A Domain Specific Language for Usage Management

Christopher C. Lamb, Pramod A. Jamkhedkar, Viswanath Nandina, Mathew P. Bohnsack, Gregory L. Heileman
University of New Mexico
Department of Electrical and Computer Engineering
Albuquerque, NM 87131-0001
{cclamb, pramod54, vishu, mbohnsack, 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 domain model, 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 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

D.2.12 [**Software Engineering**]: Interoperability—*Distributed Objects*; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*Security Policy*

General Terms

Design, Languages, Theory, Security, Policy Languages

Keywords

Access Control, Interoperability, DRM, Usage Management

DRM'11, October 21, 2011, Chicago, Illinois, USA.

1. INTRODUCTION

For the purposes of this paper, we currently define usage management as the management of the usage of resources (and data) across and within computing environments. More than access control or digital rights management, usage management concerns itself with fine-grained control of all aspects of how a given digital resource is used. As digital environments become more open over time, the need for usage management for resources that span utility computational environments (e.g. cloud provider systems) will become increasingly important [?].

With the advent and widespread use of cloud computing, those responsible for a given usage managed resource are almost never those responsible for the computing systems, except at edge devices like mobile phones or other small profile computing devices. Resources are regularly moved across national boundaries and regional areas without either the content owner's or creator's knowledge. Furthermore, this kind of transfer is generally according to pre-established algorithms or data routing protocols over which users of all stripes have no control. Managing these kinds of issues requires new usage management capabilities that can run on platforms ranging from small, hand-held devices to nodes in large data centers.

Historically research in this area has been focused on developing more expressive policy languages via either different type of mathematical logics or formalisms with greater reasoning capabilities [?, ?, ?, ?, ?, ?, ?]. These approaches however fail to address interoperability challenges posed by new commercially available distributed computing environments. Interoperability efforts have resorted to translation mechanisms, where the policy is translated in its entirety to a different language [?, ?, ?]; it has been shown recently however that such techniques are infeasible and hard to perform for most policy languages [?, ?]. Other approaches have led to

complex policy specification languages that have tried to establish themselves as the universal standard [?, ?, ?, ?]. This unfortunately tends to stifle both innovation and flexibility [?, ?, ?, ?].

To address these issues, we first applied the principles of system design to develop a framework for usage management in open, distributed environments that supports interoperability. These principles have been used by researchers in large network design create a balance between interoperability and open, flexible architectures [?, ?, ?], allowing for computing scale and power without sacrificing innovation. Initially we standardized certain features of the framework operational semantics, and left free of standards features that necessitate choice and innovation.

We have implemented this framework, including a usage management calculus providing a platform for usage management, within a Domain Specific Language (DSL) and associated evaluation environment. The DSL and its environment implements our previously defined framework, separating various roles needed for distributed policy creation and management, provides the capability to develop executable licenses, and is extensible from both a policy and constraint definition perspective.

In this paper, we will first review the problems in usage management in more detail, providing the context for this DSL and the problems it helps solve. Then, in Section 3 we will first review the model we developed to guide the DSL's syntactic and semantic development. Then, in the next section, we will cover the language itself, how it was developed, and its supporting evaluation environment. We will then close the paper with three specific implementation examples showing how the language and its runtime support usage management scenarios from three different environments — creative commons (CC), the extensible rights markup language (XRML), and the open digital rights language(ODRL).

2. MOTIVATION

As we covered in 2010 [?], usage management incorporates specific characteristics of traditional access control and digital rights management and incorporates encryption mechanisms, trust management, and trusted computing platforms. In order to be effective, it must be flexible enough to provide users with opportunities for differentiation and extension, but inter-operable enough to provide services across widely diverging computational environments.

This DSL and runtime provide flexibility through the inclusion of easily pluggable constraint and policy evaluators. Both types of evaluators simply need to be locatable by the runtime, via standard classpath-style semantics. If they are, they can be loaded and used with no other configuration required. A policy file can simply reference the evaluator via a predefined language element, and the runtime will find it and load it automatically, using it to evaluate either the policy itself or the defined constraints (depending on whichever type of evaluator is specified).

The policy runtime elements can be dynamically loaded on host systems when required. This provides interoperability by forcing the policy system itself to accept the responsibility of ensuring it is installed wherever needed. The technology upon which the runtime is based is widely available as well. Additionally, the policies can be transferred from one host to another and can be directly executed upon delivery. Furthermore, this language and system can express the semantics of common rights management environments, providing additional semantic flexibility and interoperability.

The DSL we describe, in tandem with its runtime, embodies an inter-operable framework for usage management, unlike Coral and Marlin architectures, that implements a formal calculus to reason about the relationship between a license, a computing environment, and interoperability between them. It incorporates concepts such as programmable licenses and common ecosystems used by Coral and Marlin architectures respectively. The DSL design is based on the principles of *design for choice*, eloquently described by Clarke et al. with reference to "tussles" in cyberspace [?]. They explain the importance of identify the locations in the architecture where standards need to be introduced to enable interoperability, and locations where they should *not* be applied to enable innovation and differentiation. By supporting pluggable evaluators that allow users to extend the basic language syntax, semantics, and runtime arbitrarily, this language system provides space for innovation.

3. LANGUAGE MODEL

In developing this DSL, we needed to have a clear understanding of the specific domain, and develop an 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-

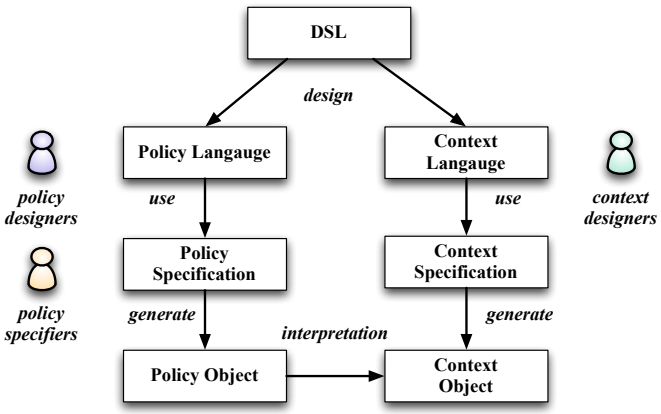

Figure 1: Conceptual Model

level extensibility to support future extension into more demanding policy implementation areas.

Our expectation, reflected in Figure 1, is that we have created a system through which three primary actors work in tandem to create executable policies that can be converted into licenses in order to manage the use of defined content. The process starts with a context designer, who specifies the environment in which a resource can be used by a specified subject. This context is then applied by a policy designer to create specific policy language, which is then used by a policy author to create a policy. The policy itself is manipulated behind a defined interface that requires an instantiation of the defined context from a usage management system.

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.

3.1 Expected Use

In order to develop the appropriate DSL giving users the power and expressiveness 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 at 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 2 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 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 Designer**. The policy designer is responsible for putting together a specification for a given policy that a policy author can use to build an instance of a policy. The person in this role is also responsible for building various policy evaluation components for the DSL if needed.
- **Policy Author**. The policy author creates a policy to control the use of a subject defined in the policy context.
- **Evaluator**. Another system element, an evaluator evaluates a given policy with a specific context.

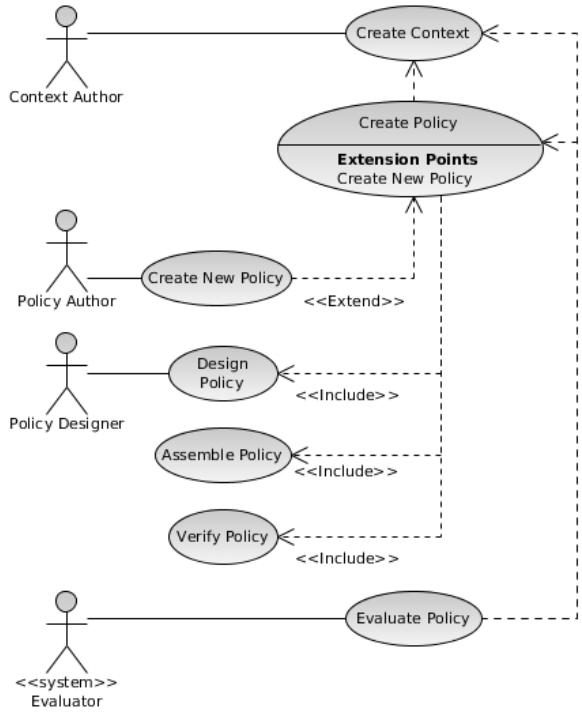

Figure 2: General DSL Use Cases

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 a policy author. 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 policy designer defines the various constraints, activities, restricted activities, and obligations and develops the appropriate DSL components required to evaluate that policy. Creating a new policy is precisely what it describes - creating a brand new policy applied to a context.

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.

Finally, a policy is evaluated by an evaluator, a system actor, after creation and association with a resource. At this point, the system 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.

3.2 Domain Model

Our domain model 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 model contains a *context*, some kind of policy-specific ontology, and a logic engine that can act over that ontology. Based on our current understanding of our needs, the ontology contains *obligations* and *constraints* applied to *activities*. We use simple first-order logic to reason over the policy elements.

This understanding leads us to the model view in Figure 3. Of special note, the specific policy ontology is contained in the usage semantics package, as are all policy evaluators. 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 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.

- **Usage Semantics**. As policies can implement arbitrary semantics, they can be based on an specific models 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.

Current usage semantics package various evaluators (specifically constraint and policy evaluators) and related entities as shown in Figure 4. Flexibility in policy evaluation is provided by pluggable *evaluators*. These evaluators essentially execute a given policy in tandem with a supplied context.

- **Evaluators**. Evaluators are pluggable components that provide flexibility in policy capability. We currently use two — one for evaluating policies, and one for evaluating constraints. Policy designers can also specify multiple evaluators to enable combinations of semantics, as long as those semantics are not in conflict. Evaluators can use any kind of computational model or logic as well. For example, one evaluator may use a simple state machine model to evaluate a policy, while another may use a more complex Turing-complete model.

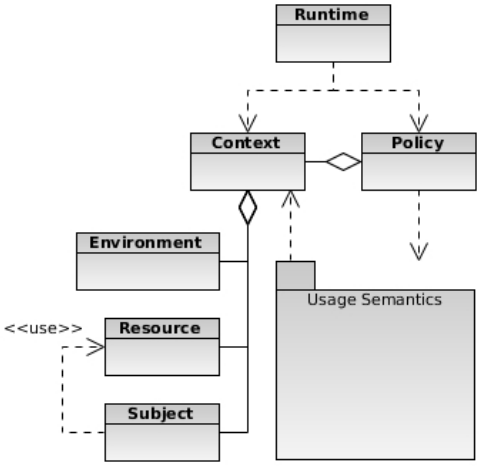

Figure 3: Basic Language Model

  - **Policy Evaluator**. This evaluator is responsible for evaluating the policy as a whole. To do so, it evaluates the relationships between permissions and obligations, and uses the constraint evaluator to examine restricted activities.
  - **Constraint Evaluator**. Constraint evaluators specifically examine constraints defined within restricted activities. They can do so using arbitrary evaluation rules ranging from simple conjunctive evaluation (where all constraints must be true) to more complex schemes where, for example, a certain number of constraints must be true while one of a set of other constraints must be false.

- **Obligation**. An obligation describes a restricted activity 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.

- **Permission**. In this context, a permission is a restricted activity that a subject can perform under the condition that certain specified obligations are met.

- **Constraint**. A constraint generally constrains 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*

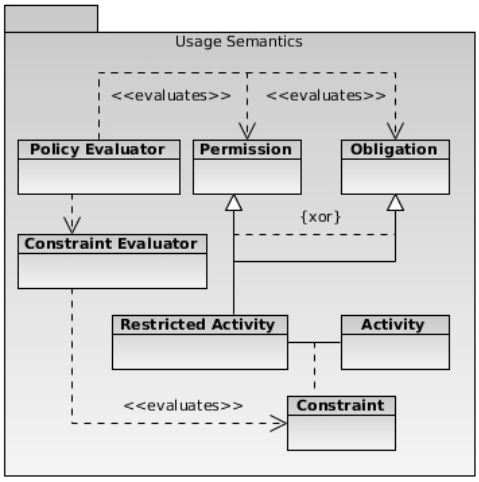

Figure 4: Current Usage Semantics

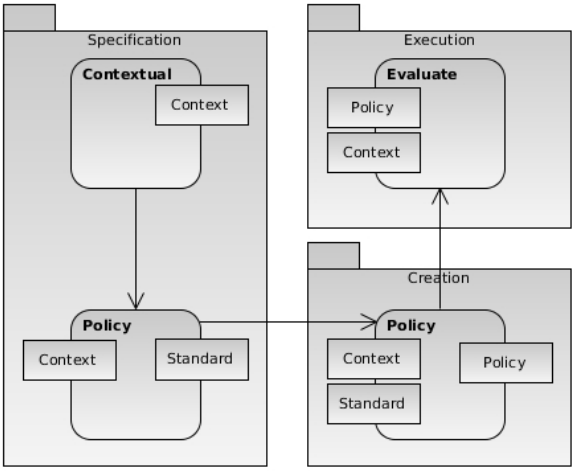

Figure 5: Policy Development Lifecycle

in an unrestricted way, or some action that *subject* performs.

- **Restricted Activity**. When an activity is embellished with constraints or obligation, it becomes restricted.

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 evaluator* and *constraint evaluator* 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 element and policies 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 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 essence, 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.

3.3 Envisioned Life-cycle

Also important to note, the primary artifacts generated within a typical life-cycle include a *context specification* (which eventually becomes a *context*) and a policy specification (eventually becoming a policy). Two of the three actors work on specification development. The context specification must be developed prior to a policy specification or policy as it contains not only the ontology describing the environment, resources, and subjects (and their relationships), it also describes the availability of these elements. The policy designer cannot develop a policy specification without a detailed understanding of the ontological elements in the domain of the policies he intends to support. Likewise, the policy author cannot create a policy until the policy specification is in place.

Policies and contexts have slightly different life-cycles however. A policy, on one hand, is created by a policy author and associated with a resource according to some kind of policy specification. A context, on the other, while specified first, is instantiated last. The context is partially instantiated when the policy author originally associated a resource with a given policy. Likewise, the eventual user becomes the subject in the context. Finally, the usage management runtime even-

tually creates the rest of the context, filling the environment with values as needed for evaluation in tandem with the policy. We illustrate this expected life-cycle in Figure 5.

3.4 Final Language Model

At this point, we have defined the primary elements in a policy life-cycle according to the usage management model we have established. Furthermore, we have established the primary domain elements associated with our identified usage semantics, as well as key use cases for the system. We have also located areas of heavy use as well as areas of low use within the domain model, so we have clear targets for our optimization efforts. With this detailed understanding of this system, and an understanding of clear extension points, we could begin to implement the DSL and the system required to evaluate it.

4. LANGUAGE

This section describes the domain specific language built upon the language model described in Section 3. 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.

This DSL is an *internal DSL* as opposed to an *external dsl*. Internal DSLs are implemented within the syntactic constraints of a given parent language; in this case, Ruby. We chose Ruby for our DSL implementation because of its perhaps excessively permissive and expressive syntax. Certainly, this kind of language flexibility can cause problems in some environments, but when assembling a DSL, it is a requirement. External DSLs on the other hand are implemented outside a specific execution environment and are developed more like a traditional computing language like C or, in fact, Ruby. This kind of DSL requires some kind of compiler or interpreter, as well as lexers and parsers to process the language syntax appropriately. External DSLs are almost always more expressive, and much more secure, but are more difficult and time consuming to develop when compared to their internal brethren. As the DSL we describe here is an ongoing research project and constantly subject to change, we opted to develop it as an internal DSL rather than external, though this may change in the future.

4.1 Language Operation

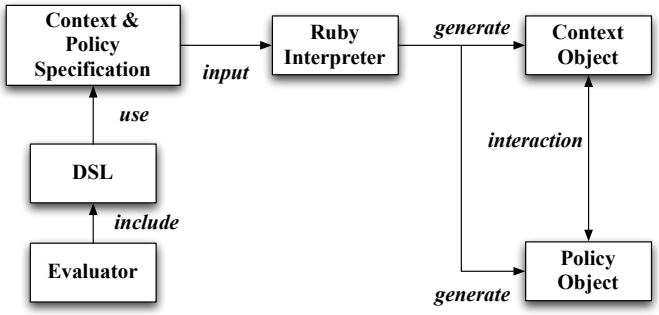

Figure 6: The role of DSL in usage management systems

The use and operation of the DSL for usage management is shown in Figure 6. The DSL provides a language for specification of contexts and policies as described in the previous 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 and is converted into a policy object. Following this, the policy and context objects interact with each other and operate within a usage management system.

4.2 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.

4.2.1 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

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

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 three entities, namely, *Subject*, *Environment* and *Resource*. The *Subject* entity has three properties, namely, *Security Clearance*, *Role* and *Project*. The *Environment* entity has three properties, namely, *OperatingSystem*, *Device* and *SecurityDomain*. The *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 "$\neq$". Comparable properties support functions that allow relative comparison of two values, namely, "$=$" , "$\neq$", "$<$", "$>$", "$\leq$", "$\geq$" 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 implement 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 :topsecret, :secret, :confidential
  functions :set, :get, :comparable
  order :topsecret, :secret, :confidential
end

property :securityclassification do
  values :topsecret, :secret, :confidential,
    :unclassified
  functions :set, :get, :comparable
  order :topsecret, :secret, :confidential,
    :unclassified
end
```

In this example, classes *SecurityDomain*, *SecurityClearance* and *SecurityClassification* that implement 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 is a less frequently carried out process, but the DSL still provides a relatively easy 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 define 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 a security domain greater than SECNet . Also, viewing can be done only after project Yuma has been project-authorized. "

This policy is expressed in terms of the DSL in the following steps:

```
view = activity :view do
  # Some activity to enable viewing
end

c1 = constraint do
  securityclassification >= :secret
    && project == :yuma
    && securityclearance >= :secret
    && device == :blackberry
    && securitydomain >= :secnet
end

restricted_view = restrict view do
  with c1
end
```

First an activity *view* is defined and associated with the constraints defined in terms of context properties described earlier. This then becomes a *restricted activity*, as we have imposed contextual constraints on the activity. 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):

```
authorization = activity :project_authorization do
  is_authorized? :yuma
end
```

Once activities *restricted_view* and *authorization* are defined, we can establish relationships between the them. 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. In this example, restricted 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 do
  policy_evaluators :standard
  constraint_evauators :propositional
  permit restricted_view do
    when authorization
  end
```

```
end
```

The above policy specification says that the policy *pol* is defined using activities *restricted_view* and *authorized*. Activity *restricted_view* is a permission that is obligated by activity *authorization*. Additional usage semantics could be added to this activity, for instance count-based limits as in the following policy:

```
pol = policy do
  policy_evaluators :standard
  constraint_evauators :propositional
  permit restricted_view do
    when authorization
    count_limit restricted_view, 5
  end
end
```

Once the policy specification is provided to the Ruby interpreter, it converts it into a policy object. A policy object is nothing but 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.

Note, in both policies we explicitly identify both the constraint and policy evaluators. This is done for clarity in this case; these values are in fact the default interpretation values used so those statements could very well be omitted.

- **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.

4.3 Language 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 are *propositional* evaluators, our current default type. This type of 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 plugin 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.

5. APPLIED

Now that the DSL has been modeled and implemented, we can apply it to a variety of common usage management domains. In order to demonstrate the DSL's power and flexibility, we have chosen to implement examples from creative commons, the open digital rights language, and the extensible rights markup language.

5.1 Creative Commons

In this section, we use our DSL to implement a policy that can be used with creative commons licensed content. The creative commons is a nonprofit organization that provides free, easy-to-use legal tools that provide a simple, standardized way to pre-clear usage rights to creative works for copyright owners. It gives copyright holders an easy way to license their works in a way that leaves "some rights reserved" as compared to the "all rights reserved" default. By providing these legal tools, the creative commons hopes to increase the amount of creativity available "in the commons" [?].

Seven main license types are defined by creative commons:

- **Public Domain**. This license does not put any restrictions on how others may remix, tweak, or build upon your work.

- **Attribution.** This license lets others distribute, remix, tweak, and build upon your work, even commercially, as long as they credit you for the original creation. This is the most accommodating of the licenses offered. It is recommended for maximum dissemination and use of licensed materials.
- **Attribution, Share-Alike.** This license lets others remix, tweak, and build upon your work even for commercial purposes, as long as they credit you and license their new creations under the identical terms. This license is often compared to free and open source software licenses. All new works based on yours will carry the same license, so any derivatives will also allow commercial use.
- **Attribution, No-Derivatives.** This license allows for redistribution, commercial and non-commercial, as long as it is passed along unchanged and in whole, with credit to you.
- **Attribution, Non-Commercial.** This license lets others remix, tweak, and build upon your work non-commercially, and although their new works must also acknowledge you and be non-commercial, they do not have to license their derivative works on the same terms.
- **Attribution, Non-Commercial, Share-Alike.** This license lets others remix, tweak, and build upon your work non-commercially, as long as they credit you and license their new creations under the identical terms.
- **Attribution, Non-Commercial, No-Derivatives.** This license is the most restrictive of the licenses, only allowing others to download your works and share them with others as long as they credit you, but they cannot change them in any way or use them commercially.

Now that we have explained the different creative commons license types, what they mean, and the restrictions they have, we will create a policy with our DSL that represents compliant usage of creative commons licensed works.

First, we use our DSL to define an activity. In creative commons there is really only one activity – sharing a creative commons licensed work – so this code is fairly simple, defining the `share` activity.

```
share = activity(:share) do
  true
end
```

Next, we define constraints on the `share` activity. There are four such constraints in creative commons, as discussed above: attribution, commercial use, derivative work, and share-alike. This results in four constraints defined in our DSL as follows:

```
attribution = constraint(:share) do
  true
end

non_commercial_use = constraint(:share) do
  true
end

non_derivative_work = constraint(:share) do
  true
end

share_alike = constraint(:share) do
  true
end
```

Finally, we define the restricted activities, covering all seven basic license types. For example, in the following code listing, the `share_by_work` restricted activity is defined by associating the `attribution` constraint with the `share` activity. Likewise, the `share_by_sa_work` restricted activity associates the `attribution` and `share_alike` constraints with the `share` activity. Sharing activites for all seven license types are defined similarly, resulting in a policy that allows for sharing all types of Creative Commons licensed works, with proper constraints.

```
# 1) Public Domain
share_pd_work = restrict share do
  # No constaints
end

# 2) Attribution
share_by_work = restrict share do
  with attribution
end

# 3) Attribution, Share-Alike
share_by_sa_work = restrict share do
  with attribution, share_alike
end

# 4) Attribution, No-Derivatives
share_by_nd_work = restrict share do
  with attribution, non_derivative_work
end

# 5) Attribution, Non-Commercial
share_by_nc_work = restrict share do
  with attribution, non_commercial_use
end

# 6) Attribution, Non-Commercial, Share-Alike
share_by_nc_sa_work = restrict share do
```

```
  with attribution, non_commercial_use,
       share_alike
end

# 7) Atribution, Non-Commercial, No-Derivatives
share_by_nc_nd_work = restrict share do
  with attribution, non_commercial_use,
       non_derivative_work
end
```

Now that we have implemented basic policy protections as defined in the creative commons environment, we will move on to ODRL.

5.2 ODRL

ODRL provides the necessary semantics for Digital Rights Management expressions in an open and trusted environment. By analyzing the ODRL language we can show that it can be mapped to our framework. In the ODRL Expression language there are different models. Specifically, we are going to demonstrate how ODRL-centric permissions, constraints, requirements, and conditions can be expressed in our DSL.

In ODRL, *permissions* are related to *actions* that a *subject* can execute on a *resource* of some kind and generally encapsulate use, reuse, transfers, and general asset management. The following analysis is modeled on ORDL v. 1.1 [?].

In our DSL we limit *activities* with *constraints*. Limiting an activity with a constraint yields a *restricted activity*. This restricted activity can be either an obligation or a permission. While evaluating these permissions, necessary steps must be taken to maintain the restrictions. On the other hand, in ODRL constraints are directly associated with the permissions rather than actions. Furthermore, constraints in our DSL are more flexible than those in ODRL generally. While ODRL constraints have two operands and a single operator, and are conjunctively evaluated, constraints in our system can be any kind of predicate with an arbitrary argument list and extensible semantics.

In ODRL, the condition model holds that if any condition is satisfied, indicating that an event did occur, then related permission is not granted. In our framework, we can express this either with specific constraints or via obligations. Constraints would generally involve the current execution context in which the policy is currently evaluated, and would render a usage decision based on that. Obligations are essentially activities, and can be activities that should have been performed prior to use, and as such can model conditions as well.

Last part is the context model. If we take into account the context model in ODRL, each and every entity has its own independent context whereas in our DSL all the entities lie inside a common context. This allows for ease of ontology sharing between the various components operating within a given context.

To demonstrate the mapping between the ODRL model and our framework, we will examine a scenario and express it as an ODRL license and then create an equivalent DSL. Consider a scenario where a user wants to listen to music from an on-line music library. To get access to the library the user has to pay the amount of $15 USD. The music file can only be played on a windows based device within United States.

```
<rights>
  <asset>
    <context>
      <uid>10.00000/content/musiclib/mus-000001</uid>
    </context>
  </asset>
  <permission>
    <listen>
      <constraint>
        <devices>'MSWindows'</devices>
      </constraint>
      <condition>
        <constraint>
          <country>'US'</country>
        </constraint>
      </condition>
    </listen>
    <requirement>
      <prepay>
        <payment>
          <currency>USD</currency>
          <amount>15.00</amount>
        </payment>
      </prepay>
    </requirement>
  </permission>
</rights>
```

The equivalent DSL would look like the following:

```
a1 = activity :listen

c1 = constraint do
  country == :US && device == :mswindows
end

ra1 = restrict a1 do
  with c1
end

a1 = activity :payment

c2 = constraint do
  amount == 15 && currency == :usd
end

ra2 = restrict a2 do
  with c2
end
```

```
policy1 = policy do
  permit ra1 do
    when ra2
  end
end
```

In the DSL, the first activity is listen which is bounded by the constraint *c1* that the user can listen to the music only within United States and that the device must be a windows device. When constraints are applied to the first activity it is then called restricted activity *ra1*. Similar constraints are reflected in the ODRL snippet.

The second activity is payment which is bounded by a constraint *c2* that the user needs to pay an amount of $15 USD to get access to the music file. We then call this activity as restricted activity *ra2* after the constraint *c2* is applied.

Then we define a policy which allows permission to the first restricted activity *ra1* when second restricted activity *ra2* is true. So, *ra2* is an obligation that needs to be fulfilled in order to get permission for *ra1*. This obligation is represented as a requirement in ODRL.

This example demonstrates clear equivalence in this particular domain. The extensible nature of the DSL's evaluators could provide any missing functionality in more detailed cases as well, but in general, any problem representable in ODRL is also representable within this DSL.

5.3 XRML

Extensible Rights Markup Language (XrML) is a language used to define a user's rights over a resource of some kind. It helps the owner of digital resources to identify the users allowed to access these resources, what rights are available to those users, and the conditions under which those rights may be applied. The XrML language has different concepts .We are interested in mapping the concepts namely grant, principal, right, resource and condition into our framework [?].

A *principal*, in XrML, is someone who has been granted *rights* to a specific *resource*. In this context, *grant* refers to the act of giving or temporarily providing those resource rights. A *right*, associated with a grant, is some kind of action the principal can perform on a given resource, like listening to a song or editing a video stream. *Conditions* specify specific terms under which a principal can wield rights over a resource.

In this DSL, the principal is a subject, grant is permission and right is an activity. A resource in XrML is the same as resource in our DSL's implementation. The concept of condition in XRML maps to our obligation and constraints.

Using the same scenario we used in the ODRL example, we have build a representative XrML file describing user rights in this particular context:

```
<license>
  <grantGroup>
    <principal>User</principal>
    <grant>
      <rights>
        <listen>
          <resource>
            <uid>10.00000/content/musiclib/mus-000001</uid>
          </resource>
          <condition>
            <country>'US'</country>
            <devices>'MSWindows'</devices>
            <grant>
              <rights>
                <payment>
                  <condition>
                    <amount>15.00</amount>
                                <currency>USD</currency>
                        </condition>
                </payment>
              </rights>
            </grant>
          </condition>
        </listen>
      </rights>
    </grant>
  </grantGroup>
</license>
```

In this XrML license, the principal is the user who wants access to the music library. The rights holder gives the right to listen to a resource given the condition that the country in which the the listening occurs is the United States and the device is windows based. Furthermore, the principal must also have already made a payment of the amount USD $15. In the DSL fragment from the previous section, payment and listen are defined as activities, whereas listening to music only on windows based device and within United States is a constraint. When the above constraints are satisfied the user gets permission to access the resource.

6. CONCLUSIONS AND FUTURE WORKS

Usage management is a common problem set with features embodied in domains ranging from security systems to video games to music production and retail. The ability to provide management of resources with regard to authorized subjects is being addressed in multiple different forums, many of which are taking remarkably different approaches. Common features however generally include the need for either ubiquitous rights expression language acceptance or for extensive translation between all supported rights languages.

In this paper, we first demonstrated the development of the initial model we used to define our problem space.

Here, we described the general use of the system, who the primary users were, what the expected lifecyle of policies was, and what the domain model looked like. We then implemented the syntax of the DSL, in Ruby, as an internal DSL with specific examples. We wrapped up the paper with demonstrations of equivalence to common rights management frameworks like the creative commons, ODRL, and XrML.

We have only begun to specify and use this particular DSL. Future focus for our group on this effort will include additional language elaboration, exploration, and use in specific scenarios. We need to spend additional time engineering the underlying software as well, so we can ensure that policies are in fact platform and environment agnostic, portable, and executable. Finally, this implementation is an internal DSL within the Ruby language; we need to explore the application of external DSL techniques to this domain to better understand the required compromises between expressiveness and development difficulty and begin to apply more stringent security models to the system itself.