Interleaving Semantic Web Reasoning and Service Discovery to Enforce Context-Sensitive Security and Privacy Policies

Jinghai Rao and Norman Sadeh

July 2005
CMU-ISRI-05-113

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

Abstract. Enforcing rich policies in open environments will increasingly require the ability to dynamically identify external sources of information necessary to enforce different policies (e.g. finding an appropriate source of location information to enforce a location-sensitive access control policy). In this paper, we introduce a semantic web framework and a metra-control model 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 privacy and security policies in response to incoming requests. These agents implement meta-control strategies to dynamically interleave semantic web reasoning and service discovery and access. The paper also presents preliminary empirical results. This research has been conducted in the context of *my*Campus, a pervasive computing environment aimed at enhancing everyday campus life at Carnegie Mellon University. The framework presented can be extended to a range of other applications requiring the enforcement of context-sensitive policies (e.g. virtual enterprises, coalition forces, homeland security, etc.).

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 security and privacy 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 some conditions. This trend requires context-sensitive security and privacy policies, namely policies whose conditions are not tied to static considerations but rather conditions whose satisfac-

tion, 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 enforce 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 privacy and security 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. While the framework is applicable to a number of domains where policy reasoning requires the automatic discovery and access of external sources of information (e.g. virtual/collaborative enterprise scenarios, coalition force scenarios, inter-agency homeland security collaboration scenarios), we look more particularly at the issue of enforcing privacy and security policies in pervasive computing environments. In this context, the owner of information sources (e.g. user, sensor, application, or organization) relies on one or more *Policy Enforcing Agents* (PEA) responsible for enforcing relevant privacy and security policies in response to incoming requests. These agents implement meta-control strategies to opportunistically interleave policy enforcement, semantic web reasoning and service discovery and access. The example used in this paper introduces one particular type of PEA we refer to as Information Disclosure Agents (IDA). These agents are responsible for enforcing two types of policies: access control policies and obfuscation policies. The latter are policies that manipulate the accuracy or inaccuracy with which information is released (e.g. disclosing whether someone is busy or not rather than disclosing what they are actually doing). The research reported herehas been conducted in the context of *MyCampus*, a pervasive computing environment aimed at enhancing everyday campus life at Carnegie Mellon University [7, 8, 19, 20].

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 an *Information Disclosure Agent* architecture for enforcing privacy and security policies. It details its different modules and how their

operations are opportunistically orchestrated by meta-control strategies in response to incoming requests. A motivating example is presented in Section 4. Section 5 details our meta-control model based on query status information. Operation of the architecture is illustrated in Section 6. Section 7 discusses our service discovery model. Section 8 presents our current implementation and discusses initial empirical results. Concluding remarks are provided in Section 9.

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 [2,11,14]) . 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, 4, 9, 12, 13, 23, 24] to name just a few). Our own work in this area has 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 [7, 8]. Semantic e-Wallets play a dual role of gatekeeper and clearinghouse 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 among 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 in enforcing 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 [7,8].

Our research builds on recent work on semantic web service languages, (e.g. OWL-S [26] and WSMO [27]) and semantic web service discovery functionality. Early work in this area by Paolucci et al. [29] 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 [31]). More recently discovery functionality has also been proposed that takes into account security annotations [30].

Other relevant work includes languages for capturing user privacy preferences such as P3P's APPEL language [25], and for capturing access control privileges such as the Security Assertion Markup Language (SAML) [17], the XML Access Control Markup Language (XACML) [16] and the Enterprise Privacy Authorization Language (EPAL) [5]. These languages do not take advantage of semantic web concepts. On the other hand [12] 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 own work on Semantic e-Wallets as

well as research described in this paper 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 [7, 8]. 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 [18], a proposed standard for a rule language, based on declarative logic programs. Another is SWRL [10], 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. In other words, while our policies are currently expressed in ROWL, they could just as easily be specified in a language like SWRL.

3 Overall Approach and Architecture

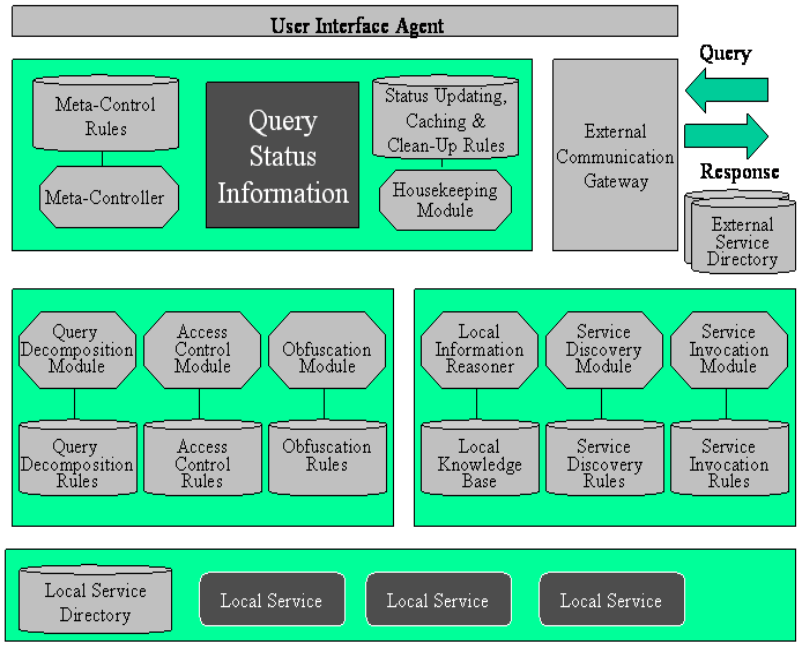

Fig. 1. Information Disclosure Agent: Overall Architecture

We consider an environment where sources of information are all modeled as services that can be automatically discovered based on rich ontology-based service profiles advertised in service directories. Each service has an owner, whether an individual or an organization, who is responsible for setting policies for it, with policies represented as rules. In this paper we focus on access control policies and obfuscation policies enforced by *Information Disclosure Agents*, though the framework we present could readily be used to enforce a variety of other policies.

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. Based on this updated *query status information*, a meta-control module ("meta-controller") dynamically orchestrates the operations of modules it has at its disposal to process queries (Fig. 1). As these modules report on the status of activities they have been tasked to perform, this information is processed by a housekeeping module responsible for updating query status information (e.g. changing the status of a query from being processed to having been processed). Simply put, the agent continuously cycles through the following three basic steps:

1. The meta-controller analyzes its latest query status information and invokes one or more modules to perform particular tasks. As it invokes these modules the meta-controller also updates relevant query status information (e.g. updates the status of a query from "not yet processed" to "being processed").
2. Modules complete their tasks (whether successfully or not) and report back to the housekeeping module – occasionally modules may also report on their ongoing progress in handling a task
3. The housekeeping module updates detailed status information based on information received from other modules and performs additional housekeeping activities (e.g. caching the results of recent requests to mitigate the effects of possible denial of service attacks, cleaning up status information that has become irrelevant, etc.)

For obvious efficiency reasons, while an IDA consists of a number of logical modules, each operating according to a particular set of rules, it is typically implemented as a single reasoning engine. In our current work we use JESS [6], a high-performance Java-based rule engine that supports both forward and backward chaining – the latter by reifying "needs for facts" as facts themselves, which in turn trigger forward-chaining rules. The following provides a brief description of each of the modules orchestrated by an IDA's meta-controller:

- *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*.
- *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*.

- *Obfuscation Module* sanitizes information requested in a query according to relevant obfuscation policies – also modeled as rules. As it evaluates relevant obfuscation policies, this module too can post requests for additional *Query Elements.*
- *Local Information Reasoner* corresponds to domain knowledge (facts and rules) known locally to the IDA
- *Service Discovery Module* helps the IDA identify potential sources of information to complement its local knowledge. External services can be identified through external service directories (whether public or not), by communicating via the agent's *External Communication Gateway*. Rather than relying solely on searching service directories, the service discovery module also allows for the specification of what we refer to as *service identification rules*. These rules directly map information needs on pre-specified services. An example of such rule might be: "when looking for my current activity, first try my calendar service". When available, such rules can yield significant speedups, while allowing the module to revert to more general service directory searches when they fail. We currently assume that all service directories rely on OWL-S to advertise service profiles (see Section 7).
- *Service Invocation Module* allows the agent to invoke relevant services. It is important to note that, in our architecture, each service can have its own IDA. As requests are sent to services, their IDAs may in turn respond with requests for additional information to enforce their own policies.
- *User Interface Agent:* The meta-controller treats its user as just another module who is modeled both as a potential source of domain knowledge (e.g. to acquire relevant contextual information) as well as a potential source of meta-control knowledge (e.g. if a particular query element proves too difficult to locate, the user may be asked whether to stop looking - she could even be offered the option of making an assumption about the particular value of the query element).

Modules support one or more services that can each be invoked by the meta-controller along with relevant parameter values. For instance, the meta-controller may invoke the query decomposition module and request it to decompose a particular query; it may invoke the access control module and task it to proceed in evaluating access control policies relevant to a particular query; etc. In addition, meta-control strategies do not have to be sequential. For instance, it may be advantageous to implement strategies that enable the IDA to concurrently request the same or different facts from several services.

4 An Example

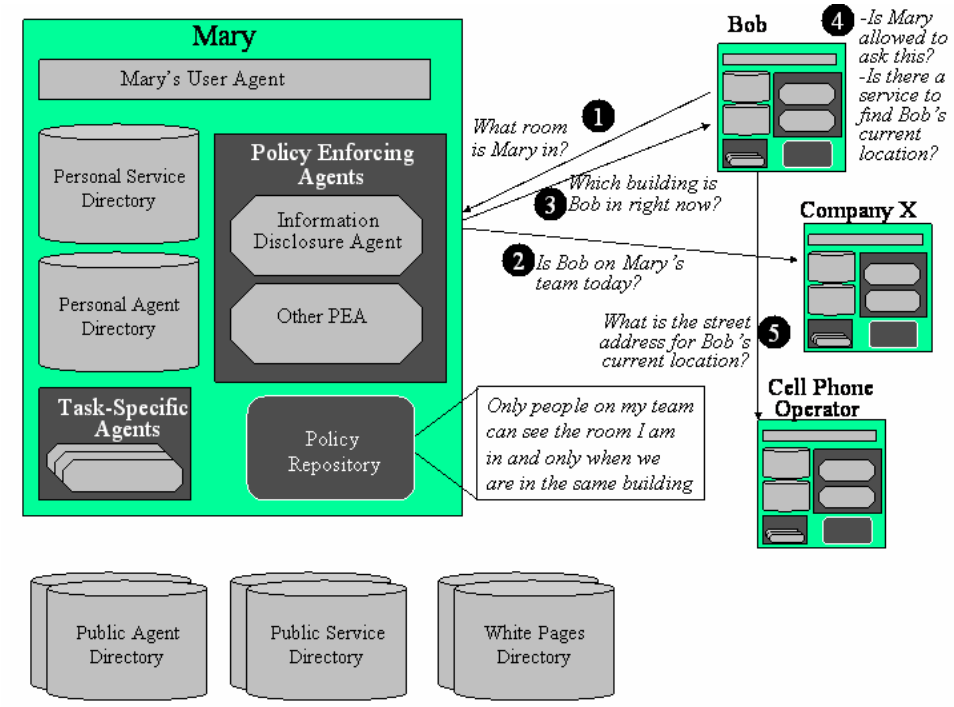

Fig. 2. Illustration of first few steps involved in processing the example

The following scenario will help illustrate how IDAs operate. Consider Mary and Bob, two colleagues who work for company X. They are both field technicians who constantly visit other companies. Mary's team changes from one day to the next depending on the nature of her assignment. Mary relies on an IDA to enforce her access control policies. In particular, she has specified that she is only willing to disclose the room that she is in to members of her team and only when they are in the same building.

Suppose that today Bob and Mary are on the same team. Bob is querying Mary's IDA to find out about her location. For the purpose of this scenario, we assume that Mary and Bob are visiting Company Y and are both in the same building at the time the query is issued. Both Bob and Mary have cell phone operators who can provide their locations at the level of the building they are in – but not at a finer level. Upon entering Company Y, Mary also registered with the company's location tracking service, which can track her at the room level. For the purpose of this scenario, we further assume that Mary's IDA needs to identify a service that can help it determine whether Bob is on her team. A discovery step helps identify a service operated by Company X (Bob and Mary's employer) that contains up-to-date information about

teams of field technicians. This requires a directory with rich semantic service profiles, describing what each service does (e.g. type of information it can provide, level of accuracy or recency, etc.). To be interpretable by agents such as Mary's IDAs, these profiles also need to refer to concepts specified in shared ontologies (e.g. concepts such as projects, teams, days of the week, etc.). Once Mary's IDA has determined that Bob is on her team today, it proceeds to determine whether they are in the same building by asking Bob's IDA about the building he is in. Here Bob's IDA goes through a service discovery step of its own and determines that a location tracking service offered by his cell phone operator is adequate. Completion of the scenario involves a few additional steps of the same type. Note that in this scenario we have assumed that Mary's IDA trusts the location information returned by Bob's IDA. It is easy to imagine scenarios where her IDA would be better off looking for a completely independent source of information. It is also easy to see that these types of scenarios can lead to deadlocks. This is further discussed later in this paper.

5 Query Status Model

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. In other words, 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 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. Some of the predicates list in the Table will be used in Section 6, when we revisit the example introduced in Section 4. 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 six categories: 1) communication; 2) query; 3) query elements; 4) access control; 5) obfuscation and 6) information collection.

	Sample Status Predicates	**Description**
1)	Query-Received	A particular query has been received.
	Sending-Response	Response to a query is being sent
	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
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)	Check-Condition	Check whether a condition is satisfied. Special type of query element.
	Element-not-locally-available	The value of a query element can not be obtained from the local knowledge base
	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
	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-response	A service failed to provide a response. Again this predicate could be refined to distinguish between different types of failure (e.g. service down, access denied, etc.)
	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.

Query status information is updated by asserting new facts (with old information being cleaned up by the IDA's housekeeping module). 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 eventual activation of one or more of the IDA's modules. As already mentioned earlier, this meta-control architecture can also be used to model the user as a module that can be consulted by the meta-controller, e.g. to ask for a particular piece of domain knowledge or to decide whether or not to abandon a particular course of action such as looking for an external service capable of providing a particular query element.

6 Updating Query Status Information: Example Revisited

The following illustrates the processing of a query by an IDA, using the scenario introduced in Fig. 2. Specifically, Fig. 3 depicts some of the main steps involved in processing a request from Bob about the room Mary is in, highlighting some of the main query status information updates. Bob's query about the room Mary is in 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" "324455")
(triple "Query#sender" "query1" "bob")
(triple "Query#element" "query1" "element1")
(triple "Ontology#office" "mary" "element1")
```

Next, the meta-controller activates the *Query Decomposition Module,* resulting in the creation of two query elements – for the sake of simplicity we omit Mary's obfuscation policy: one query element to establish whether this request is compatible with Mary's access control policies and the other to obtain the room she is in:

```
(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*. 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 and Mary are on the same team:

```
(triple "Status#predicate" "status4" "element-needed")
(triple "Query#queryId" "status4" "element2")
(triple "Query#parentId" "status4" "query1")
(triple "Query#condition" "element2" "People#same-team")
(triple "People#same-team" "mary" "bob")
```

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

```
(triple "Status#predicate" "status5" "element-not-locally-available")
(triple "Query#queryId" "status5" "element3")
(triple "Query#parentId" "status5" "element2")
(triple "People#team" "bob" "element3")
```

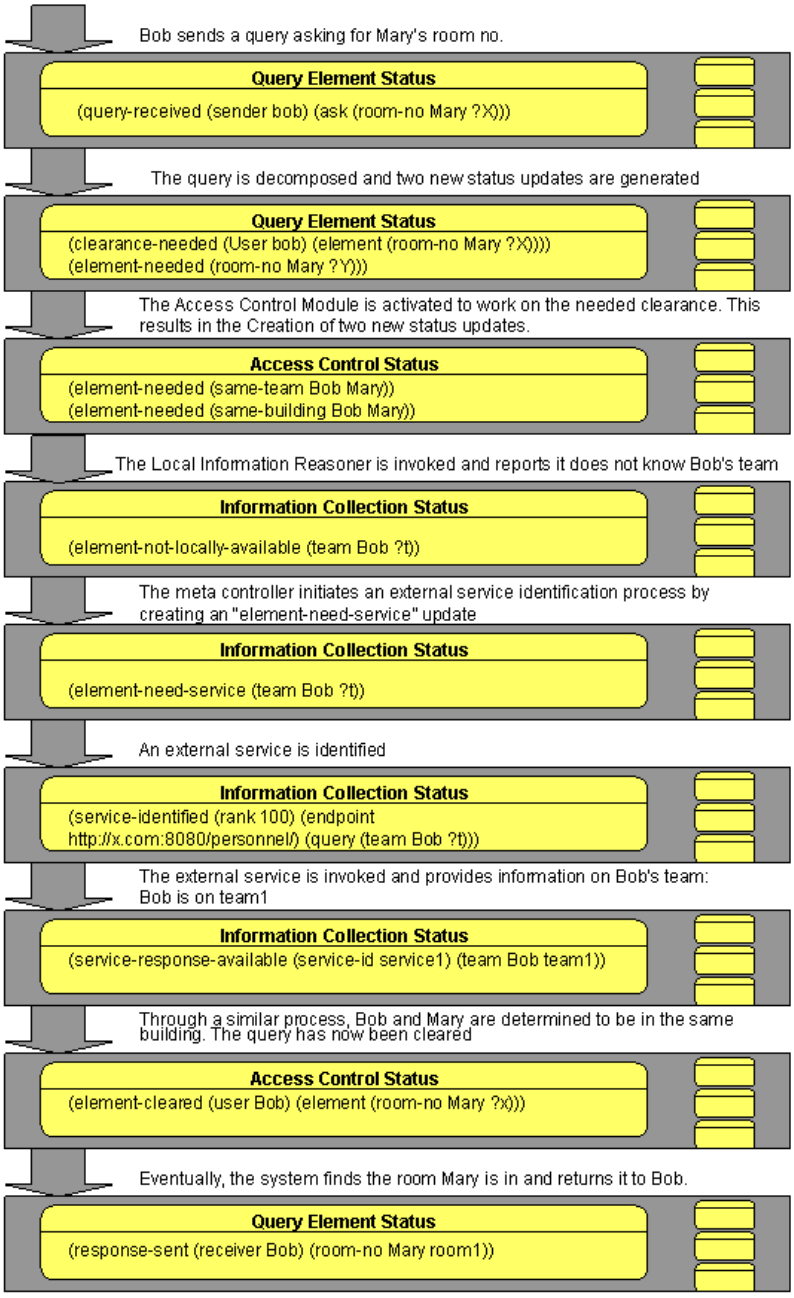
Bob sends a query asking for Mary's room no.
Query Element Status
(query-received (sender bob) (ask (room-no Mary ?X)))
The query is decomposed and two new status updates are generated
Query Element Status
(clearance-needed (User bob) (element (room-no Mary ?X))))
(element-needed (room-no Mary ?Y)))
The Access Control Module is activated to work on the needed clearance. This results in the Creation of two new status updates.
Access Control Status
(element-needed (same-team Bob Mary))
(element-needed (same-building Bob Mary))
The Local Information Reasoner is invoked and reports it does not know Bob's team
Information Collection Status
(element-not-locally-available (team Bob ?t))
The meta controller initiates an external service identification process by creating an "element-need-service" update
Information Collection Status
(element-need-service (team Bob ?t))
An external service is identified
Information Collection Status
(service-identified (rank 100) (endpoint http://x.com:8080/personnel/) (query (team Bob ?t)))
The external service is invoked and provides information on Bob's team: Bob is on team1
Information Collection Status
(service-response-available (service-id service1) (team Bob team1))
Through a similar process, Bob and Mary are determined to be in the same building. The query has now been cleared
Access Control Status
(element-cleared (user Bob) (element (room-no Mary ?x)))
Eventually, the system finds the room Mary is in and returns it to Bob.
Query Element Status
(response-sent (receiver Bob) (room-no Mary room1))

Fig. 3. Query status updates for a fragment of the scenario introduced in Fig 2.

Mary'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 [32], 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 team is identified (e.g. a service operated by company X). 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 identified by the service discovery module is now invoked and that it returns the team that Bob is on. The Housekeeping module updates the necessary Query Status Information, indicating among other things that information about Bob's team 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 (see Fig. 3)

7 The Service Discovery Model

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

Each service (or source of information) is described by a *ServiceProfile* in OWL-S [26]. In general, a *ServiceProfile* consists of three parts: (1) information about the provider of the service, (2) information about the service's functionality and (3) information about non-functional attributes [21]. Functional attributes include the service's inputs, outputs, preconditions and effects. Non-functional attributes are other properties such as accuracy, quality of service, price, location, etc. An example of a location tracking service operated on the premises of Company Y can be described as follows:

```
<profileHierarchy:InformationService rdf:ID="PositioningServ">
  <!-- reference to the service specification -->
  <service:presentedBy rdf:resource="&Serv;#PositioningServ"/>
  <profile:has_process rdf:resource="&Process;#PositionProc"/>
  <profile:serviceName Positioning_Service_in_Y />

  <!-- specification of quality rating for profile -->
  <profile:qualityRating>
    <profile:QualityRating rdf:ID="SERVQUAL">
      <profile:ratingName SERVQUAL />
      <profile:rating rdf:resource="&servqual;#Good"/>
    </profile:QualityRating>
  </profile:qualityRating>

  <profile:hasPrecondition rdf:resource="&Process;#LocateInCompanyY"/>
  <profile:hasOutput rdf:resource="&Process;#RoomNoOutput"/>

</profileHierarchy:InformationService>
```

When invoking a service it has identified, an IDA may opt to provide upfront all the input parameters required by that service or it may withhold one or more of these parameters. The latter option forces the service to request the missing input parameters from the IDA, thereby enabling the IDA to more fully determine whether the invoked service meets its policies. This option is however more computation and communication intensive.

Service outputs are represented as OWL classes, which play the role of a typing mechanism for concepts and resources. Using OWL also allows for some measure of semantic inference as part of the service discovery process. If an agent requires a service that produces as output a contextual attribute of a specific type, then all services that output the value of that attribute as a subtype are potential matches.

Service preconditions and effects are also used for service matching. For instance., the positioning service above has a precondition specifying that it is only available on company Y's premises.

8 Current Implementation: Evaluation and Discussion

Our policy enforcing agents are currently implemented in JESS, a high-performance rule-based engine in Java [6]. Domain knowledge, including service profiles, queries, access control policies and obfuscation policies are expressed in OWL [8]. As already indicated earlier ROWL the language we currently use to define rules that relate to ontologies 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. Currently all information exchange between agents is done in the clear and without digital signatures. In the future, we plan to use SSL or some equivalent protocol for all information exchange. This will include signing all queries and responses.

We have evaluated our solution 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 7.0. As part of the evaluation, we implemented the example introduced in Section 4 and 6, using a light-weight rule/fact set. The set included 22 rules and 178 facts and features a single semantic service directory with 50 services, each represented by 5 to 10 Jess rules. A breakdown of the CPU times required to process Bob's query is provided in the table below. For each module the table provides a cumulative CPU time, namely the sum of the CPU times of all invocations of that module in processing the query.

Module	*CPU time in millisecond*
Meta-Controller	28
Access-Controller	32
Local-KB	49
Service discovery / invocation	72
Total	181

While these results provide just one data point, they seem to suggest that our solution can be viewed as practical in at least some simple settings. It should be noted that our solution is not JESS-specific. In fact, other semantic web reasoners not based on Jess have been shown to be significantly more efficient than some Jess-based reasoners similar to the one used in our current implementation (e.g. see [28]). An improved version of the RETE algorithm, known as RETE2, has also been shown to be one to two orders of magnitude faster than the original RETE algorithm used in Jess. At the same time, independently of these particular implementation issues, 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 dealing with deadlocks or reasoning about provenance issues (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 [22].

9 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 privacy and security 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.

The Information Disclosure Agent presented in this paper is just one instantiation of our more general concept of Policy Enforcing Agents (PEAs). Other policies (e.g. information collection policies, notification preference policies) will typically rely on slightly different sets of modules and different meta-control strategies, yet they could all be implemented using the same meta-control architecture and many of the same principles presented in this paper. In general, PEAs 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.). Preliminary evaluation of an early implementation of our framework seems encouraging. At the same time, it is easy to see that the generality of the framework also gives rise to a number of challenging issues. Future work will focus on exploring scalability issues, evaluating tradeoffs between the expressiveness of privacy and security policies we allow and associated computational and communication requirements. 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), breaking deadlocks [15], and 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. Additional support has been provided by IBM, HP, Symbol, Boeing, Amazon, Fujitsu, the EU IST Program (SWAP project), and the ROC’s Institute for Information Industry.

This research has also benefited from interactions with Lujo Bauer, Lorrie Cranor, Fabien Gandon, Jason Hong, Bruce McLaren, Mike Reiter and Peter Steenkiste.

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, AAAI 2004 Spring Symposium Series, Stanford University, Stanford California, 2004.

[2] 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.

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

[4] 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.

[5] IBM, The Enterprise Privacy Authorization Language (EPAL 1.1) http://www.zurich.ibm.com/security/enterprise-privacy/epal/.

[6] E. Friedman-Hill. Jess in Action: Java Rule-based Systems, Manning Publications Company, June 2003, ISBN 1930110898, http://herzberg.ca.sandia.gov/jess/

[7] 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),* Florida, October 2003.

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

[9] 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*, Berkeley, California, January 2004.

[10] 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. http://www.daml.org/rules/proposal/.

[11] 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

[12] L. Kagal, T. Finin, and A. Joshi. A policy language for a pervasive computing environment. In Collection of IEEE 4th International Workshop on Policies for Distributed Systems and Networks, June 2003

[13] 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.

[14] 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. Also available as Technical Report CMU-CS-05-111, Carnegie Mellon University, February 2005.

[15] T. Leithead, W. Nejdl, D. Olmedilla, K. Seamons, M. Winslett, T. Yu, and C. Zhang, How to Exploit Ontologies in Trust Negotiation. Workshop on Trust, Security, and Reputation on the Semantic Web, part of ISWC04, Hiroshima, Japan, November 2004.

[16] OASIS, eXtensible Access Control Markup Language (XACML)

[17] OASIS, Security Assertion Markup Language (SAML)

[18] The Rule Markup Initiative. (http://www.ruleml.org)

[19] 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.

[20] N.M. Sadeh, F. Gandon, and Oh Byung Kwon. Ambient Intelligence: The MyCampus Experience. Carnegie Mellon University Technical ReportCMU-ISRI-05-123. June 2005.

[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] SweetRules. http://sweetrules.projects.semwebcentral.org/
[23] 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
[24] 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.
[25] A P3P Preference Exchange Language 1.0 (APPEL1.0). W3C Working Draft 15 April 2002, http://www.w3.org/TR/P3P-preferences/
[26] OWL-S: Semantic Markup for Web Services, W3C Submission Member Submission, November 2004. http://www.w3.org/Submission/OWL-S
[27] Web Service Modeling Ontology, WSMO. http://www.wsmo.org/
[28] Y. Gao, Z. Pan and J. Heflin, An Evaluation of knowledge Base Systems for Large OWL Datasets, In Proceedings of the Third International Semantic Web Conference, Hiroshima, Japan, November, 2004.
[29] M. Paolucci, T. Kawamura, T.R. Payne, and K. Sycara, Semantic Matching of Web Services Capabilities, In Proceedings of the First International Semantic Web Conference, Sardinia, Italy, June, 2002.
[30] G. Denker, L. Kagal, T. Finin, M. Paolucci and K. Sycara, Security For DAML Web Services: Annotation and Matchmaking, In Proceedings of the Second International Semantic Web Conference, Sandial Island, Fl, USA, October 2003.
[31] J. Rao. Semantic Web Service Composition via Logic-based Program Synthesis. PhD Thesis. Norwegian University of Science and Technology. December 10, 2004.
[32] CLIPS. http://www.ghg.net/clips/CLIPS.html.