A Policy Language for a Pervasive Computing Environment[*]

Lalana Kagal, Tim Finin and Anupam Joshi
Department of Computer Science and Electrical Engineering
University of Maryland Baltimore County
Baltimore, MD 21250
{lkagal1, finin, joshi}@cs.umbc.edu

Abstract

In this paper we describe a policy language designed for pervasive computing applications that is based on deontic concepts and grounded in a semantic language. The pervasive computing environments under consideration are those in which people and devices are mobile and use various wireless networking technologies to discover and access services and devices in their vicinity. Such pervasive environments lend themselves to policy-based security due to their extremely dynamic nature. Using policies allows the security functionality to be modified without changing the implementation of the entities involved. However, along with being extremely dynamic these environments also tend to span several domains and be made up of entities of varied capabilities. A policy language for environments of this sort needs to be very expressive but lightweight and easily extensible. We demonstrate the feasibility of our policy language in pervasive environments through a prototype used as part of a secure pervasive system.

1. Introduction

Policies guide the behavior of entities within the policy domain and have been used extensively in security, management and even network routing. Policy-based security is often used in systems where flexibility is required as users, services and access rights change frequently. As computationally enabled devices (laptops, phones, PDAs, and even household appliances) become more commonplace and short range wireless connectivity improves, there is a increased need for more automated security in the resulting pervasive environments formed by mobile users accessing these resources and other services and information using handheld devices. These environments will be populated by a large number of wirelessly networked heterogeneous users, services and semi-automated entities of varied capabilities making it necessary to ensure that all these different entities behave appropriately. Towards this end, we believe that policy based security will be the most effective as it will be possible to modify how different entities act without modifying their internal mechanism.

Policies generally require application specific information to reason over, forcing researchers to create policy languages that are bound to the domains for which they were developed. This prevents policy languages from being flexible or being applicable across domains. In order to enable entities in pervasive computing systems, which consist of different domains and systems, to understand and interpret policies correctly, we propose that they be represented in a semantic language like RDF-S [4], DAML+OIL [8] or OWL [6]. We believe that using a semantic language allows different systems to share a model of policies, roles and other attributes [3, 12].

In this paper, we describe the specification of our policy language, Rei[1]. Rei is based on deontic concepts [17, 18, 27] and includes constructs for rights, prohibitions, obligations and dispensations (deferred obligations). The language consists of a few simple constructs that are extremely flexible and allows different kinds of policies (security, privacy, management, conversation etc.) to be specified. The policy language is not tied to any specific application and permits domain specific information to be added without modification. As our policy language is geared towards environments that consist of several domains we believe that there is a potential for policy conflicts. The language includes two constructs for specifying meta-policies that are invoked to resolve conflicts; setting the modality preference (negative over positive or vice versa) or stating

[*]This research was supported in part by Hewlett-Packard Labs, Palo Alto and by DARPA contract F30602-97-1-0215.

[1]Rei, pronounced *ray*, is a Japanese 'Kanji' character which means 'universal' or 'essence'. It was chosen to indicate the universal applicability of the policy language, as its flexibility and versatility allow a large variety of policies, including security, conversation and management, to be specified.

the priority between policies. For example, it is possible to say that in case of conflict the Federal policy always overrides the State policy.

In pervasive environments, describing comprehensive policies may be time consuming, if at all possible. We believe that policies should be as simple as possible and control should be decentralized, that is authorization should be possible by more than just a few key entities. Rei models speech acts like delegation, revocation, request and cancel that allow policies to be less exhaustive and allow for decentralized security control.

Due to the large number of entities in the environment, it may not be possible to identify them accurately or even pre-determine the users of each service. Therefore we suggest developing policies associated with properties of entities instead of identities [11, 13]. These properties are established by proving them from an entity's credentials, beliefs of other entities and the appropriate security policies.

The paper is structured as follows: The discussion about the specification of the language in Section 2 includes the policy constructs, the action specifications and the representation of domain specific information. Following this, in Section 3 we describe the types of possible conflicts and the meta policies that can be used to resolve them. As delegation management is very important in the environments we work with, in Section 4 we present our approach to delegation and discuss the types of delegation Rei covers. The implementation details of the policy engine associated with Rei are covered in Section 5. In Section 6 we talk about a prototype of a secure pervasive system that we developed to test the feasibility of the policy language. After discussing related background work, we state the contributions of our research in Section 8. Finally in Section 9 we summarize our work and describe future research directions.

2. Structure of Policy Language

Our policy language is modeled on deontic concepts of rights, prohibitions, obligations and dispensations [10]. We believe that most policies can be expressed as what an entity (user, agent, service, etc.) can/cannot and should/should not do in terms of actions, services, conversations etc., making our language capable of describing a large variety of policies ranging from security policies to conversation and behavior policies. Rei is implemented in a logic language, namely Prolog [24]. We have also developed ontologies that enables the policy engine to interpret a subset of RDF-S policies.

The Rei policy language includes certain domain independent ontologies and accepts domain dependent ontologies. The former includes concepts for permissions, obligations, actions, speech acts, operators etc. The latter is a set of ontologies, shared by the entities in the system, which define domain classes (person, file, deleteAFile, readBook) and properties associated with the classes (age, num-pages, email).

Though the execution of actions is outside the policy engine, the policy language includes a representation of actions that allows more contextual information to be captured and allows for greater understanding of the action and its parameters. It similarly permits domain dependent information to be added.

Rei includes three types of constructs: (i) policy objects, (ii) meta policies and (iii)speech acts. (i) The policy objects represent rights, obligations, prohibitions and dispensations. The *has* policy construct allows these objects to be associated with different entities creating *policy rules*. This allows for reuse of policy objects. For example, the same right to read a certain file could be associated with different entities in different policy domains. (ii) The policy language contains meta-policy specifications for conflict resolution. These include constructs for specifying precedence of modality and priority of policies. (iii) Rei models four speech acts that can be used within the system to modify policies dynamically: delegate, revoke, cancel and request. In order to make correct policy decisions, we assume the presence of a monitoring service that sends all relevant speech acts to the policy engine. Associated with the policy language is the policy engine that interprets and reasons over the policies, speech acts and domain information to make decisions about users rights and obligations.

2.1. Policy Objects

The core of the policy language is the set of constructs that describe the concepts of rights, prohibitions, obligations, and dispensations. These constructs denoted by *PolicyObject* are represented as

- *PolicyObject(Action, Conditions)*

where, *Action* is a domain dependent action and *Conditions* are constraints on the actor, action and environment. A policy object defines a commonly occurring right, obligation, prohibition and dispensation. For example, the right to print to a certain printer, the obligation to brew a fresh pot of coffee when the coffee pot is empty, etc.

In order to associate a policy object with an entity, the *has* construct is used.

- *has(Subject, PolicyObject)*

where, *Subject* can either be a URI identifying an entity or a variable, allowing all entities who satisfy the conditions to be associated with the policy object to possess the policy object. A policy consists of several *has* rules.

Rei allows *role based or group based* policies to be defined by using *has* with a variable and specifying the role or group, which are domain dependent, as part of the condition of the policy object. In this way, policies can be individual, role, group - based, or any combination of the three. This mechanism is different from existing policy languages, which include special constructs for role/group based rights/obligations [5, 15].

There are four kinds of policy objects: (i) Rights, (ii) Prohibitions, (iii) Obligations, and (iv) Dispensations.

- Rights are permissions that an entity has. The possession of a right allows the entity to perform the associated action.

  An entity, *abc*, can perform an action, *actionABC* if and only if at least one of the following conditions are true

  - *has(abc, right(actionABC, Conditions))* and *abc* satisfies *Conditions*
  - *has(Variable*[2], *right(actionABC, Conditions))*, *abc* binds to *Variable* and satisfies *Conditions*

  Example 1. A rule that states that all employees of 'UMBC' can perform printAction1 is represented as

  has(Variable, right(printAction1, (employee(Variable, 'UMBC'))))

  In this policy rule, *printAction1* is an action and *employee* is a condition.

- Prohibitions are negative authorizations implying that an entity cannot perform the action.

  An entity, *abc*, is prohibited from performing *actionABC* if and only if at least one of the following conditions is true

  - *has(abc, prohibition(actionABC, Conditions))* and *abc* satisfies *Conditions*
  - *has(Variable, prohibition(actionABC, Conditions))* and *abc* satisfies *Conditions*

  Example 2. A rule that states that students of 'UMBC' are prohibited from using the faculty printer is specified as

  has(Variable, prohibition(useFacultyPrinter, (student(Variable,'UMBC'))))

  useFacultyPrinter is an action and *student(Variable, 'UMBC')* is a condition that an entity must satisfy in order to possess the prohibition.

[2]In prolog, any word starting with an uppercase letter is assumed to be a variable. All constants start with a lowercase letter.

- Obligations are actions that an entity must perform and are usually triggered when a certain set of conditions are true.

  An entity, *abc*, is obliged to perform *actionABC* if and only if one of the following conditions are true

  - *has(abc, obligation(actionABC, Conditions))* and *abc* satisfies *Conditions*
  - *has(Variable, obligation(actionABC, Conditions))* and *abc* satisfies *Conditions*

  Example 3. A policy rule specifying that all employees of 'ABC' should display their badges while at work is represented as

  has(Variable, obliged(displayBadge, (employee(X,'ABC'), atWork(X))))

  displayBadge is an action and *employee(X, 'ABC'* and *atWork(X)* are a domain dependent condition.

  There are additional parameters associated with Obligation for better processing of the policy object; MetEffects and NotMetEffects [20]. The subject can decide whether to complete the obligation by comparing the effects of meeting the obligation (MetEffects) and the effects of not meeting the obligation (NotMetEffects). However, this reasoning is not part of the policy engine and is left to the individual subject. The policy engine learns of the environment including actions being performed, relevant speech acts, and other contextual information from the monitoring service. Using this information, the policy engine is able to infer which obligations were fulfilled and which obligations still need to be fulfilled.

- Dispensations are actions that an entity is no longer required to perform. They act as waivers for existing obligations.

  An entity, *abc*, is no longer obliged to perform an action, *actionABC* if and only if

  - *has(abc, obligation(actionABC, OConditions))* and if *abc* satisfies *OConditions*
  - *has(abc, dispensation(actionABC, DConditions))* and if *abc* satisfies *DConditions*.

    OR
  - *has(Variable, obligation(actionABC, OConditions))* and if *abc* satisfies *OConditions*
  - *has(Variable, dispensation(actionABC, DConditions))* and if *abc* satisfies *DConditions*.

  Example 4. John is no longer required to pay alimony to his wife after she re-marries.

  has(john, dispensation(payAlimony, wife(john, X), remarried(X)))

2.2. Action Specifications

The policy language suggests a representation of actions that allows for greater understanding of the action and its parameters. Actions can be represented as a tuple with the following parameters

- *action(ActionName, TargetObjects, Pre-Conditions, Effects)*

where, *ActionName* is the identifier or URI of the action, *TargetObjects* is a list of objects on which the action is performed, *Pre-Conditions* are the conditions that need to be true before the action can be performed and *Effects* are the results of the action being performed. Preconditions reflect the context in which the action can be performed and Effects are used to infer the state of the environment after the action is performed.

Example 5. The action of printing a page on printerHP can be represented as

action(printOnePageHP, [printerHP], (containsCartridge(printerHP),availablePaper(printerHP, X), X $>$ 1), availablePaper(printerHP, X-1))

2.2.1. Action Operators

Though we would like the policy language to be as simple as possible, certain additional constructs are required to create complex action descriptions. For example, there is a difference between John having the permission to perform action A followed by B, and John having the permission to perform A and the permission to perform B. We tried to model these operators using the existing action specifications but were unable to come up with a satisfactory result forcing the additional complexity.

The policy language includes four action operators that allow various kinds of complex actions to be specified; sequence, non-deterministic, once, and repetition [7].

- Sequence : If A and B are actions, *seq(A,B)* denotes that action B must only be performed after action A or that action A and B must be performed in sequence.

- Non-deterministic : If A and B are actions, *nond(A,B)* denotes a choice between A and B implying either A or B can be performed, but not both.

- Repetition : If A is an action, *repetition(A)* denotes that A can be executed several times.

- Once : If A is an action, *once(A)* denotes that A can only be performed once.

Example 6. Consider a right associated with John. *John* has the right to either perform action *printBW* followed by repeated executions of *printColor* or perform action *faxBW* once. He only has the right while he is a lab member of 'AI'.

has(john, right(nond(seq(printBW, repetition(printColor)), once(faxBW)), lab-member(john,'AI'))).

2.3. Speech Acts

As Rei is geared towards highly distributed and large environments, it includes a representation of speech acts that are used to decentralize control. The policy language currently includes specifications for four speech acts that affect the policy objects of the communicating entities: (i) delegation, (ii) request, (iii) cancel, and (iv) revocation. These speech acts are also governed by policies and entities can only use a certain speech act if they have the right to it. Consider for example, John may have the right to send a *request* to Joan but not a *delegate* or a *cancel*. The structure of the speech acts allows for natural nesting of speech acts with policy objects. For example, it is possible to request a dispensation or cancel a request. The policy engine currently implements a certain subset of these nested speech acts namely delegation of rights and actions, requesting of rights and actions, cancellation of requests, and revoking of any types of rights including the right to delegate.

- Delegation : A delegation allows an entity to give a right to another entity or group of entities. A delegation, if valid, causes a *Right* to be created. Only an entity with the *Right* to delegate can make a valid delegation. A delegator always retains the right to revoke the delegated right.

  A delegation can be represented as

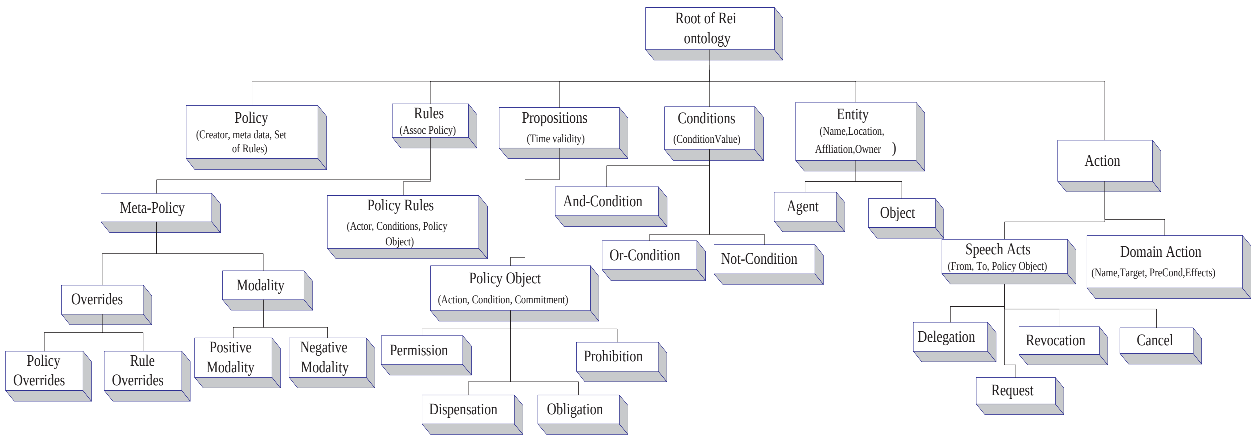
  - *delegateSpeechAct(Sender, Receiver, right(Action, Condition))* and Receiver satisfies Conditions $\longrightarrow$ *has(Receiver,right(Action, Condition))*

  Example 7. Assuming that John has the right to delegate, he delegates to Mark the right to print on the lab printer as long as he is working on the same project as John.

  delegateSpeechAct(john, mark, right(printLabPrinter,(project(john, SomeProject), project(mark, SomeProject))))

  action(printLabprinter, [laserPrinter123], queue(laserPrinter123,0), queue(laserPrinter123, 1))

- Request : There are two kinds of requests; a request for an action and a request for a right. The former, if accepted, causes an obligation on the part of the receiver. A request for a right, if valid and accepted, causes the receiver to delegate the right to the sender.



Root of Rei ontology
Policy (Creator, meta data, Set of Rules)
Rules (Assoc Policy)
Propositions (Time validity)
Conditions (ConditionValue)
Entity (Name,Location, Affliation,Owner)
Action
Meta-Policy
Policy Rules (Actor, Conditions, Policy Object)
And-Condition
Agent
Object
Or-Condition
Not-Condition
Speech Acts (From, To, Policy Object)
Domain Action (Name,Target, PreCond,Effects)
Overrides
Modality
Policy Object (Action, Condition, Commitment)
Policy Overrides
Rule Overrides
Positive Modality
Negative Modality
Permission
Prohibition
Dispensation
Obligation
Delegation
Revocation
Cancel
Request

Figure 1. Rei Policy Language Ontology

However, the delegation allows the sender the permission to perform the action only if the receiver has the right to delegate.

- *requestSpeechAct(Sender, Receiver, Action)* ⟶ disagree
  requestSpeechAct(Sender, Receiver, Action) ⟶ *has(Receiver, obligation(Action, Condition))*
- *requestSpeechAct(Sender, Receiver, right(Action, Condition))* ⟶ disagree
  request(Sender, Receiver, right(Action, Condition)) ⟶
  delegateSpeechAct(Receiver,Sender, right(Action, XCondition))

- Revoke : Revocation is the removal of a right and acts as a prohibition. A revocation is allowed in two cases; an entity can revoke those rights to which it has the right to revoke or those rights that it has itself delegated.

  revokeSpeechAct(Sender, Receiver, right(Action, Condition)) ⟶ *revocation(Receiver, right(Action, Condition)) and Sender no longer has the right to perform Action*

- Cancel : An entity can cancel any request it has sent, and this nullifies any obligation or delegation caused by the request.
  - *cancelSpeechAct(Sender, Receiver, Action) AND has(Receiver, obligation(Action, true))* ⟶ *has(Receiver, dispensation(Action, Condition))*
  - *cancelSpeechAct(Sender, Receiver, right(Action, X))* ⟶ *Sender no longer has the right to perform Action*

2.4. Domain Specific Information

The Rei policy engine accepts information about entities and their properties in any semantic language that can be converted into triples of the form Subject, Predicate, Object). It understands class hierarchies and interprets any correct instance of our ontology or any subclasses. All semantic information is stored and reasoned over in triples. Figure 1 illustrates the ontology we are developing to express policies in RDF-S [4]. The RDF-S ontology is available at *http://daml.umbc.edu/ontologies/policy/* and is being developed as a part of a larger ontology of security, trust and privacy concepts. As an example, consider a security policy being developed for a database application. Database related actions like creating/viewing/deleting/updating a table, adding/deleting/modifying a record, and creating different views will be instances or subclasses of our *Domain Action*. The databases, tables, and users of the system will be represented as classes under *Entity*. The policy itself will be specified as a set of both policy rules and meta-policy rules.

2.5. Conditions

As it is not always possible to identify entities in pervasive systems, along with allowing identity based policies, Rei also permits policies to be based on properties of the action, targets, subjects and other environmental factors like time and place. The policy languages permits complex conditions to be built from these properties and supports the following operators;

- AND : A complex condition made of two conditions associated with an AND, will be true only when both conditions are true.

For example, *(employee(X, 'UMBC'), lab-member(X, 'AI'))* will only be true if both conditions are true

- OR : A complex condition made of two conditions associated with an OR, will be true only when one of the conditions is true
- NOT : A complex condition consisting of *not(ComplexCondition)* is true when *ComplexCondition* is cannot be proved.

Example 8. As an example, consider a complex constraint made of application dependent conditions, which is when the agent is a lab member of 'AI' or if the agent is not a group member of 'IR'.
(lab-member(X, 'AI'); not(group-member(X, 'IR')))

However, Rei requires a semantic rule language that includes the unification of variables as provided by Prolog. After studying the existing semantic languages like RDF-S, DAML+OIL, OWL and rule based languages like RuleML [2], we found that this is currently not supported. We are currently looking into the feasibility of using a language which is a combination of Prolog and a semantic language to represent conditions. Our prolog engine currently accepts conditions as combinations of triples and prolog predicates.

3. Conflicts and Conflict Resolution

Due to the nature of a pervasive environment, we expect that there will be several policies applicable to every domain which could lead to potential conflicts. For example, in one policy Mary has the obligation to write the report and another policy prohibits Mary from writing the same report. Conflicts occur if policies overlap in subject, target and action but the policy objects are different.

Based on Moffett and Sloman [19] we identify two kinds of conflicts

- Conflict of modality : These conflicts occur when an entity is authorized to perform (Right) and forbidden from performing (Prohibition) a certain action on a certain set of targets or is both required to (Obligation) and waived from (Dispensation) performing a certain action on a certain set of targets. This includes Rights, Prohibitions, Obligations and Dispensations caused by speech acts.
- Conflict of Obligation and Prohibition : These conflicts occur when an entity is required to perform (Obligation) and forbidden from performing (Prohibition) a certain action on a certain set of targets. However, in Rei an entity must have the right to perform an action before performing it, which causes the conflict to reduce to a conflict between a Prohibition and a Right.

In order to resolve conflicts, Rei includes meta-policies that are policies about how policies themselves are interpreted and how conflicts are resolved at run-time. While reasoning over a security decision, if the policy engine comes across two opposing policy objects for the entities under consideration, it declares a conflict and tries to find the most appropriate meta policy to resolve the conflict. Meta policies in our system regulate conflicting policies in two ways; (i) priorities and (ii) precedence relations [16].

3.1. Priorities

Priorities can be specified between named policy rules or even entire policies. For example, it is possible to state that the school policy overrides the department policy in case of conflict. It is also possible to set priorities between any two rules. If there is a rule, A1, giving Mark the right to print and a rule, B1, prohibiting Mark from printing. By using *overrides(A1, B1)*, A1 is given priority over B1. The conflict between the two rules is resolved at run-time giving Mark the right to print. The same construct is used to specify priorities between policies. It is also possible to specify whether priorities between policies should be evaluated before or after priorities between conflicting policy rules.

3.2. Precedence

It is possible to specify which modality holds precedence over the other in the meta-policies. The policy maker can associate a certain precedence for a set of actions or a set of entities satisfying certain conditions. The constructs to be used are *metaRuleAction*, *metaRuleAgent* and *metaRule*. If negative modality holds, prohibitions hold over rights and dispensations are stronger than obligations for the set of entities that fulfill the associated conditions of the meta-policy construct, for positive modality it is the reverse.

The three kinds of meta rules:

- *metaRuleAction(ActionConditions, positive/negative)* : This allows the modality precedence to be set for a set of actions that satisfy the action conditions.
- *metaRuleAgent(ConditionOnAgent, positive/negative)* : This allows the modality precedence to be set for a set of entities that satisfy the conditions.
- *metaRule(Policy, positive/negative)* : This is the default precedence that can be set for a policy.

Example 9. A meta policy that specifies that for any conflict regarding policies for employees of ABC, negative modality holds precedence is defined as
metaRuleAgent(employee(X, 'ABC'), negative-modality)

There exists a partial ordering among the meta-policies as well and this ordering can be manipulated for every policy. The default ordering is: the meta rules associated with actions have the highest priority, followed by meta rules about subjects and if there are no rules associated with the action or the subject, then the default meta rule is considered.

4. Delegation Management

Delegation is important in highly dynamic and widely distributed systems because it allows the policy to be relatively simple and allows the rights of entities to be configured dynamically. A policy for all printers in a lab could be defined so that managers have the right to delegate the right to print and the right to re-delegate this right to any employee of the company. However, if any employee that they delegate to, misbehaves in any way, the system will hold the associated manager responsible. This forces the managers to be careful with their delegations, while at the same time allowing the rights on the printers to propagate.

The Rei policy language defines three types of interrelated rights associated with each action, out of which the last two give certain delegation rights.

- Right to execute : Possessing this right allows the agent to perform the action.

  has(Agent, right(Action, Condition))

  where Action is the action and Condition are the conditions on execution

- Right to delegate execution : If an agent possesses the right to delegate the execution of an action, it can delegate to other entities the right to perform the action but the agent cannot perform the action itself.

  NOT has(Agent, right(Action, ECondition))

  has(Agent, right(Action1, Condition1))

  where, *Condition1* are the conditions on the *Agent*

  Action1 is *delegate(right(Action, Condition))*

  This right gives the possessor the right to delegate the previous right, the right to execute.

- Right to delegate delegation right : The agent can delegate to another agent or a group of entities the right to further delegate the right to perform the action and delegate this delegation right as well. Though at this point the right should have been divided into right to delegate the right to execution and the right to delegate the right to delegation, we decided to combine them as we expect that the conditions on every right will take care of the propagation of the delegation. This right gives the possessor the right to delegate the previous right, the right to delegate execution and the right to delegate the delegation itself.

  NOT has(Agent, right(Action, ECondition))

  has(Agent, right(Action1, Condition1))

  where, *Condition1* are the conditions on the delegator, *Agent*

  Action1 is delegate(right(Action2, Condition2))

  Condition2 are the conditions on the delegatee

  Action 2 is delegate(right(Action, Condition))

These three rights force conditions on the executor of the action, the delegator of the action and whom the right can be delegated to. An agent has the right to a certain action (including speech acts) if it possesses the right or if it has been delegated the right. It should satisfy the conditions associated with the innermost right of execution of every delegation in the chain. Each delegator should satisfy the condition on the delegation of the delegation before it in the chain and the delegatee conditions of all previous delegations. If any entity fails any delegator condition, the delegations from that point on are invalid. The policy engine ensures that circular delegations are not allowed, i.e., an entity cannot delegate a right to itself or to another entity who delegates it back to the delegator or to a previous delegator in the delegation chain.

Example 10. This example demonstrates the working of cascaded delegations.

- Amy has the right to delegate the right to execute printOnePageHP and she delegates this right to Tim. Tim can only execute printOnePageHP if he satisfies both the condition associated with the speech act (employee(tim, 'UMBC') and the condition associated with Amy's delegation right (group-member(X, Group)).
  has(amy, right(amy, delegate(right(X, print, group-member(X,Group))), employee(amy, 'UMBC')))
  delegateSpeechAct(amy, tim, right(tim, print, employee(tim, 'UMBC')))

- John has the right to delegate the right to delegate the right to execute printOnePageHP and he delegates this right to Tim. Tim can delegate as long as he is satisfies group-member(X, Group) from John's right to delegate and employee(tim, 'UMBC') from the

delegation. However, John's right and the delegation also place conditions on whom Tim can delegate to, in this case to lab members of 'AI' and employees of 'UMBC'.
has(amy, right(john, delegate(right(X, delegate(right(Y, print,lab-member(Y, 'AI'))), group-member(X,Group))),employee(john,'UMBC')))
delegateSpeechAct(john, tim, right(tim, delegate(right(Y, delegate(right(Z, print, employee(Z,'UMBC'))), employee(Y, 'UMBC'))),employee(tim, 'UMBC')))

- Tim delegates to Jane the right to delegate the right to execute printOnePageHP. Jane must satisfy the conditions associated with the earlier delegations and rights in the delegation chain in order to be able to delegate the right to printOnePageHP and the delegation will only be valid if the entity she delegates to satisfy all the associated conditions as well.
delegateSpeechAct(tim, jane, right(jane, delegate(right(X, print, group-member(X, Group))),(employee(jane, 'UMBC'), time-now(morning))))

4.1. Delegation Types

We identify two types of delegation, while-delegations and when-delegations. A *while-delegation* forces all following delegators to satisfy its conditions in order to be true. Whereas a *when-delegation* requires the immediate delegator to satisfy its conditions only at the time of the delegation and not after. For example, consider a when-delegation giving Jane the right to delegate when she's an employee. All the delegations that Jane made while she was an employee hold even after she leaves. On the other hand, a similar while-delegation will fail once the delegator leaves the company. The while delegation is the default delegation type.

Example 11. The following represents the example described above

- When-Delegation

  delegateWhenSpeech(mark, matthew, right(Action, Condition)) , Matthew satisfies Condition $\longrightarrow$ *has(Matthew, right(Action, Condition)*

  Matthew no longer satisfies Condition $\longrightarrow$ *has(Matthew, right(Action, Condition)*

- While-Delegation or Default Delegation

  delegateSpeech(mark, matthew, right(Action, Condition)), Matthew satisfies Condition $\longrightarrow$ *has(Matthew, right(Action, Condition)*

  Matthew does not satisfy condition $\longrightarrow$ *NOT has(Matthew, right(Action, Condition)*

5. Policy Engine

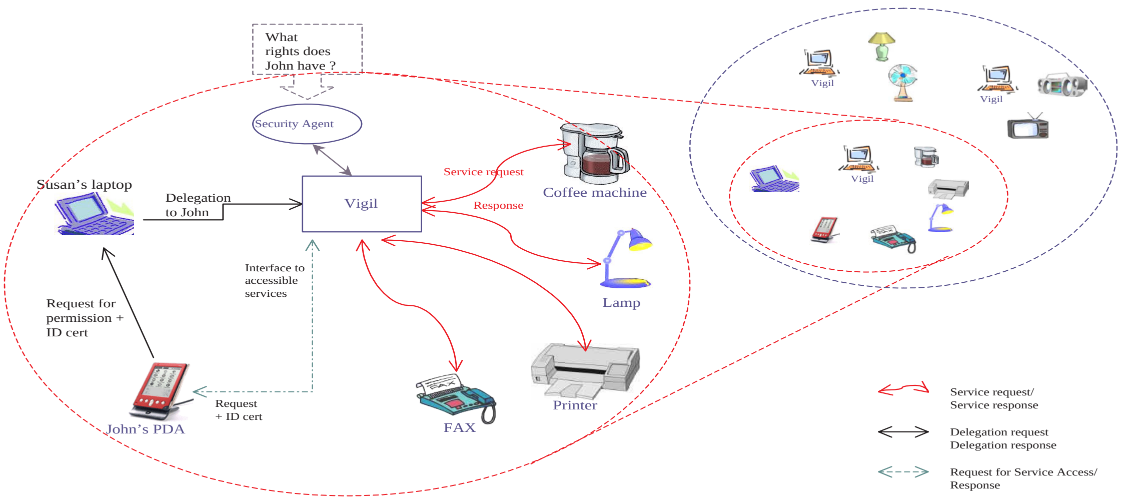
We have developed a policy engine that reasons over policies described in the Rei policy language and uses the policies, meta-policies and domain knowledge to make security decisions about access rights and obligations. Along with policies in Prolog, the policy engine also accepts policies in RDF-S that are based on the existing Rei ontology. The policy engine is developed in Java and uses Prolog as a reasoning engine. It currently has a commandline interface and a Java interface. We envision that the policy engine will be used as a security module by an application along the similar lines as our application, Vigil - a security framework for pervasive environments.

6. Application : Pervasive Environment

In the ubiquitous computing paradigm, information and services are accessible virtually anywhere and at any time via any device - phones, PDAs, laptops or even watches [22, 29]. The *SmartSpace* scenario is the first step towards realizing this vision. Smart homes and offices consist of intelligent services that are accessible to users via handheld devices connected over short range wireless links. The services will be integrated seamlessly into the environment that the user is familiar with, enabling easy and automatic usage. This is the vision that guides our research on the Vigil system. We define a SmartSpace as a dynamic environment that provides an infrastructure for providing services to mobile users via some short range wireless communication link.

We have designed and implemented Vigil, a security framework, which provides access control to services in SmartSpaces [14, 26]. Within a confined space, the client can access services provided by the nearest Vigil System via some short-range wireless technology. Vigil acts as an active proxy by executing services on behalf of any client that requests them. This minimizes the resource consumption on the client and also avoids having the services installed on each client that wishes to use them, which is a blessing for most resource-poor mobile clients.

Vigil consists of five functional components: (i) Communication Managers, (ii) Service Managers, (iii) Certificate Managers, (iv) Security Agents, and (v) Clients. Communication Managers handle communication between various entities in the system. The Communication Manager is flexible and allows any medium to be used for communication, but for implementation purposes, we have used Infrared, CDPD[25] and Bluetooth[1]. Users and services



What rights does John have ?
Security Agent
Susan's laptop
Delegation to John
Vigil
Service request
Response
Coffee machine
Lamp
Interface to accessible services
Request for permission + ID cert
Request + ID cert
John's PDA
FAX
Printer
Vigil
Vigil
Vigil
Service request/ Service response
Delegation request Delegation response
Request for Service Access/ Response

Figure 2. Overview of the Vigil system. There are several SmartSpaces in an organization. In every SmartSpace, a user uses the Vigil framework to gain access to services in that Space. A user can also request permission from another user to access a Service. Though Vigil has been conceptually shown as a central system, it is actually made up of distributed components.

are treated equally as Clients. All clients communicate via a language based on Extensible Markup Language(XML) [28]. All communication is encrypted via Public Key Infrastructure (PKI). Vigil does not assume that the end points are computationally robust and instead relies on a simplified PKI. The entities in the Vigil system enjoy non-repudiation, authentication, and protection from replay attacks vis-à-vis the simplified PKI. The Certificate Manager is responsible for generating x.509 version 3 digital certificates for each entity in the Vigil system and for responding to certificate validation queries from Service Managers. Service Managers are responsible for client and service management. The Security Agents provide access control for Services. Finally, users and services are treated equally as *Clients*.

Our infrastructure is designed to minimize the load on portable devices and provide a media independent infrastructure and communication protocol for the provision of services. Vigil, in addition to solving the issue of controlling access to services in a *SmartSpace*, also accommodates users that are foreign entities, that is entities that are not known to the system in advance. In many conventional systems, access rights are static; agents are not able to request permission to access a Service to which they are not preauthorized. The Vigil architecture allows agents to ask for permission and other agents to actually delegate rights that they have. This extends the security policy in a secure manner, as only agents that have the permission to delegate, can actually delegate.

A pervasive system is divided into *SmartSpaces*, and each SmartSpace in controlled by a Service Manager. The Service Manager acts as broker, matching client requests to registered services. A Service Manager uses one or more Security Agents to maintain security. The Security Agent enforces the security policy of the organization or SmartSpace and interprets the policy to provide controlled access to Services. There is a global policy associated with the organization and a local policy associated with a SmartSpace. All security agents in the organization will enforce the global policy and will additionally enforce a local policy, which is specific to the Space. Vigil uses a subset of the Rei constructs to specify policies which include rules for role assignment, access control (rights and prohibitions), delegation and revocation. The Security Agent stores Rei policies, meta-policies and contextual information in a knowledge base and makes security decisions by reasoning over this information.

Figure 2 illustrates the working of the Vigil system. A Service Manager on receiving a request for a service asks the Security Agent whether the request is valid. The Security Agent replies with a positive or negative response depending on the security policy. Based on this response, the Service Manager allows or denies the clients request.

When a user needs to access a service that it does not have the right to access, it requests another user, who has the right for the permission to access the Service. If the entity requested has the permission to delegate the access to the Service, the entity sends a delegation message to the Security Agent and the requester. The Security Agent checks the roles of the delegator and the delegatee and ensures that the delegator has the right to delegate, and that the delegation follows the security policy. It then adds the permission for the Client to access the Service, but sets a very short period of validity for the permission. Once this period is over, The Security Agent has to reprocess the delegation. This is very useful incase of revoked certificates, delegations or rights. If any one entity in the delegation chain loses the permission, then it is propagated down the chain very quickly, till everyone in the chain after the entity loses the ability.

7. Background

According to Sloman, policies define a relationship between subjects and targets [23]. Policy domains are groups on which the policy applies. Policies affect behavior of objects in domains. Sloman believes that it is important to represent and interpret policy information. He classifies policies into authorization and obligation and states that there are two kinds of constraints on policies; temporal, and parameter value. In contrast, Rei handles authorizations, prohibitions, obligations and dispensation policy rules and allows policies to be split into actions, constraints and policy objects. Rei also allows constraints to be domain dependent and external to the policy specifications.

Ponder [5] allows general security policies to be specified. The authors of Ponder define a policy as a set of rules that defines a choice in the behavior of a system. Ponder is a declarative, object oriented language for specifying security and management policies. It allows policy types to be defined to which any policy element can be passed to create a specific instance. This seems to be a useful ability and, in fact, Rei allows this to be done automatically. Rei allows types of policy objects to be defined and allows them to be linked dynamically to subjects. Ponder allows definition of positive and negative authorization policies (access control), information filtering (transforming requested information into a suitable format), delegation positive and negative. Ponder includes a very simple notion of delegation, which we believe is very important in distributed systems. Rei supports different kinds of delegation and includes mechanisms to control delegation propagation. Ponder describes meta policies as policies about policies, which is similar to the way Rei views them. Ponder provides a Group construct for group related policies and a Role construct for role policies. However, Rei does not distinguish between role based, group based and individual policies, allowing them to be described using the same set of constructs leading to simpler policies and more uniformity.

Lupu classifies conflicts into modality conflicts and application specific conflicts [16]. Modality conflicts arise when two or more policies with opposite modalities refer to the same subjects. Application specific conflicts refers to the consistency of what is contained in the policy and external criteria, e.g., the same manager cannot authorize payments and sign the payment checks. Lupu suggests a couple of ways of resolving modality conflicts; deciding a default priority, assigning explicit priorities to rules, finding the distance between the policy and the managed object to name a few. Rei accepts Lupu's definition of modality conflicts but does not use the suggested mechanisms. Rei provides a construct for specifying the modality which holds precedence for sets of agents and actions grouped by certain conditions. Rei also allows priorities to be assigned to policy rules and policies.

Most policy languages provide a certain set of constructs in some sort of a programming language. However, there has been some work in representing policies in logic [9, 30]. Woo and Lam [30] propose the use of default logic for authorization policies. Their access control decisions are not always conclusive and the work does not include conflict resolution mechanisms. Jajodia et al. describe the specifications of a language based on stratified logic that tries to support different access control policies [9]. Their Authorization Specification Language (ASL) allows users to not only specify authorization policies, but also specify the way the decisions over these policies are made. The language supports objects, on which actions are carried out, and subjects, which can be users, groups and roles. ASL depends heavily on the authors' understanding and interpretation of groups and roles, whereas in Rei, these concepts belong entirely to the domain in which it is being used, and can be interpreted as required by the domain. ASL defines an authorization policy as a 4-tuple consisting of an object, user, role set and an action. This language, though a step in the right direction, is complicated because it consists of a large set of interdependent rules that the user has to fully understand in order to use, and is not as expressive as Rei. ASL does not make provisions for domain dependent information and insists on only a specific set of conditions. Rei can be used to specify role based, group based and individual policies with the same constructs using certain user defined conditions. ASL does include conflict resolution but expects a set of rules (in terms of its predicates) to be defined for every access. Conflict resolution is more straightforward in Rei through its two mechanisms of modality precedence and priority specification. For example, in Rei it is possible to state that for all possible actions on colors printers in a certain lab, permissions should hold precedence over prohibitions.

8. Contributions

Rei is a flexible and expressive policy language that is based on deontic concepts and which can be used to describe several kinds of policies. For example, consider security policies. Security policies restrict access to certain resources in an organization. Rei can be used to create actions on the resources and to describe role based rights and prohibitions for the users in the organization. On the other hand, management policies define the role of an individual in terms of his duties and rights, which map directly into obligations and rights in Rei. Conversation policies are very important in semi autonomous environments [21] like pervasive environments. The order in which speech acts occur is called a conversation. By specifying what speech acts an agent can use under certain conditions (rights), and by specifying what speech acts an agent should use (obligation) under certain conditions (could include the speech acts just received), a policy for conversations can be specified in Rei. Other policies can similarly be described in terms of deontic principles making Rei versatile.

A specification is correct if it is both consistent and complete [9]. Though Rei allows inconsistent or incomplete specifications to be described, its policy engine is correct. Rei's policy engine is consistent because every request is either allowed or denied but not both. This is due to the structure of the meta policies, which resolves conflicts forcing the engine to come to either a positive or negative decision. The policy engine is complete because every request has a result.

Rei is composed of domain dependent information and domain independent information. Rei allows policy makers to use application specific information that Rei has no prior knowledge of but can still reason over while making decisions.

Rei allows *types* of policy objects to be specified. For example, all the rights on a certain resource, prohibition from printing to any color printers on the fifth floor, and the right to delete all the files belonging to your colleague.

As mentioned earlier, the same structures of Rei allow individual policies as well as group and role based policies to be specified making it uniform.

The languages in our bibliography did not take delegation into consideration. However, we believe that it is required in distributed, dynamic systems and should be included in the policy specifications. Rei's policy engine includes strong delegation management making it useful for dynamic systems, consisting of transient resources and users, and distributed systems, in which creating comprehensive policies may be time consuming. Rei includes two kinds of delegation and provides a standard way of controlling and propagating access rights through delegation.

9. Future Work and Conclusion

In this paper we described the specifications of our policy language, Rei, that we designed and developed for distributed, dynamic environments like pervasive systems. Rei is based on deontic concepts which allows policies of different types to be specified in terms of rights, obligations, dispensations, and prohibitions. We are currently working on identifying a deontic logic that Rei is most closely related to.

Our policy engine is under development and currently supports almost all the expressivity provided by the policy language. The engine supports action operators for Rights and the support for the other policy objects is currently under development. However, it currently does not handle all the nesting capable by speech acts like requesting a dispensation. The policy language accepts RDFS representations of entities and properties based on the Rei ontology but as mentioned earlier we are working on representing conditions in semantic languages. Though conflicts are detected and resolved at run-time, we believe pre-determining policy conflicts also has several practical uses. We are investigating a feasible solution for statically detecting conflicts in Rei policies. Our future work will also include developing DAML+OIL and/or OWL ontologies.

10. Acknowledgements

We would like to thank Dr. Bernard Burg, the Agents for Mobility Research group and Dr. Alan Karp at HP Labs, Palo Alto for their support and insightful comments. We would also like to thank Dr. Reed Letsinger. Without his invaluable suggestions and encouragement this work would not have been possible.

References

[1] The Official Bluetooth Website. http://www.bluetooth.com.

[2] H. Boley, B. Grosof, S. Tabet, and G. Wagner. RuleML. http://www.dfki.uni-kl.de/ruleml/indtd0.8.html, 2001.

[3] J. Bradshaw, A. Uszok, R.Jeffers, N.Suri, P. Hayes, M. Burstein, A. Acquisti, B. Benyo, M. Breedy, M. Carvalho, D. Diller, M. Johnson, S. Kulkarni, J. Lott, M. Sierhuis, and R. V. Hoof. Representation and Reasoning for DAML-Based Policy and Domain Services in KAoS and Nomads. *Under review*, 2002.

[4] D. Brickley and R. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Working Draft 30 April 2002, http://www.w3.org/TR/rdf-schema/, 2002.

[5] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *The Policy Workshop 2001, Bristol U.K.* Springer-Verlag, LNCS 1995, Jan 2001.

[6] M. Dean, D. Connolly, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. Web Ontology Language (OWL) Reference Version 1.0. http://www.w3.org/TR/2002/WD-owl-ref-20021112/, 2002.

[7] D. Harel. *First-Order Dynamic Logic*. New York: Springer-Verlag, 1979.

[8] Horrocks, van Harmelen, Patel-Schneider, Berners-Lee, Brickley, Connolly, Dean, Decker, Fensel, Fikes, Hayes, Heflin, Hendler, Lassila, McGuinness, and Stein. DAML+OIL Language Specifications. http://www.daml.org, 2002.

[9] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A Logical Language for Expressing Authorizations. In *IEEE Symposium on Security and Privacy. Oakland, CA*, 1997.

[10] L. Kagal. Rei : A Policy Language for the Me-Centric Project. HP Labs Technical Report, 2002.

[11] L. Kagal, T. Finin, and A. Joshi. Trust based security for pervasive computing enviroments. In *IEEE Communications, December 2001*, 2001.

[12] L. Kagal, T. Finin, and A. Joshi. Developing Secure Agent Systems Using Delegation Based Trust Management. In *Security of Mobile MultiAgent Systems (SEMAS 02) held at Autonomous Agents and MultiAgent Systems (AAMAS 02)*, 2002.

[13] L. Kagal, T. Finin, and Y. Peng. A Framework for Distributed Trust Management. In *Proceedings of IJCAI-01 Workshop on Autonomy, Delegation and Control*, 2001.

[14] L. Kagal, J. Undercoffer, F. Perich, A. Joshi, and T. Finin. A Security Architecture Based on Trust Management for Pervasive Computing Systems. In *Proceedings of Grace Hopper Celebration of Women in Computing 2002*, 2002.

[15] E. Lupu and M. Sloman. A Policy Based Role Object Model. In *Proceedings EDOC'97, IEEE Computer Society Press.*, 1997.

[16] E. C. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*, 25(6):852–869, November/December 1999.

[17] E. Mally. *The Basic Laws of Ought: Elements of the Logic of Willing*. 1926.

[18] Meyer and Roel. Deontic logic: A concise overview. In *Deontic Logic in Computer Science, pp. 3-16, Chichester: John Wiley and Sons*, 1993.

[19] J. Moffett and M. Sloman. Policy conflict analysis in distributed systems management. *Journal of Organizational Computing*, 1993.

[20] Morciniec, Salle, and Monahan. Towards Regulating Electronic Communities with Contracts. Technical report, HP Labs, 2001.

[21] L. R. Phillips and H. E. Link. The role of conversation policy in carrying out agent conversations. *Issues in Agent Communication 2000: 132-143*, 2000.

[22] M. Satyanarayanan. Pervasive Computing: Vision and Challenges. *IEEE Communications*, 2001.

[23] M. Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, **2**:333, 1994.

[24] Swedish Institute of Computer Science. SICStus Prolog. http://www.sics.se/ sicstus/, 2001.

[25] M. Taylor, W. Waung, and M. Banan. Internetwork Mobility: The CDPD Approach. Prentice Hall Professional Technical Reference, 1999.

[26] J. Undercoffer, F. Perich, A. Cedilnik, L. Kagal, A. Joshi, and T. Finin. A Secure Infrastructure for Service Discovery and Management in Pervasive Computing. *To be published in ACM MONET : The Journal of Special Issues on Mobility of Systems, Users, Data and Computing*, 2003.

[27] G. H. von Wright. A note on deontic logic and derived obligation. In *Mind*, 1956.

[28] W3C. Extensible Markup Language. http://www.w3c.org/XML/.

[29] M. Weiser. The Computer for the Twenty-First Century. *Scientific American, pp. 94-10, September 1991*, 1991.

[30] T. Woo and S. Lam. Authorizations in distributed systems: A new approach. *Journal of Computer Security*, 2(2-3):107–136, 1993.