Enforcing Context-Sensitive Policies in Collaborative Business Environments

Alberto Sardinha, Jinghai Rao, Norman Sadeh
School of Computer Science, Carnegie Mellon University
{alberto,jinghai,sadeh}@cs.cmu.edu

Abstract

As enterprises seek to engage in increasingly rich and agile forms of collaboration, they are turning towards service-oriented architectures that enable them to selectively expose different levels of functionality to both existing and prospective business partners. This includes enforcing access control policies whose elements are tied to changing contractual relationships or to information obtained from external sources (e.g. ratings, credit worthiness, export restrictions, etc.). To ensure maximum openness, we argue that such sources of contextual information should themselves be represented as web services that can be identified and accessed on the fly, as required to enforce relevant policies. We propose an architecture for enforcing context-sensitive access control policies in which sources of information can be annotated with rich semantic profiles. This includes a meta-control architecture for dynamically orchestrating policy reasoning together with the identification and access of external sources of information required to enforce policies. We show that this architecture can be implemented as an extension to XACML's PIP and context handler functionality. We proceed to show that our architecture extends to a broader class of corporate and regulatory policies. The paper also presents computational experiments aimed at evaluating the scalability of our architecture.

1. Introduction

Global competition is forcing companies to move towards increasingly versatile organizational structures. Business processes and supporting applications are expected to be easily reconfigurable to accommodate constantly changing business practices and relationships. In response, enterprises are increasingly turning towards service-oriented architectures in which functionality is selectively exposed in the form of composable web services - both within and across companies. This trend goes hand in hand with a need to capture and enforce ever richer sets of policies that help mediate interactions among these entities. These range from role-based access control policies, to workflow management policies, all the way to a variety of corporate, legal and regulatory policies (e.g. procurement policies, Sarbanes Oxley, taxation, ITAR, etc.). A central requirement for flexibility and openness in these emerging environments involves moving away from policies whose elements are necessarily tied to predetermined sources of information. Instead policies should be allowed to include elements whose evaluation requirements change with the context at hand. For instance, checking that an employee is authorized to request a vacation may involve verifying that he has obtained permission from his department head. Enforcing such a policy across multiple departments may involve consulting different services depending on the particular department an employee works in. Similarly checking the credit worthiness or rating of prospective business partners may involve dynamically identifying a service that can provide this information. In this paper, we propose a service-oriented architecture for enforcing such *context-sensitive policies*. Within this architecture, sources of information available to enforce context-sensitive policies are modeled as web services that can be annotated with rich semantic profiles. We describe a meta-control architecture for dynamically orchestrating policy enforcement with the identification, selection and access of relevant sources of contextual information. We show that, in the case of access control policies, this architecture can readily be implemented as an extension to XACML's PIP and context handler functionality [17]. We also discuss more general implementations of our Policy Enforcing Agents (PEAs) and report on computational experiments aimed at evaluating the scalability of our architecture.

The remainder of this paper is organized as follows. Section 2 provides a brief overview of related work. Section 3 introduces the concept of Policy Enforcing Agents (PEAs) along with a meta-control architecture for coordinating the enforcement of context-sensitive policies. An Access Control example is presented in Section 4. This includes a discussion of how our meta-control functionality can be implemented as an extension of the XACML architecture. Section 5 illustrates how the same model extends to more general sets of policies in a scenario involving a combination of corporate and regulatory policies for a fictitious aerospace contractor. Section 6 provides additional implementation details and presents results suggesting that our architecture scales fairly well. Section 7 concludes with some final remarks.

2. Related Work

The work presented in this paper builds on concepts of decentralized trust management developed over the past decade (see [3] as well as more recent research such as [2,13,16]) . Most recently, a number of researchers have started to explore opportunities for leveraging the openness and expressive power associated with semantic web frameworks in support of decentralized trust management (e.g. [1, 5, 6, 11, 14, 15, 26, 27] to name just a few). Our own work in this area evolved from the initial development of semantic web policy reasoning engines [9, 10] to that of more flexible Policy Enforcing Agents, including work in the context of mobile and pervasive computing applications [21]. The main contribution of the work discussed here is in the development and initial evaluation of a meta-control model for opportunistically interleaving policy reasoning and dynamic service identification and access to enforce context-sensitive policies. This includes a discussion of how the framework can be implemented as an extension of the XML Access Control Markup Language (XACML) standard [16]. Other relevant work on decentralized trust management languages includes the Security Assertion Markup Language (SAML) [18], the Enterprise Privacy Authorization Language (EPAL) [7] and the Platform for Privacy Preferences (P3P) [20] to name just a few. [12] also describes a semantic web policy framework for distributed policy management. The framework allows policies to be described in terms of deontic concepts and speech acts. It has been used to encode security policies of web resources, agents and web services. Work by Uszok et al. has also resulted in the integration of KAoS policy services with semantic web services [24]. Our initial work on Policy Enforcing Agents has relied on an extension of OWL Lite known as ROWL to represent security and privacy policies that refer to concepts defined with respect to OWL ontologies [9, 10, 23, 24]. While ROWL has been a convenient extension of OWL [31] to represent and reason about rules, it is by no means the only available option. In fact, ROWL shares many traits with several other languages. This includes RuleML [22], a proposed standard for a rule language, based on declarative logic programs. Another is SWRL [12], which uses OWL-DL to describe a subset of RuleML. The focus of the present paper is not on semantic web rule languages but rather on a meta-control architecture for enforcing context-sensitive policies. This architecture has been implemented to support XACML access control policies as well as more general ROWL policies. Scenarios involving both of these implementations are discussed. For the purpose of this paper, the reader can simply assume that the expressiveness of our own ROWL language is by and large similar to that of a language like SWRL, with both languages supporting the combination of Horn-like rules with one or more OWL knowledge bases.

3. Policy Enforcing Agents

As enterprises aim to (semi-)automatically enforce a variety of policies, they need to rely on reasoning engines (or “trust engines”) to make or recommend policy decisions (e.g. whether or not to authorize access to a resource, whether a new design change is consistent with relevant export restrictions, etc.). We use the term *Policy Enforcing Agents* (PEA) to refer to coordinating entities that encapsulate these reasoning engines and help orchestrate local reasoning with the collection of external information necessary to enforce policies. Rather than assume that all such external information can be obtained from predetermined sources, we believe that a truly open architecture calls for a more flexible service-oriented infrastructure within which relevant information sources are themselves modeled as web services and can be annotated with rich semantic profiles. When necessary, PEAs can use these profiles to dynamically identify and select among multiple possible sources of information.

PEAs receive *decision requests* (or simply “*queries*”) to render decisions based on policies, which themselves are encapsulated in the form of rules (see Figure 1). A meta-controller is responsible for tracking the status of each incoming query and orchestrating its processing. The latter is done by successively activating the policy reasoner itself as well as an information collection module, which can draw on both local knowledge as well as external sources of information – including possible interactions with users. All communication with the outside is assumed to be encrypted and digitally signed.

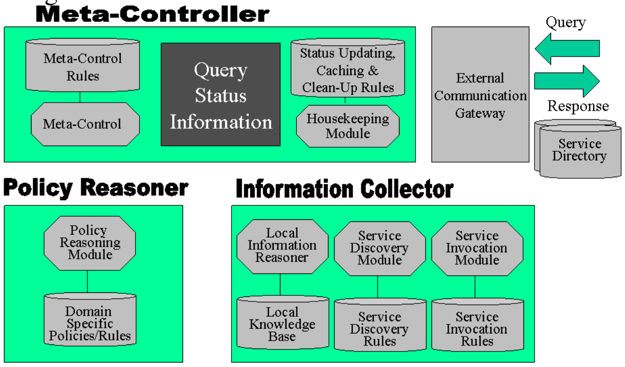


Figure 1. PEA Architecture.

Meta-control rules support the implementation of different orchestration strategies, from simple sequential control flows to more sophisticated processes capable of automatically accessing directories and concurrently collecting information from multiple sources. Strategies are executed by selectively activating different PEA modules (e.g. policy reasoning module, local information reasoner, service invocation module, etc.). This is further detailed later in this and other Sections.

In our current implementation, the meta-controller and information collection modules are rule-based engines implemented in JESS [8]. For efficiency reasons they are implemented as separate modules within the same JESS reasoning engine (i.e. each module comes with its own set of rules and control can be passed back and forth between the modules). In some domains, we have also used JESS to implement the policy reasoning module, while in others we have wrapped "legacy" policy reasoners (e.g. Sun's XACML Policy Decision Point used for the work described in Section 4).

3.1 Meta-controller

A PEA's *Meta-Controller* consists of a Meta-Control submodule, a Housekeeping submodule, and a Query Status Information knowledge base. As the PEA processes incoming queries, its meta-controller monitors progress and determines what to do next. Specifically, it continuously cycles through the following three basic steps:

1. The Meta-Control submodule analyzes the latest query status information and decides which of the PEA's module(s) to invoke next to perform particular tasks (e.g. obtaining information required to evaluate a policy or invoking the policy reasoner). As it invokes these modules the Meta-Control submodule updates relevant query status information (e.g. updating the status of a query from "not yet processed" to "being processed", identifying query elements that still need to be evaluated, etc.).
2. Modules complete their tasks (whether successfully or not) and report back to the Meta-Controller – occasionally modules may also report on their ongoing progress in handling a task.
3. The Housekeeping submodule updates detailed status information based on information received from other modules and performs additional housekeeping activities (e.g., cleaning up status information that has become obsolete, caching the results of recent requests for possible re-use and to mitigate the effects of possible denial of service attacks, etc.)

3.2. The Query Status Model

Query status information helps the PEA monitor how far along it is in obtaining the information required by each decision request (or "query") and in reaching a decision. It is expressed according to a taxonomy of predicates intended to help keep track of different activities typically involved in reaching a policy decision. This includes the status of individual queries as well as the status of query elements they give rise to. Examples of query elements include the evaluation of particular rules (e.g. "If the requester is a preferred supplier, it can have access to our component requirements forecast", or "if the employee has not exhausted his annual holiday quota and his vacation request has been authorized by his department head, the vacation request is authorized). Query elements are also used to model the need to obtain information required to evaluate individual rules (e.g. "is this particular company a preferred supplier?", "is this request for vacation authorized by the employee's department head?" or "which department does this employee work for?"). Processing query elements may in turn generate new query elements, whose statuses also need to be tracked. Accordingly, query status information includes whether a query (or query element) has been or is being processed, what individual query elements it has given rise to, whether these elements have been processed, etc. All status information is annotated with time stamps. Specifically, query status information includes:

- **Status predicates** to describe the status of a query or query element
- **A query ID** or **query element ID** to which the predicate refers
- **A parent query ID** or **parent query element ID** to help keep track of dependencies (e.g. a query element may be needed to help evaluate another query element). These dependencies, if passed between PEA agents, can also help detect deadlocks (e.g. two PEA agents each waiting for information from the other to enforce access control policies)
- **A time stamp** that describes when the status information was generated or updated. This information is critical when it comes to determining how much time has elapsed since a particular module or external service was invoked. It can help the agent look for alternative external services or decide when to prompt the user (e.g. to decide whether to wait any longer).

A sample of query status predicates is provided in Table 1. Clearly, different taxonomies of predicates can lead to more or less sophisticated meta-control strategies. For the sake of clarity, status predicates in Table 1 are organized in five categories: 1) communication; 2) query; 3) query elements; 4) service discovery and 5) service invocation. Additional categories of status predicates can also be introduced to deal with the requirements associated with different types of policies (e.g. to help orchestrate the evaluation of different types of policies such as a combination of export control policies and supplier selection policies as illustrated in Section 5).

Query status information is updated by asserting new facts (in the query status information knowledge base), with old statuses being cleaned up. As query updates come in, they trigger one or more meta-control rules, which in turn result in additional query status information updates and the invocation of one or more modules (e.g. policy reasoning module, local information reasoner, etc.).

	Sample Status Predicates	**Description**
1)	Query-Received	A particular policy decision request ("query") has been received.
	Sending-Response	A policy decision is being returned/sent
	Query-Deadlock-Detected	An incoming query has been identified as possibly resulting in a deadlock. Depending on the meta-control rules implemented in the PEA, different courses of action are possible (e.g. returning a failure message or prompting the user)
	Response-Sent	A response (or policy decision) has been successfully sent
	Response-Failed	A response failed to successfully reach its destination (e.g. message bounced back)
2)	Processing Query	Query is being processed
	Query Decomposed	Query has been decomposed (into one or more query elements)
	All-Elements-Available	All query elements associated with a given query are available (e.g. all the required information is available)
3)	Element-Needed	A query element is needed. Query elements may result from the decomposition of a query or may be needed to enforce policies. The query element's origin helps distinguish between these different cases
	Processing-Element	A need for a query element is being processed
	Element-Available	Query element is available
	Element-locally-unavailable	The value of a query element can not be obtained from the PEA's local knowledge base
4)	Element-need-service	A query element requires the identification of a source of information ("service")
	No-service-for-Element	No service could be identified to help answer a query element. This predicate can be refined to differentiate between different types of services (e.g. local versus external)
	Service-identified	One or more relevant services have been identified to help answer a query element
5)	Waiting-for-service-response	A query element is waiting for a response to a query sent to a service (e.g. query sent to an HR service or to a service operated by a trading partner, etc.)
	Failed-service-invocation	A service invocation failed. Again this predicate can be refined to distinguish between different types of failure (e.g. service down, access denied, etc.)
	Service-response-time-out	The time taken by a service to respond is greater than some threshold value. This eventually results in a failed-service-response
	service-response-available	A response has been returned by the service. This will typically result in the eventual creation of an "Element-Available" status update.

Table 1.Sample of status predicates. Status predicates typically have multiple arguments, which are not shown here (e.g. query ID or query-element ID, time-stamp, etc.)

3.3 Policy Reasoner

The PEA's policy reasoning engine is responsible for evaluating relevant policies and returning policy decisions. In the context of access control policies, this module plays a role equivalent to XACML's Policy Decision Point (PDP) [17], as further detailed in Section 4.

In this paper, for the sake of simplicity, we assume that all relevant policies are stored within the policy reasoner or in a centralized knowledge base (or database) accessible to the policy reasoner. In general, policies may come from multiple sources (e.g. combination of department policies, corporate policies and government regulations). If this is the case, a more general policy collection module similar to the PEA's information collection module might be required to identify all relevant policies. Some policies could also be embedded in other PEAs, which could themselves be modeled as external sources of information. For example, checking whether an employee has departmental approval to request a vacation could be performed by querying a departmental service, which could evaluate corresponding policies on the fly. This latter configuration is covered by the architecture presented in this paper.

While some policies can be evaluated just based on facts contained in the agent's local knowledge base, in general they require obtaining information from a combination of both local and external sources. When external sources of information are required, the *Policy Reasoner* models the missing information as "*Query Elements*". The status of query elements is recorded in the Query Information Status knowledge base, just as the statuses of original queries.

Scenarios presented in this paper rely on two different policy reasoners:

1. A generic JESS policy reasoner capable of enforcing a broad range of policies. Policies are expressed as ROWL rules [10,32] that refer to concepts specified in domain-specific ontologies written in W3C's OWL language [31]. ROWL has been used to specify a number of policies, from access control policies, to obfuscation policies, to message processing policies, etc.
2. Sun's XACML Policy Decision Point implementation, which evaluates XACML decision requests against XACML access control policies. In this configuration, the Sun PDP engine is wrapped to interoperate with our PEA architecture. This includes translating output from the Sun PDP engine into query status information.

3.4 Information Collector

The Information Collector is responsible for gathering facts (or "information") required to evaluate a given decision request. It works under the supervision of the meta-controller, which orchestrates policy reasoning and information collection. Facts required to evaluate policy decision requests may be known locally or may have to be obtained from other sources of information. Accordingly, the Information Collector comprises a Local Information Reasoner, a Service Discovery submodule, a Service Invocation submodule. Note that the users themselves could be modeled as services that can be queried for missing information. The Local Information Reasoner corresponds to domain knowledge (facts and rules) known locally to the PEA. The Service Discovery submodule helps the PEA identify potential sources of information to complement its local knowledge.

External services can be either pre-identified (using *service identification rules* such as "When checking if someone is a company employee, ask the company's HR service") or found with the help of *directories (e.g.* "find services that provide supplier ratings"), whether internal to a given organization or external to it. Clearly, service identification rules mapping information needs onto specific services can yield significant speedups. At the same time, the ability to rely on more general service discovery processes that involve querying service directories and identifying matches based on rich service annotations can provide for significantly greater levels of openness. By allowing service discovery rules to include both direct service identification rules and more complex discovery and comparison rules, PEAs allow policy developers to selectively choose between both options.

It is worth noting that, in principle, each service can have its own PEA. As requests are sent to services, their PEAs may in turn respond with requests for additional information to enforce their own control policies (e.g. access control policies). In general, if policies are not carefully coordinated, this could result in deadlocks. If all relevant policies are internal to a given organization, it is in principle possible to avoid such a situation. In general however, this needs not be the case and different services may be operated by different organizations, each with its own set of policies. Such situations can always be handled using time-outs.

As already indicated, PEAs can possibly treat users as sources of additional domain knowledge. It is worth noting that users can also serve as potential sources of meta-control knowledge (e.g. if a particular query element proves too difficult to locate, the user may be asked whether to give up).

3.5. The Service Discovery Model

A central element of our framework is the ability of PEA agents to dynamically identify sources of information needed to process queries. Sources of information are modeled as semantic web services and may operate subject to their own policies enforced by their own PEA agents. Accordingly service invocation is itself implemented in the form of queries sent to a service's PEA agent.

In this paper, we use WOWL (Web services in OWL) to annotate services, as this language has the merit of being fairly compact [10]. We have also implemented variations of our architecture using the OWL-S language [19] and could readily adopt other equivalent frameworks (e.g. WSMO [28]). A WOWL service description includes:

1. The service's output.
2. Its preconditions
3. Relevant non-functional attributes [25], if any
4. A description of how to invoke the service, including the service's endpoints and its input

In our current implementation, we use an XSLT transformation to convert WOWL service profiles into service discovery rules expressed in Jess. The discovery rules are expressed as "if-then" clauses - or "Left Hand Side" (LHS) *implies* "Right Hand Side" (RHS). The LHS refers to the types of facts a given service can provide (as specified in its output) and includes the service's preconditions and input parameters. The RHS creates a matching "*service-identified*" status predicate. In other words, given an '*element-need-service*" status predicate indicating that one is looking for a service that can provide a particular type of fact, all matching services whose preconditions and input conditions are also satisfied will trigger matching service discovery rules. As they are triggered, these rules will in turn result in the creation of matching "service identified" status predicates indicating that any of these services can possibly yield the desired information. The meta-controller can later decide which one(s) of the services to actually query – depending on its particular meta-control rules.

4. Access Control Agents: An Example of PEAs

A particularly important class of Policy Enforcing Agents (PEAs) is that of *Access Control Agents (ACA)*. These agents can be viewed as extensions of OASIS's XACML architecture for enforcing access control policies[17]. Specifically, Figure 2 shows the architecture of an ACA based on Sun's XACML *Policy Decision Point* (PDP) engine [30]. Incoming decision requests (or "queries") are directed to the agent's Meta-Controller which doubles as XACML *Policy Enforcing Point* (PEP). Queries are converted from their native format to XACML, using a language adaptor, which essentially subsumes part of the XACML *Context Handler* functionality, with the other part being handled by the meta-controller. Missing information is dynamically identified through interactions between the meta-controller and the Information Collector, the latter playing the role of XACML *Policy Information Point* (PIP).

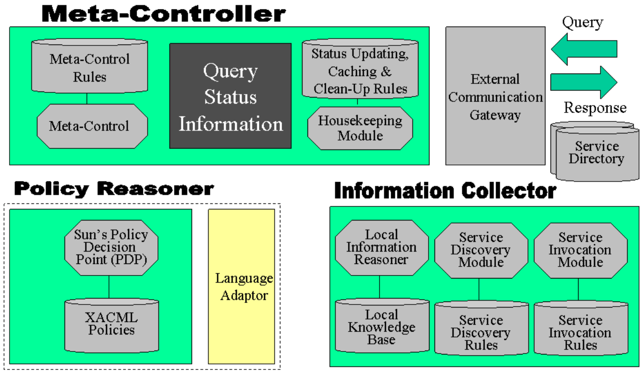


Figure 2. PEA Instantiated as an Access Control Agent using Sun's XACML Policy Decision Point engine.

In this regard, an Access Control Agent (ACA), namely an access control instantiation of a Policy Enforcing Agent (PEA), can be seen as a direct extension of the XACML standard, with the addition of a meta-controller responsible for orchestrating the collection of information required to evaluate policies, including the dynamic identification, selection and access of external sources of information.

4.1 An Aerospace Contractor Scenario

The following further illustrates our implementation of one such agent in support of access control requirements associated with a fictitious aerospace contractor, which we refer to as *United GenSat Corporation*. United GenSat is a California-based manufacturer of geostationary satellites. It builds two lines of communications satellites: the SAT 666 and the SAT 777. These two lines of satellites are designed to support mobile communications, and a series of global positioning and military communications applications.

```
<Rule RuleId="Pre-approvedSupplierRule"
                             Effect="Permit">
 <Target>
  <Subjects>
      <AnySubject/>
  </Subjects>
  <Resources>
   <Resource>
    <ResourceMatch MatchId="string-equal">
     <AttributeValue
            DataType="&XMLSchema;#string">
       ProductionSchedule
     </AttributeValue>
     <ResourceAttributeDesignator
            DataType="&XMLSchema;#string"
            AttributeId="resource-id"/>
    </ResourceMatch>
   </Resource>
  </Resources>
  <Actions>
   <Action>
    <ActionMatch MatchId="string-equal">
     <AttributeValue
            DataType="&XMLSchema;#string">
       query
     </AttributeValue>
     <ActionAttributeDesignator
            DataType="&XMLSchema;#string"
            AttributeId="action-id"/>
    </ActionMatch>
   </Action>
  </Actions>
 </Target>
 <Condition FunctionId="string-equal">
  <Apply FunctionId="string-one-and-only">
   <SubjectAttributeDesignator
            DataType="&XMLSchema;#string"
            AttributeId="SupplierCategory"/>
  </Apply>
  <AttributeValue
            DataType="&XMLSchema;#string">
    Pre-approved
  </AttributeValue>
 </Condition>
 <Condition FunctionId="string-equal">
  <Apply FunctionId="string-one-and-only">
   <SubjectAttributeDesignator
            DataType="&XMLSchema;#string"
            AttributeId="AuthorizedEmployee"/>
  </Apply>
  <AttributeValue
            DataType="&XMLSchema;#string">
    Yes
  </AttributeValue>
 </Condition>
</Rule>
```

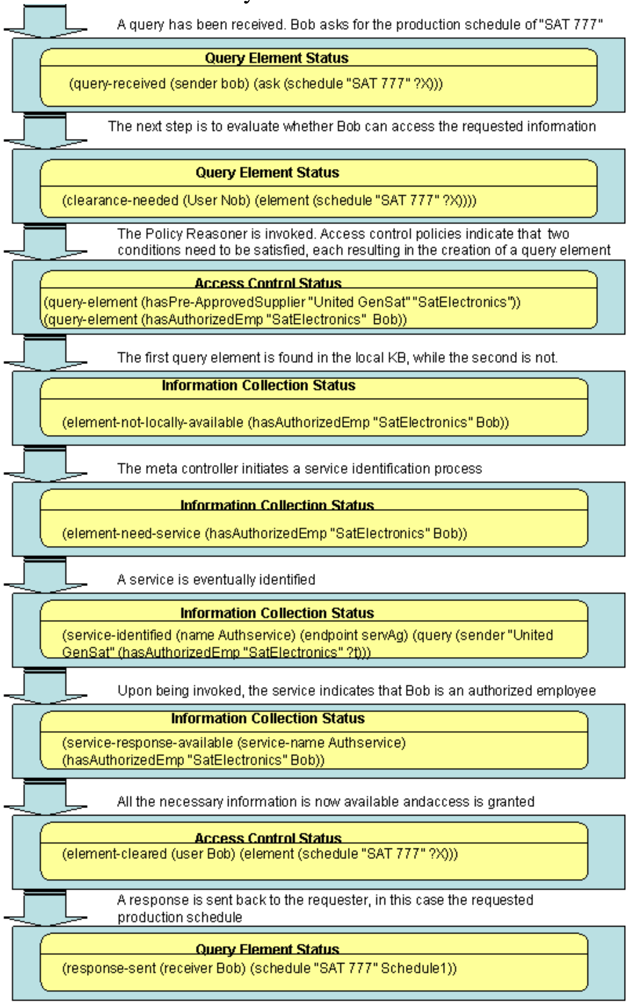
Figure 3 Sample XACML policy limiting access to Production Schedule information to authorized employees at pre-approved subcontractors.

Due to the sensitive nature of its activities and products, *United GenSat* is particularly concerned about maintaining tight control over who accesses what information both within its organization as well as in the context of interactions with its trading partners. These interactions include the selective exchange of scheduling information to ensure close coordination with key suppliers. Policies to control access to this information are expressed in XACML. An example of one such policy is provided in Figure 3. The policy only permits authorized employees (attribute of subject) of pre-approved suppliers (attribute of subject) to query (attribute of action) the production schedule of products it is contributing to (attribute of resource).

Consider Bob, an employee at *SATElectronics Corporation,* a United GenSat supplier pre-approved to access production schedule information of products it contributes to. Bob sends a request to United GenSat, requesting next month's production schedule for the SAT 777. His request, which includes the identity of his company, is forwarded to the appropriate United GenSat Access Control Agent (ACA). To determine whether to grant access to the requested information, the ACA needs additional information, namely (a) whether SATElectronics is pre-approved to obtain this information – for the sake of simplicity we will just assume that this information is maintained in the ACA's local knowledge base, and (ii) whether Bob is an authorized SATElectronics employee when it comes to accessing production schedule information. To answer this latter question, the ACA needs to identify a service at SATElectronics and send it a query.

.

Figure 4 depicts the main steps involved in processing Bob's request. Upon receiving the request, United GenSat's ACA generates an information status update indicating that a new query has been received. This information is expressed as a triple of the form *(predicate subject object)* – namely *(query-received (sender bob)(ask (schedule "SAT777" ?X)))*. Next, the meta-controller generates a new status update indicating that the request has to be cleared based on applicable access control policies, namely *(clearance-needed (User bob)(element (schedule "SAT777" ?X)))*. This status update in turn results in the meta-controller invoking the policy reasoner, which in turn leads to the creation of two query elements – one requiring to check whether Bob's company, SATElectronics, is pre-approved to access production schedule information and the other to check whether Bob is an authorized employee. The meta-control rules are assumed to first check the ACA's local knowledge base and find that SATElectronics is indeed pre-approved. On the other hand, Bob's authorized employee status cannot be determined locally. This results in the creation of an *element-not-locally-available* status predicate, which in turn leads to the creation of an “element-need-service” status predicate, followed by a service identification step. A SATElectronics service is identified and a response eventually provided indicating that Bob is an authorized employee. As a result, a status predicate is created indicating that Bob’s request has now been cleared – note that Figure 4 only depicts a subset of the steps involved in this scenario, and the production schedule is eventually returned to Bob..



A query has been received. Bob asks for the production schedule of "SAT 777"

(query-received (sender bob) (ask (schedule "SAT 777" ?X)))

The next step is to evaluate whether Bob can access the requested information

(clearance-needed (User Nob) (element (schedule "SAT 777" ?X))))

The Policy Reasoner is invoked. Access control policies indicate that two conditions need to be satisfied, each resulting in the creation of a query element

(query-element (hasPre-ApprovedSupplier "United GenSat" "SatElectronics"))
(query-element (hasAuthorizedEmp "SatElectronics" Bob))

The first query element is found in the local KB, while the second is not.

(element-not-locally-available (hasAuthorizedEmp "SatElectronics" Bob))

The meta controller initiates a service identification process

(element-need-service (hasAuthorizedEmp "SatElectronics" Bob))

A service is eventually identified

(service-identified (name Authservice) (endpoint servAg) (query (sender "United GenSat" (hasAuthorizedEmp "SatElectronics" ?t)))

Upon being invoked, the service indicates that Bob is an authorized employee

(service-response-available (service-name Authservice) (hasAuthorizedEmp "SatElectronics" Bob))

All the necessary information is now available andaccess is granted

(element-cleared (user Bob) (element (schedule "SAT 777" ?X)))

A response is sent back to the requester, in this case the requested production schedule

(response-sent (receiver Bob) (schedule "SAT 777" Schedule1))

Figure 4. Query Status Updates.

A particularly interesting step in this scenario is the one through which the ACA identifies a SATElectronics service capable of identifying whether Bob is an authorized employee. Different processing flows are possible here, depending on the particular meta-control rules and service discovery rules implemented in the ACA. In this particular implementation, the meta-controller first checks whether missing knowledge is available locally. If that fails (as in this case), it turns to the service discovery module. The service discovery module includes a number of rules aimed at making service identification as efficient as possible, as well as extremely general “fall-back” rules in case none of the more specialized rules produce results. In this example, we rely on a service identification rule for checking attributes of employees of other companies. The rule, in this simple scenario, just tells the ACA to check the company’s directory for a service capable of providing the necessary query element (i.e. whether Bob is an authorized employee). The directory is assumed to include a simple service called “AuthEmpService”, whose WOWL annotations indicate it can provide the missing information - namely whether the employee (whose name is provided as input) is an authorized employee of SAT Electronics (see wowl:output in Figure 5).

```
<wowl:ServiceRule wowl:salience="100">

<rdfs:label>SATElcEmpService</rdfs:label>
 <wowl:output>
  <scm:Company rdf:about="&var;#co">
   <scm:hasAuthorizedEmp
          rdf:resource="&var#emp"/>
  </scm:Company>
 </wowl:output>
 <wowl:precondition>
  <scm:Schedule rdf:about="&var;#sche">
   <scm:hasAccess
              rdf:resource="&var#emp"/>
   </scm:Schedule>
 </wowl:precondition>
 <wowl:precondition>
  <scm:Product rdf:about="&var;#product">
   <scm:hasSchedule
          rdf:resource="&var;#sche"/>
  </scm:Product>
 </wowl:precondition>
 <wowl:precondition>
  <scm:Company rdf:about="&var;#co">
   <scm:hasName
          rdf:resource="SATElectronics"/>
  </scm:Product>
 </wowl:precondition>
 <wowl:call>
  <wowl:Service wowl:name="AuthService">
   <wowl:endpoint>servAg</wowl:endpoint>
   <wowl:input>
    <scm:People rdf:about="&var#emp">
      <scm:hasName rdf:about="&var#nam"/>
    </scm:People>
    </wowl:input>
  </wowl:Service>
 </wowl:call>
</wowl:ServiceRule>
```

Figure 5. WOWL Service profile

The service’s precondition further indicates that this particular service is specifically to verify whether people are authorized to access production schedule information.

Admittedly, this scenario takes some short cuts. A more realistic variation would have to do a better job at dealing with confidentiality considerations and would likely involve multiple levels of indirection, with some service discovery performed by United GenSat's ACA and some performed locally by SATElectronics in response to a more general query from United GenSat. Nevertheless, once a service such as `AuthEmpService` in Figure 5 has been identified, its profile can be used to automatically generate an access request intended to verify whether Bob is an authorized employee. This step is performed by the ACA's service invocation module. It includes automatically generating the necessary service query along with additional facts required by the service as indicated in its input and precondition profile. In this particular case, the query is of the form:

```
(query
  (sender "United GenSat")
  (predicate "&scm;#SATElectronics")
  (subject "&scm;#hasAuthorizedEmp")
  (object "&scm#Bob"))
```

Based on the service's input profile, the following fact is sent along with the query:

```
(triple
  (predicate "&scm;#hasName")
  (subject "&scm;# Bob")
  (object "Bob"))
```

Clearly, this assumes that both the service provider and service requester share a common ontology. If not, semantic reasoning rules may be needed to establish a mapping between their respective ontologies.

5. Beyond Access Control Policies

PEAs are not limited to enforcing access control policies. The same meta-control architecture can be used to support more flexible processing flows when it comes to enforcing a broad range of policies. This is illustrated in this section by examining a scenario where United GenSat undertakes to develop a new satellite model, SAT 888, for a client in the UK. As it works on the design of the SAT 888 in collaboration with both current and prospective suppliers, the company needs to ensure compliance with a variety of policies. This includes compliance with corporate supplier selection policies as well as with US export control regulations (e.g. the US International Traffic in Arms Regulations, ITAR)

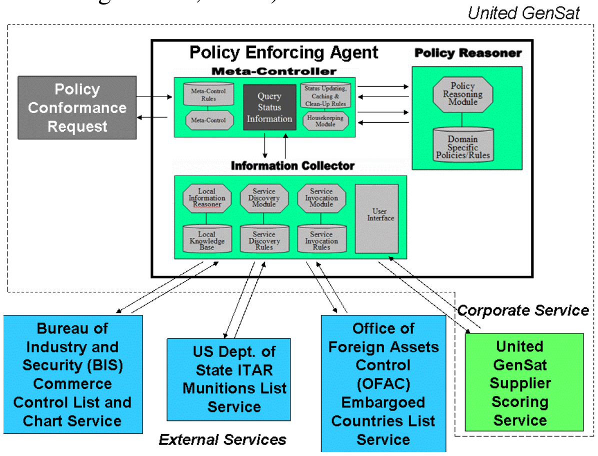

United GenSat
Policy Enforcing Agent
Policy Reasoner
Meta-Controller
Policy Conformance Request
Meta-Control Rules
Meta-Control
Query Status Information
Status Updating, Caching & Clean-Up Rules
Housekeeping Module
Policy Reasoning Module
Domain Specific Policies/Rules
Information Collector
Local Information Reasoner
Service Discovery Module
Service Invocation Module
User Interface
Local Knowledge Base
Service Discovery Rules
Service Invocation Rules
Corporate Service
Bureau of Industry and Security (BIS) Commerce Control List and Chart Service
US Dept. of State ITAR Munitions List Service
Office of Foreign Assets Control (OFAC) Embargoed Countries List Service
United GenSat Supplier Scoring Service
External Services

Figure 6. Using a PEA to check for compliance with supplier selection policies, including supplier scoring requirements and government export controls.

United GenSat relies on a specialized PEA to help it ensure compliance with these policies. As employees working on the SAT 888 refine their design and evaluate different options, they submit policy conformance requests to the PEA. This includes checking for compliance of sourcing decisions with both export control regulations and corporate supplier selection policies. These policies are expressed in ROWL and require accessing a combination of corporate and external services to obtain uptodate supply ratings and export restrictions. An example of such a ROWL rule is shown in Figure 7. It specifies that, when a product is to be exported (i.e. its country of destination is not equal to "USA"), it is approved for export if its country of destination and its Export Control Classification Number (ECCN) are not on the Bureau of Industry and Security (BIS) Commerce Control List. If the combination of the product's ECCN and export country appear in the list (in the form of a "CCLStatement"), then an export license has to be obtained.

As before, the PEA's meta-control module orchestrates the evaluation of these policies, looking for information in its local knowledge base and, when necessary, looking for services that can provide missing information. This latter step is performed with the help of the PEA's service discovery module. In this simple example, it is assumed that the required services are known ahead of time. In other words, the PEA can rely on simple service identification rules such as "When looking for a *CCLStatement*, issue a query to the *BIS Commerce Control List and Chart* Service".

```
<rowl:Rule>
 <rdfs:label>Export+Approval+Needed</rdfs:label>
 <rowl:head>
  <scm:Product rdf:ID="&var;#prod">
   <scm:hasExportApproval>
    <scm:ExportApprovalResult
rdf:resource="&scm;#true"/>
   </scm:hasExportApproval>
  </scm:Product>
 </rowl:head>
 <rowl:body>
  <scm:Product rdf:ID="&var;#prod">
   <scm:hasDestination rdf:resource="&var;#country"/>
  </scm:Product>
  <rowl:not>
   <scm:Country rdf:resource="&var;#country">
    <rowl:equal-to rdf:resource="&scm;#USA">
   </scm:Country>
  </rowl:not>
  <scm:CCLStatement rdf:resource="&var;#ccl">
   <scm:hasProduct rdf:resource="&var;#prod">
   <scm:hasECCN rdf:resource="&var;#eccn">
   <scm:hasCountry rdf:resource="&var;#country">
   <scm:hasReason rdf:resource="&var;#reason">
   <scm:hasResult rdf:resource="&var;#result">
  </scm:CCLStatement>
  <rowl:or>
   <scm:Result rdf:resource="&var;#result">
    <rowl:equal-to rdf:resource="&scm:#not_in_list">
   <scm:Result rdf:resource="&var;#result">
    <rowl:equal-to rdf:resource="&scm:#has_licence">
  </rowl:or>
 </rowl:body>
</rowl:Rule>
```

Figure 7. A ROWL export control compliance policy

Because the BIS, ITAR and OFAC services used in this scenario do not exist at this time (i.e. the current websites are not implemented as web services), our implementation of this scenario currently relies on stubs.

Going back to the ROWL policy listed in Figure 7, if there is a CCL Statement indicating that the product's ECCN and its export country are incompatible with export restrictions, the policy will result in the creation of an "Element-Needed" status predicate with attribute "has_license". In other words, the policy reasoner will let the meta-controller know that the only remaining option to satisfy this policy is to obtain an export license. This in turn could prompt the launch of a process to obtain such a license or it could lead the United SatGen employee who submitted the validation request to look for a different design. This shows how PEAs could also be integrated into workflow management functionality.

6. Evaluation

Our policy enforcing agents are currently implemented in JESS and have been integrated with two policy reasoners, a ROWL policy reasoner capable of enforcing a wide variety of policies and Sun's more specialized PDP reasoner to enforcce XACML policies. XSLT transformations are used to translate OWL classes and extensions of OWL (e.g. rules, queries and services descriptions) into CLIPS [4]. They are also used in the XACML language adaptor developed for Sun's PDP reasoner (see Figure 2).

We have evaluated the scalability of our architecture by running variations of the United GenSat Supplier Selection Policy scenario presented in the previous section. This involved running scenarios in which we increased the number of services available to obtain missing information. The experiements were run on an IBM laptop with a 1.80GHz Pentium M CPU and 1.50GB of RAM. The laptop was running Windows XP Professional OS, Java SDK 1.4.1 and Jess 6.1. After loading the initial knowledge base, the Jess engine contained a total of 3500 facts. A breakdown of the CPU times is provided in Table 2.

Number of services	100	200	500
Meta-Controller	860	861	1020
Local-KB	240	161	270
Service discovery and invocation	793	1041	1052
Total	1893	2063	2342

Table 2 Scalability Evaluation – CPU times in msec.

As can be seen here, the time required to enforce a policy conformance request is in the order of a couple of seconds and does not increase significantly with the number of available services.

7. Concluding Remarks

As enterprises seek to engage in increasingly rich and agile forms of collaboration, they are turning towards service-oriented architectures that enable them to selectively expose different levels of functionality to both existing and prospective business partners. This includes enforcing access control policies whose elements are tied to changing contractual relationships or to information obtained from external sources (e.g. ratings, credit worthiness, export restrictions, etc.). To ensure maximum openness, we have argued that such sources of contextual information should themselves be represented as web services that can be identified and accessed on the fly, as required to enforce relevant policies. We have proposed an architecture for enforcing context-sensitive access control policies in which sources of information can be annotated with rich semantic profiles. This includes a meta-control architecture for dynamically orchestrating policy reasoning together with the identification and access of external sources of information required to enforce policies. We have shown that our architecture for Policy Enforcing Agents can be implemented as an extension to XACML's PIP and context handler functionality. We proceeded to also show that it extends to a much broader class of corporate and regulatory policies and presented an example where a PEA is used to enforce sourcing policies, both corporate supplier selection policies and export control regulations .Initial experiments suggest that the computational requirements of our PEA architecture are acceptable and that it should scale relatively well.

A key advantage of our proposed architecture is that it does not prescribe a particular set of meta-control rules. Instead, it should enable companies or vendors to customize these rules based on the complexity of the policies they need to enforce and the level of flexibility they want to achieve when it comes to orchestrating policy reasoning with the dynamic identification and access of external sources of information.

Acknowledgements

The work reported herein has been supported in part by the National Science Foundation under ITR grant 0205435 ("Multiattribute Negotiation in Dynamic Supply Chains"), including a supplement to collaborate with the EU IST TrustCom project, and under Cyber Trust Grant CNS-0627513 ("User-Controllable Security and Privacy for Pervasive Computing") and in part by ARO under research grant DAAD19-02-1-0389 ("Perpetually Available and Secure Information Systems"). Early support for this work has also been provided by DARPA under contract F30602-02-2-0035 ("DAML initiative").

References

[1] R. Ashri, T. Payne, D. Marvin, M. Surridge and S. Taylor, "Towards a Semantic Web Security Infrastructure", *Proceedings of Semantic Web Services Symposium*, 2004.

[2] L. Bauer, M.A. Schneider and E.W. Felten, "A General and Flexible Access Control System for the Web", *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[3] M. Blaze, J. Feigenbaum, and J. Lacy, "Decentralized Trust Management". *Proceedings of IEEE Conference on Security and Privacy*. Oakland, CA. May 1996.

[4] CLIPS, http://www.ghg.net/clips/CLIPS.html.

[5] G. Denker, L. Kagal, T. Finin, M. Paolucci and K. Sycara, "Security For DAML Web Services: Annotation and Matchmaking", *Proceedings of the Second International Semantic Web Conference*, 2003.

[6] L. Ding, P. Kolari , T. Finin , A. Joshi, Y. Peng and Y. Yesha, "On Homeland Security and the Semantic Web: A Provenance and Trust Aware Inference Framework", *Proceedings of the AAAI Spring Symposium on AI Technologies for Homeland Security*, 2005.

[7] IBM, EPAL 1.1.
http://www.zurich.ibm.com/security/enterprise-privacy/epal/.

[8] E. Friedman-Hill, "*Jess in Action: Java Rule-based Systems*", Manning Publications Company, June 2003.

[9] F. Gandon, and N. Sadeh, "A semantic e-wallet to reconcile privacy and context awareness", *Proceedings of the Second International Semantic Web Conference (ISWC03)*, 2003.

[10] F. Gandon, and N. Sadeh, "Semantic web technologies to reconcile privacy and context awareness", *Web Semantics Journal*, 1(3), 2004.

[11] R. Hull, B. Kumar, D. Lieuwen, P. Patel-Schneider, A. Sahuguet, S. Varadarajan, and A. Vyas, "Enabling context-aware and privacy-conscious user data sharing", *Proceedings of 2004 IEEE International Conference on Mobile Data Management*, January 2004.

[12] I. Horrocks, P.F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof and M. Dean, "SWRL: Semantic Web Rule Language Combining OWL and RuleML", Version 0.6.

[13] T. van der Horst, T. Sundelin, K. E. Seamons, and C. D. Knutson, "Mobile Trust Negotiation: Authentication and Authorization in Dynamic Mobile Networks", *Eighth IFIP Conference on Communications and Multimedia Security*, Lake Windermere, England, 2004.

[14] L. Kagal, T. Finin, and A. Joshi, "A policy language for a pervasive computing environment", *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, 2003.

[15] L. Kagal, M. Paolucci, N. Srinivasan, G. Denker, T. Finin and K. Sycara, "Authorization and Privacy for Semantic Web Services", *Proceedings of Semantic Web Services Symposium, AAAI 2004 Spring Symposium Series*, Stanford University, California, March 2004.

[16] L.Bauer, S. Garriss, J. McCune, M.K. Reiter, J. Rouse, and P Rutenbar, "Device-Enabled Authorization in the Grey System", *Information Security: 8th International Conference, ISC 2005*, September 2005.

[17] T. Moses (Editor), "*Specification Document of XACML 2.0 Core: eXtensible Access Control Markup Language (XACML)*", OASIS, February 2005.

[18] OASIS, Security Assertion Markup Language (SAML), http://www.oasis-open.org/committees/security/.

[19] OWL-S: Semantic Markup for Web Services, http://www.w3.org/Submission/OWL-S

[20] A P3P Preference Exchange Language (APPEL1.0), http://www.w3.org/TR/P3P-preferences/.

[21] J. Rao and N.M. Sadeh, "A Semantic Web Framework for Interleaving Policy Reasoning and External Service Discovery", *Proceedings of International Conference on Rules and Rule Markup Languages for the Semantic Web*, Galway, Ireland, 10-12 November 2005.

[22] The Rule Markup Initiative, http://www.ruleml.org.

[23] N. M. Sadeh, T.C. Chan, L. Van, O. Kwon, and K. Takizawa, "Creating an open agent environment for context-aware m-commerce", *Agentcities: Challenges in Open Agent Environments*, 2003.

[24] N.M. Sadeh, F. Gandon, and Oh Byung Kwon, "Ambient Intelligence: The MyCampus Experience", Chapter in "*Ambient Intelligence and Pervasive Computing*", 2006.

[25] J. O'Sullivan, D. Edmond, and A.T. Hofstede, "What's in a service? Towards accurate de-scription of non-functional service properties", *Distributedand Parallel Databases*, 12:117.133, 2002.

[26] J. Undercoffer, F. Perich, A .Cedilnik, L. Kagal, and A. Joshi, "A secure infrastructure for service discovery and access in pervasive computing", *ACM Monet: Special Issue on Security in Mobile Computing Environments*, October 2003.

[27] A. Uszok, J. M. Bradshaw, R. Jeffers, M. Johnson, A. Tate, J. Dalton and S. Aitken, "Policy and Contract Management for Semantic Web Services", *Proceedings of Semantic Web Services Symposium*, AAAI 2004 Spring Symposium Series, Stanford California.

[28] Web Service Modeling Ontology (WSMO), http://www.wsmo.org/.

[29] XML Digital Signature, http://www.w3.org/TR/xmldsig-core/.

[30] Sun's XACML Implementation: http://sunxacml.sourceforge.net/.

[31] W3C: OWL Web Ontology Language Overview, W3C Recommendation, Feb. 2004. D. McGuinness & F. van Harmelen (Eds.) http://www.w3.org/TR/owl-features/

[32] F. Gandon, M. Sheshagiri, and N. Sadeh, "ROWL: Rule Language in OWL and Translation Engine for JESS". http://www.cs.cmu.edu/~sadeh/MyCampusMirror/ROWL/ROWL.html