permitted. Obligation status inconsistent with action.", "Original action permitted. Service required by obligations failed." and "Original action permitted. Obligation actor not locatable."

10.3 Checking Obligations

For checking obligation we are going to use the findPolicyDecision method and will catch a KAoSSecurityException. The operation will give us a vector of AIDs which are going to perform the action if no exception is thrown. If an exception is thrown, the statement will indicate the problem as we described on section 10.2:

```
 PolicyChecking policyChecking = CSIFactory.getPolicyChecking();
Vector<ActionInstanceDescription> obligations = policyChecking.findPolicyDecision(aid, null);
for(ActionInstanceDescription action: obligations)
{
   performAction(action);
}
```

Each AID in the vector is ordered in a sequence of execution and each AID has a property related to the execution of the action. For instance the property *hasStatus* defined in the basic KAoS ontologies as http://ontology.ihmc.us/ActionStatus.owl. We can define a property value *BEEP*. Then using the robot example, we can say that "*A robot is obligated to BEEP before the robot start moving*", where *start moving* is the trigger condition.

When a guard has to enforce an obligation action, the guard constructs an obligation AID and tries to match it to other registered agents that can perform that action. If no agent is found, the obliged action is directed to the agent with the trigger action.

Obligation policies can define kaos.policy.enforcement.obligation.ControlActionInstantiator The CAI instantiates an AID for the control action based on the specified BasicActionDescription for a control action and a trigger ActionInstanceDescription. The instantiators are added to the guard to check for obligation violations or to look for other actors to enforce an obligation action. The CAI add Role-Value-Maps (discussed in section 7) . They fill in the possible values for the role value mapped properties in the controls action by following the role value path in the trigger action and querying for the possible values. For example:

control.hasDestination = Trigger.performedBy.isTeamMemberOf.hasLeader

10.4 Policy Base Configuration

For checking configuration for the *radioTransmAID* defined in 10.1, we start getting a list of properties names and add a property *hasPower* and obtain the kaos.core.csi.policy.PolicyAdvice from the CSIFactory which defines operations to advice the user as to which property values are allowed/forbidden for the given action based on a policy. Then we call the method *getConfiguration(),*which is used when the agent/enforcer has only partial information about an action and would like to determine what range of properties can be allowed by the policy set. The agent/enforcer partially fills an ActionInstanceDescription object and sends it to the method, which finds those policies that are applicable to this action and contain the given property. The method will then select only those values for the given properties that will not conflict with higher priority policies containing the given properties. The list propertyNames is containing the properties for which values are to be found. The object *radioTransmAID* will be used to find applicable policies. The boolean argument if is set to *'true'*, will result in returning only these values for the missing specified property which would satisfy some policy if used alone. The null argument refers to the PolicyDecisionObserver, an entity interested in receiving updates whenever policy decisions change.

```
List<String> propertyNames = new ArrayList<String>();
propertyNames.add(RadioEntityConcepts.hasPower());
PolicyAdvice myPolicyAdvice = CSIFactory.getPolicyAdvice();
```

Then, we will get the power configuration as:

```
List< ActionInstanceDescription> powerConfiguration =
myPolicyAdvice.getConfiguration (propertyNames, radioTransmitAID, true,
null);
```

The *powerConfiguration* list contains multiple ActionInstanceDescription objects which contain the allowed values for those properties.

10.5 State Monitor

Sensors which monitor the state of the system must implement the kaos.core.csi.extension.StateSensor interface. A StateSensor implements the method getOntologicalAttributes(), which returns a list of state types that the StateSensor is capable of monitoring. Upon startup, the StateManager reads a configuration file, instantiates each StateSensor listed in the configuration, and registers the StateSensor with itself as a listener for policy state conditions, based on the types of states that the StateSensor is interested in.

The StateManager notifies a StateSensor about policy state conditions and obligation triggers, by calling StateSensor.registerInterest(OntClassInfo stateCondition), and its counterpart deregisterInterest() when the condition is no longer relevant (inherited from StateInterestListener interface).

The StateSensor is responsible for notifying the StateManager with information about the state(s) it is monitoring. A static instance of the StateManager can be obtained by calling StateManager.getInstance(). Upon learning about a relevant policy state condition (via registerInterest()), the StateSensor may create a new instance of a State (using OntInstanceDescriptionImpl) and register it by calling StateManager.registerState().

The StateSensor should update the State instance when the monitored state changes, by calling StateManager.updateStateProperty() (or StateManager.updateStateProperties(), if multiple properties have changed). Likewise, when a state is no longer relevant to the current policies (via deregisterInterest()), the StateSensor should call StateManager.deregisterState().

In the example below, the policy state condition specifies a particular host to monitor. The StateSensor gets the hostname from the passed-in condition, and registers a new state instance with the StateManager for the host being monitored. The StateSensor then updates a property of the state instance, marking the host as currently being “down”.

```
public void registerInterest(OntClassInfo state)
{
    // get an instance of the StateManager
    StateManager stateManager = StateManager.getInstance();

    // query the state to see which host we should be monitoring
    String host = null;
    Set<String> properties = state.getPropertyNames();
    try {
      if (properties.contains(HOST_PROP)) {
        host =   state.getInstancesForProperty(HOST_PROP).iterator().next();
      }
    }


    // create a new state instance and register it
    if (host != null) {
       String stateId = NamespaceValidator.validateURI(“fakeStateIdFor” + host);
```

```
        OntInstanceDescriptionImpl stateInstance = new OntInstanceDescriptionImpl(stateId,
state.getMainSuperClassName());

        OntInstanceDescription hostInstance = new OntInstanceDescriptionImpl(host, HOST_TYPE);
        stateInstance.addProperty(HOST_PROP, host, hostInstance);
        stateManager.registerState(stateInstance);
        stateManager.updateStateProperty(stateId, DOWN_PROP, "true");
    }
  }
```

10.6 History

As we defined in section 8, the system must maintain a history of events so that we can reason about the current action in the context of previous ones. From the CSIFactory we get the history monitor for the *RadioTransmissionAction* and then we calculate the *radioTransmAID* for the `hasTransmissionPower property`. The *logEvent* method allows tracking the actions in a way consistent with history monitoring and policy evaluation.

```
HistoryMonitor historyMonitor = CSIFactory.getHistoryMonitor();
OntPropertyDescriptionImpl hasTransmissionPower = new
OntPropertyDescriptionImpl (RadioEntityConcepts.hasPower());
hasTransmissionPower.addValue ((new Double(usedPower)).toString());
radioTransmAID.addProperty(hasTransmissionPower);
historyMonitor.logEvent(radioTransmAID);
```

The KAoS Guard also has access to the history monitor and performs the necessary checks for policy decisions.

10.7 Classifiers

When classes as video messages are not understood by KAoS, we use classifiers that are class wrappers which KAoS can load and use. These classifiers provide custom capabilities that allow applications to make sophisticated policy decisions based on specific parameters of AIDs.

10.8 Policy Callback Mechanism

KAoS policies provide a callback mechanism that add or remove a policy when a decision changes. For instance, we get the AID for an agent called *GatewayAgent* and add a property *hasPacket* and a value VIDEO_CHANNEL.

```
ActionInstanceDescription videoForwardAction = new
ActionInstanceDescriptionImpl(GatewayAgent.aidprefix + (new
VMID()).toString(), NetworkActionConcepts.ForwardDataAction(),
myGatewayAgent.getGUID());
OntPropertyDescription hasPacketForVideo = new
OntPropertyDescriptionImpl(NetworkActionConcepts.hasPacket());
hasPacketForVideo.addValue(VIDEO_CHANNEL + "Packet");
```

Then we get define a PolicyDecisionObserver *pObserver*, an entity interested in receiving updates whenever policy decisions change. When the observer is notified that a change occurred then we get the *.AllowableValuesForActionProperties* from the policy advice which returns a list of the updated AID objects.

```
PolicyDecisionObserver pObserver = new
GatewayControlAgent.GatewayLinksPolicyDecisionObserver();
List<ActionInstanceDescription> myVideoOptions =
myPolicyAdvice.getConfiguration(serachPropertyNames,videoForwardAction,
false, pObserver);
```

11 KAoS Core Ontology

KAoS provides a set of generic ontology concepts needed for basic policy creation. These are available at: *http://ontology.ihmc.us/ontology.html*. They describe actors, actions and a variety of other general concepts such as:

- Entity
- Attribute
- Group
- Actor
- Situation
- Condition
- Action
- ActionStatus
- ActionHistory
- Place
- Message
- Policy

You can also find several application specific extensions.

*** NOTE: The ontology has recently received a major update, as part of the standards efforts underway with the Federal Digital Policy Management initiative.

12 Extending the KAoS Ontology

In general, the KAoS Core Ontology usually has to be extended with concepts specific to the application under development. The application ontology should contain definitions for all the concepts for which the business logic code can provide information. For example, through code instrumentation, the business logic for the control of radios can usually provide required information about transmission parameters. Thus, all these concepts should be present explicitly in the ontology, in order that policies can refer to them. Three common extensions are for Actors, Action and Entities.

12.1 Extending Actor

The Action ontology is typically extended by some *ApplicationActor.owl*. This extension contains definitions of application actor classes (or roles) with their properties. The new Action classes should be subclasses of *http://ontology.ihmc.us/Actor.owl#Actor.*

12.2 Extending Action

The Action ontology is typically extended by some *ApplicationAction.owl*. This extension contains definitions of application action classes with their specific properties. The new Action classes should be subclasses of *http://ontology.ihmc.us/Action.owl#Action*. The properties of the Action classes should be subproperties of either *http://ontology.ihmc.us/Action.owl#hasDataContext* or *http://ontology.ihmc.us/Action.owl#hasObjectContext* in order to provide hints to KPAT about their importance.

12.3 Extending Entity

The Action ontology is typically extended by some Application**Entity**.owl. This extension contains definitions of application specific entities with their properties, which will be used to define context of the actions. The new Entity classes should be a subclasses of *http://ontology.ihmc.us/Entity.owl#Entity (or more specific subclass).*

12.4 Java Ontology Mapping Tool

Development of code linking the business logic with KAoS policies and services requires references to the URLs of ontology concepts. KAoS provides a simple tool to create Java constants for every concept defined in a given ontology with values equals to their URLs. In the *script/generateOntologyVocabulary*, create an ant build file with target running OntologyMapper for each defined ontology file. For example see target *vocabulary-selected* in *KAoS_HOME/scripts/kaos-tool. No explicit URLs should be used in the code.* Based on experience, such a practice creates difficult debugging problem (misspelled URLs, ontology changes). Re-running the script automatically updates the URLs and concepts, and keeps the code consistent with the ontology.

*** NOTE: Additional mapping tools (e.g., for Web services) are currently being defined. Should we mention some of these?

12.5 Example Ontology Extension

Here is an example of extending the ontology:

- Build the ontology:
- Create an Actor called *Printer* that extends Actor
- Create an Action called *Print* that extends Action
- Add property to Print called *Output*
- Build the java class using the ontology tool
- Host the ontology locally
- Register an actor as a Printer, see that Print is an available action in KPAT, Have somebody request the Printer to Print some Output.
- Once the ontology is build you can use a web server to store it and then run KPAT using the Ontology View tab to load the ontology. For instance, create an ontology named MyOntology.owl . Then run a web server as TOMCAT and copy the ontology to the webapps\ROOT directory. Run KPAT and select the Ontology View tab , then press the Load Namespace button and write the url of the ontology as follow: http://localhost:8080/MyOntology.owl and press the Load Namespace button to load it. The ontology will be shown in the Namespace List in KPAT. (See fig.X).

** http://localhost/test1.owl **

13 Running without Internet Access

KAoS typically runs with the expectation of Internet access. This access is used to download the current KAoS ontologies and any other required ontologies. If Internet access is not available, slow or intermittent, it may be desirable to use an Internet proxy. KAoS provides an ontology proxy tool for this purpose.

13.1 Starting the Ontology Proxy

To start the ontology proxy tool go to *your_kaos_root* \kaos\scripts\kaos-tools and type “ant ontology-proxy”. You should see the Ontology Proxy GUI similar to Figure 40 Ontology Prox.

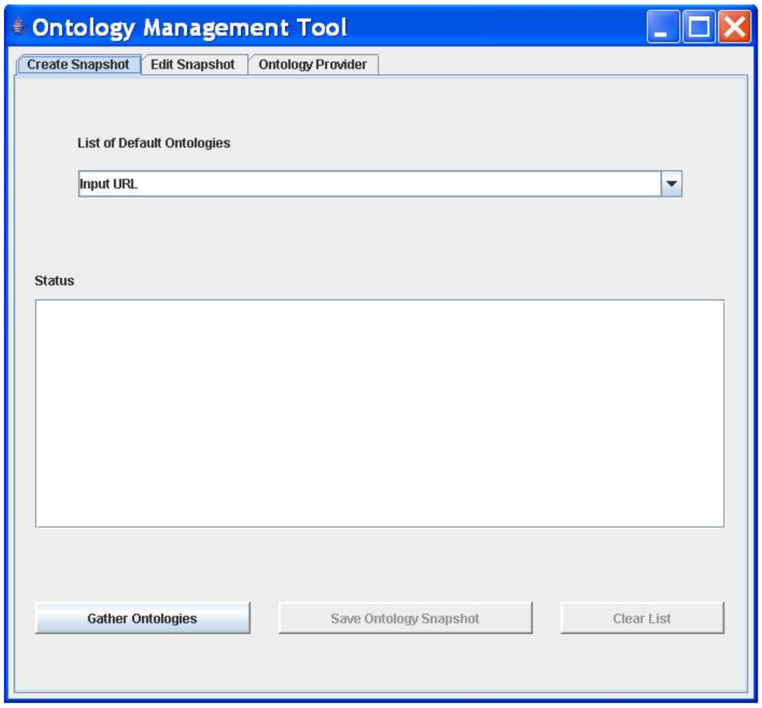
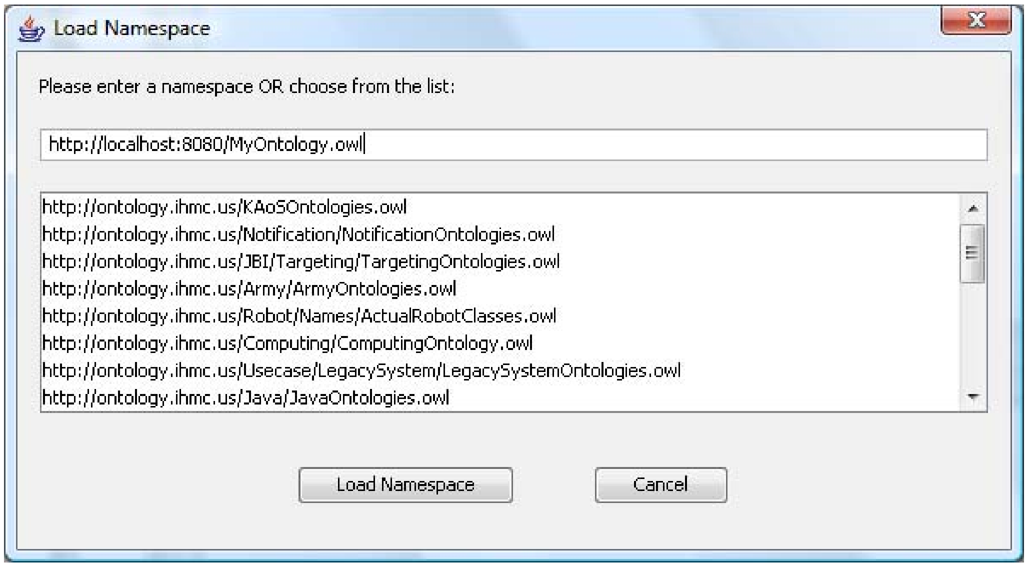


Figure 40 Ontology Proxy

13.2 Making an Ontology Snapshot

The ontology files should be configured with their *final static namespace*. Then, they should be loaded into the web server configured with a URL corresponding to this namespace. *You will need to be connected to the source of the ontology in order create the ontology snapshot***,** but once it is created, this is no longer necessary. An ontology snapshot puts all the required ontologies in a single file. On the "create Snapshot" tab, select the KAoS ontology from the drop down list. Press "Gather Ontologies" so the proxy will get the ontologies from the web. Then add any additional ontologies your application needs, gathering each one. You can type in the URLs if they are not included in the list. When you have all the ontologies you need listed, press "Save Ontology Snapshot" and a snapshot of your desired ontology configuration will be saved to a file with the extension ".ont". The tab "Edit Snapshot" allows modification or automatically refreshing of an existing snapshot. It is also very useful when ontologies have to be gathered from disparate domains (e.g. public Internet and corporate network).



Please enter a namespace OR choose from the list:

http://localhost:8080/MyOntology.owl

```
http://ontology.ihmc.us/KAoSOntologies.owl
http://ontology.ihmc.us/Notification/NotificationOntologies.owl
http://ontology.ihmc.us/JBI/Targeting/TargetingOntologies.owl
http://ontology.ihmc.us/Army/ArmyOntologies.owl
http://ontology.ihmc.us/Robot/Names/ActualRobotClasses.owl
http://ontology.ihmc.us/Computing/ComputingOntology.owl
http://ontology.ihmc.us/Usecase/LegacySystem/LegacySystemOntologies.owl
http://ontology.ihmc.us/Java/JavaOntologies.owl
```

13.3 Ontology Proxy

To run the proxy, go to the “Ontology Provider” tab. Select load snapshot and choose the ontology you want to load. You should see an indication in the status window when snapshot is loaded. Now just press “Start” and the ontology proxy is ready. Once created and tested, we provide a script that enables the automatic loading and starting of the proxy with a particular snapshot. See the *runProxyWithOntologySnap* target for an example.

14 Saving and Loading Configurations

14.1 Saving a Policy Snapshot

Created policies and other concepts created in this process should be saved in policy snapshot file. This file can be created from KPAT tab “Configuration”. The file should be saved in *config/policyConfigurationSnapshots/Name.cfg.* In *scripts/runKAoSwithPolicyConfiguration* create ant build file with target running KAoS with this policy snapshot preloaded; make sure to set up the first two properties correctly.

```
<property name="ontology.file" value="${basedir}/config/ontologySnapshots/ON.ont" />
<property name="directory.snapshot" value="${basedir}/config/policyConfigurationSnapshots/PS.cfg"/>
<target name="runKAoSwithL3078" description="Start KAoS with policy and ontolgy snapshot">
<fail unless="env.KAOS_HOME" message="Please set the environment variable KAOS_HOME" />
<echo message = "Starting ontology proxy, KAoS DS, Servlet and KPAT"/>
<parallel threadCount="4">
<ant inheritAll="true" antfile="scripts/kaos-tools/build.xml" target="ontology-proxy-autostart"
dir="${env.KAOS_HOME}"/>
<sequential>
<sleep seconds="8" />
<ant inheritAll="true" antfile="scripts/kaos-core/build.xml" target="run-kaos"
dir="${env.KAOS_HOME}"/>
</sequential>
</parallel>
```

```
</target>
```

14.2 Configuring an Ant Script

In order for existing applications to be integrated with KAoS at runtime certain parameters have to be provided. These parameters make KAoS functionality accessible from the modified application source code. The following elements are needed in the target starting the application:

```
<classpath>
<!-- Include all jar files in the KAoS ${lib} directory -->
<fileset dir="${env.KAOS_HOME}/lib">
<include name="**/*.jar" />
<exclude name="**/${optionalLib}/*.jar" />
</fileset>
<!-- Include the path to the config files -->
<pathelement path="${env.KAOS_HOME}/${cfgPath}"/>
</classpath>
<jvmarg value="-Dkaos.core.service.default=Guard.cfg" />
<jvmarg value="-
Dkaos.core.policy.service.default=${env.KAOS_HOME}/${config}/guardConfiguration.cfg" />
<jvmarg value="-Djava.util.logging.config.file=${config}/logging.properties" />
```

If the jar libraries and other configuration parameters are correctly specified, then all available KAoS functionality can be accessed by retrieving the appropriate implementation of the given subset of functionality. This is done by calling a specific factory method on static class: *kaos.core.csi.CSIFactor*. Such calls will create background connections with the KAoS Directory Service.

15 Policy Templates

To simplify policy construction, KPAT provides two additional policy creation interfaces, in addition to the generic policy creation interface:

- The *Policy Wizard* takes a user step-by-step through the policy creation process. Information selected for presentation is conditioned on whatever has been selected previously, making the experience as simple and foolproof as possible.
- The *Policy Template Editor* allows custom policy editors for a given kind of policies to be created by point-and-click methods. For instance, if an application will require the definition of several policies governing publish/subscribe actions, a custom policy editor can be quickly created by limiting choices to just what is needed, thus eliminating the requirement for repetitive selections when a given type of policy has to be created multiple times.

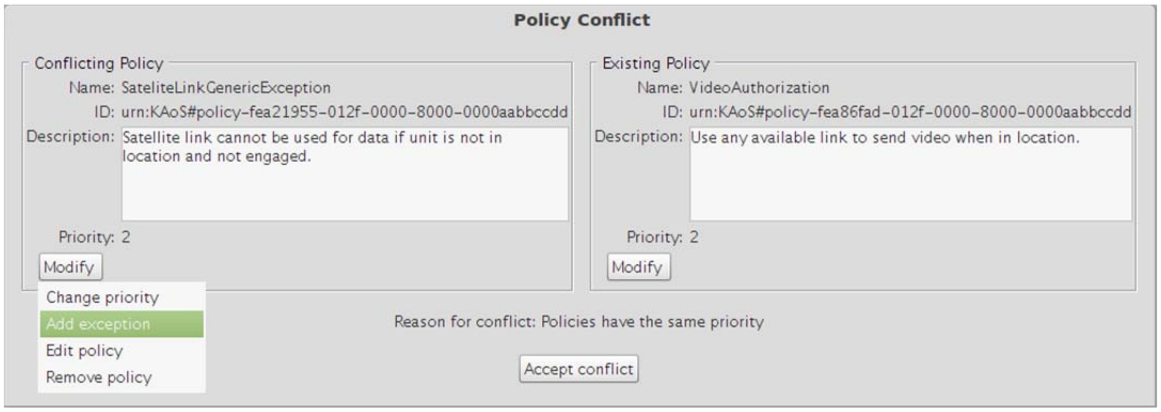
16 Policy Conflict Resolution

When a policy conflict occurs, the KPAT user is presented with a dialog for resolving the conflict.
In the case of a direct conflict (same priority), the user has some options:

- change the priority of one of the conflicting policies
- make an exception (e.g. this policy doesn't apply for mission X)
- remove one of the conflicting policies

The policy resolution dialog allows the option to "accept" policy conflict for overlapping policies, where acceptance indicates that you are OK with the overlap / redundancy. For directly conflicting policies, acceptance indicates that you do not want to resolve the conflict at this time (e.g. save it for runtime resolution at the guard level). The dialog is shown in Fig.41.

In the case of running a distributed directory service, the user may request permission to change policies forwarded by a directory service of higher authority, and provide a reason for making the change. The original directory service is notified of the request, and its user decides whether to allow the modification (via KPAT). The modified policy only affects the directory service which requested the modification. This case is depicted in Figs. 42 and 43.



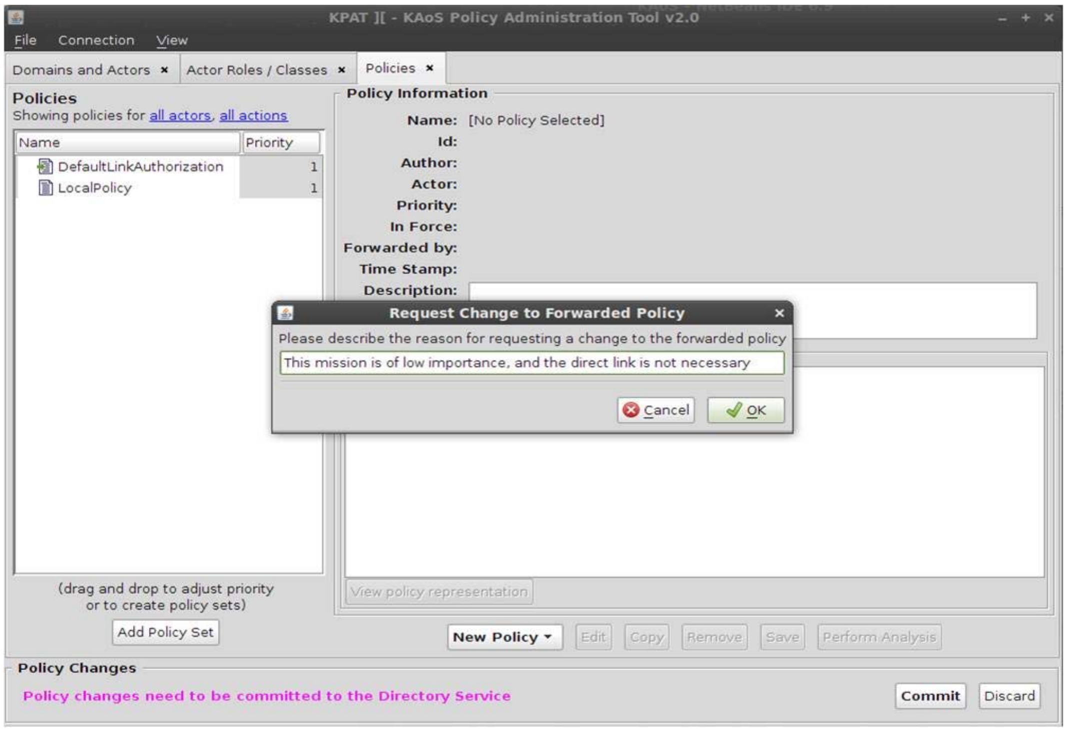
Policy Conflict

Conflicting Policy
Name: SateliteLinkGenericException
ID: urn:KAoS#policy-fea21955-012f-0000-8000-0000aabbccdd
Description: Satellite link cannot be used for data if unit is not in location and not engaged.
Priority: 2

Existing Policy
Name: VideoAuthorization
ID: urn:KAoS#policy-fea86fad-012f-0000-8000-0000aabbccdd
Description: Use any available link to send video when in location.
Priority: 2

Reason for conflict: Policies have the same priority

Fig.41. Policy Conflict Resolution Dialog



KPAT][- KAoS Policy Administration Tool v2.0

Policies
Showing policies for all actors, all actions

Name	Priority
DefaultLinkAuthorization	1
LocalPolicy	1

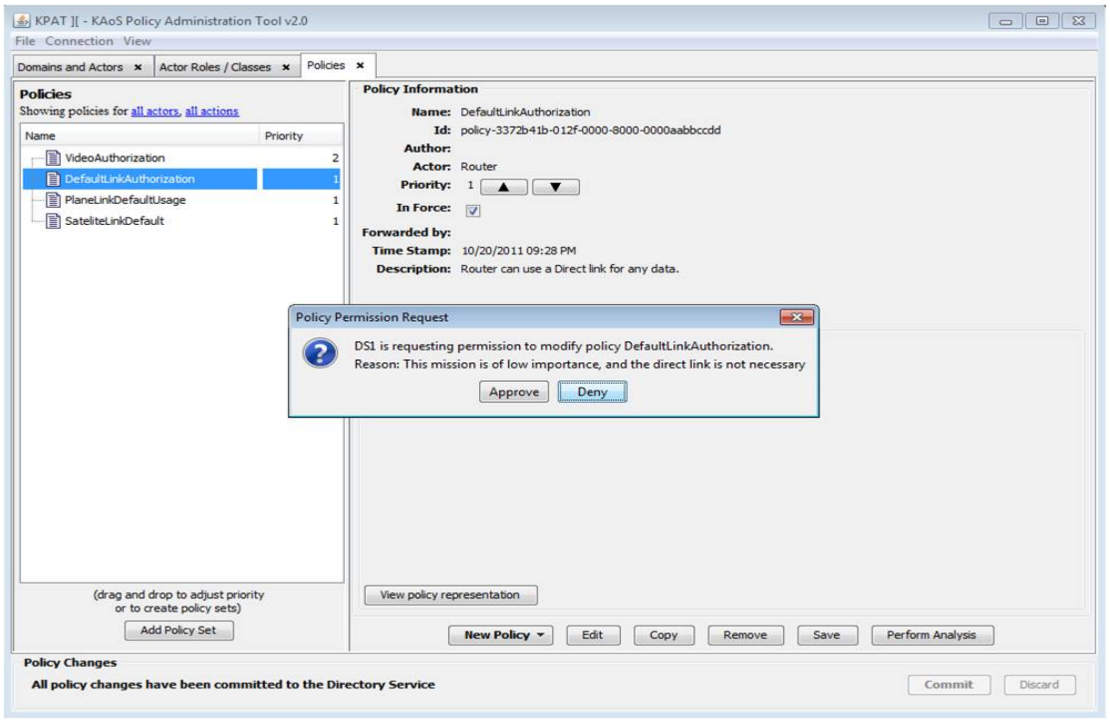
Policy Information
Name: [No Policy Selected]
Id:
Author:
Actor:
Priority:
In Force:
Forwarded by:
Time Stamp:
Description:

Request Change to Forwarded Policy
Please describe the reason for requesting a change to the forwarded policy
This mission is of low importance, and the direct link is not necessary

(drag and drop to adjust priority or to create policy sets)

Policy Changes
Policy changes need to be committed to the Directory Service

Fig.42. Request modification of forwarded policy.



Policies
Showing policies for all actors, all actions

Name	Priority
VideoAuthorization	2
DefaultLinkAuthorization	1
PlaneLinkDefaultUsage	1
SateliteLinkDefault	1

(drag and drop to adjust priority or to create policy sets)

Policy Information

Name:	DefaultLinkAuthorization
Id:	policy-3372b41b-012f-0000-8000-0000aabbccdd
Author:	
Actor:	Router
Priority:	1
In Force:	✓
Forwarded by:	
Time Stamp:	10/20/2011 09:28 PM
Description:	Router can use a Direct link for any data.

Policy Permission Request

DS1 is requesting permission to modify policy DefaultLinkAuthorization.
Reason: This mission is of low importance, and the direct link is not necessary

Policy Changes
All policy changes have been committed to the Directory Service

Fig. 43. Ask source if the modification is allowed.

17 Advanced Features

17.1 Distributed Directory Service

KAoS supports the ability to have multiple directory services running concurrently. The directories may optionally be configured to share policies between each other.

Directory services in a LAN will automatically find each other, using the KAoS P2P discovery mechanism (assuming they belong to the same discovery-group, as specified in the message transport configuration file). They must each have a unique name.

Agents also automatically find the directory services within their discovery-group, using the KAoS P2P discovery mechanism. By default, the first directory service to respond to a discovery request will be used by the agent as the primary directory service. A preferred directory service may be specified in the agent's message transport configuration file, by adding the property 'preferredHost' (for a particular hostname) or 'entityId' (for a particular agent name) to the 'PreferredDS' locator description:

```
(locators
  (PreferredDS
    (discoveryEnabled true)
    (discoveryGroup kaos)
    (entityType DirectoryService)
    (preferredHost '10.0.0.2')
  )
  …
)
```

Policy subscriptions between directories are established by using the KPAT GUI. From the policies tab, right-clicking on a policy (or policy set) presents the user with an option to “Setup peer subscriptions”. From this dialog, the user may choose one or more peer directory services with which to share a policy (or policy set). After a subscription is established, any updates to the policy or policy set will be synchronized with the remote directory. The subscriptions are saved in the configuration snapshot of the directory service used to establish the subscriptions.

To run the distributed directory service, call the ant target 'peerDistributed' before calling the normal target 'run-ds' (from the `kaos-core` ant script) . You should also pass in parameters for “agent.name” and “ds.cfg”, which specifies the name of the directory service, and the configuration file to use, respectively (the configuration file must be unique for each directory, because the agent name is also contained in the message transport section of the configuration). For example:

```
ant -Dagent.name=DirectoryService2 -Dds.cfg=DS.cfg peerDistributed run-ds
```

18 Troubleshooting

My agent seems to start, the domain shows up, but the agent does not show up in KPAT:

The most likely problem is that your agent starts registers and then terminates immediately deregistering from KPAT. Make sure you have a “Thread.wait()” in your main method.

My policies don’t seem to work at all. I made a simple authorization policy preventing an action without any properties, but it does not stop the action:

The most likely cause is forgetting to add this prefix when using names

I get an error about “java.net.BindException: Address already in use: JVM_Bind”

The most likely cause is that you already have a directory service running. Make sure you kill all previous processes before starting. If it still happens, check your running processes using Task Manager or an equivalent application. Kill any java.exe processes to ensure nothing is lingering.

Appendix

A References

1. SAFE User Documentation, Document Revision 3.0

B Definitions, Abbreviations, and Acronyms

CSI – Cougaar Software Inc. Previously NAI Labs, developers of security extensions to Ultra*Log

DM – The KAoS Domain Manager. It is responsible for registering agents consistent with policies on domain membership, for ensuring policy consistency at all levels of a domain hierarchy, for notifying Guards in the event of a policy change, and for storing policies in the repository. A domain is a collection of Cougaar agents, potentially spanning multiple hosts and Cougaar organizational structures, registered to a common domain manager as a common point of administration.

Guards. Guards interpret policies that have been approved by the DM and enforce them with appropriate mechanisms, including Cougaar Binders, Java access control, Nomads resource control, and obligation policy monitors.

IHMC – Institute for Human & Machine Cognition. A research organization associated with the University of West Florida.

IP – Internet Protocol. A widely used protocol allowing computers to share data.

JAAS – Java Authentication and Authorization Services

JDK – The Java Development Kit.

JVM – The Java Virtual Machine. A bytecode interpreter for the Java Programming Language. See also JDK.

KAoS – Knowledgeable Agent-oriented System. An agent framework developed by The Boeing Company and IHMC that is now in the public domain.

KPAT – KAoS Policy Administration Tool. The graphical user interface to managing policies in the KAoS framework.

MS – Microsoft. The large software company responsible for producing the Windows series of computer operating systems and numerous anit-trust activities.

OWL – Web Ontology Language (http://www.w3.org/TR/owl-features)

SAFE – Survivable Agent Framework Extensions. The name of the project whose goal is to extend the Cougaar agent framework with additional features to support policy-based domain management of Cougaar components and host resources.TCP – Transmission Control Protocol. Typically used in conjuction with IP.

VM – Virtual Machine. Shorthand in this context for the Java Virtual Machine (JVM).

XML – A mnemonic for eXtensible Markup Language.