KAoS Tutorial

Andrzej Uszok
auszok@ihmc.us

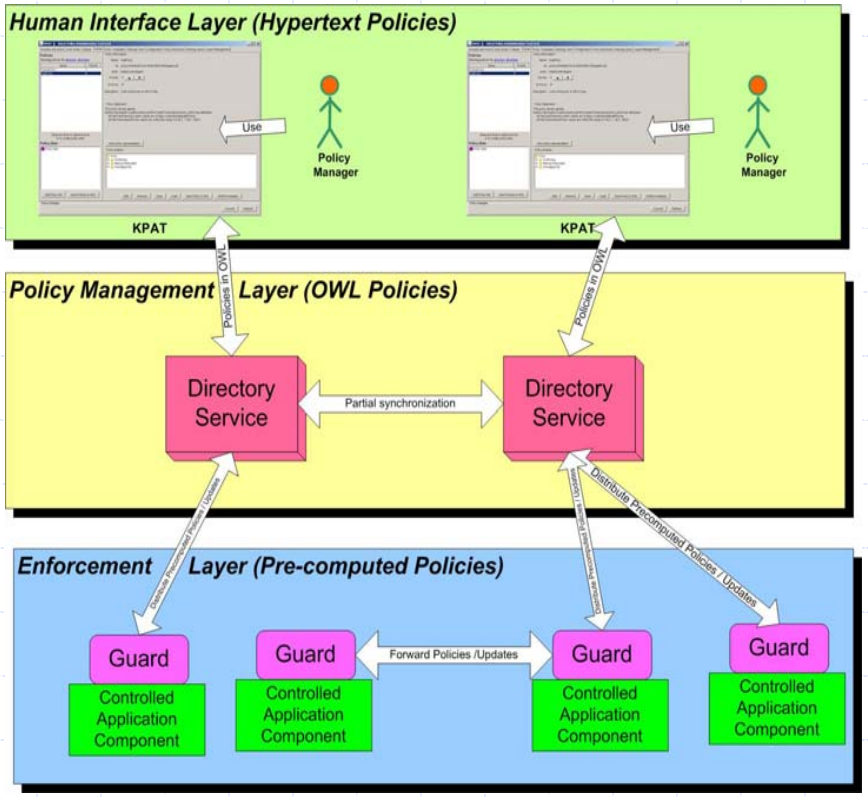
Jeffrey Bradshaw
jbradshaw@ihmc.us

KAoS Policy Service Motivations

- Allow for policies which are **human expressible but machine enforceable**
  - permit effortless extension of policy vocabulary to suite domain needs and understanding of the domain only limited by the mapping to the business logic of the application
  - present user-friendly interface to the policy system
- Provide sophisticated policy query, analysis and explanation mechanisms
- Support for interoperability by using ontology and standards for the Semantic Web
- Support extendable framework architecture allowing for easy extension, customization and integration of the policy service with diverse target application environments
- Provide policy distribution and decision infrastructure, which is highly-efficient and tolerant to disconnections

KAoS Architecture

- *Human interface:* A hypertext-like graphical interface for policy specification in the form of **natural English sentences**. The vocabulary is automatically provided from **ontology**.
- *Policy Management representation:* Used to encode and manage policy-related information in **OWL**. Inside DS it is used for policy analysis and deconfliction.
- *Policy Decision and Enforcement representation:* KAoS automatically "compiles" OWL policies to an **efficient lookup format** that provides the grounding of abstract ontology terms, connecting them to the instances in the runtime environment and to other policy-related information. These polices are sent from DS to **Guards**, which serve as local **policy decision points.**



KAoS Ontology and Policy Semantics

Use of Ontology in KAoS

- Descriptions of actors, actions and situations at different levels of abstraction
- Possibility to dynamically calculate relations among policy, platform entities, and other policies based on concepts ontology relations
- Dynamic extension of the service framework by specifying platform ontology and linking it with the generic KAoS ontology
- Extension of the KAoS framework itself by adding new ontologically-described components

KAoS Ontology Management

- Ontologies expressed in OWL
- KAoS defines core set of ontologies;
  - loaded during its bootstrap
- Ontology specific for the application extend core ontology;
  - loaded by KAoS after core ontologies
- External tools used to create these ontologies:
  - Protégé, SWOOP and validatores from daml.org
- KAoS allows to extend these ontologies by creating instances and subclasses
- When Internet connection is not available the KAoS ontology proxy can be used

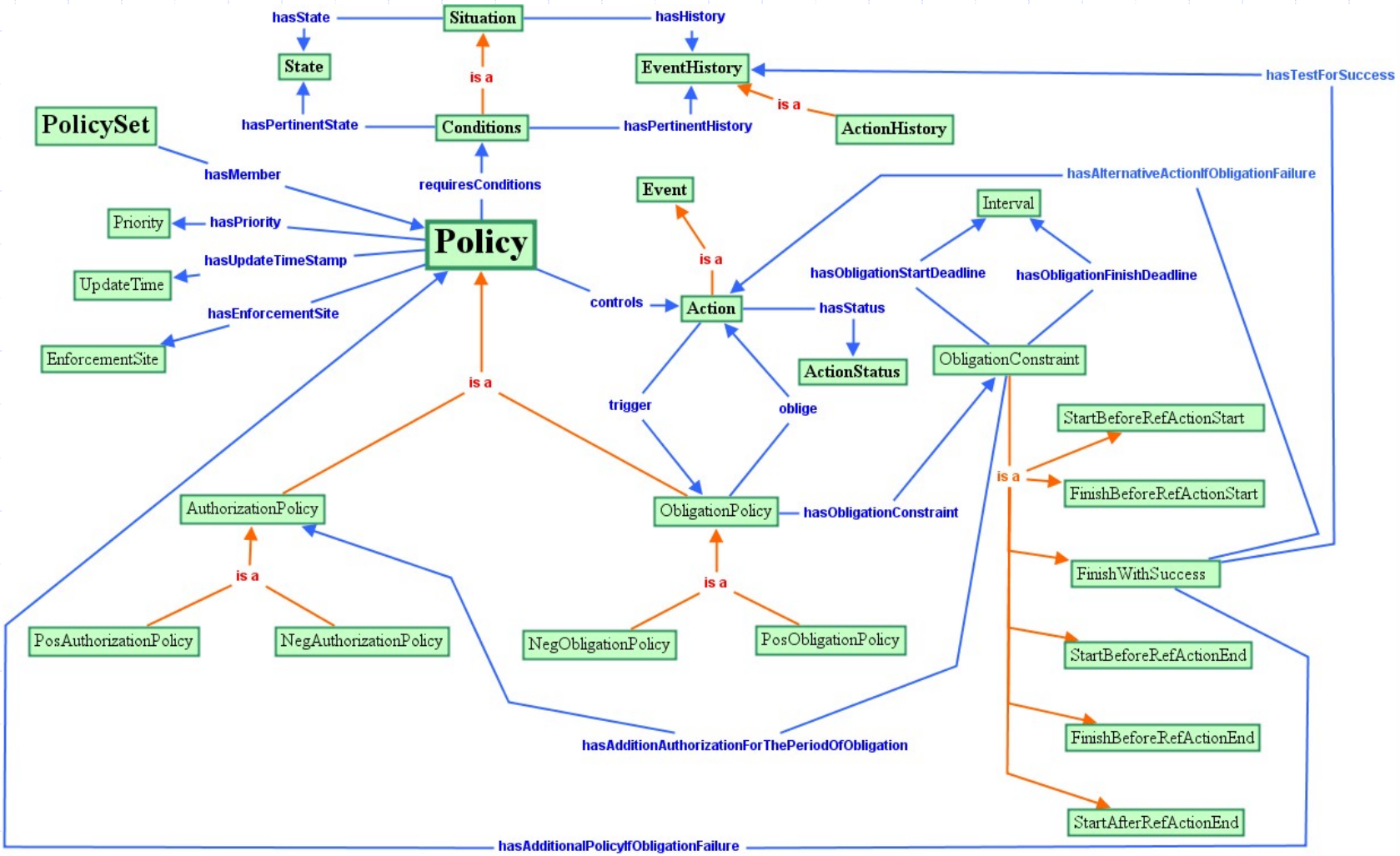
KAoS Core Ontology

- Web Site: ontology.ihmc.us contains OWL ontologies of:
  - Entity
  - Attribute
  - Group
  - Actor
  - Situation
  - Condition
  - Action
  - ActionStatus
  - ActionHistory
  - Place
  - Message
  - Policy
- Plus many application specific ontologies extending the core ones

KAoS Policies

- Policy constrains/amends user/system activity/state
- Main types of supported policies:
  - **Authorization** – Allow or Forbid **actions**
  - **Obligation** – Obliged **actions** or Waive obligation
    - Associated with **Conditions** activating the obligation
- Includes a description (*class)* of the controlled *situation*
  - Constitutes a **test** (*template)* for the applicability of the policy
  - Contain definition of action Subject – extension of traditional policy Role
- OWL vocabularies allows for declarative definition of policy applicability
- Policy posses a **priority**, which enables it to take precedence above contradicting ones

KAoS Policy Semantic



Policy Example:

Any communication outside the Arabello domain, which is not encrypted is forbidden.

OWL Policy Syntax Example

```
<?xml version="1.0" ?>
<!DOCTYPE P1 [
      <!ENTITY policy  "http://ontology.ihmc.us/Policy.owl#" >
      <!ENTITY action  "http://ontology.ihmc.us/Action.owl#" >
      <!ENTITY domains  "http://ontology.ihmc.us/ExamplePolicy/Domains.owl#" >
 ]>

<rdf:RDF
       xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
       xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
       xmlns:owl="http://www.owl.org/2001/03/owl+oil#"
       xmlns:policy="http://ontology.ihmc.us/Policy.owl#"
>
<owl:Ontology rdf:about="">
       <owl:versionInfo>$ http://ontology.ihmc.us/ExamplePolicy/ACP1.owl $</owl:versionInfo>
</owl:Ontology>

<owl:Class rdf:ID="OutsiteArabelloCommunicationAction">
      <owl:intersectionOf rdf:parseType="owl:collection">
            <owl:Class rdf:about="&action;NonEncryptedCommunicationAction" />
            <owl:Restriction>
                  <owl:onProperty rdf:resource="&action;#performedBy" />
                  <owl:toClass rdf:resource="&domains;MembersOfDomainArabello-HQ" />
            </owl:Restriction>
            <owl:Restriction>
                 <owl:onProperty rdf:resource="&action;#hasDestination" />
                 <owl:toClass rdf:resource="&domains;notMembersOfDomainArabello-HQ" />
            </owl:Restriction>
      </owl:intersectionOf>
</owl:Class>

<policy:NegAuthorizationPolicy rdf:ID="ArabelloCommunicationPolicy1">
      <policy:controls rdf:resource="#OutsiteArabelloCommunicationAction " />
      <policy:hasEnforcementSite rdf:resource="&policy;ActorSite" />
      <policy:hasPriority>10</policy:hasPriority>
      <policy:hasUpdateTimeStamp>446744445544</policy:hasUpdateTimeStamp>
</policy:NegAuthorizationPolicy>
```

Beyond Description Logic for Policy Representation

- Originally KAoS used only OWL-DL (initially DAML)
- Limited in situations when needed to define policies in which one element of an action's context depended on the value of another part of the current context:
  - Example – Loop Communication Action
  - Relation to the **current** location, time, other aspect of the current action instance context
  - Relation between Trigger Action and Obliged Action
- These requirements can be fulfill 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

Example of policies needing role-value-map semantic

Service Provider B cannot report back on results of operations to parties other than those which have provided the data, unless the data provider has authorized another party.

Spatial Ontology and Policies

- Spatial semantics is needed for policies dealing with physical objects and theirs relations: robots, radios, humans/teams, vehicles, etc.
- KAoS Spatial Reasoning Component (Ksparc)
  - Allows to querying for relative and absolute spatial relations among people, objects, and robots
  - Calculates absolute values of the relations: distance, angles, etc
  - Consist of set of local spatial reasoner (integrated with KAoS Guards) and a global spatial reasoner (integrated with KAoS DS) coordinating the reasoning

Supported Spatial relation

- threeDimensionsSpatialProperty
  - above
  - below
  - higher
  - lower
- referencedSpatialProperty
  - furtherToTheLeft
  - furtherToTheRight
  - higher
  - lower
  - between
- orientationSpatialProperty
  - inFront
  - behind
  - toTheLeft
  - toTheRight
  - above
  - below
  - towards
  - backwards
  - currentlySee
  - furtherToTheLeft
  - furtherToTheRight
  - higher
  - lower
- inside
- outside
- canSee
  - currentlySee

Inferencing Engine Integration

- Used in KAoS to reason about ontological relations and policies
- Stanford **JTP** – Java Theory Prover
  - 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
- Currently integrating with Pellet through the developed generic reasoner layer
  - Isolates from specific reasoner

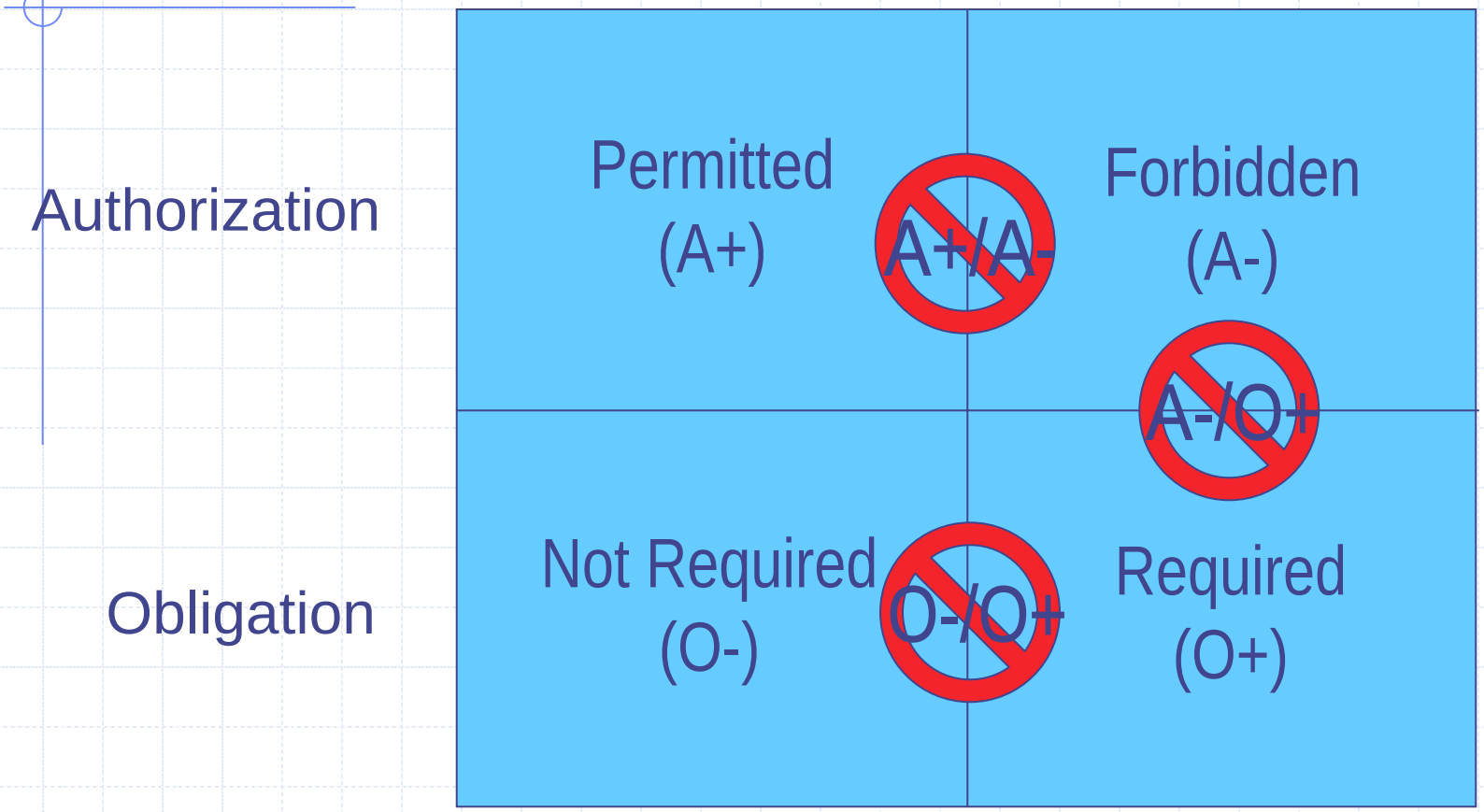
KAoS Policy Interface

- Makes **Transparent** complexity of policy reasoning
- Its input is the description of a **tested situation**
- Allow to investigate how policies affect actions:
  - ***Test Permission*** – verifies authorization to perform a given action
  - ***Get Obligations*** – gets a list of actions obliged in the given situation
  - ***Get Configuration*** – gets possible values for a questioned action property, which will make the specified action authorized
  - ***Make Compliant*** – transform the action an actor tries to perform from a policy forbidden to a closed one which is policy permitted (in progress)
- Mechanism to overwrite policy in certain situation by human or adjustable autonomy system
- Available as Java API or through remote network calls

Policy Analyses

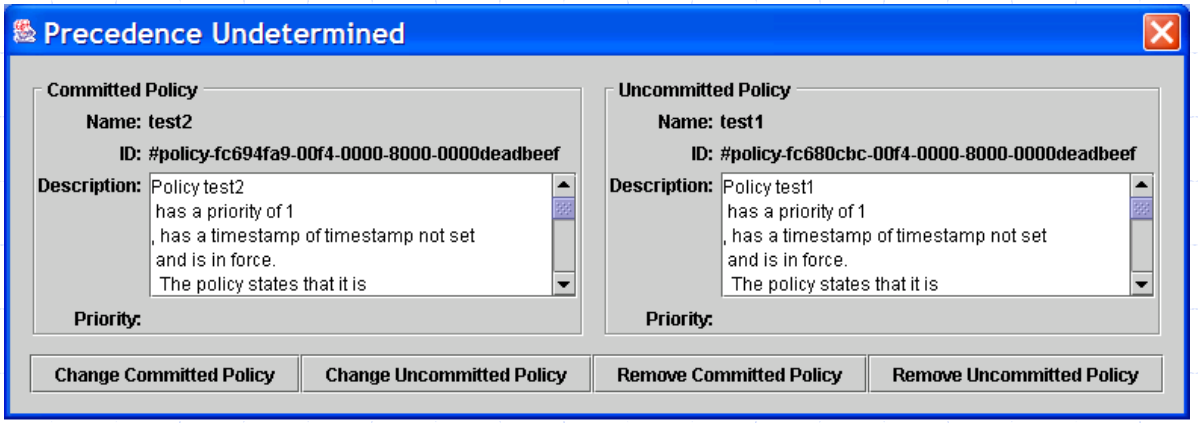
- Human needs to know policy relations and correct those resulting in incorrect or unintended policy decisions
- Given policy can be overlap by some higher priority policy (or by a sum of such policies)
  - If fully overlap then policy is irrelevant
- Policy can overlap with the same priority policy resulting in **policy conflict**

Example of KAoS Reasoning: Resolving Three Types of Policy Conflicts



- *Conflict analysis and other powerful forms of reasoning become more critical when policies and situations are changing rapidly in dynamic and tactical environments*

Notification about policy conflict



- *Remove Policy*: one of the overlapping policies can be completely removed;
- *Change Priority*: priorities of the policies can be modify so they either do not conflict or they alter the precedence relation
- *Harmonize Policy*: the controlled action of the selected overlapping policy can be modified using an automatic harmonization algorithm to eliminate their overlap
- *Split Policy*: the controlled action of the selected overlapping policy can be automatically split into two parts: one part that overlaps with the other policy and the other which does not. Then the priorities of these parts can be modified independently. The splitting algorithm is similar to the harmonization and is currently in development.

Description logic reasoning

- Subsumption-based reasoning used for determination of disjointness:
  - Finding policy conflicts by determining if two classes of controlled actions are disjoint
  - Harmonization of policies
- Instance classification:
  - Policy exploration, disclosure, and distribution

KAoS Architecture

Generic Elements of the Framework

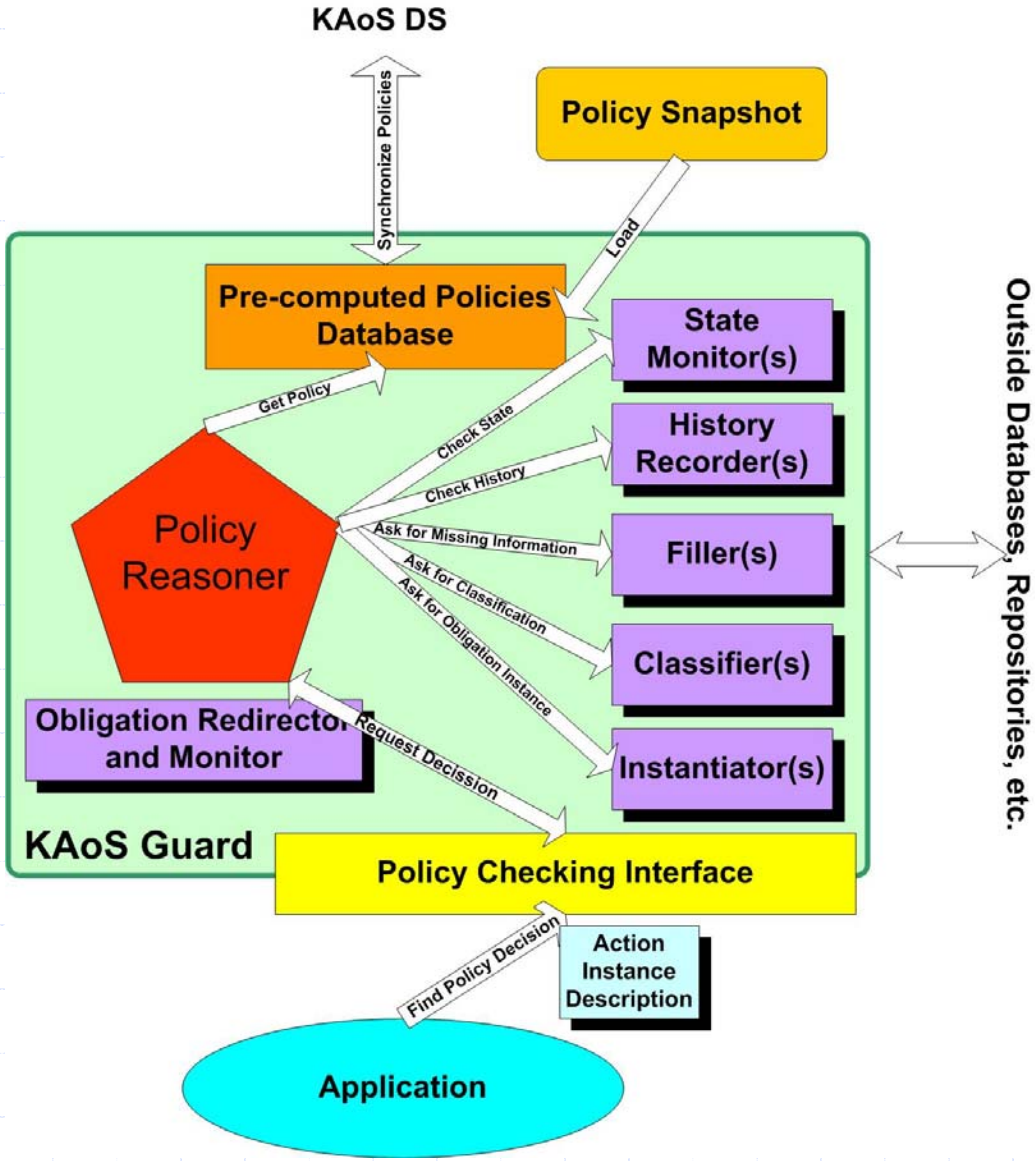
- Set of pre-defined ontologies defining concepts for: *policies*, *actions*, *actors*, *groups,* and *entities*
- Generic functionality includes:
  - **Specifying** policies using the KAoS Policy Admin Tool (KPAT)
  - **Storing, deconflicting, and querying** through the Directory Service
  - **Distribution** of policies to Guards
  - **Implementation** of policies through Enforcers
  - Policy **disclosure** through Policy Query Interface

KAoS Directory Service

- Keeps information about the domains structure of the environment,
- Contains ontological definitions of the platform and active applications
- Allows actors to register their:
  - Name and identities
  - Membership in domains
  - Ontologically specified types and capabilities
- Keeps state of policies
- Keeps ontological description of current situation by collecting history of events and monitoring states

KAoS Guard

- Where KAoS meets the application
- Policy checking traverses the policy database in policy priority order and checks to see whether the AID is in the range of actions controlled by any policy
  - The range of actions attribute is derived from an action class controlled by the policy
  - Role-value map relations, defining aspects of policy context, are checked as well
- Extensibility of Guard behavior



Policy Distribution

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

Application-Specific Extensions

- Specific **ontologies** describing new *policies*, *actions*, *actors*, *groups* and *entities*
- Framework Plug-ins:
  - **Policy Template** and **Custom Action Property** editors
  - **Enforcers** governing actions
  - **Instance Classifiers** to determine if a given instance is in the scope of the given class
- Plug-ins are linked with the framework by:
  - registration in an appropriate **Factory**
  - together with the **plug-in ontology description**
- Guard itself is **not** application specific; its extensions are.

Policy Enforcement Approaches

- Authentication policy enforcement
  - JAAS-based access control enforcement
  - Aroma-based resource control enforcement
  - Action (higher semantic) specific enforcement:
    - Enforce policies that cannot be enforced at VM-level
- Obligation policy enforcement
  - Active monitors watch for satisfaction of obligations and, if necessary, take sanctions after violations
  - Enablers assist in the performance of obligations
- Easy integration with Semantic Web Services
  - they use declarative execution mechanism

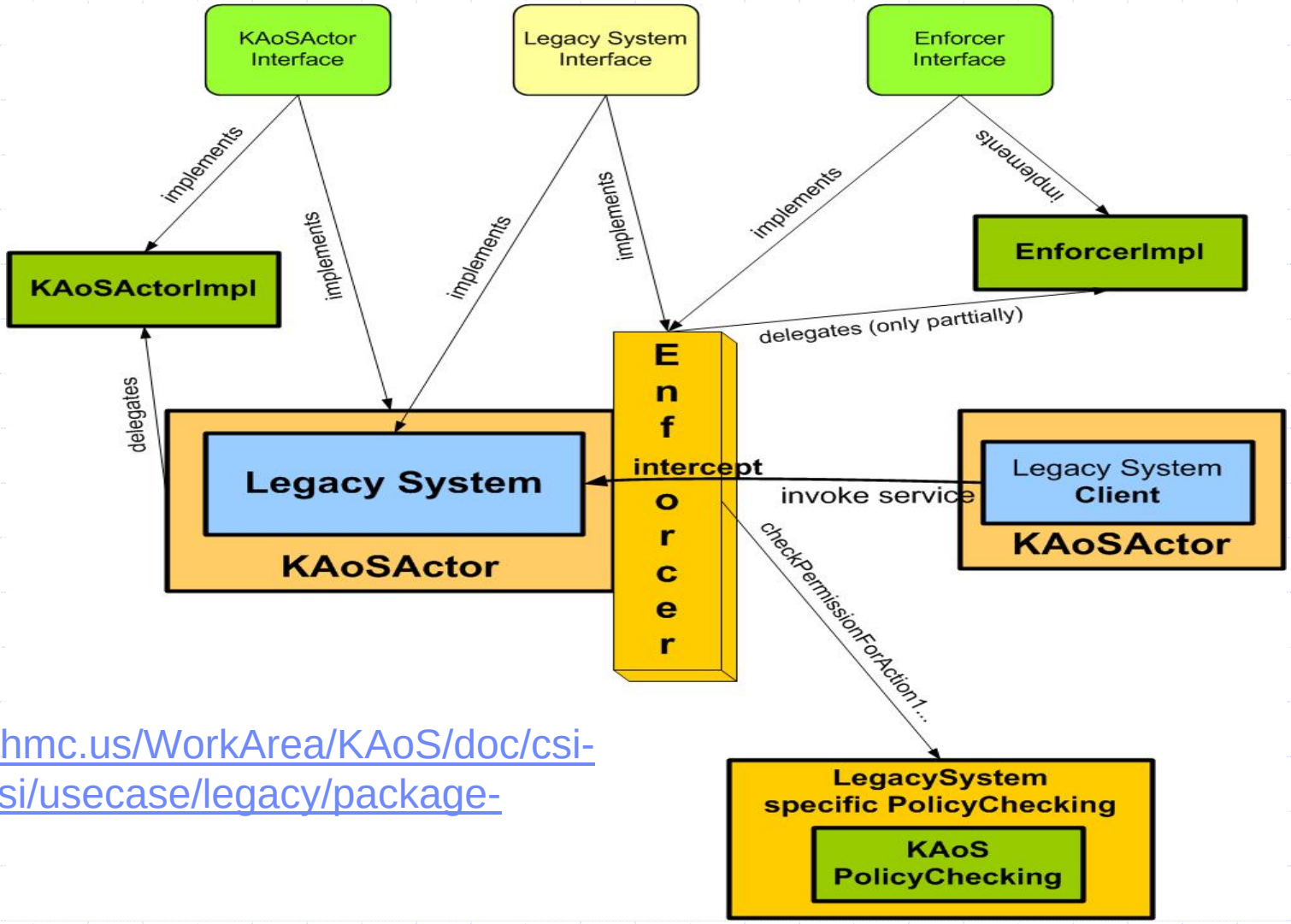
Enforcer Management

- Enforcer class name is registered in the *Enforcer Factory*
- Associated with the names of the action classes it can enforce policy on
- Registry is either stored in a Jar file or available on the network (in the future)
- When needed enforcer created through the Java Reflection mechanism

Enforcer Implementation

- Implement simple *Enforcer* interface:
  - ***getName***(),
  - ***getOntologicalAttributes****() - get names of the action classes intercepted by this enforcer on which policies can be enforced,*
  - *setEnabledStatus()/getEnabledStatus()/ – manage status*
- Implement enforcer unique action filter and the init method, which will insert the monitoring functionality into the actor VM,
- Use Policy Disclosure interface
- Register it into Enforcer Factory database.

Example of Legacy System Enforcer



See:
http://ontology.ihmc.us/WorkArea/KAoS/doc/csi-api/kaos/core/csi/usecase/legacy/package-summary.html

Enforcers for Basic and Derived Actions

- The systems needs enforcers for each of its basic action in order to be fully policy enabled
  - Usually all the system interfaces have to be wrapped into enforcers (see previous slide)
- The derived actions are usually created to enhanced policy manager experience
  - They conceptualized some specific aspect of the system activity (gives it a distinct name)
  - KAoS automatically recognizes relations between derived and basic actions; not need for special enforcer for derived actions
  - The are created from basic actions using OWL syntax:
    - Inheritance from basic action (supported)
    - Restrictions on basic action properties (in testing)
    - Unions and intersections of basic actions (planned support)
    - OWL-S extensions for process sequences (not supported yet)

Classifiers

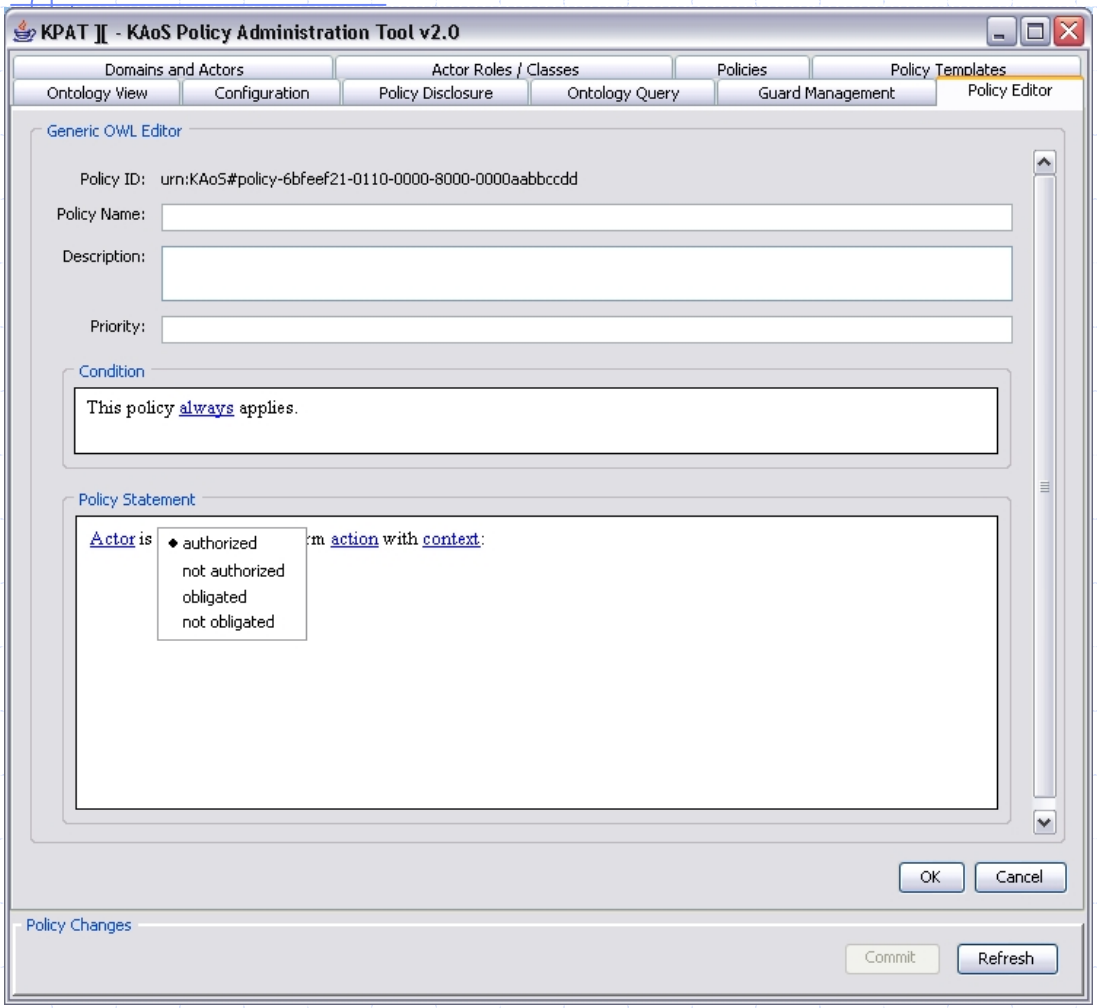
- Its role is to classify if a particular aspect of the policy controlled action is fulfill by a corresponding value in the tested action
  - For instance; test if a transmitted document/video/etc. is of particular type
- It is a Guard extension
- Can be use to handle specialized algorithm, legacy code and scalability issues
- Method *classify* - checks if the instance from the provided description is of the indicated type
- Classifiers Factory associates classes of classifiers with names of action properties

History Monitors (Loggers)

- Its role is to collect records of desired actions happening in the system
  - For instance of failed logging actions
- It is a Guard extension
- Can be plugged to existing system logging mechanism
- Method *testHistory* - checks if the specified number of occurrence of the specified action is recorded in the logger
- HistoryMonitor Factory associates classes of HistoryMonitors with names of action classes they collects

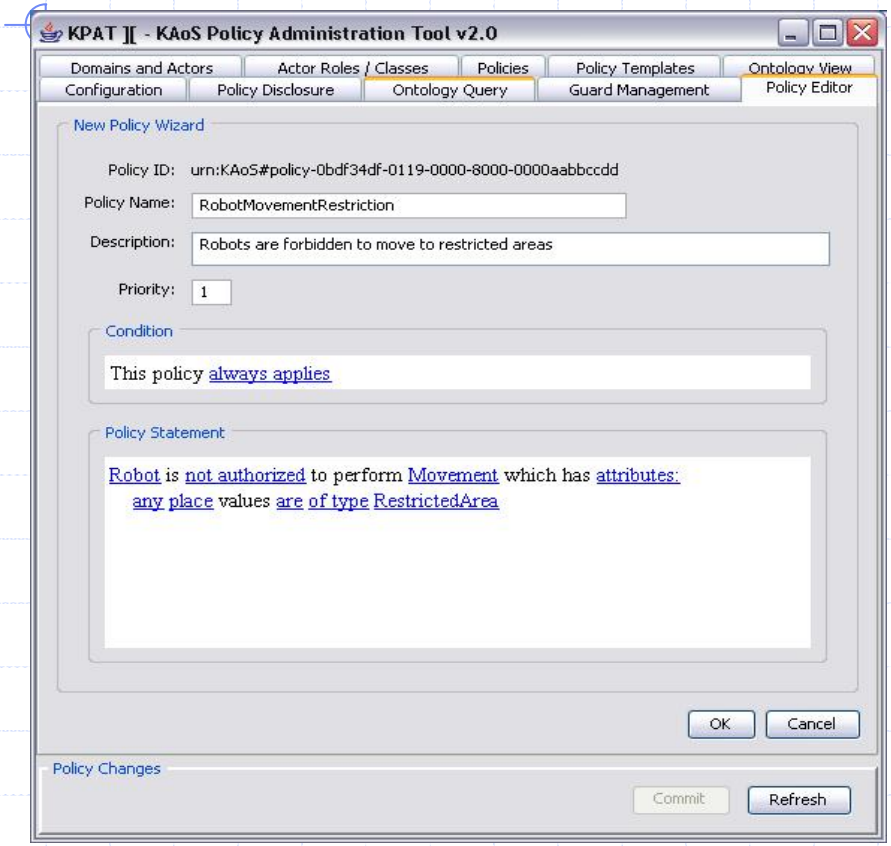
KPAT

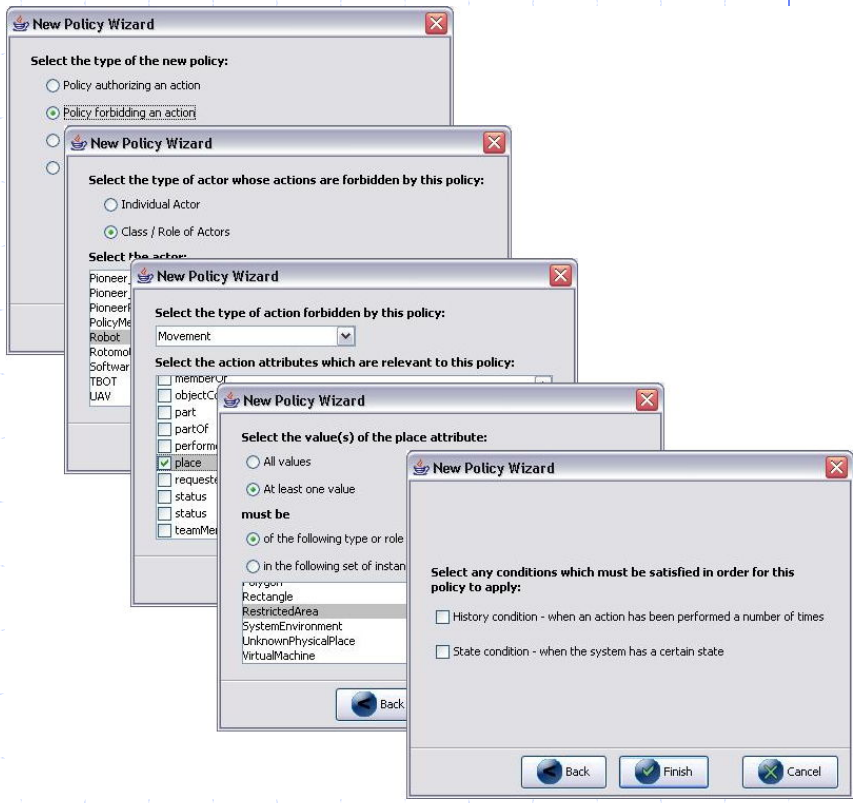
KAoS Policy Administration Tool (KPAT) Hides the Complexity of OWL for Policy Specification



- Dynamically obtains list of selections from the ontology repository based on the current context.
- Graphical template editor allows creation of simplified GUIs
- Cmap interface (COE) available for ontology definition

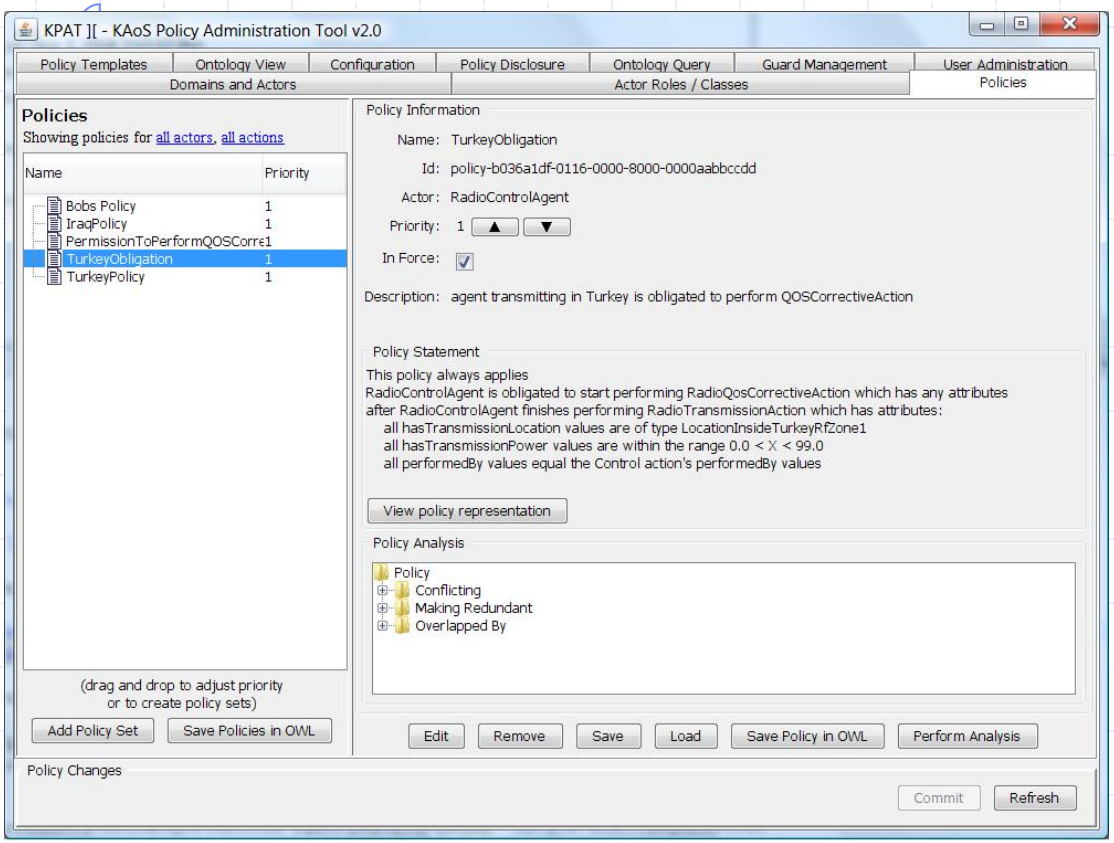
Graphical Tools for Generation of Policy: KPAT Hypertext Policy Editor and Policy Wizard





- Alternative user interfaces for policy creation:
  - Single hypertext editor
  - Policy Wizard
- Users choice lists are created from ontology based on the context

Managing Policies

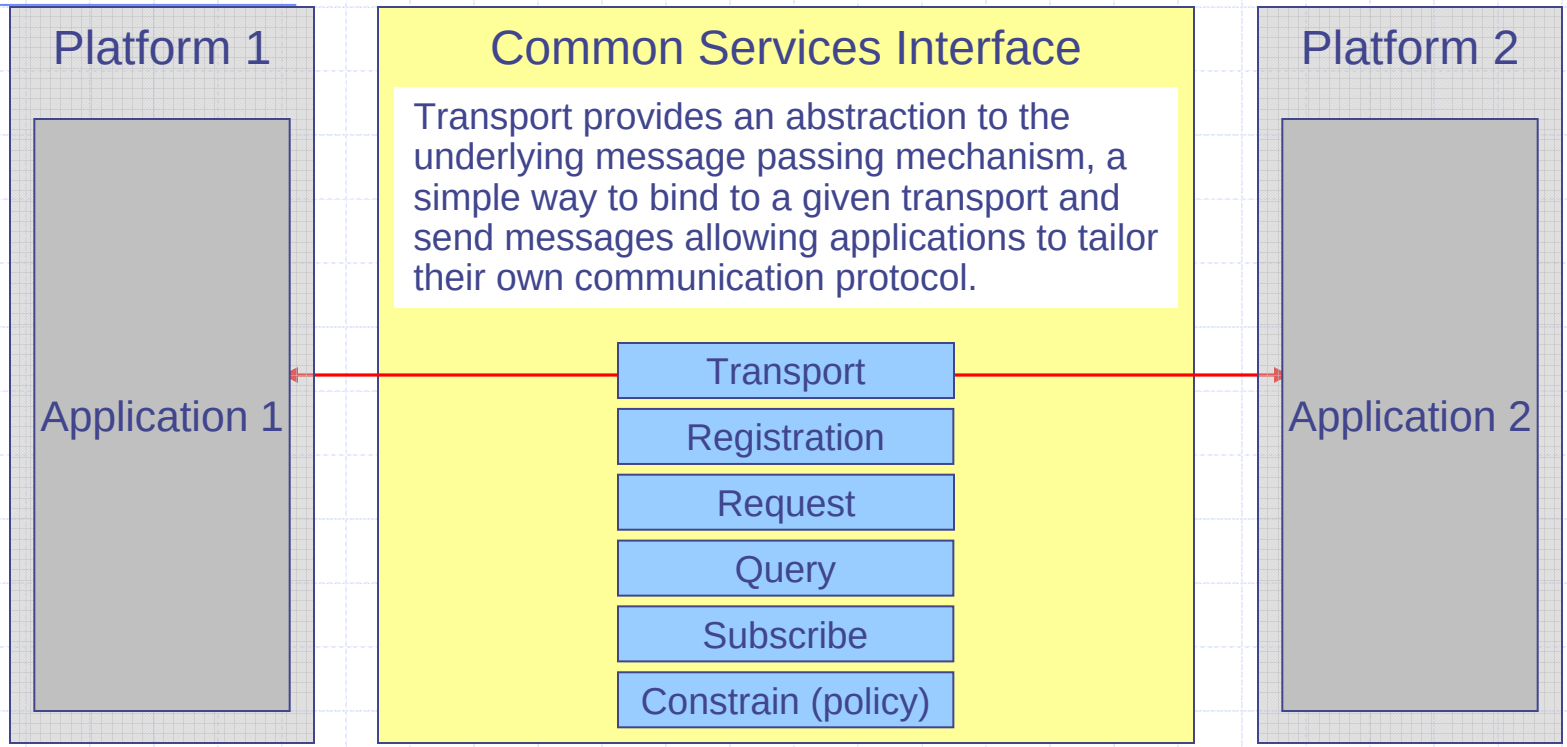


- Policy hypertext descriptions are automatically displayed
- Policies are rank ordered by importance
- The order can be adjusted by using the arrow buttons or dragging and dropping within the list.
- The rankings of other policies will adjust to accommodate the new position
- Policies can be filtered according to their actor or action

Additional KAoS Functionality Overview

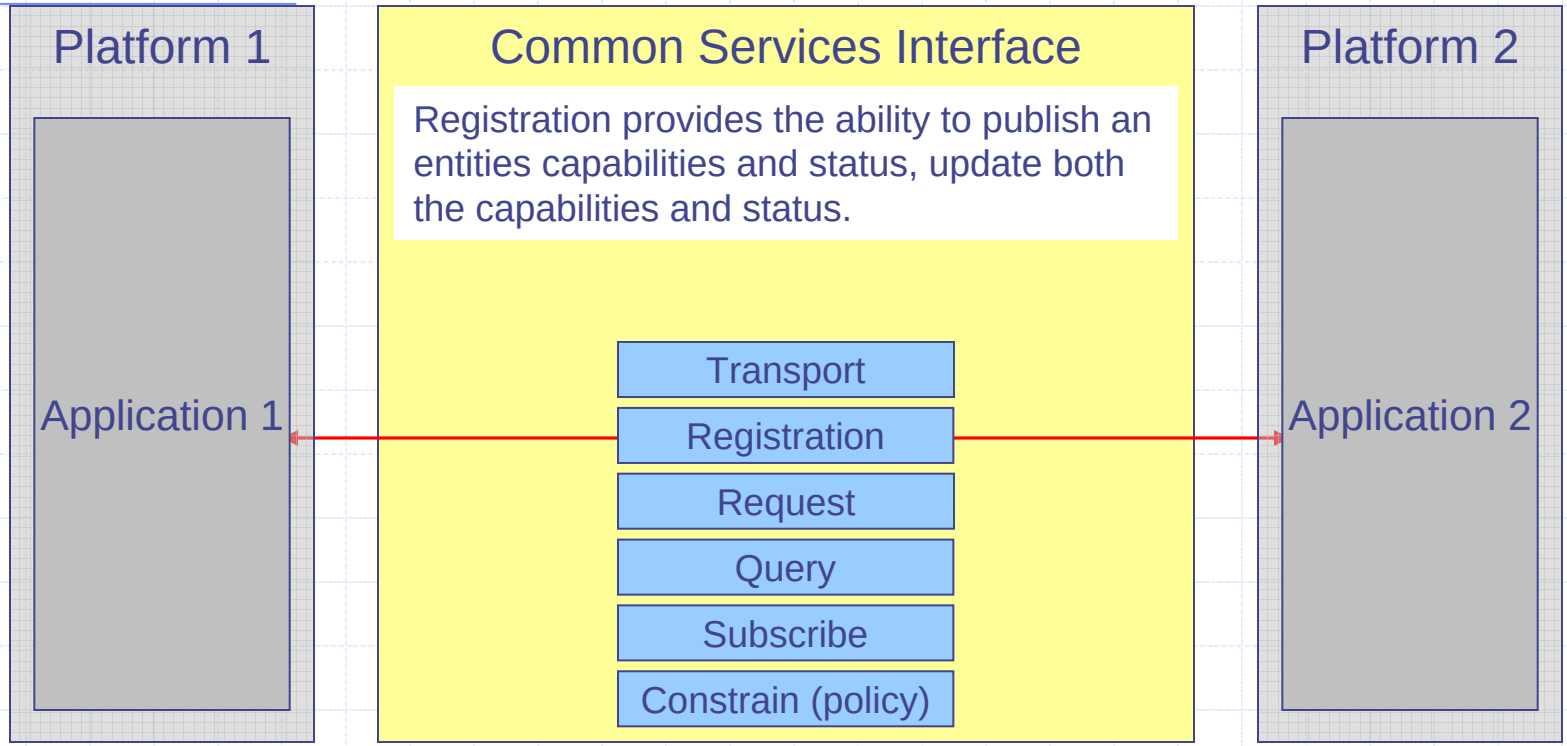
- KAoS functionality is accessed by:
  - APIs to Services through CSI (Common Services Interface) and additional platform specific layers
  - Graphical interface - KPAT

CSI: Transport Service



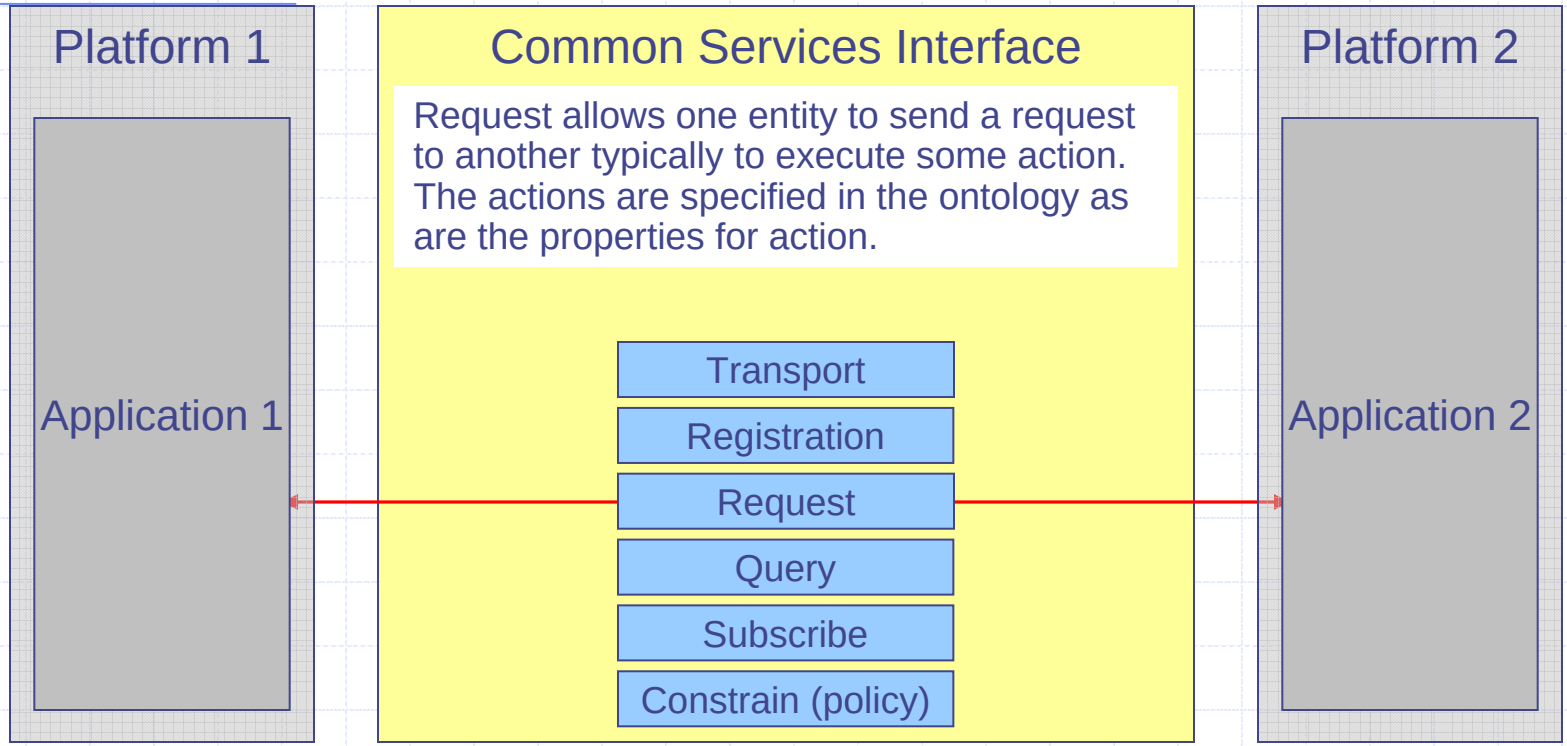
message

CSI: Functional Registration and Matching Service



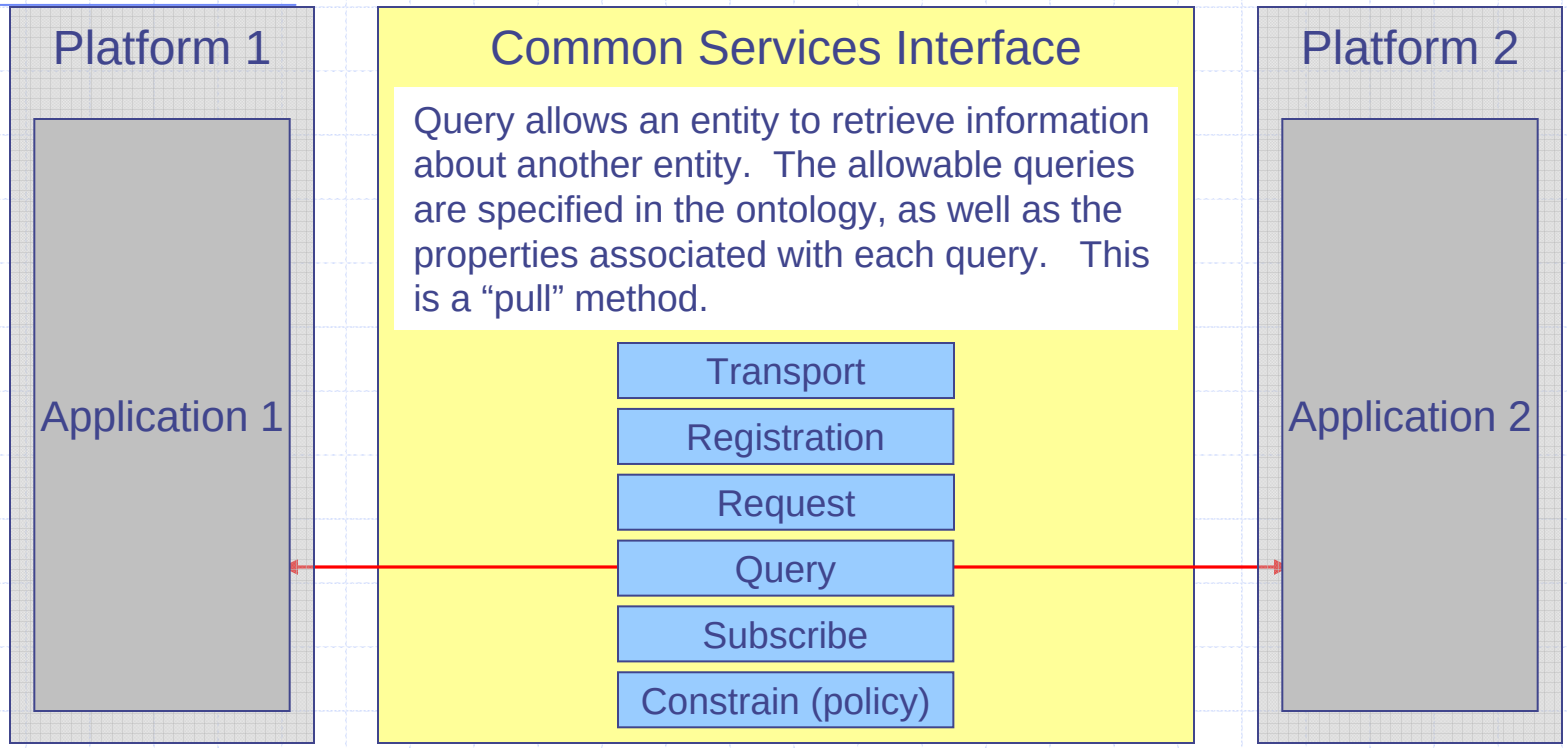
I have a camera.

CSI: Request Service



Take a picture of me.

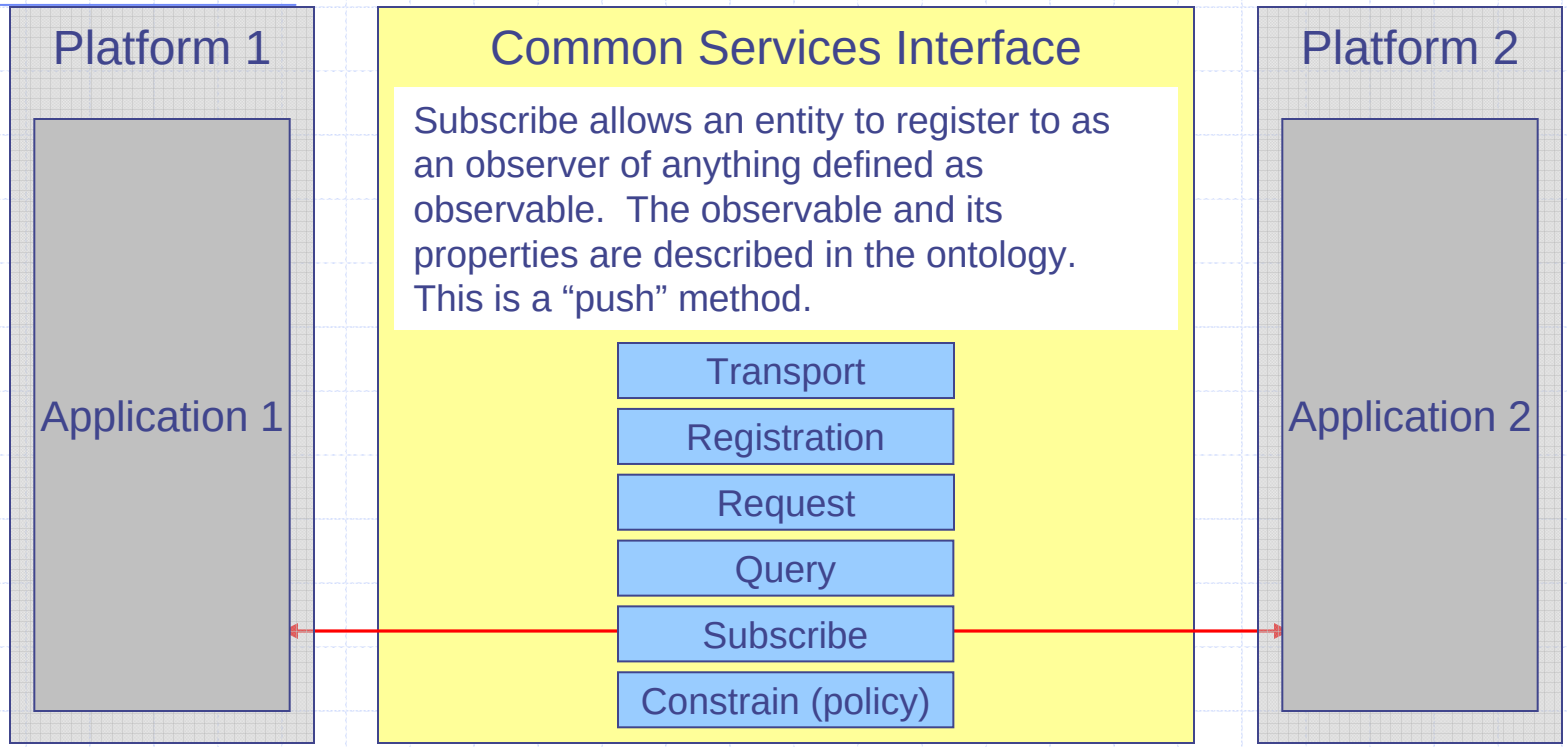
CSI: Query Service

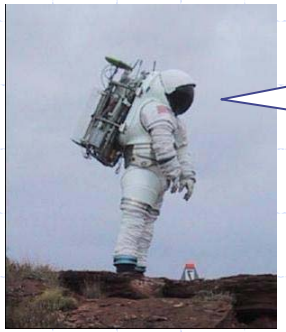


What is your current position?

I am at (32.41N, 87.26W).

CSI: Subscribe Service



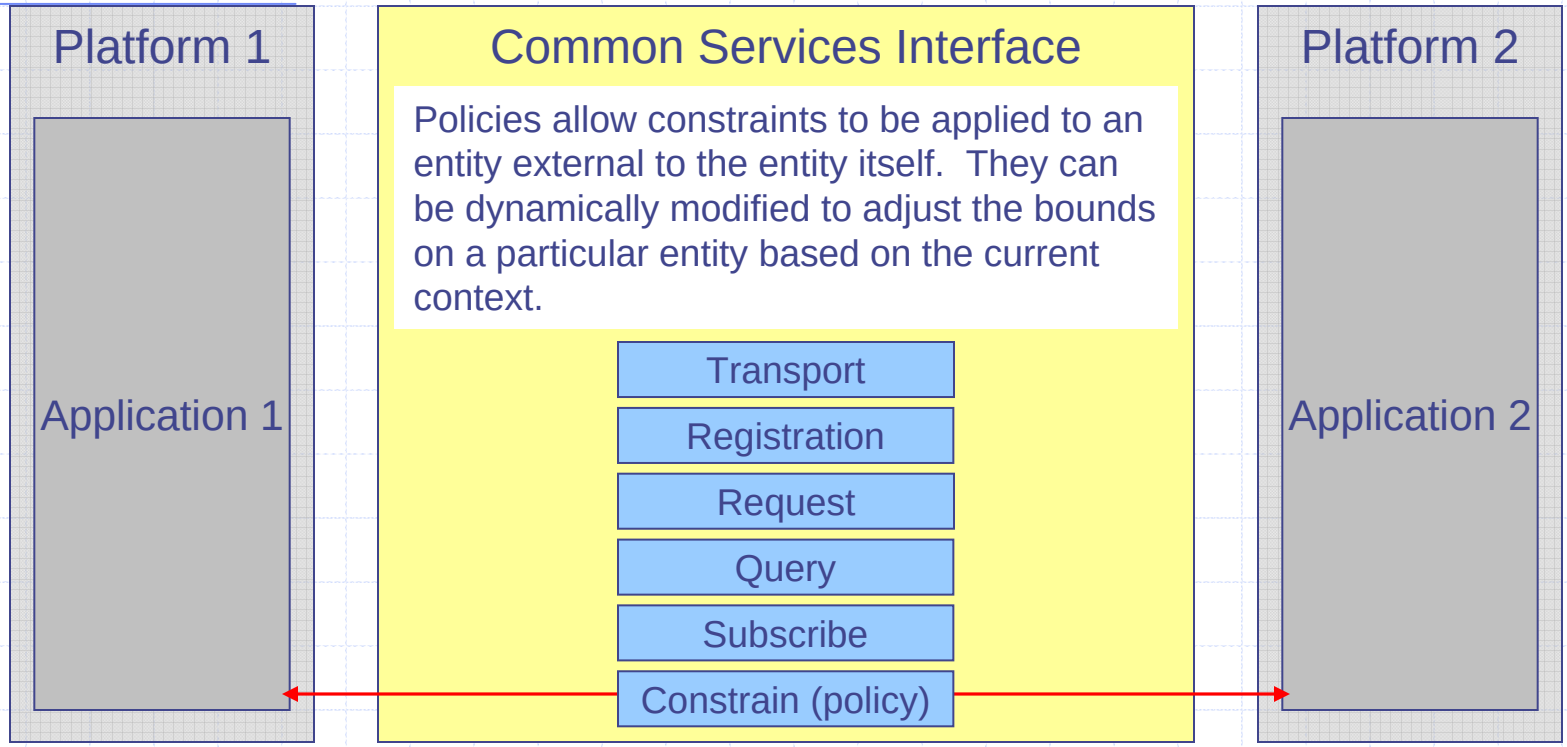


Let me know if your position changes

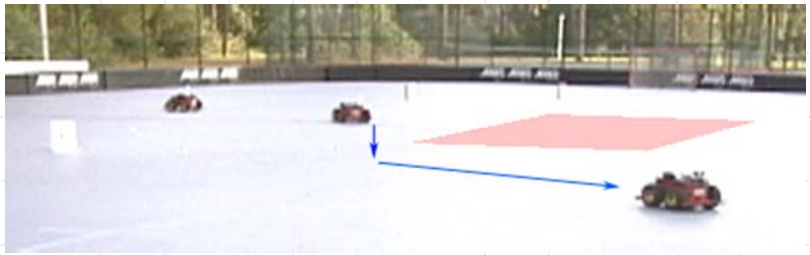
Now I am at (2,4)

I am at (2,3)

CSI: Constrain (policy) Service



Human can view constraint



Robot complies with constraint

KAoS API

Interface to KAoS Policy System

- Who uses this interface:
  - **Actor** which wants to learn about policies applicable to its current situation
  - **Enforcer** which intercepted the actor attempt to perform a given action and will enforce policies on it
- Makes Transparent complexity of policy reasoning
- Requires description of the **Action Instance**:
  - Context – includes subject ID or its credentials
  - Ontology-base Action class name
  - List of ontology-based names of the action context elements and their values for this action

KAoS CSI API data structure

The Action Instance Description (AID) is the key data.

Currently three methods to create it:

- By separate calls to the AID interface; adding action properties one be one
- By passing a hash map to the constructor containing property name and value mapping
- By passing an OWL description of the instance in a string

A user can use any one (or a combination of) the interface options.

Example of Action Instance Description

- *Actor invokes an operation with properties*

PropertyName	**Example Value**
Subject (ActorId)	IntellOfficer32
ActionClassName(s)	ActionConcepts.RetriveDocument
ActionConcepts. _documentClassification	TopSecrete
ActionConcepts. _documentSubject _	NATOActionInAsia

Basic CSI Policy Methods

- NameSpace: **kaos.core.csi.policy**
- **checkPermissionFor**
  (ActionInstanceDescription aid)
  - Checks if the given action is permitted according to the current set of policies.
- **getObligationsForTriggerCondition**
  (ActionInstanceDescription triggerAID)
  - Based on the specified trigger condition described by an ActionInstanceDescription, select all matching obligations and return them as ActionInstanceDescriptions.

KAoS Spatial Reasoning Component (Ksparc)

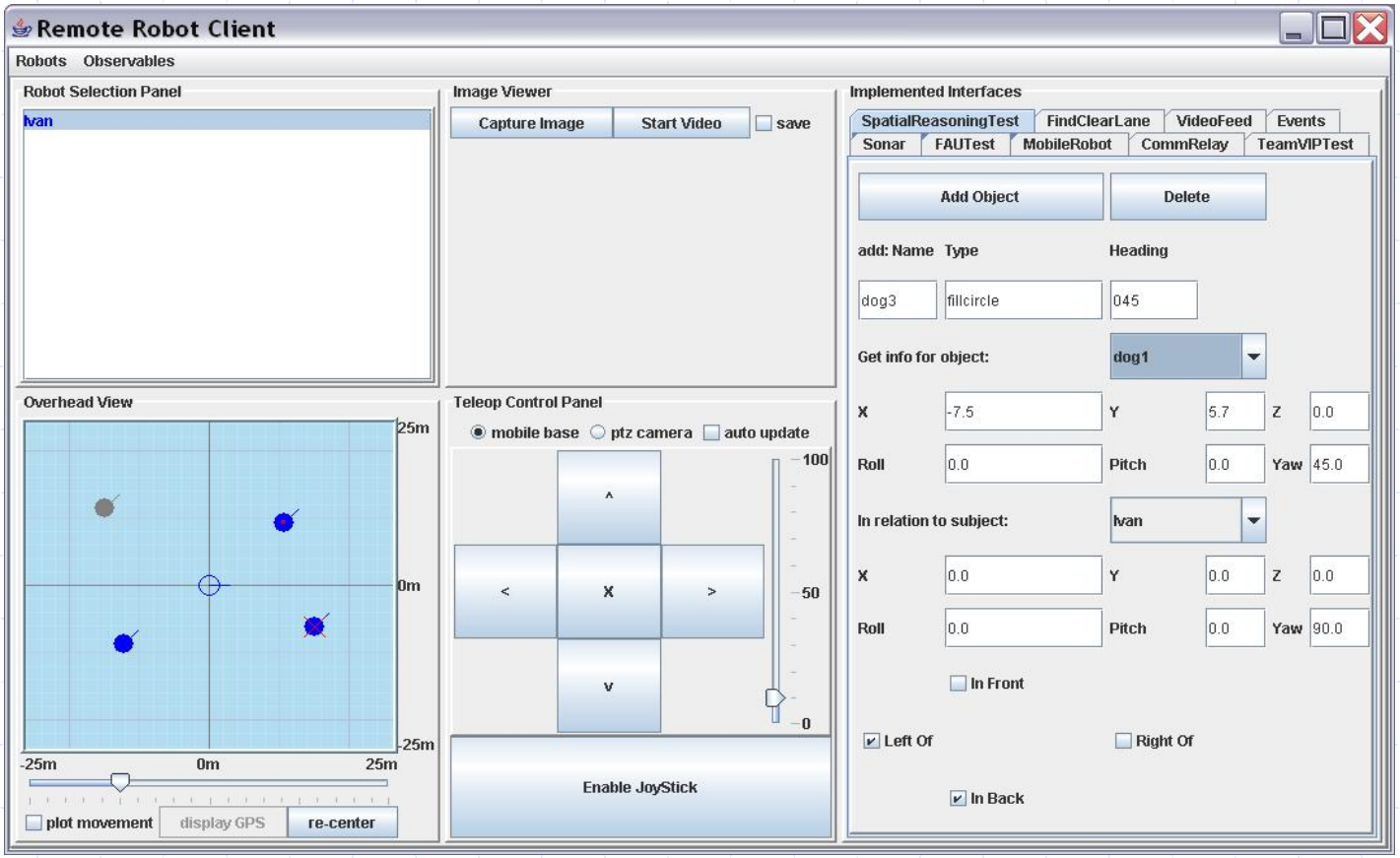
Ksparc features

- Integrated with KAoS Services Framework
- Allows to querying for relative and absolute spatial relations among people, objects, and robots
- Calculates absolute values of the relations: distance, angles, etc
- Consist of set of local spatial reasoners and a global spatial reasoner coordinating the reasoning

Ksparc usage

- Support spatial elements in robot human dialogs
- Allows for both absolute and egocentric references and recalculation of spatial point of reference
- Allows to check policies that contain spatial information

Example reasoning result



The relations are calculated between the centrally located Robot and the gray object.

Further information:

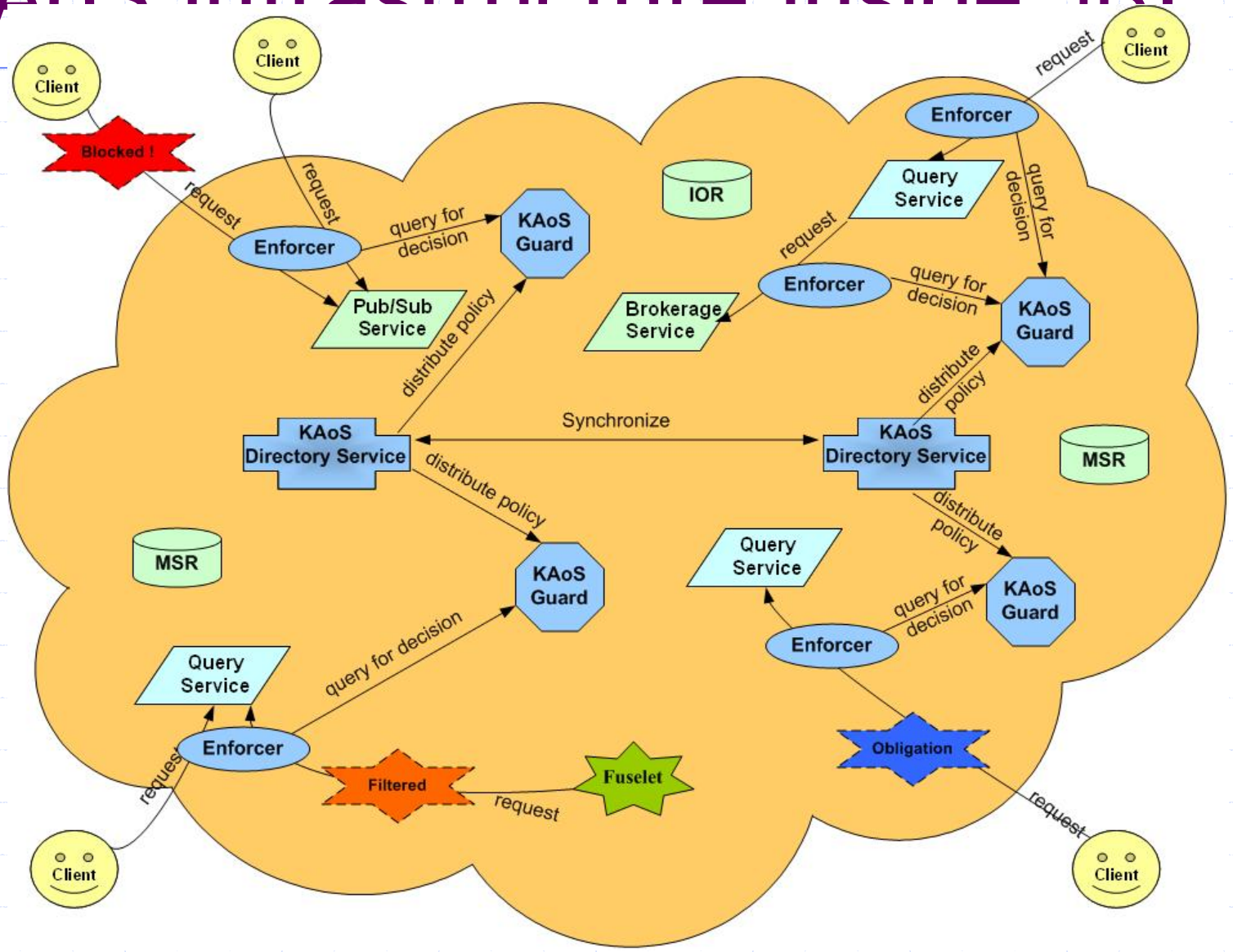
http://ontology.ihmc.us

Joint Battlespace Infosphere

KAoS Policies for JBI Access Control

- Enhancement of Air Force Research Laboratory combat information management system **together with ISX in the J-DASP project**
- Access control to information and services limited
- KAoS allows for extension of the existing JBI access control triple to full semantic description of the controlled situation
- Ontology vocabulary allow for declarative specification of policy and applicability based on context
- Quality of Service policies supporting needs of tactical environment

KAoS infrastructure inside JBI



Communities of Interest

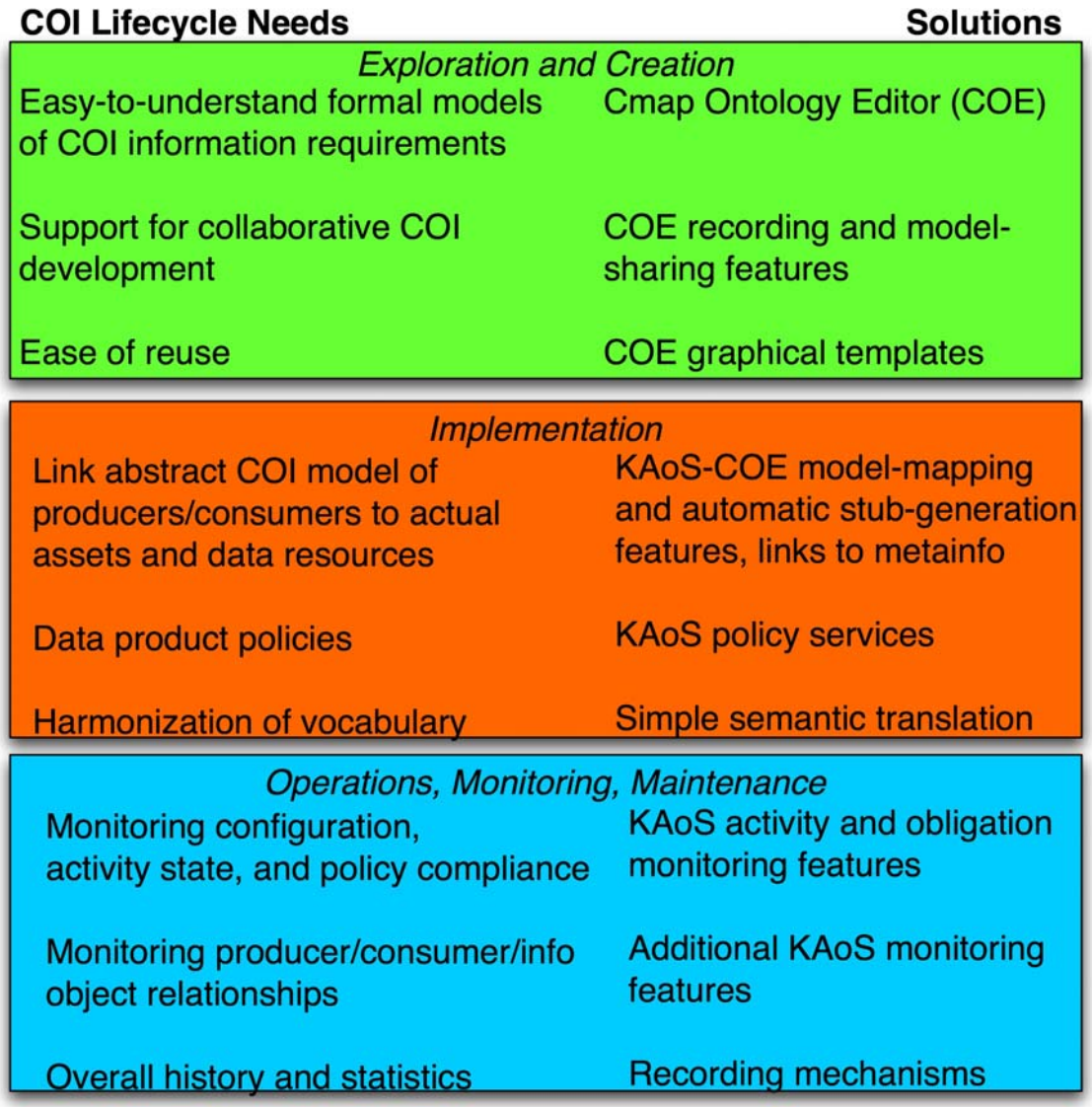
COI Concepts

- Assets and partnerships situated in the context of the mission
- Specific producers, consumers, data product and policies
- Many types of information must be captured in an easy-to-understand form:
  - information needs – COI scope,
  - types of information
  - types of consumers
  - infospace managers
  - applications used by consumers
  - degree of information integration
  - information security activities
  - consensus set of vocabulary terms and definitions
- Allows partners to determine whether the aggregated assets are adequate for performing the mission

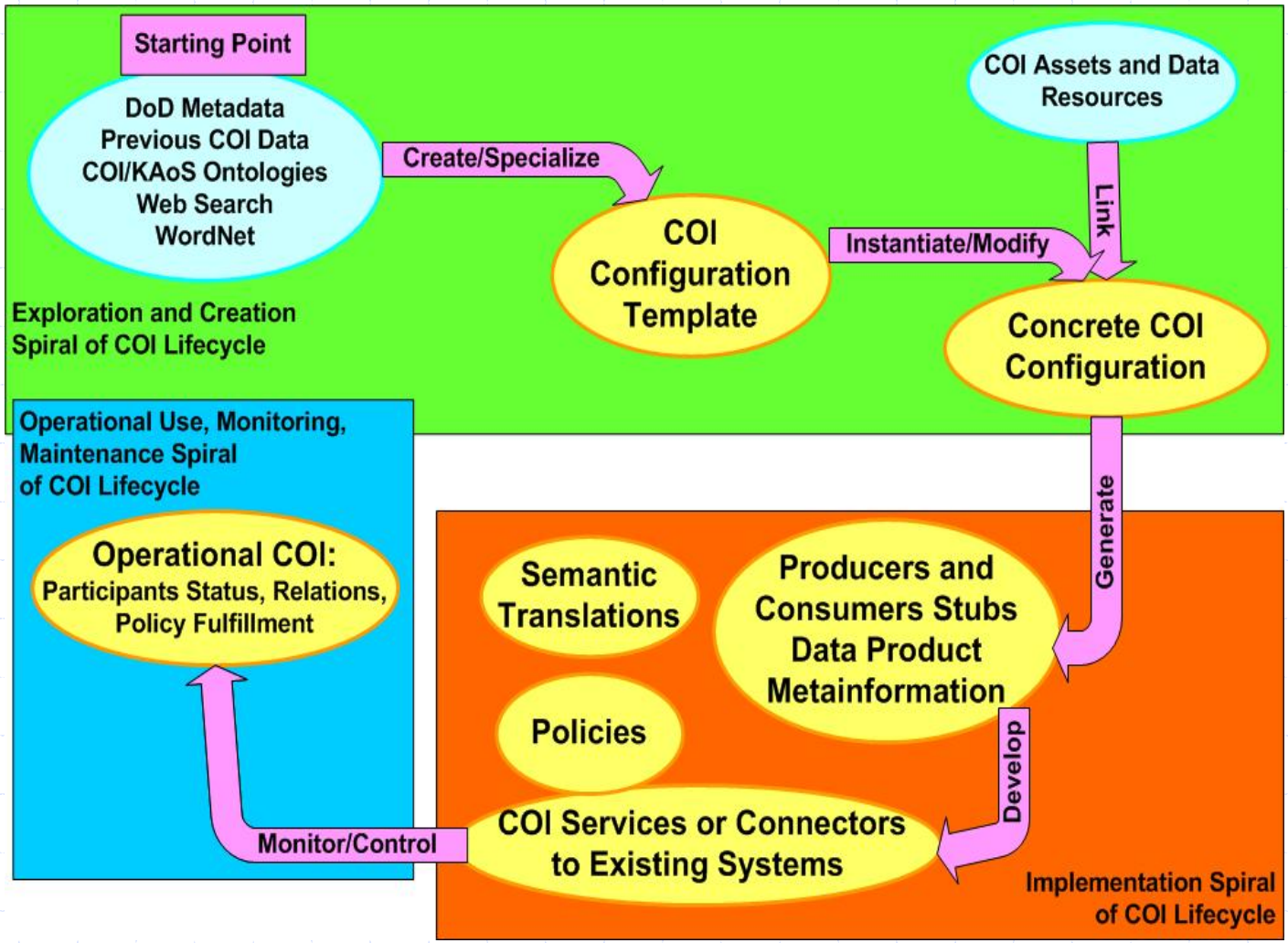
Support for the COI Lifecycle

- Exploration
  - Defining COI goals, consumers and producers of information, semantics of information, policies, etc.
- Implementation
  - Grounding to the operating platform; mapping to the JBI infrastructure
  - Definition of needed semantic transformations
- Operations
  - Monitoring community dynamics, relationships among participants, maintaining the community

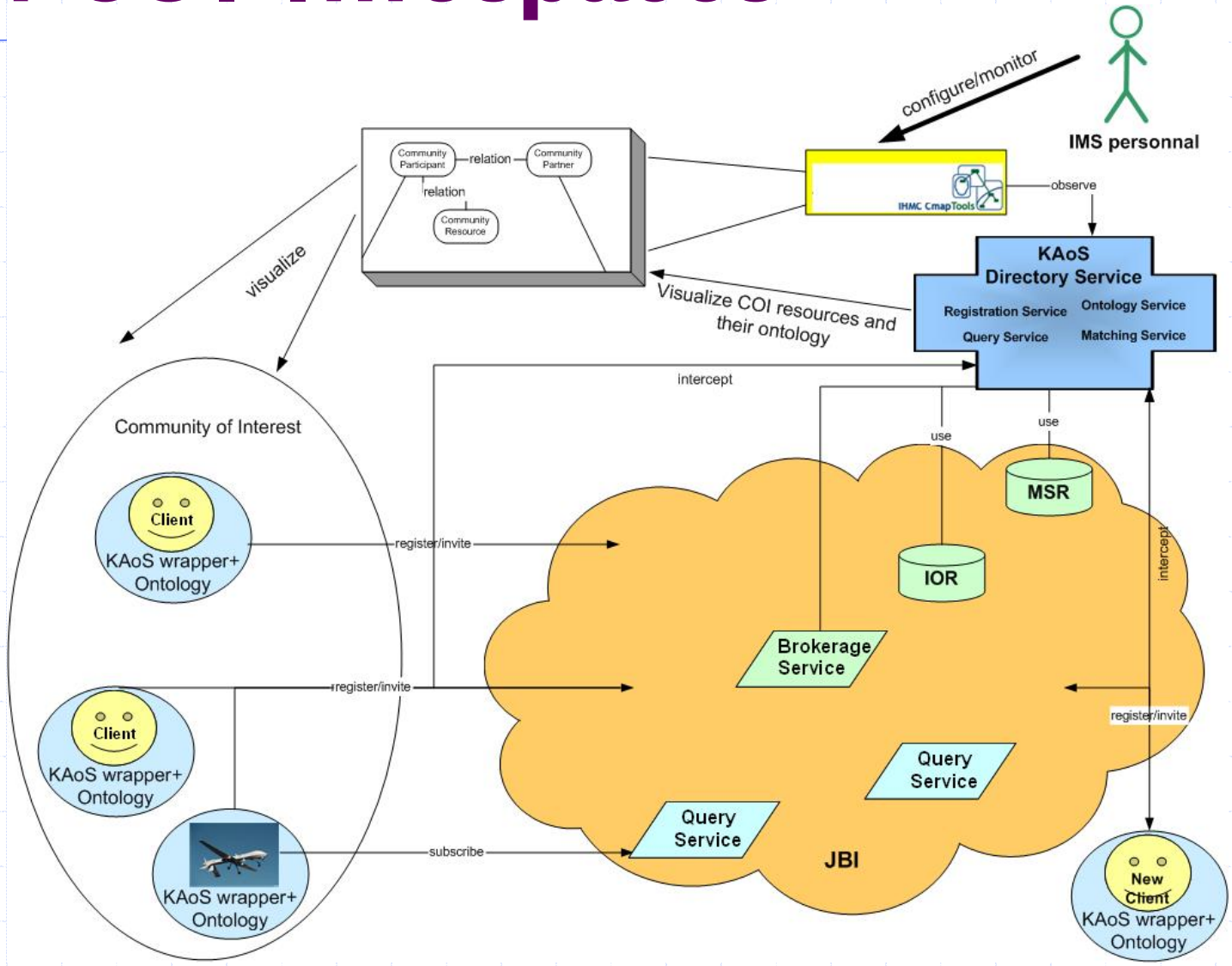
Requirements-Features Overview



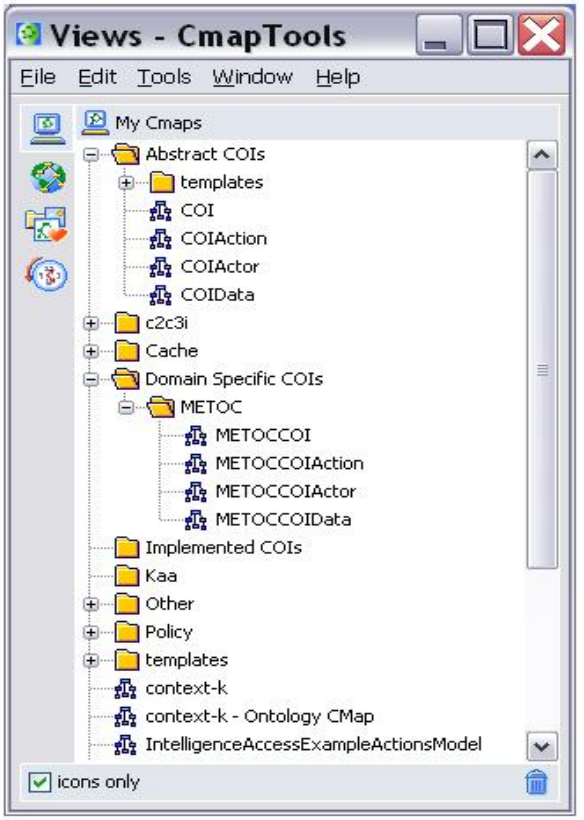
COI-Tool Dataflow



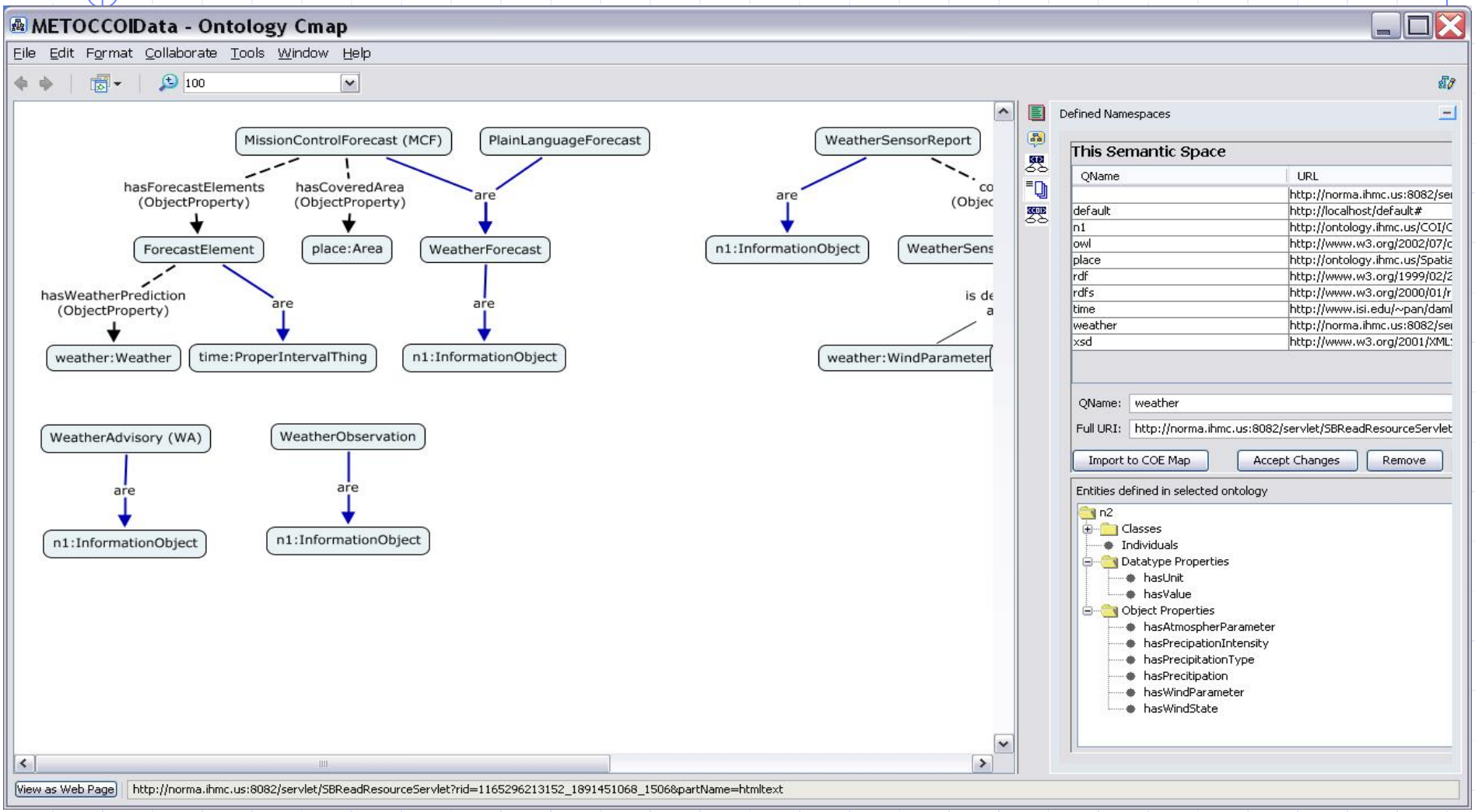
KAoS-CMap-JBI Integration for COI Infospaces



Create the New METOC COI



Graphically Define Ontology of Weather Concepts and Products



and
XML Schemas for Each Partner

Seamless and Secure Federation Among Highly- and Loosely-Connected Infospaces

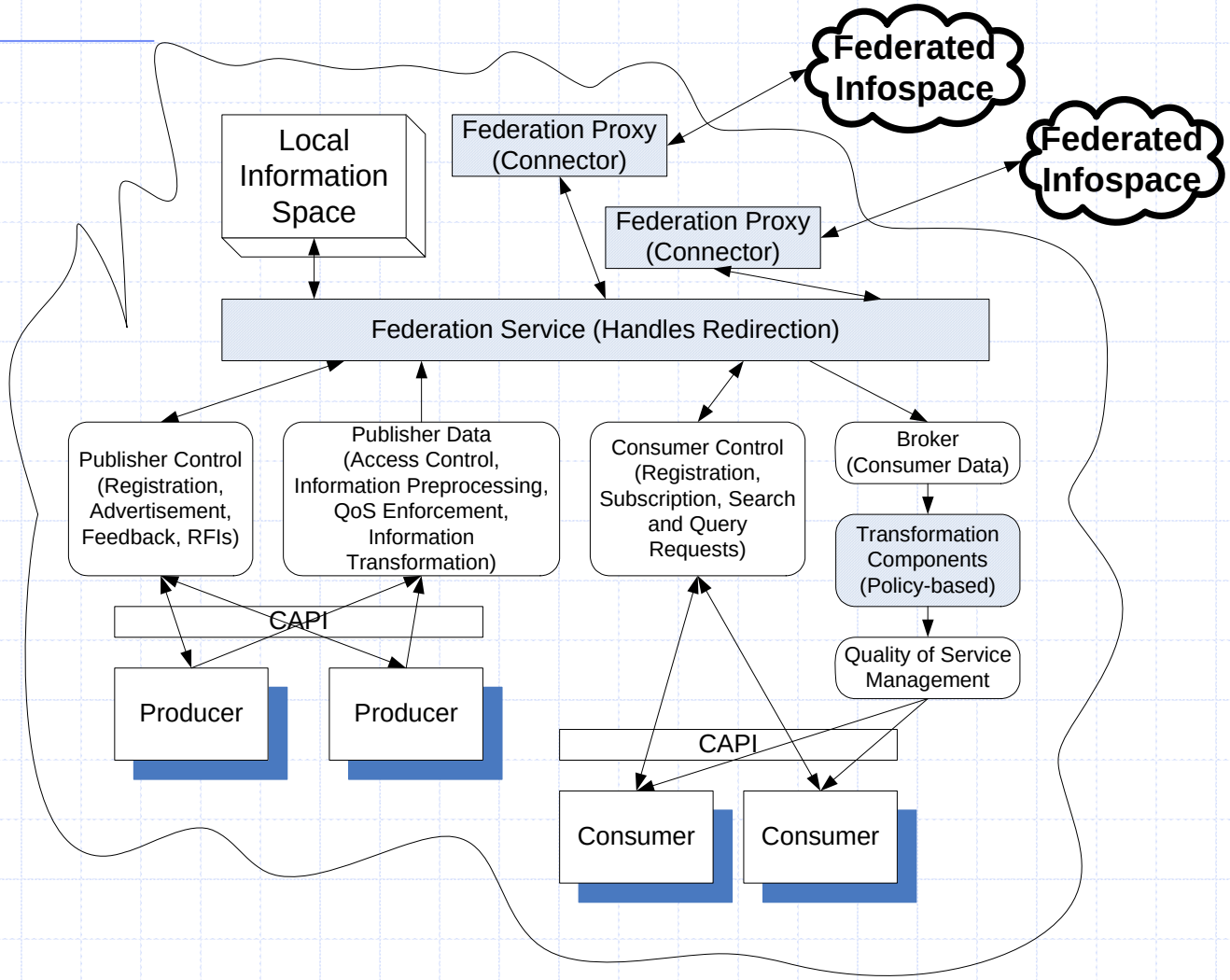
Vision for Federation

- Sharing of MIOs across infospaces
  - Seamless subscriptions and queries across infospaces
  - Transparency to CAPI clients
  - Controlled via policies – not unrestricted
  - Identity and integrity of individual infospaces preserved
- Efficiency when handling subscriptions and queries
  - Criteria: Latency, Bandwidth, Storage, Availability
- Dynamic translation of compatible MIOs
  - Different schemas
  - Restrictions on MIO content sharing
  - Bandwidth/efficiency requirements

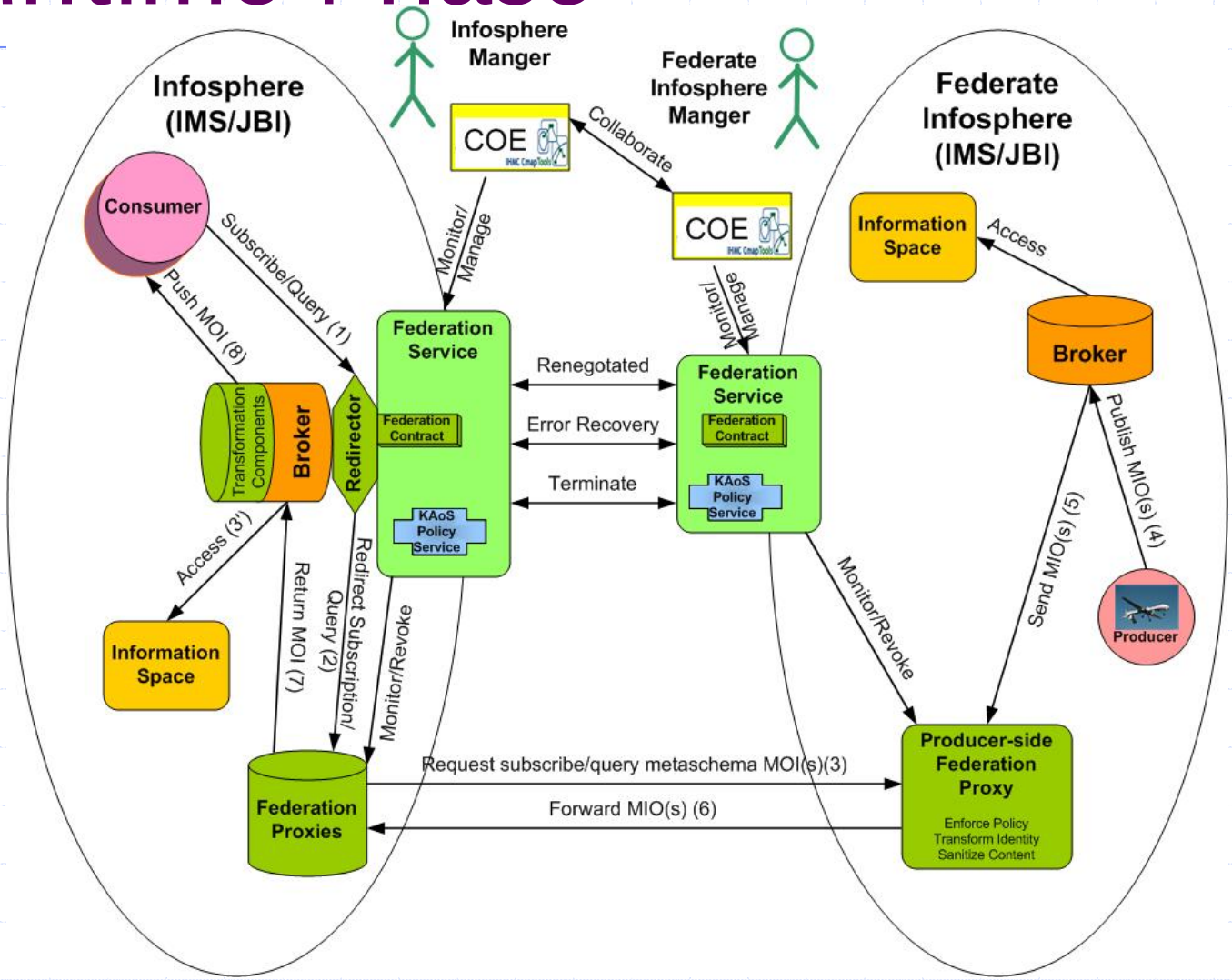
Project Goals

- Interface (API) Design for Federation
- Enable Federation Between Infospaces
  - Enterprise – Enterprise
  - Tactical – Tactical
  - Enterprise – Tactical
  - Not limited to just two instances
- Control Federation Through Policies and Contracts
- Optimize Federation via Adaptive Caching and Replication
- Work with Multiple, Existing Implementations

Overall Architecture



Runtime Phase



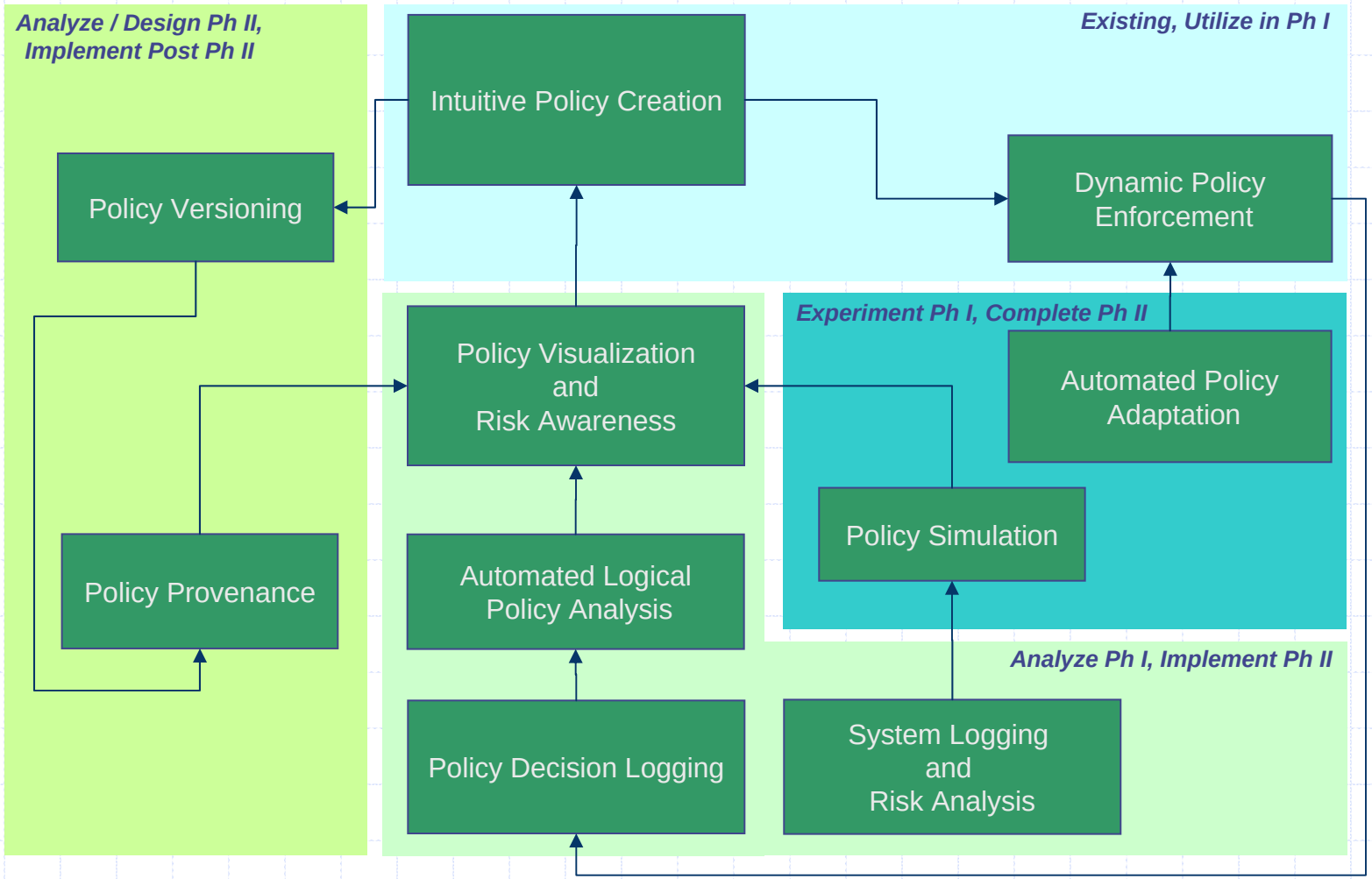
GRASP - Generic, Risk-Adaptive, Semantically-rich Policy Framework

Project with ISX for AFRL

GRASP Goals

- Provide security administrators with enough information to explicitly express who gets access to what information under varying operating conditions
- Grant assurances of secure enforcement through mathematically-grounded analysis, runtime simulation, and clear visualization of policy
- Maintaining awareness of the security of the system in the face of malicious attack is of the utmost importance, and GRASP provides the capabilities to stay aware of the weaknesses of the system and stay in control of the policies in force in the case of attack

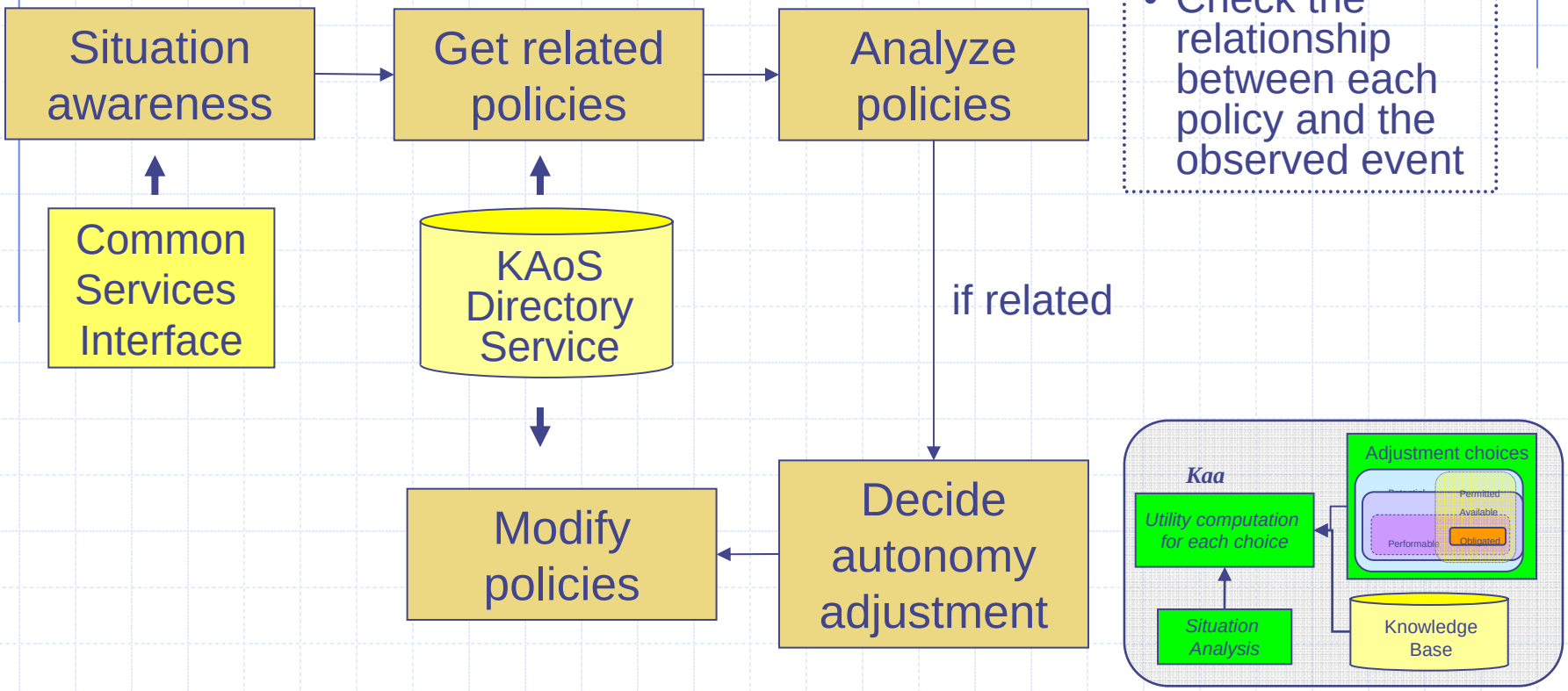
GRASP Framework



Kaa: KAoS Adjustable Autonomy

- Adjustable autonomy
  - Ability to impose and modify constraints that affect the range of actions an agent is capable of performing or is permitted or required to perform
  - Intent of adjustment is to lead to measurably better overall performance of the human-agent team in a given context
- Support for adjustable autonomy in KAoS
  - Context-based policy adaptation
    - Dynamically adjust policies to enable quick response to threats and optimization of overall system performance
  - Considers the utility of various choices for autonomy adjustment
    - Reasoning based on dynamically built decision network

Autonomy Adjustment for Observed Situation in the World

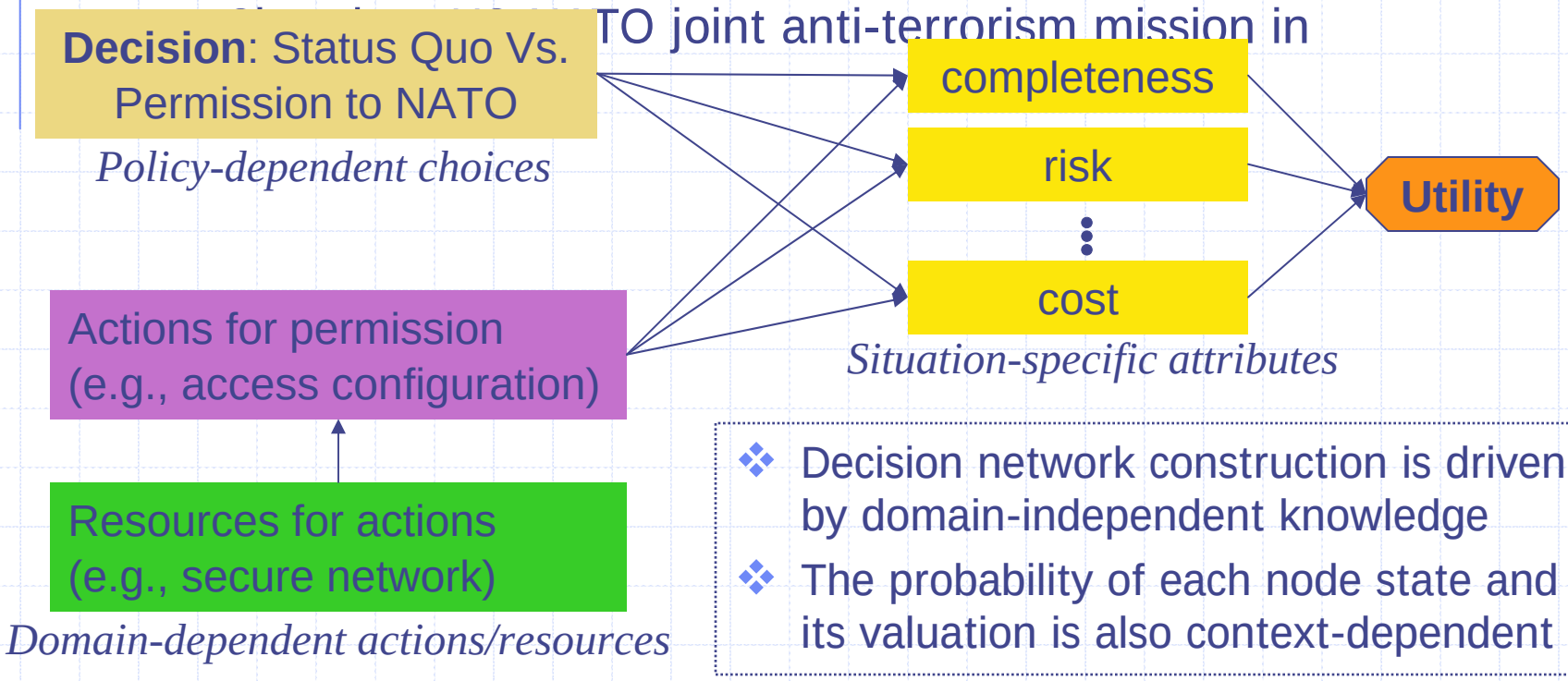


- Check the relationship between each policy and the observed event

- Decision-theoretic algorithm based on influence-diagram

Context-based Influence Diagram

- Example context
  - Policy: Deny intel-doc access to NATO (negative authorization)



- Decision network construction is driven by domain-independent knowledge
- The probability of each node state and its valuation is also context-dependent

Ontology-based Declarative Knowledge for Kaa

- Limitation of the current decision model
  - Most knowledge used to build a decision network is static
  - Difficult to handle context-specific dynamic information
  - Complicated transition to a new domain
- Solution approach
  - Represented the information required for Kaa *declaratively in the ontology*
    - Computational knowledge: node probability and valuation
    - Logical knowledge: causal relationship between nodes
  - Provided KPAT-like user interface to define the knowledge
  - Developed necessary mechanisms to dynamically construct influence diagrams
    - For a real world problem, the diagram can be very complex with multi-layers