A meta-control architecture for orchestrating policy enforcement across heterogeneous information sources

Jinghai Rao, Alberto Sardinha and Norman Sadeh

School of Computer Science, Carnegie Mellon University
5000 Forbes Avenue,
Pittsburgh, PA, 15213, USA
{jinghai; alberto; sadeh}@cs.cmu.edu

Keyword: Semantic Web, Context Sensitive Policies, Web services, security and privacy

Abstract There is increasing demand from both organizations and individuals for technology capable of enforcing sophisticated, context-sensitive policies, whether security and privacy policies, corporate policies or policies reflecting various regulatory requirements. In open environments, enforcing such policies requires the ability to reason about the policies themselves as well as the ability to dynamically identify and access relevant sources of information. This article introduces a semantic web framework and a meta-control model to orchestrate policy reasoning with the identification and access of relevant sources of information. Specifically, sources of information are modeled as web services with rich semantic profiles. Policy Enforcing Agents rely on meta-control strategies to dynamically interleave semantic web reasoning and service discovery and access. Meta-control rules can be customized to best capture the requirements associated with different domains and different sets of policies. This architecture has been validated in the context of different domains, including a collaborative enterprise domain as well as several mobile and pervasive computing applications deployed on Carnegie Mellon's campus. We show that, in the particular instance of access control policies, the proposed framework can be viewed as an extension of the XACML architecture, in which Policy Enforcing Agents offer a particularly powerful way of implementing XACML's Policy Information Point (PIP) and Context Handler functionality. At the same time, our proposed architecture extends to a much wider range of policies and regulations. Empirical results suggest that the semantic framework introduced in this article scales favorably on problems with up to hundreds of services and tens of service directories.

1 Introduction

The increasing reliance of individuals and organizations on the Web to help mediate a variety of activities is giving rise to a demand for richer policies (e.g. security and privacy policies as well as other corporate or regulatory policies) and more flexible mechanisms to enforce these policies. People may want to selectively expose sensitive information to others based on the evolving nature of their relationships, or share information about their activities under particular conditions. This trend requires context-sensitive security and privacy policies, namely policies whose conditions are not tied to static considerations but rather conditions whose satisfaction, given the very same actors (or principals), will likely fluctuate over time. Enforcing such policies in open environments is particularly challenging for several reasons:

- Sources of information available to verify these policies may vary from one principal to another (e.g. different users may have different sources of location tracking information made available through different cell phone operators);
- Available sources of information for the same principal may vary over time (e.g. when a user is on company premises her location may be obtained from the wireless LAN location tracking functionality operated by her company, but, when she is not, this information can possibly be obtained via her cell phone operator);
- Available sources of information may not be known ahead of time (e.g. new location tracking functionality may be installed or the user may roam into a new area).

Accordingly, enforcing context-sensitive policies in open domains requires the ability to opportunistically interleave policy reasoning with the dynamic identification, selection and access of relevant sources of contextual information. This requirement exceeds the capability of decentralized trust management infrastructures proposed so far and calls for policy enforcing mechanisms capable of operating according to significantly less scripted scenarios than is the case today. It also calls for much richer service profiles than those found in early web service standards.

We introduce a semantic web framework and a meta-control model for dynamically interleaving policy reasoning and external service identification, selection and access. Within this framework, external sources of information are wrapped as web services with rich semantic profiles allowing for the dynamic discovery and comparison of relevant sources of information. The framework is applicable to a number of domains where policy reasoning requires the automatic discovery and access of external sources of information. This is illustrated by drawing on examples from our work on collaborative enterprise scenarios as well as on mobile and pervasive computing applications we have deployed at Carnegie Mellon University. Independently of the particular domain, our framework relies on *Policy Enforcing Agents* (PEA) to enforce one or more types of policies The range of the policies can be broad, and may include access control policies, privacy policies or regulations such export control policies, HIPPA policies or US Safe Harbor policies. We show that, in the case of access control policies, our framework can be viewed as an extension of XACML by providing a practical and particular

powerful implementation of what the XACML architecture refers to as Policy Information Points (PIP) and Context Handler functionality [33]. We proceed to show that our framework extends to a broader class of corporate and regulatory policies and provide empirical evidence that our architecture scales favorably to problems with up to hundreds of sources of information and tens of service directories.

The remainder of this paper is organized as follows. Section 2 provides a brief overview of relevant work in decentralized trust management and semantic web technologies. Section 3 introduces a particular instantiation of a Policy Enforcing Agent developed to enforce access control policies and obfuscation policies (namely policies that manipulate the level of accuracy or inaccuracy at which information or services can be accessed). We refer to this particular class of PEAs as *Information Disclosure Agents (IDA)*. This section further details the different modules of an IDA and how their operations are opportunistically orchestrated by meta-control strategies in response to incoming requests. Section 4 details how IDAs can be viewed as extensions of the XACML architecture. Section 5 discusses the application of PEAs to a broader range of context-sensitive policies. Additional implementation details along with empirical results are presented in Section 6. Section 7 contains some concluding remarks.

2 Related Work

The work presented in this paper builds on concepts of decentralized trust management developed over the past decade (see [4] as well as more recent research such as [2,3,13]) . 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, 13, 26, 29, 30] to name just a few) and policy aware web information sharing scenarios [17, 18] . Our earlier work in this area involved the development of semantic web reasoning engines (or "Semantic e-Wallets") that enforce context-sensitive privacy and security policies in response to requests from context-aware applications implemented as intelligent agents [8, 9]. Semantic e-Wallets played a dual role of gatekeeper and clearing-house for sources of information about a given entity (e.g. user, device, service or organization). In this paper, we introduce a more decentralized framework, where policies can be distributed across any number of agents and web services. The main contribution of the work discussed here is in the development and initial evaluation of a semantic web framework and a meta-control model for opportunistically interleaving policy reasoning and web service discovery to enforce context-sensitive policies (e.g. privacy and security policies). This contrasts with the more scripted approaches to interleaving these two processes adopted in our earlier work on Semantic e-Wallets.

Our research builds on recent work on semantic web service languages, (e.g. OWL-S [35], WSMO [42], SAWSDL [38]) and semantic web service discovery functionality. Early work in this area by Paolucci et al. [22] focused on matching semantic descriptions of services being sought with semantic profiles of services being offered that include descriptions of input, output, preconditions and effects (see also our own work

in this area [24]). More recently discovery functionality has also been proposed that takes into account security annotations [16].

Other relevant work includes languages for capturing user privacy preferences such as P3P's APPEL language [36], and for capturing access control privileges such as the Security Assertion Markup Language (SAML) [34], the XML Access Control Markup Language (XACML) [33] and the Enterprise Privacy Authorization Language (EPAL) [32]. These languages do not take advantage of semantic web concepts and do not attempt to solve the problem of identifying and gathering information required to enforce policies. Rein [14, 15] is another semantic web framework for modeling and reasoning over policies that has been developed concurrently with ours. While the objectives of Rein are generally similar to ours, the work presented in this paper focuses specifically on the process of orchestrating policy reasoning with the identification and access of relevant sources of information required to verify policies. KAoS is another semantic web framework that has looked at integrating semantic web service concepts with policy reasoning [30]. Our Semantic e-Wallets as well as research described herein has relied on an extension of OWL Lite known as ROWL to represent policies that refer to concepts defined with respect to ontologies [8, 9, 10]. While ROWL has been a convenient extension of OWL 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. One better known language in this area is RuleML [37], 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 semantic web framework and a meta-control model for enforcing context-sensitive policies. 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.

Another relevant line of research involves work on trust negotiation, namely the development of interactive protocols to incrementally establish trust through the dynamic exchange of credentials. In particular, PeerTrust [20] uses logic programming to represent and reason about access control policies. This includes the delegation of terms in a Horn clause to other peers for evaluation. Another example of work in this area is Trust-Serv [28], a model-driven trust negotiation framework for Web services. While in these systems, the process of identifying sources of information to enforce policies is encoded in the form of rules, the framework presented in this article also allows for the dynamic discovery of relevant sources of information (in addition to being able to model rules such as those described in the trust negotiation literature). This additional flexibility makes it possible to support significantly more open environments, where one is not required to anticipate all possible sources of relevant information ahead of time.

3 Information Disclosure Agents

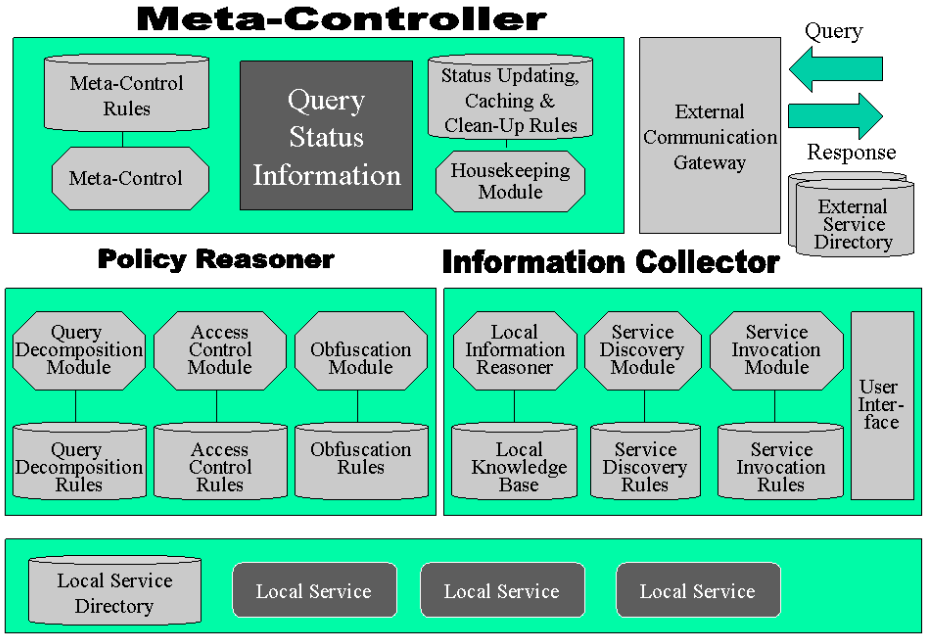

Figure 1. Information Disclosure Agents: Overall Architecture

This section introduces a particular type of Policy Enforcing Agent responsible for controlling access to an information service. We refer to this type of agent as an Information Disclosure Agent and use it to illustrate our architecture for Policy Enforcing Agents. Specifically, consider an environment where sources of information are all modeled as services that can be automatically discovered based on rich ontology-based profiles advertised in service directories. Each service has an owner, whether an individual or an organization, which is responsible for setting policies for it, with policies represented as rules. Policies include both access control policies (e.g. who has the right to access a service and under which particular conditions) and obfuscation policies (i.e. policies that manipulate the level of accuracy or inaccuracy of information being disclosed).

An Information Disclosure Agent (IDA) receives requests for information or service access. In processing these requests, it is responsible for enforcing access control and obfuscation polices specified by its owner and captured in the form of rules. As it processes incoming queries (or, more generally, requests), the agent records status information that helps it monitor its own progress in enforcing its policies and in obtaining the necessary information to satisfy the request. This typically involves submitting multiple requests to a policy reasoner module and an information collector module. The latter 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.

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 IDA modules (e.g. policy reasoning modules, 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 collector are rule-based engines implemented in JESS [7]. 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 reasoner 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.)

Query status information helps the PEA monitor how far along it is in processing individual requests, namely determining whether the request (or query) complies with relevant access control policies, gathering the requested information and applying relevant obfuscation rules, if any, to sanitize information before it is returned to the requester. It is expressed according to a taxonomy of predicates intended to keep track of different activities typically involved in processing a query. 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”). 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?”, 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.

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.).

An IDA’s *Meta Controller* relies on meta-control rules to analyze query status information and determine which module(s) to activate next. Meta-control rules are modeled as if-then clauses, with Left Hand Sides (LHSs) specifying their premises and Right Hand Sides (RHSs) their conclusions. LHS elements refer to query status information, while RHS elements contain facts that result in module activations. Query status information helps keep track of how far along the IDA is in obtaining the information required by each query and in enforcing relevant policies. Query status information in the LHS of meta-control rules is expressed according to a taxonomy of predicates that helps the agent keep track of queries and query elements - e.g., whether a query has been or is being processed, what individual query elements it has given rise to, whether these elements have been cleared by relevant access control policies and sanitized according to relevant obfuscation control policies, etc. All status information is annotated with time stamps. Specifically, query status information includes:

- **A query status ID**
- **Status predicates** to describe the status of a query or query element
- **A query ID** and **query element ID** to which the predicate refers
- **A parent query status ID** to help keep track of dependencies (e.g. a query element may be needed to help check whether another query element is consistent with a context-sensitive access control policy). These dependencies, if passed between IDA agents, can also help detect deadlocks (e.g. two IDA agents each waiting for information from the other to enforce their 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 seven categories: 1) com-

munication; 2) query; 3) query elements; 4) access control; 5) obfuscation; 6) service discovery and 7) service invocation.

	Sample Status Predicates	Description
1)	Query-Received	A particular query has been received.
	Sending-Response	Response to a query is being sent
	Query-make-deadlock	The incoming query may result in an endless loop. According to different meta control rules, the IDA may respond a failure to query sender, or consult the user to handle the problem.
	Response-Sent	Response has been successfully sent
	Response-Failed	Response failed (e.g. message bounced back)
2)	Processing Query	Query is being processed
	Query Decomposed	Query has been decomposed (into primitive query elements)
	All-Elements-Available	All query elements associated with a given query are available (i.e. all the required information is available)
	All-Elements-Cleared	All query elements have been cleared by relevant access control policies
	Clearance-Failed	Failed to clear one or more access control policies
	All-Elements-Sanitized	All query elements have been sanitized according to relevant obfuscation policies
	Sanitization-Failed	Failed to pass one or more obfuscation policies
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-Cleared	Query element has been cleared by relevant access control policies
	Clearance-Failed	Failed to pass one or more access control policies
	Element-Sanitized	Query element has been sanitized using relevant obfuscation policies
	Sanitization-Failed	Failed to pass one or more obfuscation policies
	Element-locally-available	The value of a query element can not be obtained from the local domain ontologies
4)	Clearance-Needed	A query or query element needs to be cleared by relevant access control rules
5)	Sanitization-Needed	Query or query element has to be sanitized subject to relevant obfuscation policies
6)	Element-need-service	A query element requires the identification of a relevant 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
7)	Waiting-for-service-response	A query element is waiting for a response to a query sent to a service (e.g. query sent to a location tracking service to help answer a query element corresponding to a user's location)
	Failed-service-invocation	A service failed to be invoked. Again this predicate could be refined to distinguish between different types of failure (e.g. service down, access denied, etc.)
	Service-response-time-out	The service doesn't respond the query for a time longer than the threshold. It will results in a failed-service-response
	service-response-available	A response has been returned by the service. This will typically result in the creation of an "Element-Available" status update.

Table 1. Sample list of status predicates.

Meta status is updated by asserting new facts into the working context (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 possibly additional actions. Below is an example of a meta-control rule that invokes a service after it is identified. Depending on the invocation result, the current meta-status gets updated to "waiting-for-service-response" or "failed-service-invocation"

```
(defrule invoke-service-if-identified
  ?x <- (metastatus
    (statusID ?sid)
    (predicate "service-identified")
    (parentID ?pid)
    (timeStamp ?)
    (queryID ?qid)
    (elementID ?eid)
  )
  =>
  (bind ?result (invoke-service ?eid))
  (bind ?time ((new java.util.Date) getTime))
  (if (= ?result "success")
    (assert (metastatus
      (statusID ?*statusID*)
      (predicate "waiting-for-service-response")
      (parentID ?sid)
      (timeStamp ?time)
      (queryID ?qid)
      (elementID ?eid)
    ))
    (retract ?x)
   Else
    (assert (metastatus
      (statusID ?*statusID*)
      (predicate "failed-service-invocation")
      (parentID ?sid)
      (timeStamp ?time)
      (queryID ?qid)
      (elementID ?eid)
    ))
    (retract ?x)
  )
  (bind ?*statusID* (+ ?*statusID* 1))
)
```

Meta-control rules can also be defined to consult with users, whether to ask for a particular piece of information (i.e. using a user as an external source of information) or to decide what to do next (e.g. to decide whether or not to abandon a particular course of action in situations that are taking longer than expected).

In general, different collections of query status predicates and meta-control rules will result in different behaviors. Accordingly, our meta-control architecture enables one to tailor Policy Enforcing Agents to the particular policies and scenarios associated with a given domain, with simpler domains giving rise to simpler sets of behaviors and

more complex ones allowing for more sophisticated logic to handle a wider range of situations. Figure 2 depicts the overall set of behaviors associated with a relatively simple set of status predicates and meta-control rules. In this particular case, upon receiving a request, the IDA generates an information status update indicating that a new query has been received. This information is expressed as a tuple of the form *(statusID predicate queryID elementID parentID timestamp)* such as *(status1 query-received query1 nil nil time1)*. Because receiving a query is the first step, there is no parent ID and no query element so their values are nil. Next, the meta-controller generates a new status update indicating that the request has to be run against relevant access control rules, - e.g. *(status2 clearance-needed query1 element1 status1 time2)*. This status update in turn results in the meta-controller invoking the policy reasoner, which in turn can lead to the creation of one or more query elements. Given this particular set of meta-control rules, the IDA first tries to find the information required for each query element in its local knowledge. If this does not work, the IDA creates an *element-not-known-locally* status predicate, which in turn leads to the creation of an *element-need-service* status predicate. This status predicate later triggers a service identification step. This will typically followed by the actual identification of a service, its invocation and eventually the IDA obtains the information required to determine whether its policy is satisfied, etc.

While this section focused on Information Disclosure Agents, it is easy to see that context-sensitive policies other than access control policies and obfuscation policies can easily be enforced with agents built around a similar architecture with status predicates and meta-control modules used to orchestrate policy reasoning and the collection of relevant information.. In the remainder of this paper, we generically refer to these agents as Policy Enforcing Agents (PEAs).

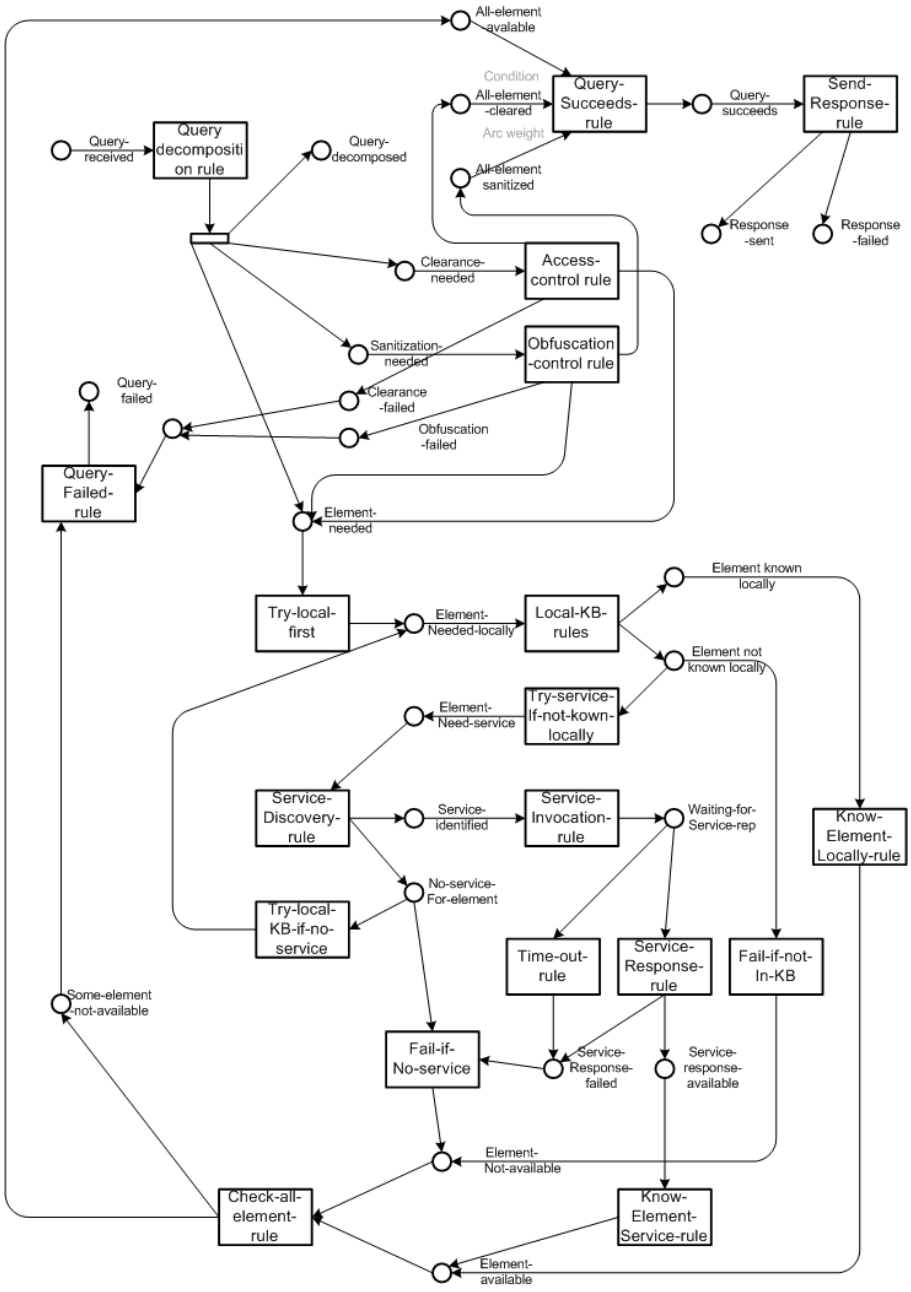

Figure 2. An example set of meta-control rules

3.2 Policy Reasoner

The PEA's policy reasoning engine is responsible for evaluating relevant policies and returning policy decisions. 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 collector 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.

In general, the policy reasoner includes the following modules:

1. *Query Decomposition Module* takes as input a particular query and breaks it down into elementary needs for information, which can each be thought of as subgoals or sub-queries. We refer to these as *Query Elements*. The value of a *Query Element* can be obtained just based on facts contained in the agent's local knowledge base, or by invoking both local and remote services.
2. *Access Control Module* is responsible for determining whether a particular query or sub-query is consistent with relevant access control policies – modeled as access control rules. While some policies can be checked just based on facts contained in the agent's local knowledge base, many policies require obtaining information from a combination of both local and external sources. When this is the case, rather than immediately deciding whether or not to grant access to a query, the *Access Control Module* needs to request additional facts – also modeled as *Query Elements*.
3. *Obfuscation Module* sanitizes information requested in a query according to relevant obfuscation policies – also modeled as rules. As it evaluated relevant obfuscation policies, this module too can post requests for additional *Query Elements*.

It should be emphasized that our architecture is not tied to a particular policy reasoner. Instead, different policy reasoning engines can be plugged in to support reasoning about different types of policies. This is illustrated in this paper by presenting examples and results obtained with two different families of policy reasoners:

1. A family of JESS-based policy reasoners capable of enforcing a broad range of policies. Policies are expressed as ROWL rules [10] that refer to concepts specified in domain-specific ontologies written in W3C's OWL language [41]. ROWL has been used to specify a number of policies, from access control policies, to obfuscation policies, to message processing policies, etc. Instantiations of this engine have been deployed in the context of several mobile and pervasive computing applications piloted on Carnegie Mellon's campus (e.g.

MyCampus [27] and PeopleFinder [23] application) as well as in the context of enterprise collaboration scenarios.

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. This is further detailed in Section 4.

3.3 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 for evaluating 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.

In our current implementation, knowledge in the *Location Information Reasoner* is represented using RDF/OWL and ROWL (for domain rules).Specifically, we use an OWL meta-model, equivalent to OWL-Lite to interpret and reason about OWL statements. Ontologies and annotations are translated into Jess facts, while inference rules are translated into Jess rules through a combination of both forward and backward chaining, with backward chaining used to express the "need" for facts and help identify sources of information (or services) that are likely to provide these facts. For instance, a rule that specifies that two people are colleagues if they have the same employer may trigger one or more backward chaining rules to determine each person's employer, e.g. first looking for local information and, if this fails, possibly looking for external services that can provide this information..

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 that map 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 a significantly greater level 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.

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).

3.4. Service Discovery and Invocation

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. We have also implemented variations of our architecture using the OWL-S language [25] and could readily adopt other equivalent frameworks (e.g. WSMO [42] or SAWSDL[38]). A WOWL service description includes:

1. The service's output.
2. Its preconditions
3. Relevant non-functional attributes [21], 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.

Given that PEAs can look for and query external sources of information, whose access may in turn be control by other PEAs, it is entirely possible to run into deadlocks, e.g. two PEAs, each waiting for a response from the other before they can proceed with a given query. A simple solution to this problem can involve using timeouts, which can themselves be implemented in the form of meta-control rules. Specifically, in our current implementation, time-outs can be specified in the form of independent threads, which periodically check the timestamp of the pending meta statuses. The rule

may simply specify the query is considered to have failed or it may ask the user whether to allocate more time for processing. In addition, all status information is annotated with time stamps which can be used to detect timeout situations. Circular analysis has also been implemented to help detect deadlocks independently of the length of time taken by a given query.

This is done using query dependency graphs, in which each query is represented as a node and query dependency is represented as a directed edge. If two queries that depend on each other have the same sender, the same receiver and ask the same information, they are in a deadlock situation. The agent that receives a query is responsible for detecting the deadlock and can either respond with a *query-failed* response or ask the sender to notify the user that a deadlock has been detected.

4. Access Control Agents based on XACML

One particular instance of an IDA is an *Access Control Agent* (ACA) which only implements access control policies. Given the amount of effort invested by industry over the past few years to define a standard for such agents, it makes sense to look at how our PEA architecture relates to the architecture developed as part of the XACML standard [33]. As it turns out, our PEA architecture can be viewed as an extension of this standard and it is possible to build instantiations of our PEAs that rely on XACML Policy Decision Points and on the XACML language to express and enforce access control policies.. This is illustrated in Figure 3, which shows the architecture of an ACA agent we have implemented using Sun's XACML *Policy Decision Point* (PDP) engine [40]. Incoming decision requests (or "queries") are directed to the agent's Meta-Controller which doubles as an 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).

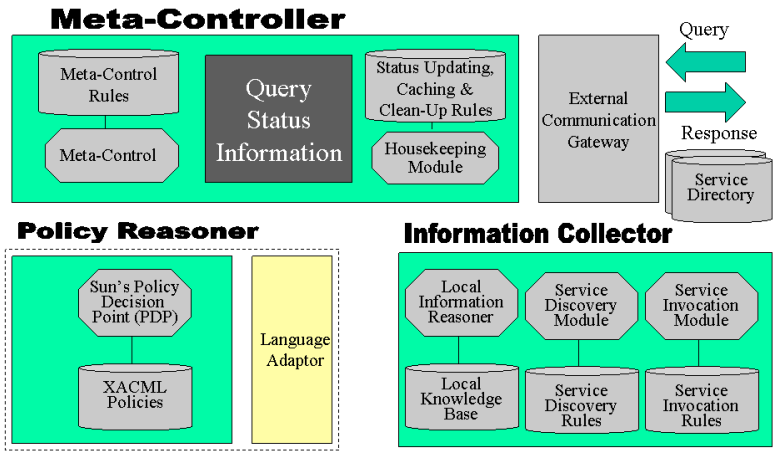

Figure 3. PEA Instantiated as an Access Control Agent using Sun's XACML Policy Decision Point engine**.**

4.1 An Aerospace Contractor Scenario

The ACA agent depicted in Figure 1 has been implement tosupport the 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 geo-stationary 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 4 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 4. 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 (i) 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.

Upon receiving the request, United GenSat's ACA generates an information status update indicating that a new query has been received. This facts of the new query is expressed as triples of the form *(predicate subject object)* – e.g. *(sender query1 Bob) or (ask query1 element1)*. Next, the meta-controller generates a new status update indicating that the request has to be cleared based on applicable access control policies, The required *element* to be cleared is also presented as a triple, like *(schedule SAT777 ?s)*. Here *?s* represents a variable whose value is unknown. 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-known-locally* 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.

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:has Access 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>SATServiceAgent</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 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.

4.2 Updating Query Status Information

The following illustrates the processing of a query by an IDA, using the scenario introduced above. Bob's query about the production schedule of SAT777 is first processed by the IDA's *Communication Gateway*, resulting in a query information status update indicating that a new query has been received. This information is expressed as a collection of *(predicate subject object)* triples of the form:

```
(triple "Status#predicate" "status1" "query-received")
(triple "Query#queryId" "status1" "query1")
(triple "Query#parentId" "status1" nil)
(triple "Query#timestamp" "querystatus1" time1)
(triple "Query#sender" "query1" "Bob")
(triple "Query#element" "query1" "element1")
(triple "Ontology#schedule" "SAT777" "element1")
```

Next, the meta-controller activates the *Query Decomposition Module,* resulting in the creation of two query elements: one query element to establish whether this request is compatible with United GenSat's access control policies and the other to obtain the production schedule of SAT777:

```
(triple "Status#predicate" "status2" "clearance-needed")
(triple "Status#predicate" "status3" "element-needed")
```

Let us assume that the meta-controller decides to first focus on the "clearance-needed" query element and invokes the *Access Control Module* which is actually an XACML PDP. This module determines that two conditions need to be checked and accordingly creates two new query elements ("check-conditions"). One condition requires checking whether Bob is an authorized SATElectronics employee:

```
(triple "Status#predicate" "status4" "element-needed")
(triple "Query#queryId" "status4" "element2")
(triple "Query#parentId" "status4" "query1")
(triple "Query#condition" "element2" "Ontology#hasAuthorizedEmp ")
(triple "Ontology#hasAuthorizedEmp" "SATElectronics" "Bob")
```

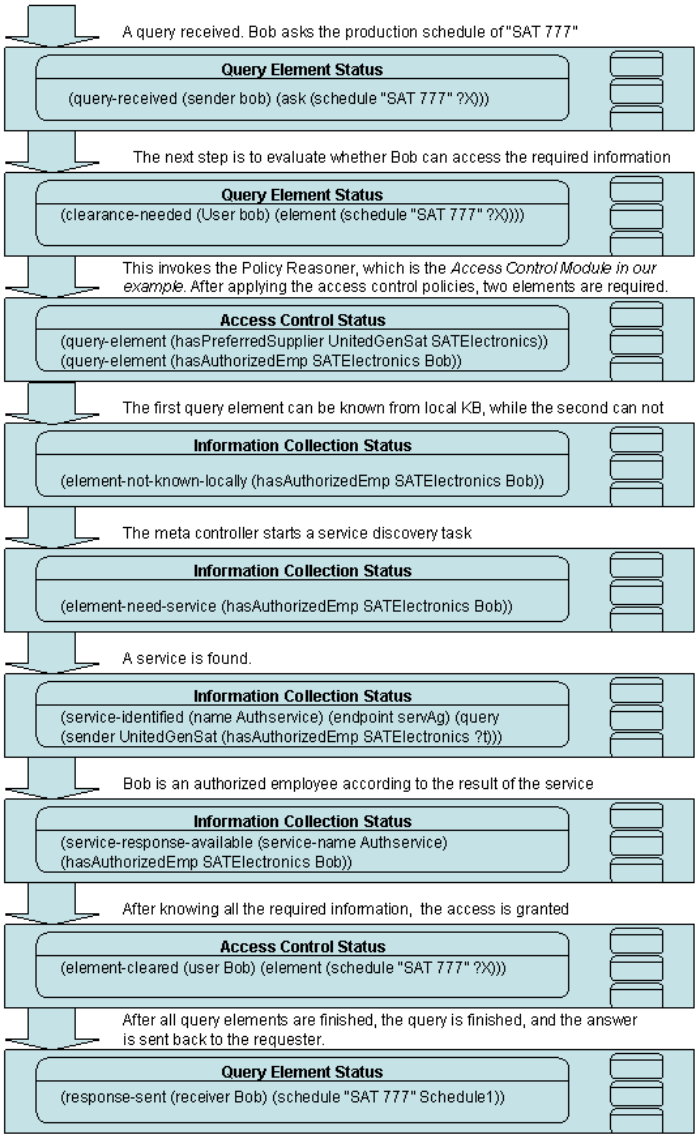

Figure 6. Query status updates for a fragment of the scenario.

This condition in turn requires a series of information collection steps that are orchestrated by the meta-control rules in United GenSat's IDA. In this example, we assume that the IDA's local knowledge base does not know Bob's employment information. According the following query status information update is eventually generated:

```
(triple "Status#predicate" "status5" "element-not-locally-available")
```

```
(triple "Query#queryId" "status5" "element2")
```

United GenSat's IDA has a meta-control rule to initiate service discovery when a query element can not be found locally. The rule, expressed in CLIPS [31], is of the form:

```
(triple "Status#predicate" ?s1 "element-not-locally-available")
(triple "Status#predicate" ?s2 "element-needed ")
(triple "Query#queryId" ?s1 ?e1)
(triple "Query#queryId" ?s2 ?e1)
=>
(assert (triple "predicate" ?newstatus "element-need-service"))
(assert (triple "Query#queryId" ?newstatus ?e1)
```

Using this rule, the meta-controller now activates the *Service Discovery Module*. A service to find Bob's employment information is identified. This results in a query status update of the type "service-identified".

```
(triple "Status#predicate" ?s1 "element-need-service")
(triple "Status#predicate" ?s2 "service-identified")
(triple "Query#queryId" ?s1 ?e1)
(triple "Query#queryId" ?s2 ?service)
(triple "Query#parentId" ?s2 ?e1)
=>
(assert (triple "Status#predicate" ?newstatus "waiting-for-service-
response"))
(assert (triple "Status#queryId" ?newstatus ?service))
```

Note that, if there are multiple matching services, the service discovery module needs rules to help select among them.

Let us assume that the service discovery module identifies that Bob's employer, SATElectronics is a trustful information source to answer if Bob is an authorized employee. The Housekeeping module updates the necessary Query Status Information, indicating among other things that information about Bob's information has been found ("element-available") and cleaning old status information. This is done using a rule of the type:

```
?x <- (triple "Status#predicate" ?s1 "waiting-for-service-response")
?y <- (triple "Query#queryId" ?s1 ?service)
(triple "Status#predicate" ?s2 "service-response-available")
(triple "Query#queryId" ?s2 ?result)
=>
(retract ?x)
(retract ?y)
(assert (triple "Status#predicate" ?newstatus "element-available"))
(assert (triple "Query#queryId" ?newstatus ?result))
```

The scenario continues through several similar steps. A full flow diagram is shown in Figure 6. For better readability, we do not restrict the syntax of the meta statuses shown in this figure.

5. Beyond Access Control Policies

PEAs are not limited to information disclosure and 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)

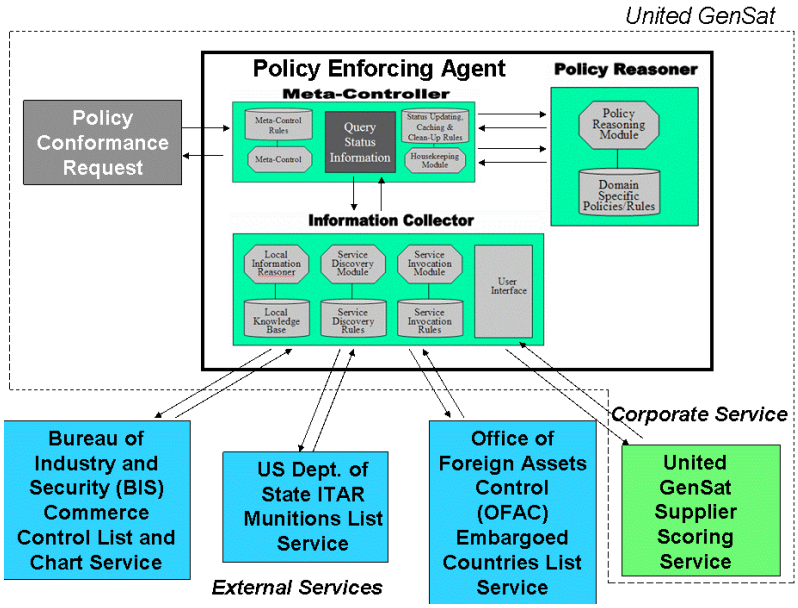

Figure 7. Using a PEA to check for compliance with supplier selection policies, including supplier scoring requirements and government export controls.

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 [10] and require accessing a combination of corporate and external services to obtain up-to-date supply ratings and export restrictions. An example of such a ROWL rule is shown in Figure . 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 appears 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>
      <scm:Result rdf:resource="&var;#result">
        <rowl:equal-to rdf:resource="&scm:#has_licence"/>
      </scm:Result>
    </rowl:or>
  </rowl:body>
</rowl:Rule>
```

Figure 8. 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 8, 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 Current Implementation: Evaluation and Discussion

Our policy enforcing agents are currently implemented in JESS, a high-performance rule-based engine in Java [7]. Domain knowledge, including service profiles, ontologies, annotations and semantic inference rules are expressed in OWL [41]. We have implemented multiple instances of PEAs, including PEAs relying on different policy languages and reasoning capabilities. This includes multiple instances of ROWL policy reasoners and Sun's more specialized PDP reasoner to enforce XACML policies. As already indicated earlier ROWL could easily be replaced with languages such as RuleML, SWRL or some similar language. XSLT transformations are used to translate OWL facts and extensions of OWL (e.g. to model rules and queries) into CLIPS. Agent modules are organized as JESS modules.

The performance of Jess is decided by the number of facts instead of the number of rules. We have tested our system under different number of policies (rules), and the CPU time required for answering a query has no significant difference. Therefore, our scalability evaluation focuses on the size of ontology and service repository.

We have evaluated our solution on an IBM server with 2 Intel Xeon 3.0GHz CPU and 3GB of RAM. The server was running Windows XP Professional OS, Java SDK 1.4.2 and Jess 6.1. Below we report empirical results obtained to evaluate the scalability of our PEA implementation.

Specifically, the first table below reports results obtained using ontologies from the Lehigh University Benchmark (LUBM) [19]. The results are based on an OWL university ontology with around 10000 triples after translation. Results are reported for repositories of 100, 200 and 500 randomly generated semantic web services in a single repository. The input and output parameter types were randomly selected from the classes in the domain ontology. CPU times are in milliseconds. We report the total time required to process a query as well as the amount of processing required by individual modules, namely the meta-controller, access control reasoning module, local knowledge reasoner and service discovery module.

Number of Services	*100*	*200*	*500*
Meta-controller	397	500	1128
Access-controller	48	44	53
Local Knowledge	38	56	66
Service discovery	31	45	62
Total	514	645	1309

The result shows that the query time increases (nearly linearly) when the service repository is growing. A second set of experiments involved distributing the service repository by limiting the number of services in each repository, but increasing the number of service repositores (or directories). Each directory had its own PEA agent with scenarios typically requiring multiple directory queries before an adequate service could be identified. In these experiments, each repository agent had a lighter ontology with about 1000 triples and 100 registered services. Again, the table below reports over processing time in milliseconds as well as the time required by individual modules.

Number of Service Repositories	*5*	*10*	*15*	*20*
Meta-controller	56	124	156	285
Access-controller	15	15	44	70
Local Knowledge	14	33	51	114
Service discovery	20	60	89	151
Total	105	232	340	620

A Policy Enforcing Agent is also used to enforce people's location privacy policies in a PeopleFinder application we have deployed on Carnegie Mellon's campus [23]. A typical PeopleFinder PEA contains 500 triples. In experiments involving 200 users, each with their own PEAs to enforce their privacy policies, a typical request for a user's location can be processed against the user's policies within 100 msec.

The performance result shows that our solution can be viewed as practical and scalable in the experimental settings. It should be noted that our solution is not JESS-specific. At the same time, a significant number of experiments still need to be conducted to gain a more comprehensive understanding of the scalability of our approach. Other complex issues such as reasoning about provenance (i.e. possible conflicts of interest of information sources used to build a proof) and inconsistent policies also require significant additional work. Differentiating between situations where a policy has been shown not to be satisfied and situations where the agent has not yet been able to determine whether a policy is satisfied will likely call for differentiating between

classical negation and "negation as failure". One possible solution here would be to use a framework such as SweetRules as an add-on to our semantic web reasoner [39]

7 Concluding Remarks

In this paper, we presented a semantic web framework for dynamically interleaving policy reasoning and external service discovery and access. Within this framework, external sources of information are wrapped as web services with rich semantic profiles allowing for the dynamic discovery and comparison of relevant sources of information. Each entity (e.g. user, sensor, application, or organization) relies on one or more *Policy Enforcing Agents* responsible for enforcing relevant policies in response to incoming requests. These agents implement meta-control strategies to dynamically interleave semantic web reasoning, service discovery and access. These meta-control strategies can also be extended to treat the user as another source of information, e.g. to confirm whether a given fact holds or to provide meta-control guidance such as deciding when to abandon trying to determine whether a policy is satisfied.

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. PEAs to enforce different types of policies or to operate on similar policies in different domains will rely on slightly different sets of modules and different meta-control strategies, yet they can all be implemented using the same meta-control architecture and many of the same principles presented in this paper. They generally rely on a taxonomy of query information status predicates to monitor their own progress in processing incoming queries and enforcing relevant security and privacy policies. They use meta-control rules to decide which action to take next (e.g. decomposing queries, seeking local or external information, etc.).

We have implemented several instances of our PEAs in the context of collaborative enterprise scenarios as well as in the context of several mobile and pervasive computing applications piloted on Carnegie Mellon's campus. Empirical results presented in this paper indicate that our existing implementation scales favorably on scenarios involving up to hundreds of sources of information and tens of service directories. Future work will focus on further exploring scalability issues, evaluating tradeoffs between the expressiveness of meta-control rules and efficiency. Other issues of particular interest include studying opportunities for concurrency (e.g. simultaneously accessing multiple web services), dealing with real-time meta-control issues (e.g. deciding when to give up or when to look for additional sources of information/web services), and better integrating the user as a source of information.

Acknowledgements

The work reported herein has been supported in part under DARPA contract F30602-02-2-0035 (“DAML initiative”) and in part under ARO research grant D20D19-02-1-0389 ("Perpetually Available and Secure Information Systems") to Carnegie Mellon University’s CyLab.

References

[1] R. Ashri, T. Payne, D. Marvin, M. Surridge and S. Taylor, Towards a Semantic Web Security Infrastructure. In Proceedings of Semantic Web Services Symposium, 2004.
[2] L.Bauer, S. Garriss, J. McCune, M.K. Reiter, J. Rouse, and P Rutenbar, “Device-Enabled Authorization in the Grey System”, Submitted to USENIX Security 2005.
[3] L. Bauer, M.A. Schneider and E.W. Felten. "A General and Flexible Access Control System for the Web", In Proceedings of the 11th USENIX Security Symposium, August 2002.
[4] M. Blaze, J. Feigenbaum, an J. Lacy. “Decentralized Trust Management”. In Proceedings of IEEE Conference on Security and Privacy. Oakland, CA. May 1996.
[5] G. Denker, L. Kagal, T. Finin, M. Paolucci and K. Sycara, Security For DAML Web Services: Annotation and Matchmaking, In Proceedings of the Second Intl 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, In Proceedings of the AAAI Spring Symposium on AI Technologies for Homeland Security, 2005.
[7] E. Friedman-Hill. Jess in Action: Java Rule-based Systems, Manning Publications Com-pany, June 2003, ISBN 1930110898, http://herzberg.ca.sandia.gov/jess/
[8] F. Gandon, and N. Sadeh. A semantic e-wallet to reconcile privacy and context awareness. In Proceedings of the Second International Semantic Web Conference (ISWC03), 2003.
[9] F. Gandon, and N. Sadeh. Semantic web technologies to reconcile privacy and context awareness. Web Semantics Journal, 1(3), 2004.
[10] 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
[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. In 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] Lalana Kagal, Tim Berners-Lee, Dan Connolly, and Daniel Weitzner, "Using Semantic Web Technologies for Open Policy Management on the Web", 21st National Conference on Artificial Intelligence (AAAI 2006).
[15] 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
[16] L. Kagal, M. Paolucci, N. Srinivasan, G. Denker, T. Finin and K. Sycara, Authorization and Privacy for Semantic Web Services, In Proceedings of Semantic Web Services Symposium, AAAI 2004 Spring Symposium Series, Stanford University, California, March 2004.
[17] Vladimir Kolovski, Yarden Katz, James Hendler, Daniel Weitzner, and Tim Berners-Lee Towards a Policy-Aware Web, In Proceedings of Semantic Web and Policy Workshop, 2005

[18] V. Kolovski, B. Parsia, Y. Katz, and J. Hendler. Representing web service policies in OWL-DL. In Proceedings of the International Semantic Web Conference (ISWC), 2005.
[19] The Lehigh Univeristy Benchmark, http://swat.cse.lehigh.edu/projects/lubm/
[20] Wolfgang Nejdl, Daniel Olmedilla, Marianne Winslett, PeerTrust: Automated Trust Negotiation for Peers on the Semantic Web, In Proceedings of the Workshop on Secure Data Management in a Connected World (SDM'04) in conjunction with 30th International Conference on Very Large Data Bases, Aug.-Sep. 2004, Toronto, Canada
[21] J. O'Sullivan, D. Edmond, and A.T. Hofstede. What's in a service? Towards accurate description of non-functional service properties. Distributedand Parallel Databases, 12:117.133, 2002.
[22] M. Paolucci, T. Kawamura, T.R. Payne, and K. Sycara, Semantic Matching of Web Services Capabilities, In Proceedings of the First Intl Semantic Web Conference, 2002.
[23] Madhu Prabaker, Jinghai Rao, Ian Fette, Patrick Kelley, Lorrie Cranor, Jason Hong and Norman Sadeh, Understanding and Capturing People's Privacy Policies in a People Finder, Accepted by 5th International Workshop on Privacy in UbiComp, September 16, 2007 - Innsbruck, Austria
[24] J. Rao. Semantic Web Service Composition via Logic-based Program Synthesis. PhD Thesis. Norwegian University of Science and Technology. December 10, 2004.
[25] 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.
[26] N. M. Sadeh, T.C. Chan, L. Van, O. Kwon, and K. Takizawa. Creating an open agent environment for context-aware m-commerce. In Agentcities: Challenges in Open Agent Environments, 2003.
[27] N.M. Sadeh, F. Gandon, and Oh Byung Kwon. Ambient Intelligence: The MyCampus Experience. Carnegie Mellon University Technical Report. CMU-ISRI-05-123. June 2005.
[28] Halvard Skogsrud, Boualem Benatallah, Fabio Casati, Trust-Serv: Model-Driven Lifecycle Management of Trust Negotiation Policies for Web Services, WWW 2004, New York, USA
[29] 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
[30] A. Uszok, J. M. Bradshaw, R. Jeffers, M. Johnson, A. Tate, J. Dalton and S. Aitken, Policy and Contract Management for Semantic Web Services. In Proceedings of Semantic Web Services Symposium, AAAI 2004 Spring Symposium Series, Stanford California.
[31] CLIPS. http://www.ghg.net/clips/CLIPS.html.
[32] IBM, EPAL 1.1. http://www.zurich.ibm.com/security/enterprise-privacy/epal/.
[33] OASIS, eXtensible Access Control Markup Language (XACML)
[34] OASIS, Security Assertion Markup Language (SAML)
[35] OWL-S: Semantic Markup for Web Services. http://www.w3.org/Submission/OWL-S
[36]A P3P Preference Exchange Language(APPEL1.0) http://www.w3.org/TR/P3P-preferences/
[37] The Rule Markup Initiative. (http://www.ruleml.org)
[38] Semantic Annotation for Web Services Description Language, http://www.w3.org/2002/ws/sawsdl/
[39] SweetRules. http://sweetrules.projects.semwebcentral.org/
[40] Sun's XACML Implementation: http://sunxacml.sourceforge.net/.
[41] W3C: OWL Web Ontology Language Overview, W3C Recommendation, Feb. 2004. D. McGuinness & F. van Harmelen (Eds.) http://www.w3.org/TR/owl-features/
[42] Web Service Modeling Ontology, WSMO. http://www.wsmo.org/