Rei[*]: A Policy Language for the Me-Centric Project

Lalana Kagal
HP Labs, Palo Alto

September 26, 2002

Abstract

Policies guide the way entities within a domain act, by providing rules for their behavior. Most of the research in policies is within a certain application area, for example security for databases, and there are no general specifications for policies. Another problem with policies is that they require domain dependent information, forcing researchers to create policy languages that are bound to the domains for which they were developed. This prevents policy languages from being flexible and being usable across domains.

This report describes the specifications of the *Rei* policy language, which provides constructs based on deontic concepts. These constructs are extremely flexible and allow different kinds of policies to be stated. This simple policy language is not tied to any specific application and allows domain dependent information to be added without any modification. The policy engine associated with Rei accepts policies in first order logic and RDF. The report also discusses the functionality of the policy engine that interprets and reasons over Rei policies.

1 Introduction

Policies influence the behavior of entities within the policy domain. By separating policies from mechanisms, it is possible to change the behavior of entities dynamically without changing the implementation. However it is not enough to have a policy, there should also be a mechanism that interprets the policy correctly. This necessitates the presence of a policy engine that is able to understand the policy and evaluate the properties of an entity to decide how the entity should act.

The Me-Centric project is an approach to developing context aware systems, which are able to provide more relevant support to the users by understanding the contextual information of the user and the environment under consideration. The Me-Centric project views the world as divided into overlapping domains. Each domain is either a physical or a virtual domain. Associated with each domain is a set of policies that serve as guidelines to restrict the behavior of the entities within that domain. Using this combination of domains and policies, it is possible to provide optimal information and services to users, and encourage them to act appropriately. Along with users, the Me-Centric world will also be occupied by agents deployed by users for

[*] Rei, pronounced *ray*, is a Japanese word that means "universal"

automating several tasks. These agents will also require the same kind of context aware infrastructure. In this report, we use the word agents to represent entities in the system including agents, users, and services.

Though policies seem to be very attractive in dynamic systems like those handled by the Me-Centric project, it was difficult to choose the appropriate policy language. A policy language for the Me-Centric project should be able to express security policies, management policies and even conversation policies. Unfortunately most research in policies has been within a specific application domain, i.e. security, network management. Due to this, there are no general specifications for policies or mechanisms for policy verification. and the Me-Centric project would require several policy languages to describe domain policies. Secondly, to allow integration with the Semantic Web, some sort of semantic language like Resource Description Framework (RDF) [13], DAML+OIL [3], Web Ontology Language (OWL) [2] will be used in a Me-Centric system. This requires that the policy language be compatible with at least one of them and allow for translation among the other languages. Very few policy language have a semantic interface or allow policies to be described in a semantic languages. We also believe that the notion of delegation is essential to widely distributed and dynamic environments and should be captured by the policy language. However this is something most policy languages tend to overlook. Even after extensive research, we were unable to find a single policy language capable of such expressibility and that met all the requirements of the Me-Centric project.

This report describes the specification of a general policy language that we developed, and its associated policy engine that interprets policies written in this language. Our language is modeled on deontic logic and includes notions of rights, prohibitions, obligations and dispensations. We believe that most policies can be expressed as what an entity can do and what it should 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. The language is described in first order logic that allows for easy translation from/to RDF, DAML+OIL and OWL. This report includes a discussion of the RDF schemas that can be used to describe policies.

Our policy language is named *Rei*, which in Japanese means "universal" or "essence", 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 .

2 Related Work

According to Sloman, policies define a relationship between subjects and targets [15]. 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 interprete policy information. He classifies policies into authorization and obligation and states that there are two kinds of constraints on policies; temporal, and parameter value. 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 [1] 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. Separating policy from mechanism allows behavior of the system to be changed by changing the policies without changing the implementation. 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 Rei allows this to be done naturally. Rei allows actions and policy objects to be defined separately 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. It includes a very simple notion of delegation, refrain policies that are negative authorization enforced by subjects instead of targets, obligation policies that are event triggered. 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.

We found the work by Lupu et al in policy conflicts very interesting as we believe conflict resolution is an important section of policy specifications. Lupu classifies conflicts into modality conflicts and application specific conflicts [8]. 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, and finding the distance between the policy and the managed object. Rei uses Lupu's definition of modality conflicts but does not use the suggested mechanisms. Rei provides a method of specifying which modality holds precedence for sets of agents and actions grouped by certain conditions.

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 [5, 17]. Woo and Lam [17] 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. In "A Logical Language for Expressing Authorizations", Jajodia et al. describe the specifications of a language based on logic that tries to support different access control policies [5]. The Authorization Specification Language (ASL) allows users to not only specify authorization policies, but also specify the way the decisions over these policies are made. Jajodia et al. classify policies as closed, where all positive authorizations have to be specified, and open, where all negative authorizations have to be specified, and show how ASL supports them both. 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. It is based on ten predicates: *cando* (authorization), *dercando* (derived authorizations), *do* (conflict resolution), *grant* (access control), *done* (executed accesses), *active* (current roles), *dirin* (direct membership), *in* (indirect membership), *typeof* (grouping relationship), and *error* (error in specifications). An authorization rule is a cando based on a set of conditions made of dirin or typeof. A derivation rule is made of a dercando and a set of conditions on cando, dercando, done, in and dirin. A resolution rule is a rule made containing do as its head and conditions on cando, dercando, in, dirin, done or typeof as the body of the rule. An access control rule has grant as its head and conditions in cando, dercando, done, do, in, dirin or typeof as its body. An integrity rule is

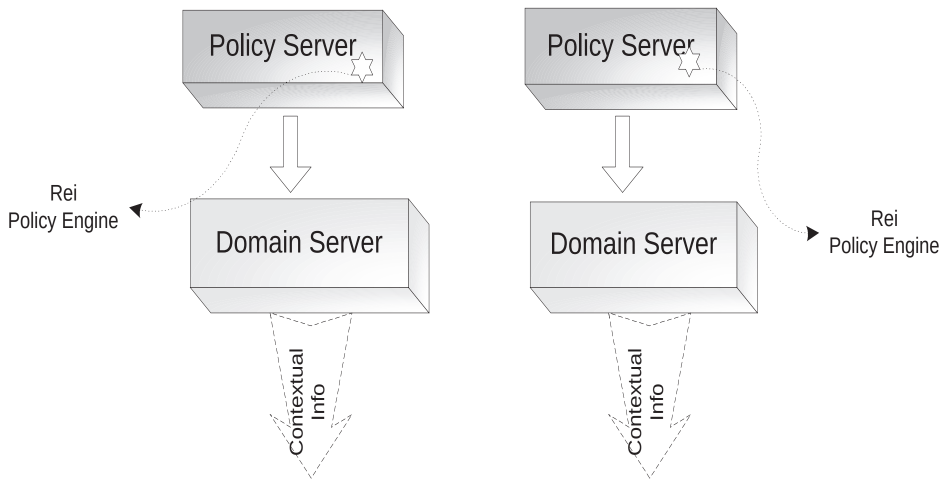

Figure 1: Me-Centric Project is composed of Domain Servers and their associated Policy Servers.

a specification of erroneous conditions. Using these rules, it is possible to describe different access control policies. This language, though a step in the right direction, is complicated, because it consists of interdependent rules that the user has to fully understand, and is not as expressive as Rei. It 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, but ASL expects these policies to described using three predicates, active, dirin, and in. 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 as it allows conflict resolution to be specified for sets of actions and agents. For example, in Rei you can specify that for all possible actions on colors printers in a certain lab, permissions should hold precedence over prohibitions.

3 The Role of Policies in the Me-Centric Project

The Me-Centric project is aimed at providing personalization to users using contextual information. It is currently composed of two main components, Domain Server and the Policy Server, as illustrated in Fig. 1. The Domain Server interacts with sensors and other contextual information sources, to decide domain memberships of users. The domain specifications include the location of the policies. The Policy Server gets the domain memberships and retrieves these policies. It then reasons over the policies and conditions associated with users to decide what the rights, prohibition, dispensations and obligations of the users are. These rules guide the behavior of the users, forcing the user to act in a certain way depending on the domains the user is in.

Policy makers create policies for domains that consist of several resources. The policies are specified in Rei and contain actions that can be performed on the domain's resources, properties of agents/users that should be allowed to use these resources and the policy objects themselves. Rei's policy engine resides within the Policy Server. The Policy Server retrieves policies of different domains and feeds them to the policy engine, after mapping the domain specific names to unique names within the policy engine. The Policy Server is also responsible for capturing speech acts that occur and add them to the policy engine's knowledge base. The policy engine will only reason about policies that it contains and speech acts that it has knowledge of. Every Domain Server will be associated with one or more Policy Servers that will reason over the domains that the Domain Server is interested. The Me-Centric system will be aware of which Policy Server to query, based on the domains of interest.

4 Design of the Policy Language

According to our bibliography, we found most policy languages to be complicated and unintuitive. The languages we studied introduced several constructs for different specifications, which we believe made the specifications difficult to understand. Our design choice was to keep our policy language as simple as possible. Our policy language includes few constructs for describing rights, prohibitions, obligations and dispensations. However these constructs are general enough to cover a range of policies by allowing the behavior of an entity to be modified (Please refer to Section 9 for more information) . We believe that policies specified in a language based on First Order Logic (FOL) are easy for users to understand and describe. There exists a simple translation between FOL and RDF and DAML+OIL, enabling the FOL system to use a semantic language for representing the knowledge. Our policy language is based on First Order Logic and currently includes an RDF interface based on an ontology. We believe that other interfaces can be developed by extending the same ontology.

The Rei policy language has some domain independent ontologies but will also require specific domain ontologies. The former includes concepts for permissions, obligations, actions, speech acts, operators etc. The latter is a set of ontologies, shared by the agents in the system, which defines domain classes (person, file, deleteAFile, readBook) etc. and properties associated with the classes (age, num-pages, email).

Rei allows actions and conditions to be external to the system. Though it provides a way to specify them, it assumes that the *meaning* of actions or conditions are domain dependent and that their complete processing is outside the policy. Using these actions and conditions, a policy maker is able to create policy objects. There are four kinds of policy objects: rights, obligations, prohibitions, and dispensations. In order to associate these policy objects with users, the policy maker uses the has predicate, creating rights, obligations etc. for the users. 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, it expects all relevant speech acts about the resources and users in its policies to be input to it. The Rei policy language contains meta-policy specifications for conflict resolution. Associated with the policy language, is a policy engine that interprets and reasons over the policies and speech acts to make decisions about users rights and obligations. We envision that Rei will be used as a querying engine by the Policy Server. The Policy Server will retrieve policies associated with domains, map the domain specific names into unique names for the Policy Server and insert the policies into Rei. Every time an agent requests a certain action, the Rei policy engine will be queried. Rei will check if the requester has the right, by checking its policy objects, or if the right has been delegated to the requester by an entity with the right to delegate. It will also check for prohibitions and revocations. It will use the meta-policy associated with the agent and the requested action to resolve any conflicts. If the agent has the right, Rei will inform the owner of the action and allow the owner to interpret the action and handle its execution.

The following subsections describe the FOL specifications for actions, constraints, policy objects, speech acts and meta policies.

4.1 Policy Objects

The core of the policy language are the constructs that describe the deontic concepts of rights, prohibitions, obligations, and dispensations[1]. These components (@) are represented as

@(Action, Conditions)

where, *Action* is a domain dependent action and *Conditions* are domain dependent restrictions on the actor and environment.

In order to associate a policy object with an agent, we use the *has* construct. *has* represents the possession of a policy object.

has(Subject, Policy Object)

where, *Subject* can either be a URI identifying an agent or a variable, allowing all agents who satisfy the conditions to be associated with the policy object to possess the policy object. 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 application 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 [7, 1].

- Rights (∼O∼) are permissions that an agent has. The possession of a right allows the agent to perform the associated action. However the agent should also have the ability, in terms of resources, to perform the action.

  An agent, *AgentA*, can perform an action, *ActionB* iff at least one of the following conditions are true

  – *has(AgentA, right(ActionB, Conditions))* and *AgentA* satisfies *Conditions*
  – *has(Variable*[2]*, right(ActionB, Conditions))*, *AgentA* binds to *Variable* and satisfies *Conditions*

  Example 1. A rule that states that all employees of HP labs can perform the action printAction1, it is represented as

  has(Variable, right(printAction1, [employee(X, HPLabs]))

  In this policy rule, *printAction1* is an action and *employee* is a domain dependent condition, which is foreign to the system. (Please refer to Section 4.3 for more information on how to create different constraints and Section 4.2 for more information on actions.)

- Prohibitions (O∼) are negative authorizations and if an agent has a prohibition, it cannot perform the action.

  An agent, *AgentA*, is prohibited from performing *ActionB* iff at least one of the following conditions is true

[1]The structure of policy objects was a result of discussion between the author and Dr. Reed Letsinger

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

- *has(AgentA, prohibition(ActionB, Conditions))* and *AgentA* satisfies *Conditions*
- *has(Variable, prohibition(ActionB, Conditions))* and *AgentA* satisfies *Conditions*

Example 2. In Rei, a rule that states that no students can use the faculty printer is specified as

has(Variable, prohibition(useFacultyPrinter, [student(X)]))

useFacultyPrinter is an action and *student* is a condition that an entity must satisfy in order to possess the prohibition.

- Obligations (O) are actions that an agent must perform and are usually event driven, i.e. when a certain set of conditions are true.

  An agent, *AgentA*, is obliged to perform *ActionB* iff one of the following conditions are true

  - *has(AgentA, obligation(ActionB, Conditions))* and *AgentA* satisfies *Conditions*
  - *has(Variable, obligation(ActionB, Conditions))* and *AgentA* satisfies *Conditions*

Example 3. A policy rule to specify that all employees should display their badges while at work, the obligation is represented as

has(Variable, obliged(displayBadge, [employee(X), atWork(X)]))

displayBadge is an action and *atWork(X)* is a a domain dependent condition.

- Dispensations ($\sim$O) are actions that an agent is no longer obliged to perform. They are used to cancel the associated obligation.

  An agent, *AgentA*, is no longer obliged to perform an action, *ActionB* iff

  - *has(AgentA, obligation(ActionB, OConditions))* and if *AgentA* satisfies *OConditions*
  - *has(AgentA, dispensation(ActionB, DConditions))* and if *AgentA* satisfies *DConditions*.

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

has(john, dispensation(payAlimonyJoan, [remarried(joan)]))

4.2 Action Specifications

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.

Actions are represented as a tuple

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

where, *ActionName* is identifier of the action, *TargetObjects* are the 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.

Along with individual actions, this representation enables sets of actions to be described that are grouped by target objects.

Example 5. John has the right to read papers at work, as long as they are technical papers.

- The action of reading technical papers
  action(readingTechPapers, X, [technical-paper(X), not-read(X)], [assert(read(X))])
- John has the right to read technical papers at work
  has(john, right(action(readingTechPapers, X, [technical-paper(X), not-read(X)], [assert(read(X))]), [atWork(john)]))

4.2.1 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, which justifies the additional complexity.

The policy language includes four action operators that allow various kinds of complex actions to be specified.

- 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.
- Iteration : If A is an action, *iteration(A)* denotes that A can be repeated
- 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 *fax* once. He only has the right while he satisfies the associated conditions.
has(john, right(nond(seq(printBW, iteration(printColor)), once(fax)),[lab-member(X,ai)])).

The policy engine currently only supports action operators for *rights* due to time limitations. Support for the other policy objects is currently under development.

4.3 Constraint Specifications

As conditions are application dependent, they cannot be pre-specified and are handled as placeholders by this system. This gives the policy maker a lot of flexibility and control at the same time. The policy maker can define his/her own domain specific conditions for the different policy rules.

As time constraints are also dependent on the application, they are also left as templates. The user can define his/her own time constraints, like range, morning, 9-5, start-time etc. The time constraints are handled as part of the conditions associated with policy objects. Along with this, each policy rule is valid for a certain time and the policy maker can change this time validity by creating new time conditions or modifying the macro (calculateTTL) of the policy engine.

4.4 Creating a New Constraint

In order to evaluate the condition correctly a certain clause needs to be provided to the policy engine. The *newConstraint* clause allows the user to describe the parameters of the condition and specify positions that the associated agent can be bound to. All policy objects only have one value, *Agent*, that can be used for binding with variables in the conditions. The engine should know where these agent based variables are, for proper evaluation of the condition. The parameters are described by a string containing the name of the field, followed by a ':', and then the type of the field. Though this clause is not very useful for conditions in prolog, it is helpful while executing conditions in RDF.

Example 7. The following example illustrates how *newConstraint* is used. Consider a condition, "employee X of Y", where only X can be bound to *Agent*, represented as employee(X,Y).
newConstraint(employee, [employee:String, company:String], [1])

Example 8. The it newConstraint clause also works for conditions which are not related to agents. For example *time-now* represented as time-now(X), where X is the current time.
newConstraint(time-now, [time:Time], [])

Currently the policy engine does not check the type of the parameters and only checks the property name while reading policies in RDF (Please refer to 7.2 for more information on the RDF Interface) .

4.5 Adding Instances of Constraints

In order to add constraint instances in prolog, the user can simply use the *assert* clause. *assert* is a builtin clause that allows the information to be added to the existing knowledge base.

Example 9. To add instances of *employee* stating John is an employee of HP labs and Marty is an employee of Xerox, the user would add to following to the current knowledge base
assert(employee(john, hpLabs))
assert(employee(marty, xerox))

The policy engine also accepts constraint instances in RDF. It contains a parser that reads the RDF class specification and converts it into a clause *addConstraint*. This clause contains the name of the class and the parameters (not necessarily in any order). When this clause is used, it causes a new constraint instance to be asserted which conforms to the values in the associated *newConstraint* clause.

4.5.1 Constraint Operators

Constraints can be joined together by certain boolean operators to create complex contraints.

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

  For example, *and(employee(X, hpLabs), 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 false.

Example 10. As an example, consider a complex constraint made up of application dependent conditions, which will be true if the agent is a lab-member of ai or if the agent is not an employee of hpLabs.
or(lab-member(X, ai), not(employee(X, hpLabs)))

4.6 Speech Acts

Along with representing policy objects, actions and conditions, the policy language also describes speech acts. Speech acts are used to model interactions between different entities in the Me-Centric system. These speech acts are based on the FIPA specifications [4]. Agents can only use a certain speech if they have the right to it. John may have the right to send a *request* but not a *revoke*. The policy engine does not capture the speech acts while they occur and expects all relevant speech acts, regarding resources and users included in its policies, to be explicitly sent to it for evaluation.

The policy language currently includes specifications for four speech acts that affect the policy objects of the communicating entities; delegation, request, cancel, and revocation.

- Delegation : A delegation allows an agent to transfer its right to another agent or group of agents (by not specifying a Receiver). A delegation, if valid, causes a right to be created. Only an agent with the right to delegate can make a valid delegation. (Please refer to section 5 for more details about delegation management.)

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

  Example 11. Assuming that Mark has the right to delegate, he delegates to Joan the right to use his car as long as she fills it up.
  delegateSpeechAct(mark, joan,
  right(useMarkCar,[assert(has(joan, obligation(fillGasMarkCar,[])))]))
  action(useMarkCar, [mark-car], [], [assert(used(joan, mark-car))])
  action(fillGasMarkCar, [mark-car], [used(X, mark-car)], [])
  newConstraint(used, [driver:String, car:String], [1])

- Request : There are two kinds of requests; a request for action and a request for a right. The former causes an obligation, if the receiver decides to accept it. A request for a right, if valid and accepted, causes the receiver to send a delegation to the sender. A request is only valid, if the sender has the right to use *request*.

- *requestSpeechAct(Sender, Receiver, Action)* $\longrightarrow$ disagree
  requestSpeechAct(Sender, Receiver, Action) $\longrightarrow$ *has(Receiver, obligation(Action, Condition))*
- *request(Sender, Receiver, right(Action, Condition))* $\longrightarrow$ disagree
  request(Sender, Receiver, right(Action, Condition)) $\longrightarrow$
  delegation(Receiver, Sender, right(Action, XCondition))

- Revoke : Revocation is the removal of a right and acts as a prohibition. An agent can only 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)) $\longrightarrow$ *prohibition(Receiver, right(Action, Condition))*

- Cancel : An agent can cancel any request it has sent, and this causes the obligation or delegation formed by the request to be canceled.

  - *cancelSpeechAct(Sender, Receiver, Action)* $\longrightarrow$ *has(Receiver, dispensation(Action, Condition))*
  - *cancelSpeechAct(Sender, Receiver, right(Action, X))* $\longrightarrow$ *remove has(Sender ,right(Action, Condition))*

4.7 Meta Policies

There is generally more than one applicable policy in a domain. This may lead to policy conflicts. For example, one policy could give Mary the right to print and the other policy could prohibit Mary from printing. Meta-policies are policies about how policies are interpreted and how conflicts are resolved statically. Conflicts only occur if the policies are about the same action, on the same target but the modalities (right/prohibition, obligation/dispensation) are different.

Meta policies in our system regulate conflicting policies statically in two ways; by specifying priorities and precedence relations [8].

4.7.1 Priorities

Every policy rule can be associated with an identifier. Using a special construct, *overrides*, the priorities between any two rules can be set. 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)*, the conflict between the two rules can be resolved as A1 will be given priority over B1. In order to re-order the rules that were affected by the *overrides* clause, the policy maker must execute a *orderRules* function. This is not done automatically because it is a batch process, and several priorities can be specified before the affected rules are re-ordered according to their new priorities.

Example 12. The example described above can be represented as

- *A rule named a1*
  *a1**has(mary, right(print, [time-now(12.00)]))*

- *A rule named b1*
  *b1**has(mary, prohibition(print, [lab-member(X, ai)]))*
- *overrides(a1, b1)*
- *orderRules*

4.7.2 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 agents satisfying the associated conditions. The constructs to be used are *metaRuleAction* and *metaRuleAgent*.

Example 13. Consider a meta policy specifies that negative modality holds precedence for all employees of Xerox Labs, the rule will be
metaRuleAgent([employee(X, xeroxLabs)], negative-modality)

As with policies the conditions are application dependent and the format is not forced by the policy engine.

There exists a partial ordering among the meta policies as well. The meta rules associated with actions have the highest priority, and are followed by meta rules about agents. If there are no rules associated with the action or the agent, then the default meta rule is considered. The policy language allows the policy maker to decide what the default policy is for all the policies in a domain, whether negative modality holds precedence for all policies, or positive modality holds by using the *meta-rule* clause. If negative modality holds, prohibitions hold over rights and dispensations are stronger than obligations, for positive modality it is the reverse. However the meta-rules themselves can be prioritized as required by using the *overrides* clause.

The three kinds of meta rules that are possible;
metaRuleAction([action(..), positive-modality/negative-modality)
metaRuleAgent([Condition], positive-modality/negative-modality)
metaRule(positive-modality/negative-modality)

5 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 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 it allows the rights on the printers to propagate.

The Rei policy language recognizes three types of inter-related 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 possess the right to delegation the execution of an action, it can delegate to other agents the right to perform the action, but it cannot perform the action itself.

  NOT has(Agent, right(Action))

  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 agents the right to further delegate the right to perform the action and delegate this right. 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 delegation itself.

  NOT has(Agent, right(Action))

  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 while-delegations. The delegator conditions for when-delegations are only checked at the time of the delegation (Please refer to next subsection for more details about when-delegation). If any one agent fails any required delegator condition, the delegations from that point on are invalid.

5.1 Delegation Types

There are generally 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 which gives 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 known as the default delegation type.

Example 14. The following represents the example described above

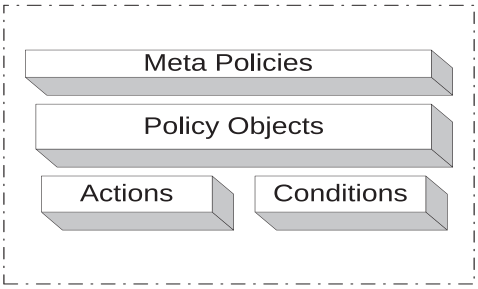

Figure 2: Structure of a Rei policy

- 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)*

6 Structure of Policies

Policies allow the behavior of agents to change dynamically without changing the implementation of. These policies are made of rules that change the rights and obligations of agents, thereby deciding what the agents should and can do next. Rei allows policies to be defined as a set of policy objects specifying the rights, obligations, prohibitions and dispensations of agents in the policy domain. Along with these policy objects, the policy maker can include meta-policies for resolving conflicts among policies. If there is no meta-policy, a default meta policy is assumed, in which negative modality holds precedence over positive modality for modality conflicts. As policy objects are composed of actions and constraints, the policy could also include action and constrain specifications and constraint instances. Figure 2 depicts the structure of a Rei policy.

7 Policy Engine

As mentioned earlier, along with specifications for a general policy language, we also developed a policy engine to interpret policies and provide replies to queries about specific entities. The policy engine is a Java class (PolicyEngineUI.java) which has a command line interface. It enables users to load policies, and speech acts in prolog [16] and RDFS [13] and allows users to query the knowledge base. As the policy engine has a Java wrapper, it is possible to use its functionality without going through the interface.

The command line interface has the following functionality

- EXIT

  This causes the command line interface to end. Shortcut : e

- HELP or h

  This command prints out all the allowable commands with their appropriate usage. Shortcut : h

- LOADP

  Loadp loads the file that follows the command. It assumes that this file is in prolog and searches for policy components, speech acts, constraint values or meta policies. Shortcut : lp

- LOADR

  A user can load RDF policies using this command. The associated function reads through the RDF and locates appropriate classes, builds prolog clauses and inserts them into the prolog knowledge base. Shortcut : lr

- SAVE

  The user can save the current state of the policy engine. Shortcut : s

- RESTORE

  The user can restore the state of the policy engine from a file containing a saved state. Shortcut : r

- QUERY

  The user can query the knowledge base using prolog and ask the following questions :

  - canPerformAction(Agent, Action)
    This will return true if Agent has the right to Action. However if either Agent or Action are variables[3] , the query will will return appropriate values for Agent and Action.
  - doPerformAction(Agent, Action)
    This will check if Agent can perform Action, and then go ahead with the execution. Though executing the actual action will be outside prolog, the policy engine goes through with the effects of the action.
  - getObligations(Agent, Actions)
    It returns the current obligations of Agent.

  This command directly queries the knowledge base, so any sort of prolog query can be made as well. For example,

  action(Action, printerHP, PreCond, Effect)

  This query is asking for an action specification that has *printerHP* as a target object.

  Shortcut : q

- ASSERT

  Instead of loading an entire file, the user can use assert to insert a single policy/meta-policy rule, constraint, speech act etc. Shortcut : a

[3] In prolog strings starting with uppercase letters are assumed to be variables

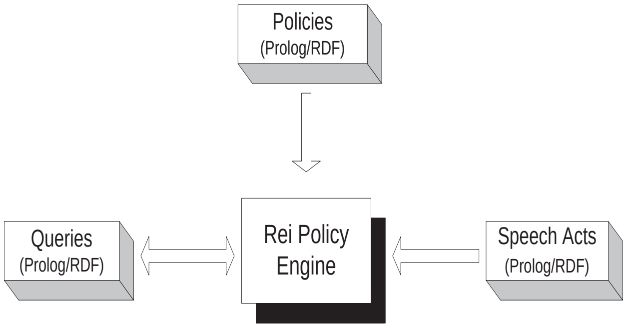

Figure 3: Rei Policy Engine

7.1 Prolog Interface

Policies can be specified directly in prolog if required. Policies are composed of actions, constraint values, and policy objects. Speech acts can also be represented in prolog and added to the policy engine's knowledge base. Queries can be made to policy engine about the rights, obligations, prohibitions and dispensations of entities in the policy domain. (Please refer to earlier section about queries in prolog)

The clauses accepted by the prolog interface have already been discussed while describing the policy language specifications. This section will briefly mention them and how they are used.

- Constraints : There is only clause to be used while specifying policies in prolog, *new-Predicate*, which includes the name of the constraint, the parameters, and the positions of variables that can be bound by the agent.

  newPredicate(Name, [ParName1:Type1, ParName2:Type2 ....], [AgentPositionList])

  Instances of the constraint can be added by using asserting them as *Name(Par1, Par2 ....)* or using another clause *addPredicate*.

  addPredicate(Name, [ParName2:Par2Value, ParName1:Par1Value..])

  While using *addPredicate* the parameters need not be in order.

- Actions : The clause to add actions is *action* and has five parameters, name of action, a list of objects it acts on, a list of conditions that must be true before it can be executed and the effects of executing the action.

  action(ActionName, [TargetObjects], [Pre-Conditions], [Effects])

- Policy Objects : There are four policy objects, all of which have the same structure.

  right(Action, [Condition])

  prohibition(Action, [Condition])

  obligation(Action, [Condition])

  dispensation(Action, [Condition])

  To associate a policy object with an agent or a group of agents, the *has* clause is used.

  has(Agent, PolicyObject)

  During evaluation of the *has* clause, the agent is bound with all possible conditions in the list of conditions associated with the policy object.

- Speech Acts : Our policy language models four speech acts that cause the policy objects associated with the communicating agents to dynamically change.
  - Delegate : This speech acts allows a right to be transfered from one agent to another agent or group of agents satisfying the delegation conditions.
    delegateSpeechAct(Sender, Receiver, right(Action, Condition))
  - Request : An agent can request another agent to perform a certain action or for a right.
    requestSpeechAct(Sender, Receiver, right(Action, Condition))
    requestSpeechAct(Sender, Receiver, Action)
  - Revoke : An agent can revoke a right of another agent or group of agents. *revoke-SpeechAct(Sender, Receiver, right(Action, Condition))*
  - Cancel : An agent can cancel an action or a right it previously requested for.
    cancelSpeechAct(Sender, Receiver, Action) cancelSpeechAct(Sender, Receiver, right(Action, Condition))
- Meta policies There are two kinds of meta policies allowed in Rei; one for specifying priorities between rules and the other to represent which modality holds precedence.

  Rules are named by ıRuleName**Rule. The policy maker can use the *overrides(RuleName1, RuleName2)* clauses to specify priorities. After all the priorities have been specified, the *orderRules* rule should be executed.

  The modality can be set for a set of actions or agents.

  metaRuleAction([action(..), positive-modality/negative-modality)

  metaRuleAgent([Condition], positive-modality/negative-modality)

  The policy maker can decide the default modality for all the policies, by using the *metaRule* clause. *metaRule(positive-modality/negative-modality)*
- Queries : Any query valid under prolog is acceptable. The list of queries supplied by the Rei policy engine is described in the section above.

7.2 RDF Interface

We have defined an ontology in RDF, representing domain independent information, that users can extend in order to create policies in RDF. The ontology is portrayed in the Figure 4. There are three main classes under the root of the ontology; *State*, *Entity* and *Action*. *Entity* is divided into *Agent* and *Object*. Under *Agent*, all users and agents are specified, whereas resources are described under *Object*. *Action* has two subclasses; *Speech Acts* and *Domain Actions*. *Speech Acts* are the imperatives that change the policy objects, and *Domain Actions* are actions on resources in the domain. The *Domain Action* class has the same parameters as the action tuple described in Section 4.2. *State* has four subclasses; *Meta-Rules*, *Policy Objects*, *Conditions* and *Has*. Any RDF policy that extends the appropriate classes in our ontology can be loaded into the policy engine. For example, in order to create a new condition or add new instances of a condition, the condition should be made a subclass of *Condition*.The policy engine parses these conditions and asserts the required clauses. Our ontology allows automated translation from databases and other sources of information to policies that the Rei policy engine can understand. The policy engine extracts information about the schema classes and creates appropriate prolog clauses that are inserted into the knowledge base.

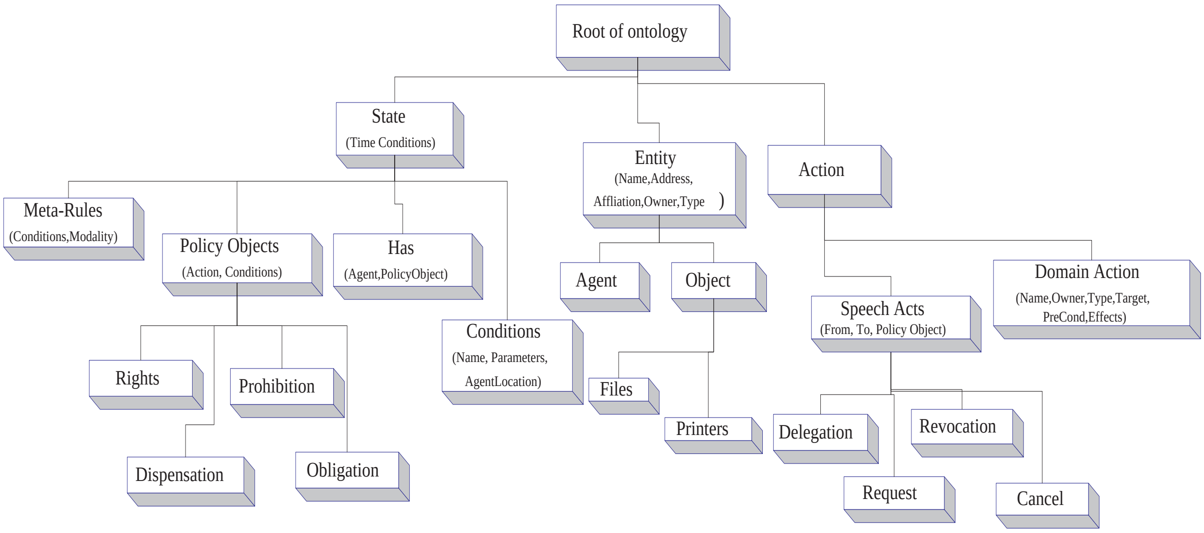

Figure 4: The RDF ontology for Rei

Example 15. The RDF representation for a policy object that specifies that John has the right to print would be as follows

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xmlns:policy="http://www.csee.umbc.edu/˜lkagal1/policy/policy-schema.rdf#"
   xmlns:policyobjects="http://www.csee.umbc.edu/˜lkagal1/policy/policy-objects-schema.rdf#"
   xmlns:condition="http://www.csee.umbc.edu/˜lkagal1/policy/some-conditions.rdf#"
   xmlns="http://www.csee.umbc.edu/˜lkagal1/policy/example1.rdf#">


  <policyobjects:Has rdf:about="http://csee.umbc.edu/˜lkagal1/policy/example1.rdf#hasright1">
     <policyobjects:Actor>
       <policy:Agent rdf:about="http://csee.umbc.edu/˜lkagal1/policy/some-conditions.rdf#john" />
     </policyobjects:Actor>
     <policyobjects:Ability rdf:resource="http://csee.umbc.edu/˜lkagal1/policy/example1.rdf#printright"/>
   </policyobjects:Has>


  <policyobjects:Right rdf:about="http://csee.umbc.edu/˜lkagal1/policy/example1.rdf#printright">
     <policyobjects:PolicyAction  rdf:resource="http://csee.umbc.edu/˜lkagal1/policy/example1.rdf#printing" />
     <policyobjects:PolicyCondition rdf:resource="http://csee.umbc.edu/˜lkagal1/policy/example1.rdf#employee1" />
  </policyobjects:Right>

  <policy:DomainAction rdf:about="http://csee.umbc.edu/˜lkagal1/policy/example1.rdf#printing">
     <policy:ActionName>print</policy:ActionName>
     <policy:Owner>john</policy:Owner>
     <policy:Type>read</policy:Type>
     <policy:TargetObjects rdf:resource="http://csee.umbc.edu/˜lkagal1/policy/example1.rdf#printerHP" />
     <policy:PreConditions rdf:resource="http://csee.umbc.edu/˜lkagal1/policy/example1.rdf#employee1"/>
     <policy:Effects rdf:resource="http://csee.umbc.edu/˜lkagal1/policy/example1.rdf#employee1"/>
   </policy:DomainAction>


   <condition:employee rdf:about="http://csee.umbc.edu/˜lkagal1/policy/example1.rdf#employee1">
      <condition:company>
        <condition:companyclass rdf:about="http://csee.umbc.edu/˜lkagal1/policy/example1.rdf#hpLabs"/>
      </condition:company>
   </condition:employee>


   <condition:group_member rdf:about="http://csee.umbc.edu/˜lkagal1/policy/example1.rdf#group_member1">
      <condition:group>
         <condition:groupclass rdf:about="http://csee.umbc.edu/˜lkagal1/policy/example1.rdf#ai"/>
      </condition:group>
   </condition:group_member>


   <rdfs:Class rdf:about="http://www.csee.umbc.edu/˜lkagal1/policy/some-conditions.rdf#employee">
<rdfs:comment>Employee schema [Agent, Organization]</rdfs:comment>
<rdfs:subClassOf
```

```
          rdf:resource="http://www.csee.umbc.edu/~lkagal1/policy/policy-schema.rdf#Condition"/>
   </rdfs:Class>

   <rdf:Property rdf:about="http://www.csee.umbc.edu/~lkagal1/policy/some-conditions.rdf#agent">
      <rdfs:comment>Agent field associated with employee class</rdfs:comment>
      <rdfs:domain rdf:resource="http://www.csee.umbc.edu/~lkagal1/policy/some-conditions.rdf#employee"/>
      <rdfs:range rdf:resource="http://www.csee.umbc.edu/~lkagal1/policy/policy-schema.rdf#Agent"/>
   </rdf:Property>


   <rdf:Property rdf:about="http://www.csee.umbc.edu/~lkagal1/policy/some-conditions.rdf#company">
      <rdfs:comment>Company field associated with employee class</rdfs:comment>
      <rdfs:domain rdf:resource="http://www.csee.umbc.edu/~lkagal1/policy/some-conditions.rdf#employee"/>
      <rdfs:range rdf:resource="http://www.csee.umbc.edu/~lkagal1/policy/some-conditions.rdf#companyclass"/>
   </rdf:Property>



   <rdfs:Class rdf:about="http://www.csee.umbc.edu/~lkagal1/policy/some-conditions.rdf#group_member">
<rdfs:comment>Group_Member schema [Agent, Group]</rdfs:comment>
<rdfs:subClassOf
          rdf:resource="http://www.csee.umbc.edu/~lkagal1/policy/policy-schema.rdf#Condition"/>
   </rdfs:Class>

   <rdf:Property rdf:about="http://www.csee.umbc.edu/~lkagal1/policy/some-conditions.rdf#agent">
      <rdfs:comment>Agent field associated with employee class</rdfs:comment>
      <rdfs:domain rdf:resource="http://www.csee.umbc.edu/~lkagal1/policy/some-conditions.rdf#group_member"/>
      <rdfs:range rdf:resource="http://www.csee.umbc.edu/~lkagal1/policy/policy-schema.rdf#Agent"/>
   </rdf:Property>


   <rdf:Property rdf:about="http://www.csee.umbc.edu/~lkagal1/policy/some-conditions.rdf#group">
      <rdfs:comment>Company field associated with employee class</rdfs:comment>
      <rdfs:domain rdf:resource="http://www.csee.umbc.edu/~lkagal1/policy/some-conditions.rdf#group_member"/>
      <rdfs:range rdf:resource="http://www.csee.umbc.edu/~lkagal1/policy/some-conditions.rdf#groupclass"/>
   </rdf:Property>

    <rdfs:Class rdf:about="http://www.csee.umbc.edu/~lkagal1/policy/some-conditions.rdf#companyclass">
     <rdfs:comment>Company</rdfs:comment>
     <rdfs:subClassOf
          rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Resource"/>
   </rdfs:Class>

    <rdfs:Class rdf:about="http://www.csee.umbc.edu/~lkagal1/policy/some-conditions.rdf#groupclass">
     <rdfs:comment>Company</rdfs:comment>
     <rdfs:subClassOf
          rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Resource"/>
   </rdfs:Class>


</rdf:RDF>
```

8 Implementation Details

The core of the policy engine has been implemented in SICStus Prolog [16], because of its powerful reasoning capabilities. The policy engine has a Java [11] wrapper and the parsing of RDF is done by Jena [10].

9 Contributions

Rei is a flexible and easy-to-use policy language. It includes few constructs, based on deontic concepts, that are powerful because they 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. These map directly into obligations and rights in Rei. Conversation policies are very important in semi automatic environments [12]. 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), the 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 both consistent and complete [5]. 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, the access being allowed or prohibited.

Rei is composed of domain dependent information and domain independent information. Rei provides specifications for representing domain independent information (constraints, actions etc.) allowing the policy makers to use 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. Though the policy specifications allow these kinds of policy objects, based on properties of actions, the policy engine does not support this functionality as yet.

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.

10 Future Directions

The policy engine is currently under development. There are certain sections that are not complete; supporting action specifications for all policy objects, providing explicit support for conversation policies, and providing greater conflict resolution. Certain sections of the RDF interface, though completely designed, are incomplete.

There are several issues that we would also like to explore further. Though we have obligation specification, we believe that another representation is required;

has(Agent, obligation(Action, OnConditions, MetEffects, NotMetEffects))

The obligation is triggered when *Agent* satisfies OnConditions. The agent 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). We have also not considered the delegation of obligations, though the structure of the policy supports it. We believe that while delegating an obligation, two additional obligations are created [14]. The delegator is now obliged to inform the agent he/she was obliged to earlier that he/she is not responsible any longer and provide the name of the agent who is now obliged. The delegator is also obliged to check if the delegatee fulfills the obligation.

Though the policy engine interacts with RDF, we would like to study the feasibility of moving to DAML+OIL [3] or OWL [2].

We would also like to extend Rei to include specifications for distributed trust management [9, 6], in which trust is relative to every entity.

11 Summary

In this report we described the constructs that Rei, our policy language, provides for specifying policies of most types. Rei includes few constructs, based on deontic logic, that allow security policies, management policies and even conversation policies to be described in terms of rights, obligations, dispensations, and prohibitions. We specified the functionality of the Rei policy engine that interprets and reasons over Rei policies. This policy engine accepts policies in first order logic and RDF. We showed through examples how the policy engine can be used.

We believe that Rei provides a good set of specifications for describing policies. Though there are several improvements that can be made, we believe that Rei is the first step towards realizing our vision of a general policy language.

References

[1] 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.

[2] Dean, Connolly, Van Harmelen, Hendler, Horrocks, McGuinness, Patel-Schneider, and Stein.

[3] Ian Horrocks et al. DAML+OIL Language Specifications. http://www.daml.org/2000/12/daml+oil-index, 2001.

[4] Foundation for Intelligent Physical Agents. FIPA Specification, 2001.

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

[6] Lalana Kagal, Tim Finin, and Anupam 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.

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

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

[9] M.Blaze, J.Feigenbaum, and J.Lacy. Decentralized Trust Management. *Proceedings of IEEE Conference on Privacy and Security*, 1996.

[10] Brian McBride. Jena: Implementing the rdf model and syntax specification. http://www-uk.hpl.hp.com/people/bwm/papers/20001221-paper/, 2001.

[11] Sun Microsystems. Java technology. http://java.sun.com/.

[12] Laurence R. Phillips and Hamilton E. Link. The role of conversation policy in carrying out agent conversations.

[13] RDF. Resource Description Framework (RDF) Schema Specification, 1999.

[14] Andreas Schaad and Jonathan D. Moffett. Delegation of obligations. In *Third International Workshop on Policies for Distributed Systems and Networks, 5-7 June 2002, Monterey, CA*, 2002.

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

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

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