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 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

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

1. INTRODUCTION

For the purposes of this paper, we current 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 [4, 5, 7, 9, 10, 18, 22]. 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 [11, 17, 20]; it has been shown recently however that such techniques are infeasible and hard to perform for most policy languages [16, 19]. Other approaches have led to complex policy specification languages that have tried to establish themselves as the universal standard [1, 2, 21, 23]. This unfortunately tends to stifle both innovation and flexibility [11, 12, 13, 15].

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 [3, 6, 8], 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 usage management, within a Domain Specific Language (DSL) and associated evaluation environment. The DSL and its environ-

ment 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 implemenation 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 [14], usage management incorporates specific characteristics of traditional access control and digital rights management and incorporate 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 interoperable 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 where required. 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 interoperable 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 [8]. They explain the importance of identify the locations in the architecture where

Figure 1: Conceptual Model

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 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.

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 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
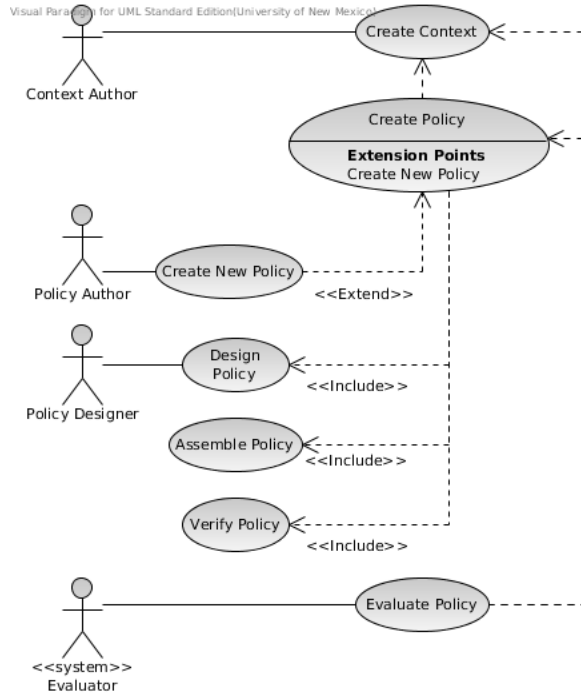
Figure 2: General DSL Use Cases

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 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.

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 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.

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 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 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 sub-ontology is contained in the usage semantics package, as is 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
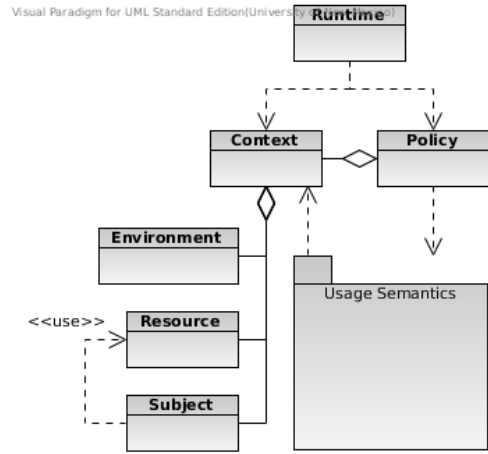

Figure 3: Basic Language Model

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.

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

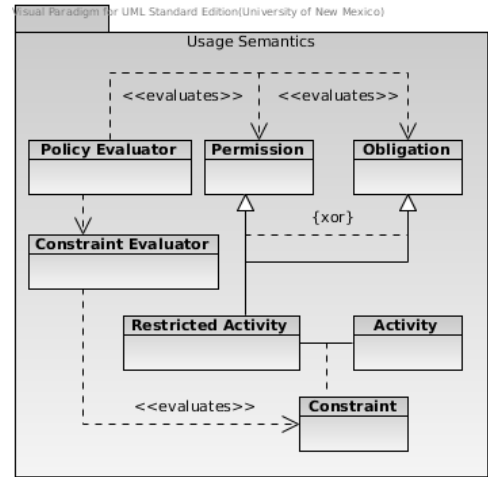- *Restricted Activity.* When an activity is embellished with constraints or obligation, it becomes restricted.

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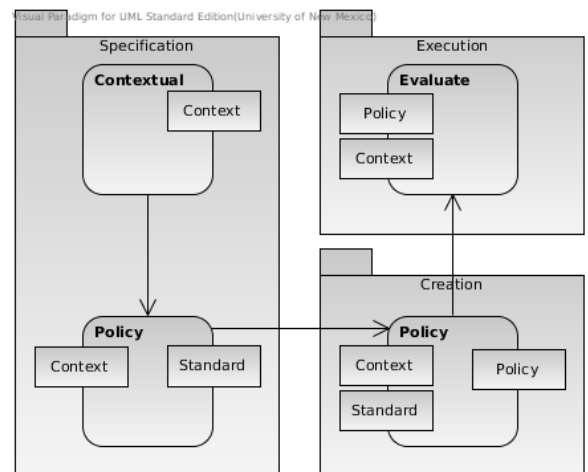
The base domain model however, based on contextual elements, policies, and licenses does not change. This stability allows for ease of runtime integration as it hides any policy evaluation-specific changes. As long as a given logic and policy ontology is delivered with a given policy, the policy evaluation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

3.3 Envisioned Lifecycle
3.4 Language Components
4. LANGUAGE
Language implementation

5. APPLIED
Application section

Figure 4: Current Usage Semantics

Figure 5: Policy Development Lifecycle

5.1 Creative Commons
Creative commons applied

5.2 Creative Commons
The Creative Commons is a nonprofit organization that works to increase the amount of creativity available in "the commons". It provides free, easy-to-use legal tools that provide a simple, standardized way to pre-clear usage rights to creative work for copyright owners. It lets copyright holders easily change from default of "all rights reserved" to "some rights reserved".

In this section, we use our DSL to implement a policy that can be used with Creative Commons licensed content, covering all seven of the main Creative Commons license types:

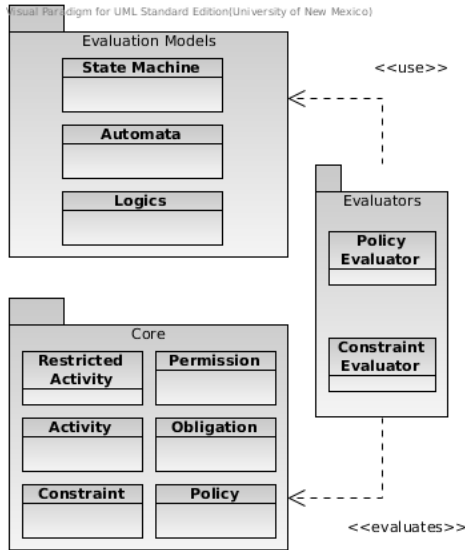- *Public Domain.*

- *Attribution.*

- *Attribution, Share-Alike.*

Figure 6: Language Elements

- *Attribution, No-Derivatives.*
- *Attribution, Non-Commercial.*
- *Attribution, Non-Commercial, Share-Alike.*
- *Attribution, Non-Commercial, No-Derivatives.*

First, we create an activity. This activity represents sharing a Creative Commons licensed asset.

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

Next, we define constraints on this share activity. There are four such constraints in Creative Commons: attribution, commercial use, derivative work, and share-alike.

```
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 7 basic license types.

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

# 2) Attribution
share_by_work = restrict share do
```

```
  with attribution_constraint
end

# 3) Attribution, Share-Alike
share_by_sa_work = restrict share do
  with attribution_constraint,
       share_alike_constraint
end

# 4) Attribution, No-Derivatives
share_by_nd_work = restrict share do
  with attribution_constraint,
       non_derivative_work_constraint
end

# 5) Attribution, Non-Commercial
share_by_nc_work = restrict share do
  with attribution_constraint,
       non_commercial_use_constraint
end

# 6) Attribution, Non-Commercial, Share-Alike
share_by_nc_sa_work = restrict share do
  with attribution_constraint,
       non_commercial_use_constraint,
       share_alike_constraint
end

# 7) Atribution, Non-Commercial, No-Derivatives
share_by_nc_nd_work = restrict share do
  with attribution_constraint,
       non_commercial_use_constraint,
       non_derivative_work_constraint
end
```

5.3 XRML
XRML applied

5.4 ODRL
ODRL applied

6. CONCLUSIONS AND FUTURE WORKS
Conclusion & future works

7. REFERENCES

[1] Enabler release definition for DRM V2.0. Technical report, Open Mobile Alliance, 2003.
xml.coverpages.org/OMA-ERELD_DRM-V2_0_0-20040401-D.pdf.

[2] Open digital rights language ODRL version 2 requirements. ODRL, Feb. 2005.
odrl.net/2.0/v2req.html.

[3] H. Alverstrand. The role of the standards process in shaping the internet. *Proceeding of the IEEE*, 92(9):1371–1374, 2004.

[4] A. Arnab and A. Hutchison. Persistent access control: A formal model for drm. In *DRM '07: Proceedings of the 2007 ACM workshop on Digital Rights Management*, pages 41–53, New York, NY, USA, 2007. ACM.

[5] A. Barth and J. C. Mitchell. Managing digital rights using linear logic. In *LICS '06: Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science*, pages 127–136, Washington, DC, USA, 2006. IEEE Computer Society.

[6] M. S. Blumenthal and D. D. Clark. Rethinking the design of the Internet: The end-to-end arguments vs. the brave new world. *ACM Transactions on Internet Technology*, 1(1):70–109, Aug. 2001.

[7] C. N. Chong, R. Corin, S. Etalle, P. Hartel, W. Jonker, and Y. W. Law. LicenseScript: A novel digital rights language and its semantics. In *Third International Conference on the Web Delivery of Music*, pages 122–129, Los Alamitos, CA, Sept. 2003.

[8] D. D. Clark, J. Wroclawski, K. R. Sollins, and R. Braden. Tussle in cyberspace: Defining tomorrow's internet. In *SIGCOMM*, pages 347–356, Pittsburg, Pennsylvania, USA, Aug. 2002.

[9] J. Y. Halpern and V. Weissman. A formal foundation for XrML licenses. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*, pages 251–265, Asilomar, CA, June 2004.

[10] J. Y. Halpern and V. Weissman. A formal foundation for XrML. *J. ACM*, 55(1):1–42, 2008.

[11] G. L. Heileman and P. A. Jamkhedkar. DRM interoperability analysis from the perspective of a layered framework. In *Proceedings of the Fifth ACM Workshop on Digital Rights Management*, pages 17–26, Alexandria, VA, Nov. 2005.

[12] P. A. Jamkhedkar and G. L. Heileman. DRM as a layered system. In *Proceedings of the Fourth ACM Workshop on Digital Rights Management*, pages 11–21, Washington, DC, Oct. 2004.

[13] P. A. Jamkhedkar and G. L. Heileman. *Handbook of Research on Secure Multimedia Distribution*, chapter Rights Expression Languages. IGI Publishing, 2008.

[14] P. A. Jamkhedkar, G. L. Heileman, and C. C. Lamb. An interoperable usage management framework. In *Proceedings of the tenth annual ACM workshop on Digital rights management*, DRM '10, pages 73–88, New York, NY, USA, 2010. ACM.

[15] P. A. Jamkhedkar, G. L. Heileman, and I. Martinez-Ortiz. The problem with rights expression languages. In *Proceedings of the Sixth ACM Workshop on Digital Rights Management*, pages 59–67, Alexandria, VA, Nov. 2006.

[16] R. H. Koenen, J. Lacy, M. MacKay, and S. Mitchell. The long march to interoperable digital rights management. *Proceedings of the IEEE*, 92(6):883–897, 2004.

[17] J. Polo, J. Prados, and J. Delgado. Interoperability between ODRL and MPEG-21 REL. In *Proceedings of the first international ODRL workshop*, Vienna, Austria, Apr. 2004.

[18] R. Pucella and V. Weissman. A logic for reasoning about digital rights. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, pages 282–294, Nova Scotia, Canada, June 2002.

[19] R. Safavi-Naini, N. P. Sheppard, and T. Uehara. Import/export in digital rights management. In *Proceedings of the Fourth ACM Workshop on Digital Rights Management*, pages 99–110, Washington, DC, Oct. 2004.

[20] A. U. Schmidt, O. Tafreschi, and R. Wolf. Interoperability challenges for DRM systems. In *IFIP/GI Workshop on Virtual Goods*, Ilmenau, Germany, 2004. http://virtualgoods.tu-ilmenau.de/2004/program.html.

[21] X. Wang. MPEG-21 rights expression language: Enabling interoperable digital rights management. *IEEE Multimedia*, 11(4):84–87, Oct./Dec. 2004.

[22] J. Xiang, D. Bjorner, and K. Futatsugi. Formal digital license language with OTS/CafeOBJ method. In *Proceedings of the sixth ACS/IEEE International Conference on Computer Systems and Applications*, Doha, Qatar, Apr. 2008.

[23] eXtensible Rights Markup Language (XrML) 2.0 Specification, November 2001. www.xrml.org.