A Domain Specific Language for Usage Management

Christopher C. Lamb
University of New Mexico
Department of Electrical and Computer Engineering
Albuquerque, NM 87131-0001
cclamb@ece.unm.edu

Pramod A. Jamkhedkar
University of New Mexico
Department of Electrical and Computer Engineering
Albuquerque, NM 87131-0001
pramod54@ece.unm.edu

Viswanath Nandina
University of New Mexico
Department of Electrical and Computer Engineering
Albuquerque, NM 87131-0001
vishu@ece.unm.edu

Mathew P. Bohnsack
University of New Mexico
Department of Electrical and Computer Engineering
Albuquerque, NM 87131-0001
mbohnsack@ece.unm.edu

Gregory L. Heileman
University of New Mexico
Department of Electrical and Computer Engineering
Albuquerque, NM 87131-0001
heileman@ece.unm.edu

ABSTRACT

In the course of this paper we will develop a domain specific language (DSL) for expressing usage management policies and associating those policies with managed artifacts. We begin by framing a use model for the language, including generalized use cases, a loose ontology, an general supported lifecycle, and specific extension requirements. We then develop the language from that model, demonstrating key syntactic elements and highlighting the technology behind the language while tracing features back to the initial model. We also compare and contrast this DSL with others developed for rights management (e.g. Ponder). We then demonstrate how the DSL supports common usage management and DRM-centric environments, including creative commons, the extensible rights markup language (XRML), and the open digital rights language (ODRL).

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

DRM, usage management, software architecture

1. INTRODUCTION

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse varius mi et dui consequat id faucibus massa elementum. Nunc iaculis magna nec lectus feugiat vulputate. Sed eu pretium nisl. Sed ipsum urna, vulputate pharetra tincidunt in, aliquet in magna. Nam faucibus suscipit nibh a eleifend. Nulla nisi nunc, vulputate eget sollicitudin ac, interdum quis tortor. Aenean fermentum mollis dui, vel bibendum lorem placerat non. Suspendisse eget sollicitudin orci. Pellentesque metus erat, sagittis eu laoreet sed, ultricies molestie leo. Donec pellentesque massa sed nisl feugiat tempor. Suspendisse convallis purus ac mi posuere varius. Fusce at enim id metus mattis aliquet. Donec elementum adipiscing purus quis egestas.

Quisque euismod pulvinar diam, ut porttitor turpis pharetra non. Proin eu tincidunt lectus. Aenean quis nibh vitae nulla cursus venenatis. Aliquam non dolor eu erat elementum sagittis. Vestibulum viverra tristique eros sit amet elementum. Pellentesque at tellus lorem, vitae convallis purus. Morbi at rhoncus dolor. Integer luctus elit sed elit auctor aliquam. Nulla tincidunt neque in augue adipiscing et consequat dolor fringilla. Cras vulputate dignissim lectus vel consectetur. Ut lorem enim, pulvinar commodo hendrerit vitae, dapibus sit amet turpis. Duis bibendum, turpis et porttitor molestie, magna dolor imperdiet quam, ut scelerisque ante libero sed sem.

Vivamus dictum pellentesque metus quis convallis. Quisque consequat, lacus non hendrerit posuere, nulla risus posuere purus, sit amet vestibulum urna tortor vel erat. Curabitur malesuada sodales diam quis ullamcorper. Aenean lectus lacus, volutpat eu mollis id, semper volutpat leo. Cras turpis neque, rutrum in tempor vitae, egestas ut diam. Nam luctus sagittis euismod. Fusce et lorem tortor, vel bibendum purus. Duis egestas, velit ac faucibus fringilla, magna lectus adipiscing ipsum, vitae porta metus nisi id metus. Morbi eu erat augue. Suspendisse potenti. Maecenas bibendum ultrices urna vitae porta. Aliquam feugiat neque at elit facilisis non aliquet mi adipiscing. Phasellus non lacus quis eros viverra rutrum. Phasellus suscipit suscipit dapibus. Cras quis malesuada augue. Proin hendrerit cursus lectus.

Maecenas vestibulum felis in elit interdum venenatis. Fusce consequat rhoncus justo ut blandit. Pellentesque viverra tellus eget enim cursus quis imperdiet urna consequat. Morbi

nec metus sit amet arcu lacinia tempor non eu tortor. Cras ut massa sed eros porttitor aliquam. Integer justo nisi, scelerisque quis fermentum a, interdum id dui. Aliquam sed arcu eget velit malesuada rutrum. Nullam viverra lectus id tellus scelerisque ac varius metus pellentesque. Nullam leo leo, aliquam id imperdiet a, pellentesque scelerisque ipsum. Phasellus mi velit, ullamcorper ut pellentesque ut, euismod in nisi. Curabitur tincidunt, sapien condimentum congue commodo, lectus ipsum fermentum nunc, tincidunt porttitor mauris augue non massa. Aenean sollicitudin consequat arcu vel congue.

Vivamus vestibulum pulvinar quam nec consectetur. Phasellus bibendum vulputate iaculis. Nullam dictum, mauris a adipiscing porttitor, massa purus mattis magna, id porta erat nibh sed turpis. Morbi laoreet, diam ac iaculis venenatis, libero risus dignissim orci, sit amet consequat diam nisi a ante. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Proin feugiat ullamcorper nulla sed pellentesque. Nulla facilisi. In varius tincidunt velit, sagittis suscipit dolor pulvinar a. Pellentesque quis lorem turpis, placerat dictum orci. Pellentesque at mattis eros. Nunc luctus, elit id fermentum viverra, leo lorem mattis nisi, et faucibus magna augue dapibus ipsum. Integer pharetra felis non dui fermentum sagittis. Suspendisse sit amet justo sapien, eu aliquet nisl.

1.1 Previous Work

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur vitae dolor elit, vel sagittis justo. Nullam vehicula scelerisque fermentum. Quisque pulvinar, neque sit amet tempor sagittis, risus felis consequat massa, at euismod quam lacus sed sapien. Integer nec viverra mi. Mauris ultricies tellus non nisl tincidunt dictum. Pellentesque pretium lectus consectetur mauris tempor rutrum. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Duis dictum quam tellus, eu scelerisque elit. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Nullam massa leo, commodo id suscipit eu, consectetur at quam. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Etiam eu diam leo, in rutrum justo. Phasellus ut turpis at orci tempor varius. Suspendisse iaculis faucibus bibendum. Etiam eget mollis nunc. Suspendisse potenti. Pellentesque lectus leo, ornare a ultrices a, lacinia in diam. Vivamus eu justo at lacus sodales eleifend consequat id dui. Nullam laoreet rhoncus malesuada. Curabitur sit amet cursus diam.

In hac habitasse platea dictumst. Suspendisse potenti. Nunc eros libero, eleifend eu scelerisque sit amet, varius rutrum felis. Nulla facilisi. Phasellus tincidunt sapien a libero blandit blandit. Fusce rutrum aliquet lacus, non rhoncus risus facilisis at. Mauris semper varius quam et placerat. Fusce sed suscipit mi. Vivamus a ligula ante, in adipiscing velit. Nullam auctor molestie tincidunt. Duis blandit diam et ante congue vitae laoreet elit sodales. Proin eu nulla vitae est tincidunt rhoncus. Duis at ultrices odio.

2. LANGUAGE MODEL

In developing this DSL, we needed to have a clear understanding of the specific domain, and develop and appropriate domain model to guide our efforts. Admittedly, there exist many possible models that can describe this area of policy and policy management, and the model that we chose to initially use is purposefully simple to help ease development and implementation efforts. We did however provide arbitrary language-level extensibility to support future extension into more demanding policy implementation areas.

We developed this model to help us understand how policy-centric DSLs would be used, to visualize how the various elements are inter-related, and to clarify important areas upon which to focus effort. Through this model, we were able to conceptualize the initial language structure and generate performance hierarchies, as well as to tailor expected DSL use.

2.1 Expected Use

In order to develop the appropriate DSL giving users the power and expressivity they need to easily express usage management concepts, we begin by developing a model describing how we expect it to be used, and by whom, identifying key functional and non-functional characteristics. We use roles codified as actors to identify the primary user base, and link those roles to specific use cases we expect to be common in day to day DSL use. We also identify common inputs and outputs from expected activities, and show how those input and output elements are related. We finally specify the essential core structure of the DSL, as well as extension points and default implementations of those points.

In general day to day use, we expect that certain activities will be much more common that others. For example, each *policy* requires a *context* in order to both be developed and to run. That *context* describes the actors using an artifact protected by a policy, the artifact itself, and the environment in which the artifact is both expected to be used (during policy design) and is being used (at evaluation). That said, the expectation is that the number of policies is much greater than the number of contexts associated with those policies.

Likewise, we expect that the number of times a policy is evaluated is much greater than the number of times that policy is designed and created. Policies should be read, evaluated, or combined with other policies frequently. This gives us a magnitude ordering for these activities, where the number of supported contexts is much less than the number of created policies, which is in turn much less than the number of times that policy is evaluated or otherwise used.

This has specific implications on both the DSL syntax and performance profile. For example, as it is much more common for policies to be evaluated than contexts to be created, our efforts and tuning the system and increasing performance are best focused on policy evaluation rather than contextual activities. In a similar vein, the language itself should be as simple to comprehend as possible for policies at the expense of contextual elements if necessary.

Figure 1 shows the primary system actors we have identified as well as the use cases with which they will be involved. Actors include:

- *Context Author.* The context author is responsible for defining the context in which a policy will be applied

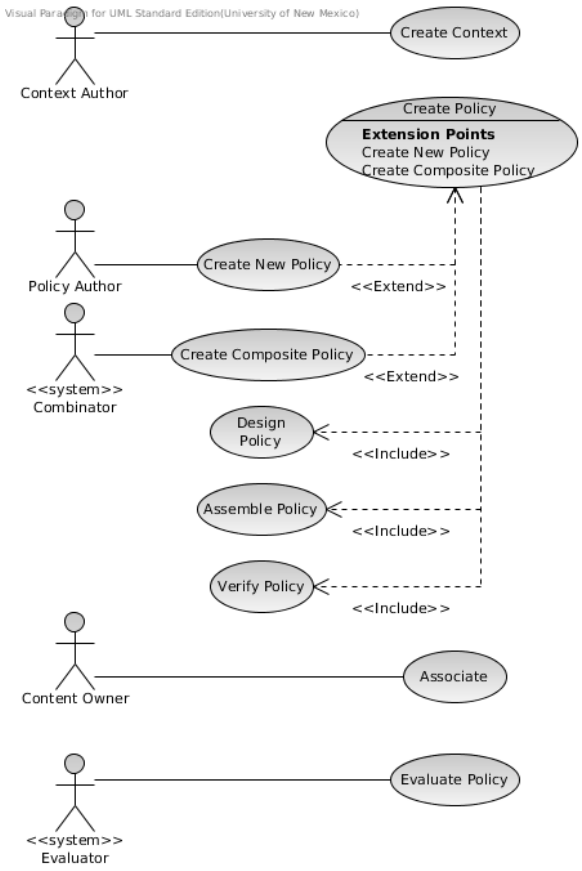

Figure 1: General DSL Use Cases

to a given resource. The context itself defines the environment in which the policy executes, the resource to which the policy is applied, and the subject that attempts to use the resource.

- *Policy Author*. The policy author creates a policy to control the use of a subject defined in the policy context.
- *Combinator*. A system element, generally. A combinator in this context combines two or more policies into a single composite policy.
- *Content Owner*. The owner of a protected resource.
- *Evaluator*. Another system element, an evaluator evaluates a given policy with a specific context.

When a context author creates a context, that author compiles the elements of that context for use both at policy creation and policy execution. When the policy is initially created, the resource is the only defined element. Generally both the subject, representing the eventual user, and the environment, containing information describing the evaluation environment, are only defined at the classifier level. That is to say, they both are defined, but individual properties have yet to be assigned.

Creating a policy is an activity undertaken by either a policy author or a combinator. This step requires a *declared* (but not *defined*) context. This is also undertaken in tandem with some kind of policy specification that describes roughly what the policy should manage and how it should be managed. In the ontology we have defined, this is the step at which the author defines the various constraints, activities, restricted activities, and obligations. Creating a new policy is precisely what it describes - creating a brand new policy applied to a context. Creating a composite policy, on the other hand, involves creating a new derivative policy from two or more previously existing policies.

The included cases define policy, assemble policy, and verify policy are common development steps through which the policy is essentially designed, developed, and then tested against a context.

Once a policy has been created, the content owner can then associate that policy with a resource, essentially instantiating the resource in the associated context.

Finally, a policy is evaluated by an evaluator, a system actor, after creation and association with a resource. At this point, the context has a fully instantiated context, with defined resource, subject, and environmental elements.

Now we have a general understanding of the expected use of a given policy, and have defined the expected roles. With this in place, we begin to look at the elements the DSL should have to allow it to express the use cases we expect we need to support as the next step in refining our understanding of what this DSL should look like.

2.2 Domain Ontology

Our domain ontology will be the foundation of our DSL. It will allow us to begin to understand the various language elements and how they are related, leading us to an eventual syntax to represent these classifiers and relationships. Not understanding this structure well, or developing a structure that does not support our defined use cases will lead us to develop a DSL that inadequately supports our expected use.

Based on our use cases, we know the ontology contains a *context*, some kind of policy-specific sub-ontology, and a logic engine that can act over that sub-ontology. Based on our current understanding of our needs, the sub-ontology contains *obligations* and *constraints* applied to *activities*. We use simple propositional logic to reason over the policy elements.

This understanding leads us to the Ontology view in Figure 2. Of special note, the specific policy sub-ontology is represented as a realization of a more general *policy ontology* type. Likewise, the logic used to reason over the policy elements is propositional logic. Both of these can change within this model to allow for inclusion of more complex policy systems and powerful reasoning capabilities.

Primary elements within this ontology are:

- *Runtime*. This is the system that manages use of a given *resource* by a *subject* in accordance with a *policy*. It is responsible for providing and managing context

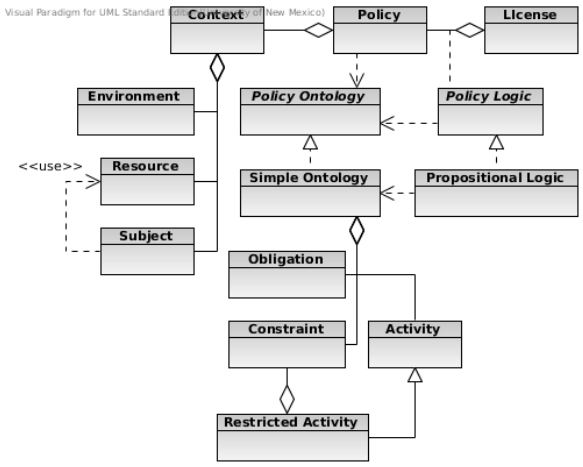

Figure 2: Basic Language Ontology

elements, controlling policies and licenses, and handling requests from subjects. Realizations of this system must be cross platform to support distributed use as well.

- *Context.* The context describes the operating environment of the policy. This information must be available at runtime, and parts of it must be understood when the policy is initially designed. In order to effectively control use of a given artifact, the parameters that artifact can be used under must be understood when the policy is created and must be read when that policy is evaluated.
  - *Environment.* The environment in which a given policy is evaluated. This must be understood in order to constrain the conditions where a policy will allow or disallow artifact access. This is essentially an associative array, where the keys are specific expected properties of a given environment.
  - *Resource.* A resource is the artifact over which the policy controls use. This can be any type of artifact whatsoever, ranging from documents to media files to streaming data. A resource may also have arbitrary properties like an associated URI, a canonical name, a MIME type, or creation metadata.
  - *Subject.* Subjects use a given resource. Acceptable use is described by the policy.
- *Policy.* A policy describes the conditions of use for a given resource. In our example, this includes information on acceptable contexts and subjects, as well as obligations and constraints. Policies can be configured in this DSL to use arbitrary evaluators. This allows users to implement specific policy semantics tailored to their domain if needed, though they are free to use packaged syntax evaluators if those evaluators fit their needs.
- *Policy Ontology (*Abstract*).* As policies can implement arbitrary semantics, they can be based on an ontology tailored to the needs for the particular policy system. For example, this DSL currently implements obligations and constraints restricting defined activities. Other domains may need to use more descriptive semantics, perhaps addressing causality or ordering.
- *Simple Ontology.* The ontology currently used for policy development and packaged with the DSL.
  - *Obligation.* An obligation describes something that must have occurred or must occur in the future for a restricted activity to be performed. For example, a media stream may wish to obligate users to purchase access to that stream on the third access.
  - *Constraint.* A constraint generally constrains the a restricted activity. This could be as simple as limiting use to a single identifiable subject or as complex as limiting use based on time and date, user identity, and geographic location.
  - *Activity.* A general activity is something a subject would wish to do in association with an artifact. It describes how a *subject* would use a *resource* in an unrestricted way.
  - *Restricted Activity.* When an activity is embellished with constraints or obligation, it becomes restricted.
- *Policy Logic (*Abstract*).* As the general policy ontology can be changed to reflect different policy conditions, so to can the logic used to evaluate that policy change. Recognizing this logic as a first order system element helps facilitate that substitution.
- *Propositional Logic.* The logic currently used for policy evaluation.
- *License.* An instance of a defined policy that would be applied to a resource.

Now we have rigorously defined the domain elements our DSL will address. Keep in mind, this domain model allows us to dynamically replace ontology elements in that all *policy ontology* and *policy logic* elements can be replaced on per-policy basis. This would allow us to create multiple policies described using disparate ontologies and related evaluation logics if needed to more fully describe restrictions in a specific evaluation domain.

We have also separated the definition of *activities* from *restricted activities*. This separation of concerns allows policy developers to define a single activity which can then be reused across a large number of restricted activities based on specific varying constraints. For example, if I have a write activity, I can constrain that activity in slightly different ways to create a relatively large number of related restricted activities. I could restrict write by geographic area, by subject identification, by date and time, or by having contributed to some political cause, creating four restricted activities from the same base activity.

The base domain model however, based on contextual elements, policies, and licenses does not change. This stability allows for ease of runtime integration as it hides any policy evaluation-specific changes. As long as a given logic and

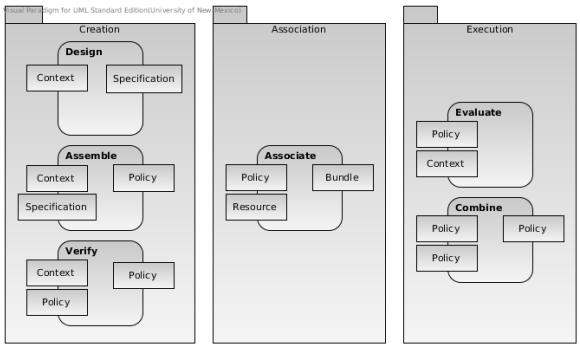

Figure 3: Policy Development Lifecycle

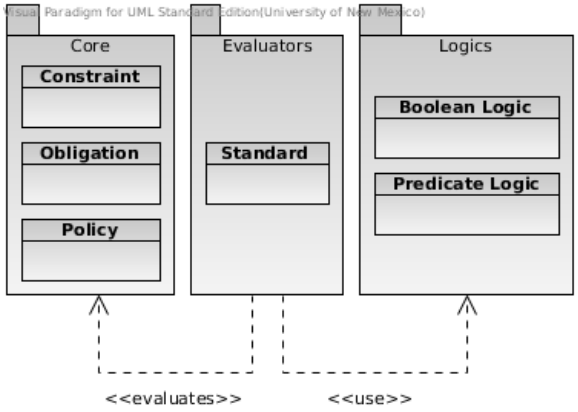

Figure 4: Language Elements

policy ontology is delivered with a given policy, the policy evaluation runtime will be able to evaluate that policy against resources and subjects in a given environment. In essense, this is a layered model where the context and policy elements compose the user interoperability layer while policies and policy logics form a more dynamic policy expression layer. This way, resource users have a stable runtime view of policies and policy managed resources, while policy and context developers have the flexibility to implement policies at arbitrary levels of expressibility.

2.3 Envisioned Lifecycle

2.4 Language Components

3. LANGUAGE

This section describes the domain specific language built upon the language model described in Section 2. First the deployment of the language is provided that explains how the DSL is operates and is used in usage management systems. This is followed by a detailed example of the syntax and semantics, and major components of the language. Finally, we demonstrate how the language can be extended to include different types of evaluators deploying different types of logics.

3.1 Language Operation

The use and operation of the DSL for usage management is shown in Figure 5. The DSL provides a language for specification of contexts and policies as described in the previous

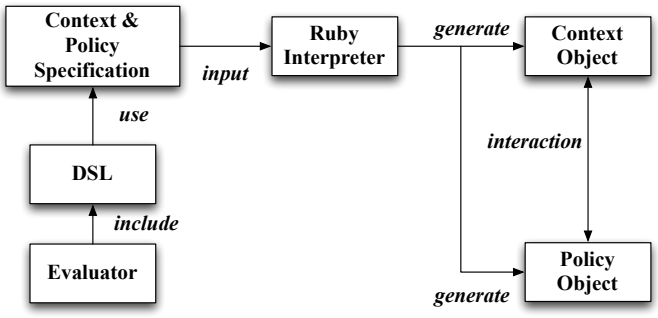

Figure 5: The role of DSL in usage management systems

section. As shown in the figure, the DSL can incorporate different types of evaluators for expression of different types of usage semantics. A DSL, along with an appropriate evaluator is then used to specify contexts and policies. Users are allowed to choose from different types of evaluators that provide the right type of semantics needed by the users. The policy specification and context specification, described using the DSL is then taken by a Ruby compiler to generate a corresponding context and policy object. In usage management systems, context objects are generated on the client side and maintained within the computing platform. The policy specification is provided by the resource owners, is converted into a policy object. Following this, the policy and context objects interact with each other and operate within a usage management system.

3.2 Language Description

The language is based on the model description provided in the previous section, and enables specification of the various policy semantics and context descriptions.

3.2.1 Context Specification

The DSL provides a mechanism for specification of different types of contexts based on the context structure explained in the previous section. A context consists of a set of entities, such as Subject, Resource, and Environment, and each of these entities possess a set of properties. Every property supports a set of functions that operate over the property. The steps involved in defining a context first includes the description of different types of properties. The properties are then allocated to different entities, and then entities are assigned to a given context.

Consider a multi-level security context as shown in Table 1. The context consists of two entities, namely, *Subject* and *Environment* and *Resource*. *Subject* entity has three properties, namely, *Security Clearance*, *Role* and *Project*. *Environment* entity has three properties, namely, *OperatingSystem*, *Device* and *SecurityDomain*. *Resource* entity has one property, *SecurityClassification*. Every property supports a set of functions depending on the type of that property. For the purpose of this discussion, we explain two types of properties, namely, equatable and comparable. Equatable properties primarily support equality functions "=" and "! =". Comparable properties support functions that allow relative comparison of two values, namely, "=" , "! =", "<",

Table 1: An example structure of context.

Context			
Entity	**Property (p)**	**Domain (D_p)**	**Functions (F_p)**
Environment (E)	OperatingSystem	{Windows, OSX, SELinux}	equatable
	Device	{Workstation, Handheld, Blackberry, Terminal}	equatable
	SecurityDomain	{ ABNet, SECNet, TELNet, OMNINet}	comparable
Subject (S)	SecurityClearance	{Top Secret, Secret, Confidential}	comparable
	Project	{Zebra, Yuma, Lion}	equatable
	Role	{Alpha, Beta, Delta}	equatable
Resource(R)	SecurityClassification	{ Top Secret, Secret, Confidential, Unclassified}	comparable

">", "≤", "≥" and "*between*()". Both, equatable and comparable property types support "*get*()" and "*set*()" functions to retrieve and set property values. The properties *OperatingSystem*, *Device*, *Project* and *Role* are equatable properties, and *SecurityDomain*, *SecurityClearance* and *SecurityClassfication* are comparable properties. *SecurityDomain* property values have the ordering $ABNet > SECNet > TELNet > OMNINet$, *SecurityClearance* property values have the ordering $Top\ Secret > Secret > Confidential$, and *SecurityClassification* property values have the ordering $Top\ Secret > Secret > Confidential > Unclassified$.

In order to specify this context, individual properties are defined first as follows.

```
Property (:OperatingSystem) do
        values :Windows, :OSX, :SELinux
        functions :set, :get, :equatable
end

Property (:Device) do
        values :Workstation, :Handheld, :Blackberry,
                :Terminal
        functions :set, :get, :equatable
end

Property (:Project) do
        values :Zebra, :Yuma, :Lion
        functions :set, :get, :equatable
end

Property (:Role) do
        values :Alpha, :Beta, :Delta
        functions :set, :get, :equatable
end
```

In this example, classes *OperatingSystem, Device, Project* and *Role* that inherit the type *Property* are specified. The term *values* specify the set of valid values and *functions* define the set of functions supported by the class as described earlier. Similarly, classes for comparable properties are defined, with the addition that the ordering of the valid values is specified by the user as shown below.

```
Property (:SecurityDomain) do
        values :ABNet, :SECNet, :TELNet, :OMNINet
        functions :set, :get, :comparable
        order :ABNet > :SecNet > :TELNet > :OMNINet
end

Property (:SecurityClearance) do
        values :Top Secret, :Secret, :Confidential
        functions :set, :get, :comparable
        order :Top Secret > Secret > Confidential end

Property (:SecurityClassification) do
        values :Top Secret, :Secret, :Confidential,
                :Unclassified
        functions :set, :get, :comparable
        order :Top Secret > :Secret > :Confidential >
                :Unclassified
end
```

In this example, classes *SecurityDomain*, *SecurityClearance* and *SecurityClassification* that inherit the type *Property* are generated. Each of these classes are comparable and provide a set of comparison functions. The *order* keyword allows context designers to specify the value ordering in an easy manner. The property classes defined here are now assigned to the entities *Subject*, *Resource* and *Environment*. These assignments are described in the DSL as follows.

```
Entity (:Subject) do
        contains :Project, :Role, :SecurityClearance
end

Entity (:Environment) do
        contains :Device, :OperatingSystem,
                :SecurityDomain
end

Entity (:Resource) do
        contains :SecurityClassification
end
```

The above specification generates the *Subject*, *Resource* and *Environment* classes. The DSL generates a *Subject* class that contains properties of type *Role*, *Project*, and *SecurityClearance*. Entity type classes, by default, provide functions that

provide information about the properties contained in these entities. Finally, each of these entities are included in the context, which in this case is a multi-level security context. The syntax for context specification is as follows.

```
Context (:MultilevelSecurity) do
        contains :Subject, :Resource, :Environment
end
```

This specification generates the *MultilevelSecurity* class that contains entities *Subject*, *Resource* and *Environment*. The process of context specification is carried out by context designers. This a less frequently carried out process, and the DSL provides an easier way to provide this specification. The *MultilevelSecurity* class is instantiated, and the corresponding object of this class is maintained on the client system. The property values maintained by this object defines the circumstances under which activities are carried out in the client system.

It must be noted that various kinds of properties can be described using this method. For example, properties having set-based characteristics can be defined that support set functions such as "*in*()" to determine if a given element is a part of the set.

The context generated in this manner is then used to define policies. The DSL provides a policy specification component that provides an easy way to specify policies. Policy specification are read by the Ruby interpreter and converted into policy objects. For example, consider the following policy.

"A document with a security classification greater than or equal to Secret can only be viewed by subjects working on project Yuma having security clearance greater than or equal to Secret on only Blackberry devices operating in security domain greater than SECNet . Also, viewing can be done only after project Yuma has been project-authorized. "

The above policy is expressed in terms of the DSL in the following steps.

```
a1= activity (view) do
    constraint_evaluators :propositional
    SecurityClassification ≥ Secret ∧ Project = Yuma ∧
    SecurityClearance ≥ Secret ∧ Device = Blackberry ∧
    SecurityDomain ≥ SECNet
end
```

First an activity *view* is defined and associated with the constraints defined in terms of context properties described earlier. Such activities with restrictions are called *restricted activities* as described in the language model. It must be noted that the activities defined in the DSL are agreed upon *a priori* with the client system. Now we define another activity, *project-authorization* along with its restrictions (or constraints)

```
a2= activity (project-authorization) do
    constraint_evaluators :propositional
    Project = Yuma
end
```

Once activities activities $a1$ and $a2$ are defined, relationship between the two activities is defined. Such relationships are called usage policies, and are defined the policy definition of the DSL. Usage semantics establish relationships among different activities along with behavioral or or history-based characteristics such as count limits on a given activity. In the present form, the DSL allows permissions, obligations and count limits. However, the DSL can be extended, and usage semantics can be added or changed by using different types of evaluators. This aspect of the DSL is discussed later. In this example, activity *view* is a permission, and activity *project-authorization* is an obligation for exercising this permission. These semantics are expressed in the DSL as follows.

```
pol = policy a1, a2 do
      policy_evaluators :standard
      permit a1 do
            when a2
      end
end
```

The above policy specification says that the policy *pol* is defined using activities *a1* and *a2*, and activity *a1* is a permission that is obligated by activity *a2*. Additional usage semantics could be added to this activity, for instance count-based limits as follows.

```
pol = policy a1, a2 do
      policy_evaluators :standard
      permit a1 do
            when a2
      end
      count_limit a1, 5
end
```

Once the policy specification is provided to the Ruby interpreter, it converts it into a policy object. A policy object is nothing by an executable ruby object with a well-defined policy interface. An example interface provided by the standard policy evaluator defined here contains the following set of functions.

- **permissions?()** Returns the set of permissions for a given policy.
- **obligations?(a)** Returns the set of all obligations associated with a given permission.
- **remaining_obligations(a)** Returns the set of remaining obligations for a given permission.
- **remaining_count(a)** Returns the set of remaining count for a given permission.

- **allowed?(a, ctx)** A boolean function that returns *true/false* whether a given activity can be carried out under a given context.
- **reset()** Resets the policy by resetting its state.

The policy DSL proposed here is simply a skeleton which can extended by incorporating different types of logics expressing various usage semantics. The manner in which these extensions may be carried out is described next.

3.3 Langauge Extensions

The DSL allows inclusion of different types of evaluators that provide users with different sets of usage semantics. The evaluators provide a design space for innovation and extensibility. In the examples provided here, the evaluators used for evaluating constraint (or restriction) semantics is a *propositional* evaluator. This evaluator allows constraints to be expressed as boolean formulas constructed from property functions. This evaluator provides a design space for expressing restrictions in terms of context properties. This evaluator is merely a pulugin that can be replaced by a different constraint evaluator that allows constraints to be expressed in a different manner.

Similarly, the policy construct uses a policy evaluator for expressing and reasoning about usage semantics. The *standard* evaluator used here allows expression of permissions, obligations and count-based limits. The usage semantics is similarly a design space where different types of evaluators can be used to express various types of usage semantics such as parallel actions, partial ordering, and interleaving semantics among others.

In the next section we show how different rights expression languages such as ODRL, XrML and creative commons can be mapped on to this DSL.

4. APPLIED

Application section

4.1 Creative Commons

Creative commons applied

4.2 XRML

XRML applied

4.3 ODRL

ODRL applied

5. CONCLUSIONS AND FUTURE WORKS

Conclusion & future works