LicenseScript: A Logical Language for Digital Rights Management

Cheun Ngen Chong[1,2], Ricardo Corin[1], Jeroen Doumen[1], Sandro Etalle[1], Pieter Hartel[1], Yee Wei Law[1], and Andrew Tokmakoff[2]

[1]University of Twente, P.O.Box 2100, 7500 AE Enschede, The Netherlands
{chong,corin,doumen,etalle,pieter,ywlaw}@cs.utwente.nl
[2]Telematica Instituut, P.O.Box 589, 7500 AN Enschede, The Netherlands
{Jordan.Chong,Andrew.Tokmakoff}@telin.nl

Abstract

We propose LicenseScript, a language for digital rights management (DRM) based on multiset rewriting and logic programming. LicenseScript enjoys a precise syntax and semantics, and it is rich enough to embed other rights expression languages (REL). We show that LicenseScript is expressive and flexible by exploring several application domains representing different aspects of DRM. We present an implementation. Finally, we extend the core of the language to account for multiple devices in authorized domains.

1 Introduction

Losses to the film and music industry due to illegal distribution of content on the Internet amount to billions of dollars annually [Hartung and Ramme, 2000]. The easy production and distribution of pirated content makes copyright violations of digital content a problem with far-reaching consequences. Many sorts of solutions have been proposed to tackle this problem. On one hand of the spectrum we find hardware-based solutions, which unfortunately are provider-centric and user-unfriendly, and even infringe upon the legal rights of the user. On the other hand we have DRM systems, which - at least in principle - allow one to flexibly express and to enforce how a certain digital asset should be used.

Figure 1 illustrates the main components of a typical DRM system [Rosenblatt et al., 2002]. Here a license issuer can issue a license for digital content. The content is encrypted by using (one or more) content key(s). Content keys are contained in the license, as are the usage rights associated with the content. This license is bound to the content, e.g. by containing the cryptographic key with which the content is encrypted. Usage permissions or restrictions, such as a (limited) ability to make copies of the content, can be specified in the license. A license may also be bound to a specific Identity (which may refer to a single person or device, or even a group of them). The license itself is expressed in a Rights Expression Language (REL).

Among the many DRM solutions that have been proposed, we should mention Windows Media Rights Manager [Birney and Gill, 2003] and IBM's Electronic Media Management System (EMMS) (`http://www.ibm.com/software/data/emms/`). Most of these solutions comply with the conceptual DRM model described above.

The license is an important part of a DRM system, since it carries the usage rights as well as content keys, content metadata (e.g. location of the content) and other critical information (e.g. binding to an identity) [Chong et al., 2004]. The usage rights can only be described (let alone enforced) correctly if the license

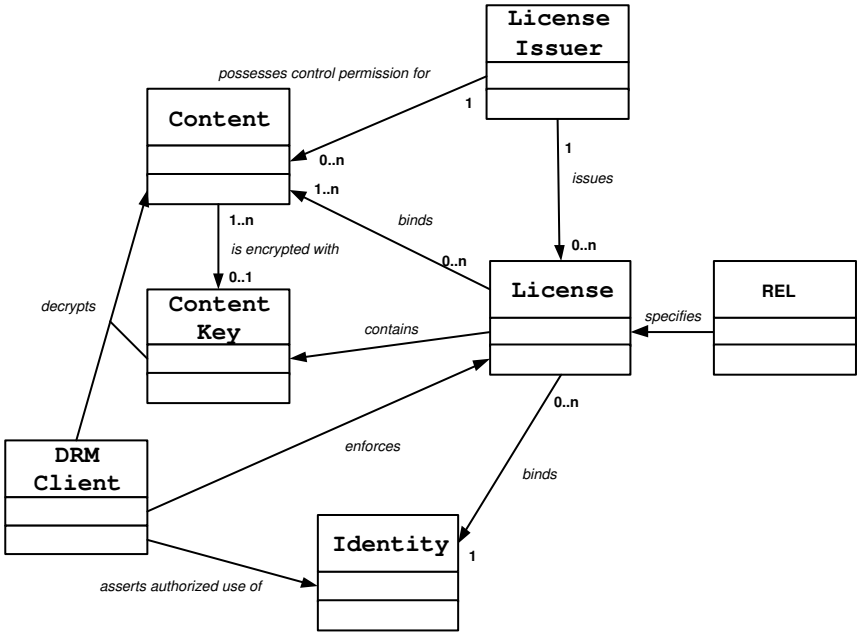

Figure 1: Main conceptual consumption DRM model.

has a well-defined semantics. Therefore, the rights expression language (REL), which is the language used to construct the license, plays a crucial role in DRM [Guth, 2003].

A good REL enjoys the following properties. It has to be sufficiently expressive, so that it can express complex usage rights, terms and conditions. Flexibility is also an important aspect, since the REL should handle all kinds of novel usage scenarios. It should be verifiable, so that one can assure that the created license expresses the intended rights and that they can be correctly enforced. Simplicity, i.e. ease of use is an important aspect as well. Lastly, the REL should be both machine-understandable and human-readable - it needs to be interpreted and enforced by a machine (e.g. personal computer or mobile device), but it should still be possible for a human to read and interpret it.

Most current RELs are based on the eXtensible Markup Language (XML) due to its flexibility, machine-understandability, human-readability, and expressivity. The two main examples are the eXtensible rights Markup Language (XrML) [Guo, 2001] and the Open Digital Rights Language (ODRL) [Iannella, 2001].

However, it is now widely acknowledged that the above-mentioned XML-based RELs have some important shortcomings: first of all, the syntax is complicated and obscure when the usage rights become more complex. Secondly, these languages lack a formal semantics – the exact meaning of licenses relies heavily on the human interpretation. Thirdly, Mulligan and Burstein [2002] found that these languages cannot express many useful copyright acts. Gunter et al. [2001] overcome some of these drawbacks by introducing an abstract model and a language with corresponding formal semantics. Pucella and Weissman [2002] follow up this effort by providing a semantics for a fragment of ODRL. Last (but not least), new high-impact scenarios are emerging in which licenses from different vendors might be combined to have full access to a given digital asset; current XML-based languages do not allow to capture this scenarios.

We propose LicenseScript, a language that is able to express usage rights of a dynamic and evolving license. LicenseScript is based on multiset rewriting, which is able to capture the *dynamic* evolution of licenses, and on logic programming, which captures the static terms and conditions in a license, and a judicious choice of the interfacing mechanism between the static and dynamic domains. LicenseScript makes it possible to express a multitude of sophisticated usage patterns clearly and precisely. The formal

basis of LicenseScript (multiset rewriting and logic programming) provides for a concise and explicit formal semantics.

We use Prolog programs for the license clauses because of the simultaneous declarative and procedural reading of Prolog. Thanks to its declarative reading, Prolog is suitable for rendering licenses, which can be easily read and reasoned about by humans. In fact, Prolog is often used as a language to describe policies and business rules [DeTreville, 2002]. On the other hand, the procedural reading of Prolog allows for an direct execution of the code.

In this paper, using LicenseScript we explore different aspects of DRM, namely technical, business, commercial, and legal aspects. We use LicenseScript to define different licensing models, which are useful to the content providers. We then study the usage scenarios specified in XML-based RELs, and translate the XML code to LicenseScript, exploring its interoperability. We use LicenseScript to specify different types of payment policies, which suit the needs of different users. We also model a problematic aspect of copyright law, namely Fair Use, which allows users to exercise their lawful rights without violating those of the content providers. In short, we show that LicenseScript is technically expressive, flexible, interoperable, and implementable.

The remainder of this paper is organized as follows: Section 2 presents an anatomy of RELs based on related work. Section 3 introduces the LicenseScript core language and its semantics. Section 4 analyzes the safety aspects of LicenseScript. Section 5 explores different aspects of DRM when applying LicenseScript. Section 6 describes the full LicenseScript language, including the concept of authorized domains. Section 7 illustrates our implementation of LicenseScript. Section 8 discusses some related work. Finally, section 9 concludes and presents future work.

2 Setting the Stage: Anatomy of Rights Expression Languages

Based on Stefik's axiomatic principles, the XrML and ODRL specifications, and the requirements of RELs proposed by Parrott [2001], we conclude that RELs have a structure which is shown in Fig. 2. The figure is presented in the form of a class diagram because this exhibits the logical relations between the components. This figure provides an abstract view of a REL.

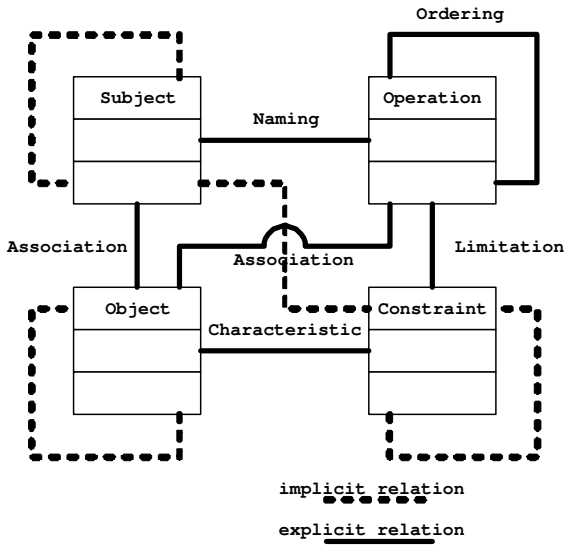

Figure 2: The components and their relations in a REL.

We identify four main components, which we explain in section 2.1. Each of these components is logically related to other components. We elaborate these *relations* in section 2.2. Components and relations support a wide variety of *models* of the rights management systems. We elaborate on the models in section 2.3. Most of what follows originates from the literature, including Parrott [2001] and XML-based RELs specification, but we explicitly indicate the additional features of a REL.

2.1 Components

From Parrott's requirements of RELs [Parrott, 2001] and the XML-based RELs specification, we conclude that there are four main components in a REL, namely: (1) A *subject* is an actor who performs some operations; (2) An *object* is the content acted upon by a subject; (3) An *operation* is what a subject can do to an object; and (4) a *constraint* describes when an operation can be performed. There are two types of operations: a *right* is an operation that can be performed directly on the object; and an *obligation* is an operation that must precede or follow another operation. As an illustration consider the following:

Example 1. Alice wants to play a high-quality movie three times, but she must pay $5 up front.

Intuitively, the subject is "Alice", the objects are "movie" and "$5", the right is "play", the obligation is "pay", and the constraints are "high-quality", "for three times" and "up front".

A REL typically describes the *subject* by naming, e.g. "Alice". Additionally, the REL must be able to distinguish the type of the subject by using roles, e.g. creator, end-user, distributor and so on. A REL must also be able to specify the identification mechanism employed by common rights management systems to describe the subject, e.g. a digital certificate and public key . This is discussed in section 2.3.

A REL uses names to describe an *object*, such as the title of the object or the artist. A REL is required to support (1) generalized types of objects, for instance, multimedia (e.g. MP3), personal data (e.g. DOC) or meta-data (e.g. XML); (2) classification of similar objects, for instance, "publisher Addison's ebooks"; (3) fuzzy (or implicit) matching criteria of the object, for instance, "looks like" and "sounds like"; and (4) the delivery methods of the object, for example, by downloading, streaming or by means of physical storage (e.g. CD).

As mentioned earlier, there are two types of *operation*: *right* and *obligation*. Rights are further divided as follows [Rosenblatt et al., 2002]: (1) *Render right*, which indicates a set of rights in which the object can be consumed, e.g. play; (2) *Reuse right*, which indicates a set of rights in which the object can be re-utilized, e.g. modify. (3) *Transport right*, which indicates a set of rights in which the subject's rights over the object can be transferred, e.g. lend. (4) *Object management*, which indicates a set of rights to handle the management over the object, e.g. move and duplicate. None of these rights cover the regulation of the rights themselves. Therefore, in addition, we propose a further set of rights, namely (5) *Rights management*, which indicates a set of rights that regulate the subject's rights over the object, e.g. update and renew.

On the other hand, an obligation may be an operation that enables or activates the rights over the objects. The *pay* and *register* operations are two common examples.

Parrott [2001] and XML-based RELs specifications recognize several common *constraints*: (1) *temporal*, such as date and time (e.g. the ebook can be viewed before 20 March 2004); (2) accumulated (e.g. the ebook can be viewed for 2 weeks); and interval (e.g. the ebook can be viewed within 20 days from the time of issuing this license); (3) *bound*, for instance, the number of distinct times the ebook can be viewed, and the range of the page numbers of the ebook that can be printed; (4) *environment*, which may be a physical environment (e.g. geographic territory) or logical environment (e.g. network address or system environment); (5) *aspect*, which mainly relates to the technical perspectives of the object, for example, quality and format of the content; (6) *purpose*, for instance, educational purpose and commercial reason. There may be more unique constraints required when new scenarios emerge.

We introduce another constraint, namely (7) the *status* constraint. Real time content access requires this constraint to indicate the current state, e.g. availability and accessibility of the content at the time the rights are exercised.

XrML and ODRL are able to represent render, reuse, transport, and object management rights. However, XrML and ODRL do not accommodate (explicitly) the descriptions of rights that manage other rights. For example, "a user can *renew* the rights to play a movie within a fixed period (after the expiry time of the rights) with a discount". However, XrML and ODRL do cater for the revocation of rights and obligations. XrML does not provide explicit facilities to specify the purpose constraints. ODRL and XrML cannot express the status of the object. As we shall see, LicenseScript is able to accommodate all the listed constraints.

2.2 Relations

A REL must specify *relations* between components. As can be seen in Fig. 2, there are two distinct types of relations, namely *explicit* relations and *implicit* relations. We use example 1 to elaborate some of the relations discussed in this section.

Parrott [2001] identifies two classes of explicit relations, namely *ordering* relation, e.g. "*pay* $5 before *play* the movie" (operation–operation); and *association* relation, "*Alice* owns the *movie*" (subject–object) and "*play* the *movie*" (operation–object). The *ordering* relation describes how operations are linked to each other. For example, "pay before play" is an example of antecedent obligation; and "play then pay" is an example of consequent obligation. An ordering can be total or partial. A total ordering fully specifies the order of all operations, for example, "register, pay and then play". A partial ordering implies that there is no explicit order between all items, for example, "register and then play, user can pay before or after". The *association* relation covers the subject–object and operation–object relations.

We identify three additional types of explicit relations, namely, (1) *naming* relation (subject–operation), which specifies the name of the operation the subject can perform, e.g. "*Alice plays* the music"; (2) *limitation* relation, which implies that the operations are restricted by the constraints, in the same example, (constraint–operation); and (3) *characteristic* relation (constraint–object), which describes the object (that the operations can be acted upon), e.g. "*high-quality movie*".

We also identify several implicit relations (see Fig. 2), which include: subject–subject, subject–constraint, object–object, and constraint–constraint. These implicit relations are embedded and indirect. To elaborate these relations, we use two additional examples:

Example 2. Alice needs Bob to prove her identity so that she can play the movie.

Example 3. Alice can reuse the image in the ebook on her Web site, for educational purpose and for 2 years.

Example 2 exhibits the implicit subject–subject relation between "Alice" and "Bob", as well as implicit subject–constraint between "Alice" and "prove her identity". Example 3 exhibits implicitly the object–object relations between the "image" and "ebook", and the constraint–constraint relations between the "educational purpose" and "2 years".

2.3 Models

A *model* describes a typical way of using a REL; we can distinguish: (1) *revenue* model, (2) *provision* model, (3) *operational* model, (4) *contract* model, (5) *copyright* model and (6) *security* model. A rights management system typically fits different models simultaneously.

The *revenue* model, is normally related to the payment architecture of the system. There are many revenue models, for example, pay-per-use, pay-up-front, pay-flat-rate, tiered payment (e.g. free now pay later), pay to multi-entities (e.g. pay half to publisher and half to distributor), and fractional payment (e.g. discount and tax). New revenue models emerge every day.

The *provision* model may provide an *alternative solution* more than yes or no to the situations when the rights and obligations fail to meet the constraints. For instance, if viewing a high-resolution video is not allowed, it should be possible to switch to low-resolution video. Additionally, the provision model should be able to *reconcile the conflicts* caused, for example, when there is more than one subject performing the same

operation on the same object simultaneously. The provision model also accommodates the *default settings of operations* over an object when the object is not associated with any operations.

The *security* model defines a variety of security mechanisms, for instance, identification, authentication and authorization (IAA), access control, non-repudiation, integrity, audit trails and privacy.

The *operational* model handles the technological aspects of the system, such as quality-of-service, watermarking, caching, network operations, bandwidth and other operational aspects of the system.

The *contract* model establishes the agreement of the terms and conditions (over the operations offered over the object and constraints) established between different subjects.

We include the *copyright* model in this category because the copyright enforcement from the user's standpoint is always a source of controversy [Camp, 2002]. The *copyright* model enforces copyright acts (especially from the end-user's standpoint), such as fair use, first sale and so on.

Not all RELs are able to support the 6 revenue models above. XrML and ODRL are not able to support the provision model of reconciling the rights conflicts. This model handles the dynamic license evolutions and content access patterns. XrML and ODRL are static RELs that are not sufficiently flexible to meet this requirement. None of the RELs can as yet support the copyright model [Samuelson, 2003]. However, Mulligan and Burstein [2002] provide several suggestions to incorporate copyright into the XML-based RELs.

In next section, we introduce the core language of LicenseScript. We will show that LicenseScript supports all features of a REL discussed earlier.

3 LicenseScript

We now introduce our REL LicenseSript. As mentioned earlier, LicenseScript is based on multiset rewriting. We start this section with the description of the core language.

3.1 Preliminaries on Logic Programming

For the sake of completeness, we first shortly recall the basic results and notation of Logic Programming that will be used in the rest of the paper. The reader is assumed to be familiar with the terminology and the basic results of logic programs [Apt, 1990, 1997; Lloyd, 1987].

In what follows we often refer to pure prolog programs, i.e., to definite (negation-free) logic programs executed by means of *LD-resolution*, namely SLD-resolution with the leftmost selection rule (the Prolog selection rule). We work with *queries* that are *sequences* of atoms, $B_1, \ldots, B_n$, instead of *goals*. Apart from this, we use the standard notation of Lloyd [1987] and Apt [1997].

Let $\mathcal{T}$ be the set of terms built on a finite set of *data constructors* $\mathcal{C}$ and a denumerable set of *variable symbols* $\mathcal{V}$. A *substitution* θ is a mapping from $\mathcal{V}$ to $\mathcal{T}$ such that $Dom(\theta) = \{x|\ \theta(x) \neq x\}$ is finite. A substitution $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ is called *grounding* if $t_1, \ldots, t_n$ are ground (variable-free) terms and is called a *renaming* if $t_1, \ldots, t_n$ is a permutation of $x_1, \ldots, x_n$. For any syntactic object o, we denote by $Var(o)$ the set of variables occurring in o. The *composition* $\theta\sigma$ of the substitutions θ and σ is defined as the functional composition, i.e., $\theta\sigma(x) = \sigma(\theta(x))$. The result of the application of a substitution θ to a term t is said an *instance* of t and it is denoted by $t\theta$. We denote by *mgu*(t, t') any *most general unifier* (*mgu*, in short) of t and t'. A substitution σ is called a *matching substitution* of terms t and s if $t\sigma = s$, and *Dom*$(\sigma) =$ *Var*(t). In that case, we say that t *matches* s. If a term matches with another one, then it follows that there exists a unique matching substitution.

Let $\mathcal{P}$ be a finite set of *predicate symbols*. An *atom* is an object of the form $p(t_1, \ldots, t_n)$ where $p \in \mathcal{P}$ is an n-ary predicate symbol and $t_1, \ldots, t_n \in \mathcal{T}$. Given an atom A, we denote by $Rel(A)$ the predicate symbol of A. A *query* is a finite, possibly empty, sequence of atoms $A_1, \ldots, A_m$. The empty query is denoted by $\Box$. A *clause* is a formula $H : -B_1, \ldots, B_n$ where H is an atom (the *head*) and $B_1, \ldots, B_n$ is a possibly empty query (the *body*). When the body is empty, the clause is written $H.$ and is called a *unit clause*. A *program*

is a finite set of clauses. We denote atoms by $A, B, H, \ldots$, variables by $x, y, z \ldots$ (when referring to actual programs, we will use the standard Prolog notation in which variables start with an uppercase letter), clauses by $c, d, \ldots$, and programs by P.

Computations are constructed as sequences of resolution steps as follows. Consider a non-empty query $A_1, \ldots, A_m$ and a clause c. Let $H : -B_1, \ldots, B_n$ be a variant of c, which is variable disjoint from $A_1, \ldots, A_m$. Let A_1 and H unify with mgu θ. The query $(B_1, \ldots, B_n, A_2, \ldots, A_m)\theta$ is called a *resolvent of* $A_1, \ldots, A_m$ *and* c *with mgu* θ. A *derivation step* is obtained by resolving the (given) query with any clause in P and is denoted by

$$A_1, \ldots, A_m \stackrel{\theta}{\Longrightarrow}_P (B_1, \ldots, B_n, A_2, \ldots, A_m)\theta$$

The clause $H : -B_1, \ldots, B_n$ is called its *input clause*. A derivation is obtained by iterating derivation steps. A maximal sequence of derivation steps

$$\delta := Q_0 \stackrel{\theta_1}{\Longrightarrow}_{P,c_1} Q_1 \stackrel{\theta_2}{\Longrightarrow}_{P,c_2} \cdots Q_n \stackrel{\theta_{n+1}}{\Longrightarrow}_{P,c_{n+1}} Q_{n+1} \cdots$$

where each Q_i is a query, is called a *derivation of* $P \cup \{Q_0\}$ provided that for every step the standardization apart condition holds, i.e., the input clause employed is variable disjoint from the initial query Q_0 and from the substitutions and the input clauses used at earlier steps. A derivation ending in the empty query is called *successful*, and given a successful derivation $\delta := Q_0 \stackrel{\theta_1}{\Longrightarrow}_{P,c_1} \cdots \stackrel{\theta_n}{\Longrightarrow}_{P,c_n} \square$, we call the composition of the resolving substitutions $\theta = \theta_1 \cdots \theta_n$, the *computed answer substitution* of $P \cup \{Q_0\}$ (*c.a.s.*, for short).

If the leftmost atom in a query does not unify with the head of any clause in P, we say that the query *fails* (in P).

Definition 4. Given a program P, and a query (i.e., a conjunction of atoms) Q, we write $P \vdash_{LD} Q$ (or simply $P \vdash Q$) when there is a successful LD-derivation for $P \cup Q$.

3.2 Licenses

In our approach, licenses are stored in a multiset; intuitively, one can think of each device as containing a multiset of licenses that can be used when necessary. A *multiset* (also known as a *bag*) is a set with possibly repeated elements, and is written using square brackets (e.g. $[a, b, b, c]$). The concepts of unification and matching trivially extend to multisets by taking into account multiplicities.

A license defines the terms and conditions of use for music, videos etc. Therefore, a license must contain at least two relevant pieces of information: (i) a reference to the *data* that is being licensed, and (ii) the *conditions of use* on that data. The conditions of use are specified (to some extent) by a prolog program. It is well-known that Prolog – thanks to its declarative reading – is very suitable for defining access control policies [DeTreville, 2002]. On the other hand, Prolog is not suitable to model *change of state*, which is a feature we also need; consider for instance at a license stating that a given piece of music might be played only a given number of times. To implement it, we need to use a counter keeping track of how many times the music has been played already. This cannot be done in pure logic programming, a paradigm where variables cannot be re-instantiated and in Prolog the only way to store a state is by asserting and retracting new unit clauses. This, however, leads to a disruption of the declarativity of the language.

Here, we solve this problem by storing temporary values in a separate set of *bindings*, which contain only the mutable values the prolog policies need to refer to.

Definition 5. A *license* is a term of the form $lic(content, \Delta, B)$ where:

- *content* is a unique identifier representing the data the license refers to.
- Δ is a set of *clauses*, i.e., a Prolog program.

- B is a set of *bindings*, i.e., a set containing elements of the form $name \equiv value$, where both $name$ and $value$ are ground terms.

For instance $\{expires \equiv 10/10/2003\}$ is a set of bindings with just one element. As explained above, bindings provide us with a flexible way of storing modifiable data. A license could be regarded as a database in which Δ is the intensional part, while B is the extensional part. To interface licenses with the external world, we have to define a set of reserved calls that form the "API" of the license. In this paper we suggest a minimal API, the full API is implementation dependent. Before we define the precise operational semantics of LicenseScript (which involves the use of multiset rewriting *rules*), let us give an intuition of how clauses and bindings are related to each other: In the sequel, we use *canplay(·)* to indicate when a license allows a given piece of music to be played: if the query $canplay(B, B')$ succeeds in the program Δ, this means that the license $lic(a, \Delta, B)$ allows the content a to be played. Notice that we passed the set of bindings B as an argument to the query; after $canplay(B, B')$ has succeeded, B' will contain the new bindings of the license.

Example 6.

1. The license $lic(mus, \{canplay(X, X) : -true.\}, \{\})$ allows one to play mus, and playing the music does not modify the bindings.

2. The license $lic(mus, \{\}, \{\})$ does not allow any operation on mus.

3. The license $lic(a, \Delta, \{expires \equiv 10/10/2003\})$ where Δ is consists of the single clause

   $$canplay(B, B) : -today(D),\ get\_value(B, expires, Exp), Exp > D.$$

   allows one to play *a* until the given expiration day. $today(D)$ and $get\_value(B, n, V)$ are two primitives of our minimal API that work as follows: $today(D)$ binds the variable D to the current system date, while $get\_value(\ B, n, V)$ reports in V the value of the name n according to the set of bindings B. In the remainder, we gather all such primitives in a special program that we call the *domain*, denoted D.

4. In some situations the "execution" of a license should be followed by a change in the bindings, like in the case of a license that allows to play a piece of music only a given number of times: every time a *play* action is carried out, a counter should be incremented. This is done by means of the primitive $set\_value(OldB, name, value, NewB)$, which associates $name$ with a new value $value$ in the "new" bindings $NewB$[1]. Consider now the following license: $lic(a, \Delta, \{played\_times \equiv 3\})$, where Δ consists of the following clause:

   $$\begin{aligned} canplay(B, B') \quad : - \quad & get\_value(B, played\_times, R), R < 10, \\ & set\_value(B, played\_times, R + 1, B'). \end{aligned}$$

   Here, we first fetch the value of $played\_times$. Then, if this value is less than 10, we increment $played\_times$ and store this new value in the set of bindings B', which is equal to B except for in the value of $played\_times$.

3.3 Multiset Rewrite Rules

So far, we have seen the structure of licenses, and we have said that licenses form a multiset which typically resides inside a device. What we now need to show is how licenses are actually addressed from the outside world.

[1] This primitive can be easily implemented in pure logic programming

The modeling of communication between devices and the licenses is done by means of *multiset rewrite rules*. These rules can be thought as the *firmware* of the device. Intuitively, while licenses are mutable (they can be acquired and deleted) rules are fixed into the device. The syntax of rules is similar to Gamma [Banâtre et al., 2001].

Definition 7. A rewrite rule is a 4-tuple

$$rule(arg) : l_{ms} \rightarrow r_{ms} \Leftarrow cond$$

where $rule(arg)$ is a term called the *rule label*, l_{ms} and r_{ms} are two multisets, and $cond$ is a sequence of elements of the form $P_i \vdash Q_i$.

An example of a rule is the following one. Let ms be the multiset of available licenses:

$$play(X) : lic(X, \Delta, B) \rightarrow lic(X, \Delta, B') \Leftarrow \Delta \vdash canplay(B, B')$$

Intuitively, this rule can be fired when a command matching with its head (like *play(bach)*) is given; when this happens the left-hand side of the rule ($lic(X, \Delta, B)$) is first matched against one of the licenses in ms (this instantiates Δ and B to the Prolog program and the bindings of the license, respectively), if no matching license is found, the action fails. Next, the query $canplay(B, B')$ is fired in the program Δ. If this query succeeds, the command *play(bach)* succeeds as well and $lic(X, \Delta, B)$ is replaced with $lic(X, \Delta, B')$ within the multiset.

More in general, when the rule $rule(arg) : l_{ms} \rightarrow r_{ms} \Leftarrow cond$ is fired, then first, the left multiset l_{ms} has to be matched to a sub-multiset of *ms*. Second, all the queries in $cond$ are fired. If they succeed, then the action succeeds and (the instance of) l_{ms} is replaced with the instance of r_{ms} within *ms*.

3.4 LicenseScript Execution Model

We formalize the meaning of rule execution in this section.

Definition 8. Execution Step. Let *ms* and *ms*′ be two multisets of licenses, and R be a set of rules, and let a be a ground term (the action). We write *ms* $\xrightarrow{a}_R$ *ms*′ iff there exists a rule $l : l_{ms} \rightarrow r_{ms} \Leftarrow cond \in R$, renamed apart wrt *ms* and substitutions $\sigma_1, \sigma_2, \delta_1, \cdots, \delta_n$ such that:

1. a matches with l, with matching substitution σ_1.
2. $l_{ms}\sigma_1$ matches with i_{ms}, with matching σ_2, for some sub-multiset i_{ms} of *ms* (so, $ms = i_{ms} \cup rest_{ms}$ for some multisets $rest_{ms}$ and $l_{ms}\sigma_1\sigma_2 = i_{ms}$).
3. After step 2, $cond\ \sigma_1\sigma_2$ has the form $P_1 \vdash Q_1, \cdots, P_n \vdash Q_n$, for some programs $P_1, \cdots, P_n$ and query $Q_1, \cdots, Q_n$. Now we require that for each $P_i \vdash Q_i$, in $cond\ \sigma_1\sigma_2$, $Q_i\delta_1 \cdots \delta_{i-1}$ succeeds in P_i, with computed answer substitution δ_i;
4. *ms*′ is the result of removing i_{ms} from *ms*, and adding the multiset $r_{ms}\sigma_1\sigma_2\delta_1 \cdots \delta_n$ to it.

Since R is always clear from the context, we omit explicit references to R and simply write *ms* $\xrightarrow{a}$ *ms*′. *Step 1* of this definition represents the choice of a rule for executing a given request action a (e.g., $play(mus)$) from the environment. In principle there may be more than one rule matching the request action. The request action *fails* immediately if no rule matches with the request. After a rule is chosen, in *Step 2* we fetch from *ms* the licenses i_{ms} that match with the left hand side of the rule. These licenses are needed to complete the process. There might different sub-multisets of *ms* matching the lhs of the rule This corresponds to the possible situation in which the user possesses more than one license allowing her to execute the desired action. In our theoretical treatment we consider this as a source of non-determinism; in

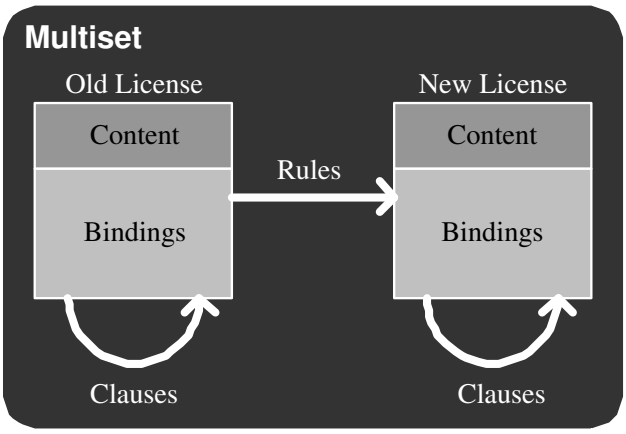

Figure 3: The transformation of licenses with *content* and *bindings* in a multiset caused by rules.

practice, however the system might want to ask the user which license should be used. *Step 3* checks that the conditions of the rule hold, by executing the associated queries. Each query generates a new substitution which is carried over to the next query. If any of the queries fail, the whole procedure fails. Finally, *Step 4* realizes the actual rewriting.

Example 9. Consider the following multiset: *ms* $= [lic(music, \Gamma, C), lic(video, \Sigma, D)]$, where $C = \{played\_times \equiv 2\}$, $D = \{played\_times \equiv 10\}$, and $\Gamma = \Sigma$ and contains the clause

$$canplay(B, B') : -get(B, played\_times, N), N < 10, set(B, played\_times, N + 1, B').$$

Let R be the singleton rule set containing the following rule:

$$play(X) \quad : \quad lic(X, \Delta, B) \rightarrow lic(X, \Delta, B') \Leftarrow \Delta \vdash canplay(B, B')$$

- Suppose the environment requests the action $play(music)$.
  1. First, this action will match rule $play(X)$, giving matching $\sigma_1 = \{X/music\}$.
  2. The next step involves looking for occurrences of $lic(music, \Delta, B)$ in *ms*. The only possible match is $lic(music, \Gamma, C)$. This gives us matching $\sigma_2 = \{\Delta/\Gamma, B/C\}$.
  3. Condition $\Gamma \vdash canplay(C, B')$ has to be evaluated. Since variable $played\_times$ is less than 10 in C, the query $canplay(C, B')$ succeeds in the Prolog program Γ, hence the condition is satisfied. We get the computed answer substitution $\delta_1 = \{B'/\{played\_times \equiv 3\}\}$.
  4. Finally, *ms* is updated. License $lic(music, \Gamma, C)$ is removed from *ms*, and replaced by $lic(music, \Gamma, C')$, where $C' = \{played\_times \equiv 3\}$.
- Suppose now request action $play(video)$ is issued. This action, even though it has a matching rule and a matching license in the multiset, fails. This is so since, in the unique matched license (that is, $lic(video, \Sigma, D)$), the condition $\Delta \vdash canplay(B, B')$ does not hold as the video has been played 10 times already.

Definition 8 shows a multiset rewritten after a license has been invoked. This brings us to the notion of *execution*.

Definition 10 (Execution). Given a multiset ms and a set of rules R, an *execution* is the (possibly infinite) sequence of rule applications $ms \xrightarrow{a_1} ms_1 \xrightarrow{a_2} ms_2 \cdots$. We refer to the sequence of actions $a_1 \cdot a_2 \cdots$ as the *trace execution* of ms.

The semantics of executing a multiset and a rule set is then defined as all possible trace executions, according to the above definitions.

4 Safety

The language we have introduced might be regarded as an access or usage control system (though it must be remarked that in LicenseScript, users are not first-class objects, unlike in the usual access and usage control systems). In this perspective, one of the characteristics worth investigating is whether or not it can be decided whether a configuration is safe with respect to a given action (or sequence of actions). Given that our semantics consists of the traces of possible actions that can be carried out starting in a given configuration, we say that the system is safe with respect to an action a if no trace starting in the current configuration contains a. More complex notions of safety (like availability, bounded safety) can also be considered (see e.g., Li et al. [2004]); the concepts we present in this section can be extended to them as well.

Before we continue, we would like to remark that in most access control systems, safety is undecidable; for instance Harrison et al. [1974] formalize a simple safety analysis for the HRU system to determine whether it can reach a state in which an (unwanted) access is allowed. They prove three safety results:

1. Safety is undecidable in general.
2. Safety is decidable for a mono-operational system (albeit **NP**-complete).
3. Safety is decidable within the **PSPACE**-complete complexity class with the restriction that no subjects or objects are allowed to be created.

Undecidability is shown by encoding a Turing machine in the HRU system.

In the case of LicenseScript, it is immediate that safety is undecidable in general. LicenseScript is an extension of gamma, and therefore it is Turing-complete. In fact, it is not even possible to restrict the system to one in which the safety problem is decidable without losing one of the essential features of LicenseScript: that one should be able to acquire new licenses. Since a license contains a whole new usage control policy, acquiring a new license has unpredictable consequences and may well spoil safety.

While safety is in general undecidable, it is useful to isolate (necessarily restrictive) situations in which this is not the case. Here we want to point out that there exist at least two techniques that can be used to prove the safety of a given configuration. The first is by using *invariants*, the second is by showing that the system is *terminating* and *finitely branching*. In this second case, it is possible to carry out a full search of the multisets space generated by the (repeated) application of the rules and thus to establish whether a given unsafe action is reachable. In the rest of this section we show how this can be done in practice, by borrowing some techniques from the study on the termination of logic programs.

Let ms be a multiset, R be a set of rules, and C be a set of commands (the admissible commands). In the light of what we said in the previous paragraph, we need to establish a number of conditions:

1. The starting multiset ms contains only terms of the form $i(o, c, b)$, where o is ground, c is a set of clauses, and b is a set of bindings. This condition is always met in practice.
2. Let

   $$R' = \{(r : l_{ms} \rightarrow r_{ms} \Leftarrow c)\delta \mid r : l_{ms} \rightarrow r_{ms} \in R \, and \, \delta = mgu(r, c) \; for \; some \; c \in C\}$$

   be the set of rules obtained from the rules in R by instantiating their heads with the commands in C. We assume that (R and C are such that) R' is finite.

   If this condition is not satisfied we immediately obtain a system that is not finitely branching. This condition is met if C is finite or R contains only rules with ground heads (one could avoid this restriction by applying abstract interpretation to the head's positions that might take an infinite number of arguments, but this is outside the scope of this paper).

3. Rules in R' have the form:

$$\begin{array}{rcl} r & : & a_1(o_1, c_1, b_1), \ldots, a_m(o_m, c_m, b_m) \rightarrow \\ & & i'_1(o'_1, c'_1, b'_1), \ldots, i'_n(o'_n, c'_n, b'_n) \rightarrow \\ & \Leftarrow & c_1 \vdash p_r(s_1, \ldots, s_k, b_1, \ldots, b_m, b'_1, \ldots, b'_n) \end{array}$$

   Where (1) for each i, s_i, o_i and o'_i are ground, while c_i, c'_i, b_i and b'_i are variables. In addition, (2) each c'_i is equal to some c_j.

   The condition (1) is always met in practical deployments of the system and simplifies the discussion, while (2) is needed to make sure that the programs carried by the licenses are not modified at run-time; notice that the bindings might be changed. Here, for the sake of simplicity, we also assumed that the "condition" contains only one query check $(c_1 \vdash p_r(s_1, \ldots, s_k, b_1, \ldots, b_m, b'_1, \ldots, b'_n)$; nevertheless, the reasoning applies as well to the general case in which we have $\cdots, c_i \vdash p_r(s_{i,1}, \ldots, s_{i,k_i}, b_{i,1}, \ldots, b_{i,m_i}, b'_{i,1}, \ldots, b'_{i,n_i})., \cdots$.).

4. Given: any rule r (notation as in point 3.), any term $a(o, c, b)$ in the initial multiset, any set of ground terms $s_1, \ldots, s_k, t_1, \ldots, t_m$, and distinct variables $x_1, \ldots, x_n$, we require that the query $p_r(s_1, \ldots, s_k, t_1, \ldots, t_m, x_1, \ldots, x_n)$ is terminating and finitely branching in c.

   This is needed to ensure that the decision process terminates and that the whole system remains finitely branching. In general, we expect the rules in c not to interact with the outside world, in which case we can exploit the fact that in pure logic programming a terminating system is always also finitely branching (in which case one can use standard LP techniques [Apt and Pedreschi, 1993; Bossi et al., 1991; De Schreye and Decorte, 1994; Etalle and Gabbrieli, 1999]).

5. There exists a level mapping $|\ | : \mathcal{B} \rightarrow \mathcal{W}$, where $\mathcal{B}$ is the set of all possible binding sets, and $\mathcal{W}$ is a set with a well-founded ordering $<$, such that given any rule r (notation as in point 3.), any term $a(o, c, b)$ in the initial multiset, and $b_1, \ldots, b_m, b'_1, \ldots, b'_n \in \mathcal{B}$, if $p_r(s_1, \ldots, s_k, b_1, \ldots, b_m, b'_1, \ldots, b'_n)$ succeeds in c then

   $$|b_1|, \ldots, |b_m| \succ_m |b'_1|, \ldots, |b'_n|$$

   Where $\succ_m$ is the (well-founded) multiset ordering induced by the ordering $>$ on $\mathcal{W}$[2]

We can finally state the result we are aiming at.

Proposition 11. *If the initial multiset ms, the set of rules R and the set of allowed commands C satisfy the above conditions $1, \ldots, 5$, then the system is terminating and finitely branching, and therefore safety is decidable.*

Proof (sketch). Each action of the system requires (a) selecting a rule, (b) selecting a multiset m_1 (the lhs of the rule), (c) firing a condition in a program c (contained in an term of m_1), (d) rewriting m_1 with m_2. Action (a) is terminating and finitely branching thanks to the fact that by condition 1 rules have no nonground arguments. Action (b) is clearly terminating (the multiset is finite). Notice that condition 3 guarantees that, no matter how the multiset is being rewritten, the set of programs (set of clauses in the licenses) we have to deal with does not change; therefore, (c) terminates because of condition 4. Finally, condition 5 guarantees that the new multiset is smaller than the original one according to a well-founded ordering. This implies that it is not possible to have an infinite chain of actions. QED.

[2] Quoting from [Apt, 1997]: The *multiset ordering* is an ordering on finite multisets of natural numbers. It is defined as the transitive closure of the relation in which X is smaller than Y if X can be obtained from Y by replacing an element a of Y by a finite (possibly empty) multiset of natural numbers each of which is smaller than a. In symbols, first we define the relation $\prec$ by: $X \prec Y$ iff $X = Y - \{a\} \cup Z$ for some $a \in Y$ and Z such that $b < a$ for $b \in Z$, where X, Y and Z are finite multisets of natural numbers and then define the multiset ordering $\prec_m$ as the transitive closure of the relation $\prec$.

5 Application Aspects

In this section, we explore different application domains of DRM, which include technical, business, commerce and legal aspects (as shown in Figure 4). These application aspects also represent the perspectives of different parties involved in a DRM value chain, such as the content providers and the consumers.

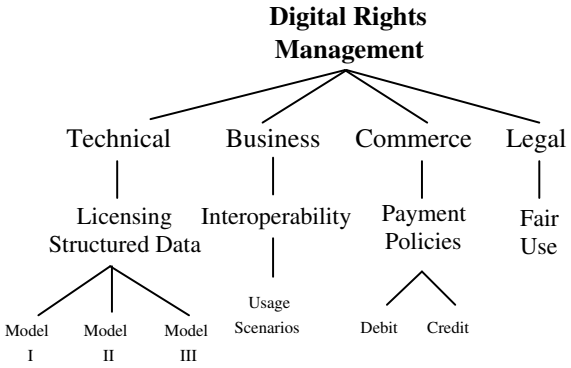

Figure 4: Exploring different application aspects of DRM with LicenseScript.

We use LicenseScript to handle the licensing of structured data in section 5.1. We introduce three different licensing models, showing that LicenseScript is technically capable of handling different business models.

Next, we study a wide variety of usage scenarios, which have been specified in XrML and ODRL [Chong et al., 2003b]. We model these usage scenarios by using LicenseScript in section 5.2 to show that LicenseScript can easily model typical usage scenarios and is interoperable with XrML and ODRL – a critical feature of DRM from a business perspective [Koenen et al., 2004].

Furthermore, we use LicenseScript to model two types of payment methods, namely debit and credit in section 5.3. We show by some examples that LicenseScript is able to specify different payment policies.

Finally, we use LicenseScript to model a legal aspect of DRM, namely fair use in section 5.4. Fair use allows users to *legally* exercise rights without abiding by the content provider's rules.

5.1 Licensing Structured Data

From a content provider's perspective, a REL should be able to capture different types of licensing models, corresponding to the content provider's requirements. In this section, we will show how the handling of structured data unleashes the power of Prolog as a component of LicenseScript.

An example of structured data is the digital library. We define a digital library as a structured collection of multimedia resources, e.g. electronic documents, music files, movie files etc. Not all or parts of the library need to be online. The practical requirement of transparent and seamless access does not concern us here. Based on this scenario, we propose three content licensing models, which exploit salient features of Prolog. The licensing models themselves are not necessarily new, what *is* new is how we put them in the context of digital licensing, and justifying our approach to REL at the same time.

5.1.1 Licensing Model I

The first model is based on *locking* and *unlocking*. In this model, an imaginary content provider distributes a digital library for free. To optimize the use of bandwidth, the digital library is distributed in parts. However, some of these parts are locked. Locked content allows only for a *restricted* access, for example, if it is an

audio resource, it is not playable, or only playable at low sampling rate and/or not to the full length. This licensing model works by arousing the user's interest in the locked parts of the digital library.

We now show how the licenses can be constructed. First, a locked audio resource can be associated with a license of the form:

$$
\begin{array}{l}
lic(track1, \\
\{canunlock(B, \_, Code) :- \\
\quad get\_value(B, unlock\_code, Unlock\_code), Code == Unlock\_code.\}, \\
\{unlock\_code \equiv 1234\})
\end{array}
$$

Call this the *base license*. Here, $track1$ is just an arbitrary resource identifier and the $canunlock$ clause provides an interface for inputting an unlocking code, through the $Code$ parameter. The clause checks if the input code is valid by comparing it to the (hashed) value of the $unlock\_code$ binding. To unlock this resource, a user has to obtain an *unlocking license* from the content provider of the form:

$$
\begin{array}{l}
lic(track1, \\
\{canplay(B, \_, Code) :- \\
\quad get\_value(B, unlock\_code, Code') \; where \; Code' \; is \; Hash(Code).\}, \\
\{unlock\_code \equiv 1234\})
\end{array}
$$

Notice that the $canplay$ clause is in the unlocking license instead of in the base license. The $canplay$ clause seems readily satisfied, but the following $play$ rule ensures that the $play$ action is only allowed if the $unlock\_code$ of the unlocking license matches with the $unlock\_code$ of the base license:

$$
\begin{array}{l}
play(N) \; : lic(N, C1, B1), lic(N, C2, B2) \longrightarrow lic(N, C1, B1'), lic(N, C2, B2') \\
\qquad \Leftarrow C1 \vdash canunlock(B1, B1', Code), C2 \vdash canplay(B2, B2', Code)
\end{array}
$$

Observe that there are two licenses in each of the multisets (i.e. the left and right multisets): $lic(N, C1, B1)$ for the base license and $lic(N, C2, B2)$ for the unlocking license. The multisets accommodate as many licenses as needed to be checked. The same applies to the condition, and additional information is allowed to flow from one license to another (e.g. through the $Code$ variable). This information flow is made particularly easy by Prolog, since Prolog's *unification* is a two-way matching process.

In this example, the advantage of having two sub-licenses, i.e. a base license and an unlocking license, for a single resource is that we can avoid changing the base license and put all volatile information like expiry date or usage limits in the unlocking license. Renewing the license for the resource means that only the unlocking license needs to be renewed.

5.1.2 Licensing Model II

Our second licensing model is inspired by the current computer games market. For example, most multiplayer games on the market nowadays are engineered in such a way that the gamers themselves can easily extend the games in terms of content and ways of playing, by building so-called 'mods' (short for 'modifications') on top of the games. These mods are usually free, so part of the reason a customer buys these games is due to her anticipation of the up-coming free mods. Moreover, the game publisher themselves are usually expected to release free 'bonus packs' from time to time. Transplanting such model to the context of digital licensing will certainly bring some fresh air into how we can use RELs.

Now consider the context of the digital library. Imagine a licensing model in which the content provider charges for basic content but gives away the bonus content that can be added on top of the basic content for free. An example is a digital library of maps. Suppose a basic set of maps only contains route information,

a potential free add-on to the maps might be directions for tourists, or annotations of public facilities, and so on. It is not for us to argue from a business perspective whether this model will work, but from a technical perspective, we can readily support this model with LicenseScript.

The first observation is that the license of an add-on depends on the license of the basic content P, whatever P is. However, since *every single license has a separate namespace*, i.e. a license l_1 cannot reference the bindings of another license l_2 to find out if l_2 is appropriate, we have to combine l_1 and l_2 to see if the resultant combined license is valid. The following sample licenses for the add-on and the basic content will make things clearer:

$$lic(tourist\_attractions,\\ \{canrender(B, \_) : -get\_value(B, acceptAddon, AcceptAddon), AcceptAddon == 1.\},\\ \{\})$$

$$lic(basic\_map,\\ \{canrender(\_, \_)\},\\ \{acceptAddon \equiv 1\})$$

Notice that the $canrender$ clause in the add-on's license explicitly refers to the binding $acceptAddon$, which does not exist in the add-on's license itself, and yet exists in the license of the basic content! Therefore to allow the add-on's $canrender$ clause to access the $acceptAddon$ binding, we have to combine the bindings of the two licenses. Also, by the logic that "the combination of basic and add-on content can be rendered iff the basic and the add-on content can individually be rendered", we also need to combine the $canrender$ clauses of the two licenses. The resultant combined license would look like:

$$lic(enhanced\_map,\\ \{canrender(B, \_) : -get\_value(B, acceptAddon, AcceptAddon), AcceptAddon == 1.\},\\ \{acceptAddon \equiv 1\})$$

The *algorithm* of this combination operation cannot be encoded in the licenses themselves nor in the rule, and we must handle this in the license interpreter. At this point, we must stress that this is *not* a limitation of LicenseScript, but in fact a feature, because by writing these $canrender$ constraints in Prolog, we can combine the clauses (almost) as easily as concatenating the bodies of the clauses. In the following, we give a formalization of the combination operation, starting with the definition of names:

Definition 12. Given a license $lic(N, C, B)$, the set of *names* of B is the set of names of the bindings in B. Similarly, the set of names of C is the set of all symbols representing the head of a clause (head symbols) in C, and the set of names referring to the bindings in B. By convention, we write $\eta(P)$ to represent the set of names of P.

Definition 13. Given two sets of names X and Y, the *renaming function* α is a bijection from X to Z such that $Z \cap Y = \emptyset$. We write $z \equiv \alpha(x, Y)$ by convention where $z \in Z$ and $x \in X$.

Definition 14. Let $N_1 \equiv \eta(C_1) \cup \eta(B_1)$, $N_2 \equiv \eta(C_2) \cup \eta(B_2)$ and $N_3 \equiv \eta(C_3) \cup \eta(B_3)$. A *combination operation* on two licenses $l_1 = lic(N_1, C_1, B_1)$ and $l_2 = lic(N_2, C_2, B_2)$ is a function φ that maps l_1 and l_2 to the combined license $l_3 = (N_3, C_3, B_3)$, such that the following conditions are satisfied:

- l_3 is a valid license;
- $|\eta(B_3)| = |\eta(B_1)| + |\eta(B_2)|$;
- The number of clauses in C_3 equals the number of clauses in C_1 plus the number of clauses in C_2 minus the number of clauses having the same head symbol in both C_1 and C_2;

- $\forall b \in N_1$ and $b \notin N_2$, $\exists! b_3$ such that $b_3 = b$ and $b_3 \in N_3$; if b is a head symbol, then the corresponding Prolog clause in C_1 also exists in C_3;
- $\forall b \in N_2$ and $b \notin N_1$, $\exists! b_3$ such that $b_3 = b$ and $b_3 \in N_3$; if b is a head symbol, then the corresponding Prolog clause in C_2 also exists in C_3;
- $\forall b \in N_1 \cap N_2$, $\exists! b_{3(1)}$, $\exists! b_{3(2)}$ such that $b_{3(1)} = \alpha_1(b, N_1 \cup N_2)$, $b_{3(2)} = \alpha_2(b, N_1 \cup N_2)$, $b_{3(1)} \neq b_{3(2)}$ and $b_{3(1)}, b_{3(1)} \in N_3$; if b is a head symbol, then the clause b:-$b_{3(1)}, b_{3(2)} \in C_3$ (the parameter/argument lists have been left out for clarity).

The definition may look daunting but the only difficult part of the combination operation is resolving name conflicts. In our example, there is no name conflict, so the combination operation does not need to invoke the renaming function α. If the constraints are coded in any imperative programming language, the combination operation might not be as straightforward, if possible at all. Observe also that the combination operation has obviated the need for a license to explicitly refer to another license by name.
To summarize, the implication of this approach is that

1. By taking advantage of the structural relationship between data units, we have avoided the need for static naming, and enabled a dynamic referencing mechanism.
2. Without static naming, the content providers can now introduce more products without incurring unnecessary management overhead.
3. With dynamic referencing, context-based instead of fixed licensing schemes can now be easily realized.

5.1.3 Licensing Model III

In fact, the combination operation comes in so handy that we easily have Licensing Model III. In this model, the content provider licenses two parts of a resource separately, for example the text part and audio part of a resource. An example of such a resource might be an multimedia e-book. With the text part, a user can only, obviously, read; while with the audio part, the user can listen to the e-book like an audio book. The selling point is when we combine the text part and the audio part, we can get an e-book with *synchronized* text and audio. Such an e-book potentially appeals very much to a language learner who would likely be interested in not only following the text but also mastering the pronunciation. Another example might be karaoke, some users might only be interested in the lyrics which is the text part, others might only be interested in the music video which is the multimedia part, but most users would probably like a synchronized whole, in the form of a karaoke with synchronized music video and lyrics display.

For the e-book example, we have a license for the text part:

$$lic(ebook\_text, \{candisplaytext(\_, \_)\}, \{\})$$

and a license for the audio part:

$$lic(ebook\_audio, \{canrenderaudio(\_, \_)\}, \{\})$$

To activate the synchronized rendering feature, the e-book renderer combines the two licenses and obtain:

$$\begin{array}{l} lic(ebook, \\ \{(candisplaytext(\_, \_)), \\ \ (canrenderaudio(\_, \_)), \\ \ (cansyncrender(\_, \_) :- \\ \quad candisplaytext(\_, \_), canrenderaudio(\_, \_))\}, \\ \{\}) \end{array}$$

As can be seen, the $cansyncrender$ clauses ensures that the feature is only activated if both the right to display text ($candisplaytext$) and the right to render audio ($canrenderaudio$) are licensed. In this example, $candisplaytext$ and $canrenderaudio$ are trivial on purpose, because our main idea is to show how easy it is to merge licenses by just using the technique of combination. The rule for $syncrender$ is given below for completeness:

$$syncrender(N) : lic(N, C, B) \longrightarrow lic(N, C, Bprime)$$
$$\Leftarrow C \vdash cansyncrender(B, Bprime)$$

To conclude this section, we have proposed three licensing models that make good use of Prolog as a component of LicenseScript. Indeed, to construct these licenses using other RELs would require much more effort compared with LicenseScript, if currently possible at all.

5.2 General Scenarios

To assess LicenseScript with respect to XrML and ODRL, i.e. the mainstream DRM languages, we have studied 20 scenarios taken from the XrML and ODRL documentation and we have encoded them in LicenseScript. In this section we report on these experiments.

5.2.1 XML-based RELs Usage Scenarios

We have translated the scenarios described in XML-based RELs documentation in LicenseScript. However, due to space constraints, for the LicenseScript code we refer to our previous work [Chong et al., 2003b]. The usage scenarios that we have studied and translated into LicenseScript are:

- ODRL scenarios, which are specified in the ODRL specification (`http://odrl.net/1.1/ODRL-11.pdf`): ebook #1 (E1), #2 (E2), #3 (E3), video (VD), super distribution (SD), software(SW), image(IM) and audio(AD);
- XrML scenarios, which are specified in XrML Use Case Examples (`http://www.xrml.org/spec/2001/11/ExampleUseCases.htm`): preview/ promotional (PP), subscription (SB), territory restriction (TR), temporal ordering of rights (TO), usage of part of a work (PW), site license (SL), personal lending (PL) and giving (GV), super distribution (SD), unrestricted sales (US), personal copies (PC), web service access (WS), software execution (SW), confidentiality of rights (CR), operational model (OM) and secure device (SV).

An analysis of the scenarios is summarized in Table 1 and Table 2.
In the next section, we analyze the interoperability of LicenseScript with ODRL and XrML.

5.2.2 Interoperability of LicenseScript

Table 3 shows the capabilities of XML-based RELs and LicenseScript from our studies of the usage scenarios. We put a '✓' where we can show that XML-based RELs and LicenseScript cover the corresponding feature of the REL. Our conclusion is as follows.

LicenseScript can render most of the aspects of XML-based RELs, for instance, payment, user authentication etc. We use user authentication as an example: in XML-based RELs, the user identity (e.g. certificate, public key, etc.) is described by XML tags. In LicenseScript, we use license bindings to express the user identity, certificate and keys, and (in)equalities in the license clauses to model user authentication explicitly.

In addition, LicenseScript can capture the dynamic transformation of licenses (along with the content) caused by the user operations, using the multiset rewrite rules. For instance, we can model the transformation

Property	Scenario							
	E1	E2	E3	VD	SD	SW	IM	AD
Right-Render	✓	✓	✓	✓	✓	✓	✓	✓
Right-Reuse			✓			✓		
Right-Transport			✓		✓			✓
Right-Manage Object								
Right-Regulation Right								
Obligation	✓	✓		✓		✓		
Object-General	✓	✓	✓	✓	✓	✓	✓	✓
Object-Class								
Object-Delivery	✓	✓		✓		✓		✓
Object-Fuzzy								
Subject	✓	✓	✓	✓	✓	✓	✓	
Constraint-Temporal			✓		✓	✓	✓	
Constraint-Bound	✓	✓	✓	✓		✓	✓	
Constraint-Environment		✓	✓			✓		
Constraint-Aspect			✓	✓				
Constraint-Purpose								✓
Constraint-Status								
Relation-Ordering-Ant. Obligat.	✓	✓		✓		✓		
Relation-Ordering-Con. Obligat.								
Relation-Ordering-Total								
Relation-Ordering-Partial								
Relation-Association	✓	✓	✓	✓	✓	✓	✓	✓
Relation-Naming	✓	✓	✓	✓	✓	✓	✓	
Relation-Limitation			✓		✓	✓	✓	
Relation-Characteristic			✓	✓				
Model-Revenue	✓	✓		✓		✓		
Model-Provisional-Conflicts								
Model-Provisional-Alternative								
Model-Provisional-Default								
Model-Operational								
Model-Contract	✓	✓	✓	✓	✓	✓	✓	✓
Model-Copyright								
Model-Security-IAA	✓	✓	✓	✓	✓	✓	✓	✓
Model-Security-Access Control								
Model-Security-Confidentiality								
Model-Security-Nonrepudiation								
Model-Security-Integrity								
Model-Security-Audit Logging								

Table 1: The properties of the usage scenarios specified in ODRL (The symbol '✓' indicates the scenario exhibits the corresponding feature).

of an offer to a license, when the user *agrees* on the terms and conditions stated in offer. We can also capture the semantics of license evolution when the license is copied, clipped or mixed, etc.

LicenseScript syntax is more compact and readable than the XML-based syntax of ODRL and XrML. From our study, we can translate XrML or ODRL licenses into LicenseScript licenses easily due to the declarative and procedural expressivity of Prolog. Since our LicenseScript Interpreter, discussed in Section 7, is built on top of a generic Prolog engine, we can make use of LicenseScript as an intermediate language, and by using some available techniques for mapping XML-based languages to Prolog [Boley, 2001; Eyers et al., 2002], the interoperability of LicenseScript with XML-based RELs can readily be realized.

5.3 Payment Modeling

In this section, we show how LicenseScript can be used to model electronic wallets. We show this by means of specifying payment policies in LicenseScript, with the help of a few realistic examples.

5.3.1 Wallets

Each money source, or "wallet", is represented as a term of the form $wallet(\Gamma, B)$,[3] where Γ are the clauses (as in the licenses above) and B the bindings of the wallet. We assume that B always contains at least a binding *money* $\equiv M$, representing how much money the wallet contains.

Example 15. Consider wallet $wallet(\Gamma, B)$ where $B = \{type = bank\_account, money = 1500\$, interest =$

[3] $wallet(\Gamma, B)$ can be considered a shorthand for $lic(money, \Gamma, B)$.

Property	Scenario															
	PP	SB	TR	TO	PW	SL	PL	GV	SD	US	PC	WS	SW	CR	OM	SV
Right-Render	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Right-Reuse					✓				✓							
Right-Transport							✓	✓	✓		✓					
Right-Manage Object											✓	✓				
Right-Regulation Right																
Obligation	✓	✓	✓		✓			✓	✓					✓		✓
Object-General	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Object-Class			✓		✓											
Object-Delivery	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓					
Object-Fuzzy																
Subject	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Constraint-Temporal			✓	✓	✓	✓	✓	✓		✓	✓					✓
Constraint-Bound											✓				✓	
Constraint-Environment				✓									✓		✓	✓
Constraint-Aspect															✓	
Constraint-Purpose																
Constraint-Status																
Relation-Ordering-Ant. Obligat.	✓	✓			✓			✓	✓							
Relation-Ordering-Con. Obligat.																
Relation-Ordering-Total			✓	✓												
Relation-Ordering-Partial				✓								✓				
Relation-Association	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Relation-Naming	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Relation-Limitation			✓	✓	✓	✓	✓	✓		✓	✓					✓
Relation-Characteristic															✓	
Model-Revenue	✓	✓		✓	✓			✓	✓							
Model-Provisional-Conflicts																
Model-Provisional-Alternative																
Model-Provisional-Default																
Model-Operational															✓	
Model-Contract	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓	
Model-Copyright																
Model-Security-IAA	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Model-Security-Access Control																
Model-Security-Confidentiality														✓		
Model-Security-Nonrepudiation	✓	✓		✓	✓	✓				✓	✓	✓			✓	✓
Model-Security-Integrity				✓						✓						
Model-Security-Audit Logging					✓		✓	✓	✓	✓						

Table 2: The properties of the usage scenarios specified in XrML (The symbol '✓' indicates the scenario exhibits the corresponding feature).

0.5%, *bank_charges* $= 1\$\}$ and Γ contains clauses to load and transfer money:

$$
\begin{aligned}
&\mathit{canload}(B, B', A) \ :- \\
&\qquad \mathit{get\_value}(B, m, M), \\
&\qquad \mathit{set\_value}(B, m, M + A, B'). \\
&\textit{cantransfer}(B, B', P, A) \ :- \\
&\qquad \mathit{get\_value}(B, m, M), \\
&\qquad \mathit{get\_value}(B, \textit{bank\_charges}, C), \\
&\qquad C + A \leq M, \\
&\qquad \mathit{set\_value}(B, m, M - (A + C), B'), \\
&\qquad \textit{transfer}(P, A).
\end{aligned}
$$

Here, A is the amount of money the user wants to load onto the wallet and the primitive *transfer*(P, A) models the money transfer to entity P of the amount of money A.

In the rest of this paper, we assume that each wallet has a clause defining *cantransfer*.

In the multiset of LicenseScript, all wallets of a user are gathered in a special term, namely the *wallet manager*, denoted $wm(\Psi, L)$. Here, L is the list of wallets, while Ψ contains clauses that operate over the wallets. For example, a valid $wm(\Psi, L)$ is one where $L = [wallet(\Gamma, B)]$ as in Example 2 and Ψ contains clauses for managing wallets (e.g. *addwallet* and *removewallet*, which are straightforward and therefore omitted in the presentation.

Property	REL		
	XrML	**ODRL**	**LS**
Right-Render	✓	✓	✓
Right-Reuse	✓	✓	✓
Right-Transport	✓	✓	✓
Right-Manage Object	✓		✓
Right-Regulation Right			✓
Obligation	✓	✓	✓
Object-General	✓	✓	✓
Object-Class	✓		✓
Object-Delivery	✓	✓	✓
Object-Fuzzy			
Subject	✓	✓	✓
Constraint-Temporal	✓	✓	✓
Constraint-Bound	✓	✓	✓
Constraint-Environment	✓	✓	✓
Constraint-Aspect	✓	✓	✓
Constraint-Purpose		✓	✓
Constraint-Status			✓
Relation-Ordering-Ant. Obligat.	✓	✓	✓
Relation-Ordering-Con. Obligat.			✓
Relation-Ordering-Total	✓		✓
Relation-Ordering-Partial	✓		✓
Relation-Association	✓	✓	✓
Relation-Naming	✓	✓	✓
Relation-Limitation	✓	✓	✓
Relation-Characteristic	✓	✓	✓
Model-Revenue	✓	✓	✓
Model-Provisional-Conflicts			✓
Model-Provisional-Alternative			
Model-Provisional-Default			
Model-Operational	✓		✓
Model-Contract	✓	✓	✓
Model-Copyright			
Model-Security-IAA	✓	✓	✓
Model-Security-Access Control			✓
Model-Security-Confidentiality	✓		✓
Model-Security-Nonrepudiation	✓		✓
Model-Security-Integrity	✓		✓
Model-Security-Audit Logging	✓		✓

Table 3: A comparison of properties supported by XML-based RELs and LicenseScript (LS). '✓' indicates the property is supported by the corresponding REL.

5.3.2 Payment Policies

Now that we have the wallets and the wallet manager, we need to specify a payment policy that decides how to perform the payments. More specifically, we need a *weight* predicate $p(Wallet, Weight)$, that assigns to each wallet $Wallet$ a value $Weight$ (s.t. $Weight$ is a real number in the $\{0, 1\}$ interval.) A wallet with higher weight assigned by p is preferred for payment over a wallet with lower weight. Given the list of wallets L from $wm(\Psi, L)$, we say that payment policy p selects wallet $wallet(\Gamma, B)$ for payment if $p(wallet(\Gamma, B), W)$ and $W = max(\{W_i \mid w \in L \wedge p(w, W_i)\})$.

5.3.3 Implementing payment policies in LicenseScript

To implement the selection procedure of a wallet by a payment policy in LicenseScript, we proceed as follows: First, we create a special term that resides in the multiset, *policy(*Γ*)*. In Γ we have two clauses:

- First, the weight predicate $p(Wallet, Weight)$ that assigns weight $Weight$ to wallet $Wallet$. This predicate can be implemented in Prolog easily, and depends on our payment requirements. We illustrate three different choices for p:

    1. *Random policy.* We first model the simplest policy of Chong et al. [2003a]: we just look for a wallet with enough money, and perform the payment. Let $p(Wallet, Weight) : -random(Weight)$ where $random(R)$ returns a random number between $\{0, 1\}$ in R. Intuitively, this weight predication assigns *any* weight to a wallet, thus modeling the free choice. Notice that, alternatively, defining $p(Wallet, c).$ with $c \in \{0, 1\}$ would also have been possible.

2. *Minimum bank charges.* Consider now the following stronger requirement: we would like to find the wallet with *minimum* bank charges to perform the payment. Suppose binding *bank_charges* $\equiv$ C is in B for wallet $wallet(\Gamma, B)$, and $C > 0$. We set

$$\begin{aligned} p(wallet(\Gamma, B), Weight) : - \\ \quad get\_value(B, bank\_charges, C), \\ \quad Weight = \frac{1}{C}. \end{aligned}$$

Thus, $p(Wallet, Weight)$ will set as $Weight$ of wallet $Wallet$ the inverse of the bank charges of $Wallet$; The higher the bank charges are, the lower the weight that p assigns.

3. *Paying with loyalty points.* Suppose we have wallets of three kinds: *airmiles*, *moneycoupon*, and *bankaccount*, specified in the *type* binding as in Example 1. We would like to specify a payment policy in which we want to choose, for payment, first wallets of kind *airmiles*. If no wallet of that kind is possible to pay, we want to choose *moneycoupon*. Otherwise, we want to choose *bankaccount*. A weight predicate p can be defined as follows:

$$\begin{aligned} p(wallet(\Gamma, B), Weight) : - \\ \quad get\_value(B, type, Type), \\ \quad Weight = \begin{cases} 1 & if\ Type = airmiles \\ 0.5 & if\ Type = moneycoupon \\ 0 & if\ Type = bankaccount \end{cases} \end{aligned}$$

- Secondly Γ includes clause *select*. Intuitively, *select*(L, C, W) takes wallet W from L, which is the wallet with maximum weight according to p, and which has enough money to pay C:

$$\begin{aligned} select(L, C, W) \quad : - \quad & map(L, L'), \\ & sort(L', L''), \\ & choose(L'', C, W). \end{aligned}$$

Here $map(L, L')$ returns L' such that $L' = \{(w, weight) \mid w \in L \wedge p(w, weight)\}$ and clause $sort(L', L'')$ sorts list L' in L'': $L'' = [(w_1, weight_1), ..., (w_n, weight_n)]$ for $n = card(L')$ and $weight_i \geq weight_{i+1}$ for $i = 1..n-1$. Both map and $sort$ can easily be expressed in Prolog, so we omit their implementations. Finally, $choose(L, C, W)$ selects the first wallet W in list L that has enough money (in W's *money* binding) to pay C:

$$\begin{aligned} & choose([H|T], C, H) : -H = wallet(\_, B), get\_value(B, money, M), M \\ & choose([H|T], C, Y) : -choose(T, C, Y). \end{aligned}$$

Recall that we assume that binding *money* is always in the bindings.

5.3.4 Specifying Credit and Debit payment schemes for Commerce

Now we are ready to describe an example of a rule that checks the payment condition on a license, and then performs the payment by using an appropriate policy. Two different payment schemes are considered. First, we model *debit* payment, in which the money is extracted from the wallet when the content is accessed (in the example, when the content is *played*). Secondly, we model *credit* payment, in which the payment is

delayed until later. In our model, we extend the bindings of the wallets with a *Credit* value, which records the amount of money credited. Further, we assume that credit is always granted as long the wallet has enough money to cover all the money already credited (recorded in *Credit*) plus the money of the actual credit request. The actual payment can be triggered at will from the user with the rule *pay_credit*. Of course, more complex types of credit payment can be modeled in LicenseScript, for example credit payment done monthly (like credit card companies) although for simplicity and space constraints we do not show them here.

Debit payment Consider:

$$
\begin{aligned}
& play\_debit(X) : lic(X, \Delta, B), policy(\Psi), wm(\Gamma, L) \rightarrow \\
& \qquad lic(X, \Delta, B'), policy(\Psi), wm(\Gamma, L') \\
& \Leftarrow \quad \Delta \vdash canplay(B, B', C, P), && (1) \\
& \qquad \Psi \vdash select(L, C, wallet(\Theta, E)), && (2) \\
& \qquad \Theta \vdash cantransfer(E, E', P, C), && (3) \\
& \qquad \Gamma \vdash set\_value(L, wallet(\Theta, E), \\
& \qquad\qquad\qquad wallet(\Theta, E'), L'). && (4)
\end{aligned}
$$

The *play_debit* rule first selects a license $lic(X, \Delta, B)$, a policy $policy(\Psi)$ and the wallet manager $wm(\Gamma, L)$. The execution of this rule is carried out successfully if the above conditions (1)-(4) hold. First, it is checked that license X can indeed be played (1). Then, a wallet $wallet(\Theta, E)$ is chosen in (2), and then used to perform the transfer (3). The new wallet with the updated money value is then saved in wm (4), illustrating the debit payment.

Credit payment To model credit payments, we extend the bindings of our wallets to contain an element *Credit*, recording the amount of money credited. We now introduce a new clause defining *cancredit*:

$$
\begin{aligned}
& \textit{cancredit}(B, B', P, A) \ : - \\
& \qquad get\_value(B, m, M), \\
& \qquad get\_value(B, \textit{bank\_charges}, C), \\
& \qquad get\_value(B, \textit{Credit}, Cr), \\
& \qquad C + A + Cr \leq M, \\
& \qquad set\_value(B, Credit, Cr + A + C, B').
\end{aligned}
$$

This clause checks that the credit already given (stored in Cr) plus the actual required payment A and bank charges C are not greater than the total amount of money M. Then, the new credit is granted and the new *Credit* is updated.

Now, our rule *play_credit* is:

$$
\begin{aligned}
& play\_credit(X) : lic(X, \Delta, B), policy(\Psi), wm(\Gamma, L) \rightarrow \\
& \qquad lic(X, \Delta, B'), policy(\Psi), wm(\Gamma, L') \\
& \Leftarrow \quad \Delta \vdash canplay(B, B', C, P), && (5) \\
& \qquad \Psi \vdash select(L, C, wallet(\Theta, E)), && (6) \\
& \qquad \Theta \vdash cancredit(E, E', P, C), && (7) \\
& \qquad \Gamma \vdash set\_value(L, wallet(\Theta, E),\ wallet(\Theta, E'), L'). && (8)
\end{aligned}
$$

This rule does not make any actual money transfer, different from *play_debit*, but just credits the payment. Now, whenever a credit payment is performed by the user, the following clause is triggered:

$$
\begin{aligned}
&\textit{canpaycredit}(B, B', P) \ :- \\
&\quad get\_value(B, m, M), \\
&\quad get\_value(B, \textit{Credit}, Cr), \\
&\quad Cr \leq M, \\
&\quad set\_value(B, m, M - Cr, B'), \\
&\quad set\_value(B, Credit, 0, B'), \\
&\quad \textit{transfer}(P, Cr).
\end{aligned}
$$

This clause pays the credited money in *Credit*, realizing the money transfer and resetting the credit binding. Finally, the rule to trigger the clause for credit payment is:

$$
\begin{aligned}
\textit{pay\_credit}(E, E', L, L')(P) : wm(\Gamma, L) \rightarrow & \qquad (9)\\
\Leftarrow \quad \Gamma \vdash L = [wallet(\Theta, E)|\_], & \qquad (10)\\
\Theta \vdash canpaycredit(E, E', P), & \qquad (11)\\
\Gamma \vdash set\_value(L, wallet(\Theta, E), & \\
\qquad wallet(\Theta, E'), L'). & \qquad (12)
\end{aligned}
$$

From the examples of payment schemes given above, it is clear that LicenseScript is well suited to integrate payment policies with rights control.

5.4 Fair Use Modeling

In this section, we explain how LicenseScript may be used to approximate fair use. For this, we use a two-pronged approach (Fig. 5): we employ (1) *Rights assertion* to allow the user to assert new fair use-compliant rights in addition to the rights dictated by the license; and (2) *Audit logging* to keep a record of the rights asserted and exercised by the user and to keep track of the copies of the licenses created.

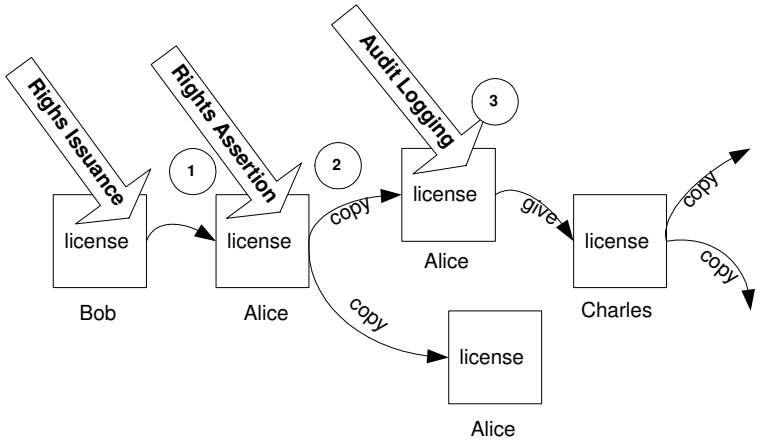

Figure 5: Approximating fair use.

Fig. 5 shows that Bob issues a license to Alice, allowing Alice to make copies of the license (and therefore she is able to make copy of the licensed content too). Alice may assert new rights including making copies of the license. Alice in turn gives a copy of the license to Charles. All the actions performed by the users (Alice and Charles) are logged in the appropriate license.

Using this approach, on one hand the users can freely exercise their statutory rights; on the other hand, the copyright owner can track the source of possible copyright infringement. Note that our proposal is more advanced than *rights issuance*, which is performed by the copyright owner for issuing and granting rights to a user. In subsequent sections, we elaborate how the two-pronged approach mentioned earlier can be used to approximate fair use.

5.4.1 Rights Assertion

We use the following illustrative scenario of a *digital library* to aid in our explanation:

Example 16. Alice borrows an ebook, entitled "An Example Book" from Bob's Digital Library (herefrom we simply call it Bob). Bob sends the license to Alice, allowing her to view, copy and give the rendered copies of the ebook to other users.

Suppose Alice wants to print the ebook. The license does not include the *print* right. Therefore, Alice must assert a new *print* right by adding it to the license. We believe that the users ability to assert new rights contributes to fair use because the user can express their rights according to their legal rights (i.e. print for personal use), in addition to the rights granted by the copyright owner.

We make a few what we believe to be reasonable assumptions. Alice must have a content renderer, in this case an ebook viewer to use the ebook. A set of rules are embedded in the firmware of this ebook viewer. Bob may define these rules. Bob may not trust Alice, but he trusts the rules he defines. If Alice's asserted right in the license can be exercised by any corresponding rule Bob defines, Bob may logically trust this right (unless the asserted right causes conflicts in the license, which we will discuss later). This is because Alice's asserted rights must conform to the semantics of the rules. The implicit assumption is that the content renderer is secure.

Bob may specify some of the contextual information by using LicenseScript. This information may be, for instance, the usage purpose, the location of use etc. that the Fair Use Doctrine refers to. Then, Bob can write the rules in such a way that Alice is obliged to provide the contextual information. The rules then validate the provided information using the contextual information stated in the doctrine. In other words, Alice must declare her intention to perform fair use and the circumstances in which this is done. The attestation of this declaration is performed by the underlying architecture (presumably by using some cryptographic means). We consider architectural support to enforce all this as our future work. Here we are concerned only with a higher level of abstraction that defines *what* may or may not happen, and not *how* actions may be performed and/or enforced.

5.4.2 Audit Logging

Alice should not be able to assert arbitrary rights nor must she be able to override existing rights (in the license) that may undermine the rights management system. While we might be able to avoid some problems by syntactic checks (e.g. to check for duplication of rights in the license caused by the rights assertion), many other potential ambiguities will remain (e.g. if the rights asserted can expire). Therefore, we record all the asserted rights (along with the user's identity, the date the right is asserted and the purpose of asserting the rights) in the license.

Bob may check the record and the license if the asserted rights have overridden or violated the contract rights. Therefrom, Bob may take further action, e.g. to allow/disallow the Alice's asserted right or to take Alice to court if the asserted rights violate the copyrights or the contract rights.

Additionally, Bob also tracks the copies of the licenses distributed by Alice. He can put a history record in the license to log this distribution pattern. Thus, Bob (i.e. the copyright owner) can trace the distribution of the licenses by inspecting the history record in these licenses. This helps the copyright owner track possible sources of copyright infringement. Audit logging requires cryptographic support from the underlying architecture. We have already addressed the issue of secure audit logging in our previous work [Chong et al., 2003c].

This concludes the introduction to our two-pronged approach towards approximating fair use, i.e. a priori specification and a posteriori verification. We will now present the details of the approach using LicenseScript.

5.4.3 Modeling of Fair Use in LicenseScript

We now use LicenseScript to define (1) the *license* issued by Bob to Alice, which allows her to *view*, *copy* and *give* the ebook, as well as *assert* new rights (section 5.4.3); (2) the *doctrine* that carries the contextual information consistent with fair use (section 5.4.3); (3) the *record* that logs Alice's asserted rights (section 5.4.3); and (4) the *rules* defined by Bob as the firmware of Alice's system, which include *view*, *copy*, *give*, *print* and *assert* (section 5.4.3).

The License lic Following Example 16, the license that Bob issues to Alice is:

$$
\begin{array}{l}
lic(ebook : an\_example\_book, \\
\{(canloan(B1, B2, Loaner, User) :- \\
\quad get_value(B1, digital\_library, L), L == Loaner, \\
\quad get\_value(B1, loaned, Loaned), Loaned == false, set\_value(B1, loaned, true, B2), \\
\quad set\_value(B1, user, User), today(D), set\_value(B1, expires, D + 7, B2)), \\
(canreturn(B1, B2) :- \\
\quad set_value(B1, loaned, false, B2)), \\
(canview(B1, B2, User) :- \\
\quad get\_value(B1, user, U), U == User, get\_value(B1, expires, Exp), \\
\quad today(D), D > Exp), \\
(cancopy(B1, B2, B3, User) :- \\
\quad get\_value(B1, user, U), U == User, get\_value(B1, expires, Exp), \\
\quad today(D), D > Exp, get\_value(B1, copies, N), append(\{(User, D)\}, N, NN), \\
\quad set\_value(B1, copies, NN, B2), set\_value(B1, copies, NN, B3)), \\
(cangive(B1, B2, User1, User2) :- \\
\quad get\_value(B1, user, U), U == User1, set\_value(B1, user, User2, B2), get\_value(B1, trace, T), \\
\quad today(D), append(\{(User1, D, User2)\}, T, T2), set\_value(B1, trace, T2, B2)), \\
(canassert(C1, C2, B1, B2, Cs, Bs, User, Purpose) :- \\
\quad get\_value(B1, user, U), U == User, get\_value(B1, expires, Exp), today(D), D > Exp, \\
\quad get\_value(B1, asserted, As), append(\{(User, D, Purpose)\}, Clause, NC), append(NC, As, As2), \\
\quad set\_value(B1, asserted, As2, B2), append(C1, Cs, C2), append(B1, Bs, B2)), \\
(canperform(B1, B2, User) :- \\
\quad get\_value(B1, expires, Exp), today(D), D > Exp, get\_value(B1, user, U), U == User)\}, \\
\{(user \equiv alice), (digital\_library \equiv bob), (loaned \equiv true), \\
\quad (asserted \equiv \{\}), (trace \equiv \{\}), (copies \equiv \{\}), (expires \equiv 15/8/03)\})
\end{array}
$$

The function $append(L1, L2, L3)$ is a built-in Prolog program that produces a new list ($L3$) by combining two lists ($L1$ and $L2$).

The license clause $canloan$ determines if the ebook can be loaned to the user, and *only* by the digital library. The return date (represented by $expires$) is set at the seventh day from the date this ebook is loaned.

The license clause $canreturn$ is the counterpart of the license clause $canloan$, which resets the binding $loaned$ to $false$.

The license clause $canview$ determines that *only* Alice (the user) can view the ebook and the return date $expires$ has not expired. The license clause $canassert$ allows Alice to assert new rights (represented as the $Clause$ with necessary bindings $Binds$). The license clause $canperform$ determines if the user who performs fair use is the genuine user who owns the license.

The license binding $asserted$ records all the rights asserted by the user. The license binding $trace$ records the distribution of the license when it is given away. The license binding $copies$ records the user who generates a new copy of this license.

The Record rec This is the record that belongs to Bob, and which logs the rights asserted by Alice:

$$
\begin{array}{l}
rec(ebook : an\_example\_book, \\
\{(canlog(B1, B2, User, Action) : - \\
\quad get\_value(B1, history, H), today(Date), \\
\quad append([(User, Action, Date)], H, NH), set\_value(B1, history, NH, B2))\}, \\
\{(history \equiv \{\}), (digital\_library \equiv bob)\})
\end{array}
$$

The term $record$ records (clause $canlog$) all the actions (the argument $Action$) performed by the user (the argument $User$) at the current time (the value $Date$) on the ebook.

The Doctrine doc The doctrine (defined by Bob) encodes the contextual information of fair use:

$$
\begin{array}{l}
doc(fairuse, \\
\{(canallow(B1, B2, Purpose) : - \\
\quad get\_value(B1, purposes, Ps), member(Purpose, Ps), \\
\quad identify(Loc), get\_value(B1, location, L), L == Loc)\}, \\
\{(purposes \equiv \{criticism, comment, newsreport, eduction, scholarship, research\}), \\
\quad (location \equiv USA)\})
\end{array}
$$

The function $member(E, L)$ checks if the element E belongs to the list L. The license clause $canallow$ determines if the purpose attested by the user for using the content is under the fair use context and if the user is in the U.S. The license binding $purposes$ state all the usage purposes allowed under the fair use doctrine. The license binding $location$ indicates that this doctrine applies in USA. The copyright owner can define different types of doc, e.g. the first sale doctrine to encapsulate the corresponding contextual information.

The Rules The rules that Bob defines for Alice's firmware are:

$$
\begin{array}{rl}
loan(Ebook, User) : & lic(Ebook, C, B1) \longrightarrow lic(Ebook, C, B2) \\
& \Leftarrow C \vdash canloan(B1, B2, User) \\
return(Ebook) : & lic(Ebook, C, B1) \longrightarrow lic(Ebook, C, B2) \\
& \Leftarrow C \vdash canreturn(B1, B2) \\
view(Ebook, User, Purpose) : & lic(Ebook, C, B1) \longrightarrow lic(Ebook, C, B2) \\
& \Leftarrow C \vdash canview(B1, B2, User) \\
view(Ebook, User, Purpose) : & doc(fairuse, Cp, Bp1), lic(Ebook, C, B1) \longrightarrow \\
& \quad doc(fairuse, Cp, Bp2), lic(Ebook, C, B2) \\
& \Leftarrow C \vdash canperform(B1, B2, User), \\
& \quad Cp \vdash canallow(Bp1, Bp2, Purpose)
\end{array}
$$

$$
\begin{aligned}
copy(Ebook, User) : lic(Ebook, C, B1) &\longrightarrow lic(Ebook, C, B2), lic(Ebook, C, B3) \\
&\Leftarrow C \vdash cancopy(B1, B2, B3, User) \\
copy(Ebook, User, Purpose) : doc(fairuse, Cp, Bp1), lic(Ebook, C, B1) &\longrightarrow \\
& \quad doc(fairuse, Cp, Bp2), lic(Ebook, C, B1), lic(Ebook, C, B2) \\
&\Leftarrow C \vdash canperform(B1, B2, User), \\
& \quad Cp \vdash canallow(Bp1, Bp2, Purpose) \\
give(Ebook, User, Purpose) : lic(Ebook, C, B1) &\longrightarrow lic(Ebook, C, B2) \\
&\Leftarrow C \vdash cangive(B1, B2, User1, User2) \\
give(Ebook, User, Purpose) : doc(fairuse, Cp, Bp1), lic(Ebook, C, B1) &\longrightarrow \\
& \quad doc(fairuse, Cp, Bp2), lic(Ebook, C, B2) \\
&\Leftarrow C \vdash canperform(B1, B2, User), \\
& \quad Cp \vdash canallow(Bp1, Bp2, Purpose) \\
print(Ebook, User, Purpose) : lic(Ebook, C, B1) &\longrightarrow lic(Ebook, C, B2) \\
&\Leftarrow C \vdash canprint(B1, B2, User) \\
print(Ebook, User, Purpose) : doc(fairuse, Cp, Bp1), lic(Ebook, C, B1) &\longrightarrow \\
& \quad doc(fairuse, Cp, Bp2), lic(Ebook, C, B2) \\
&\Leftarrow C \vdash canperform(B1, B2, User), \\
& \quad Cp \vdash canallow(Bp1, Bp2, Purpose) \\
assert(Ebook, Cs, Bs, User, Purpose) : lic(Ebook, C1, B1), rec(Ebook, Cr, Br1) &\longrightarrow \\
& \quad lic(Ebook, C2, B2), rec(Ebook, Cr, Br2) \\
&\Leftarrow C1 \vdash canassert(C1, C2, B1, B2, Cs, Bs, User, Purpose), \\
& \quad Cr \vdash canlog(Br1, Br2, User, Cs)
\end{aligned}
$$

The $assert$ rule says: to assert a new clause Cs with corresponding bindings Bs, the user must show her identity $User$ and state the $Purpose$ of asserting this right. The user's asserted clause is recorded by the term rec (see section 5.4.3).

The $view$ rules say: to view the $Ebook$, the user must present her identity $User$ and declare to the usage $Purpose$; the license must contain the license clause $canview$. If the first rule does not apply to Alice's execution, the second rule may be executed: the usage $Purpose$ attested by the $User$ must conform to the contextual information stated in the doc.

The rules show how the various objects in the multiset are used, in a cooperative fashion to achieve fair use. For example, as shown in Figure 5, in step ① Alice has asserted a print right to the license (as shown in this section) on '$1/8/2003$" for the purpose of education (by executing the $assert$ rule). The new license is as follows:

$$
\begin{aligned}
& lic(ebook : an\_example\_book, \\
& \{\cdots, \\
& (canprint(B1, B2, User) : - \\
& \quad get\_value(B1, onlyalice, U), U == User)\}, \\
& \{\cdots, \\
& \quad (asserted \equiv \{(alice, 1/8/2003, education), \\
& \quad\quad (canprint(B1, B2, User) : -get\_value(B1, onlyalice, U), U == User)\}), \\
& \quad\quad (onlyalice == alice)\})
\end{aligned}
$$

(Herefrom, the symbol "$\cdots$" represents the unchanged part of the object.)

The asserted clause $canprint$ allows *only* Alice to print the ebook (she adds a new binding, namely $onlyalice$ to the license). Alice can execute the $print$ rule to print the ebook with the asserted $print$ right. The asserted right is logged at the license binding $asserted$ and Bob's rec (from section 5.4.3) becomes:

$$
\begin{array}{l}
rec(ebook : an\_example\_book, \\
\{\cdots\}, \\
\{\cdots, \\
\quad (history \equiv \{(alice, (canprint(B1, B2, User) : -get\_value(B1, user, U), U == User), 1/8/03)\})\})
\end{array}
$$

Now, in step ② Alice makes an additional copy of the license by executing the $copy$ rule. The third version of the license becomes:

$$
\begin{array}{l}
lic(ebook : an\_example\_book, \\
\{\cdots, \\
\quad (canprint(B1, B2, User) : -get\_value(B1, onlyalice, U), U == User)\}, \\
\{\cdots, \\
\quad (asserted \equiv \{(alice, 1/8/2003, education), \\
\quad\quad (canprint(B1, B2, User) : -get\_value(B1, onlyalice, U), U == User))\}), \\
\quad (onlyalice \equiv alice), (copies \equiv \{(alice, 1/8/2003)\})\})
\end{array}
$$

Alice's copy action is logged in the binding $copies$. Alice then gives a copy of the license to Charles on "$2/8/03$" (by executing the $give$ rule) in step ③. Charles' license (given by Alice) will look like this:

$$
\begin{array}{l}
lic(ebook : an\_example\_book, \\
\{\cdots, \\
\quad (canprint(B1, B2, User) : -get\_value(B1, onlyalice, U), U == User)\}, \\
\{\cdots, \\
\quad (asserted \equiv \{(alice, 1/8/2003, education), \\
\quad\quad (canprint(B1, B2, User) : -get\_value(B1, onlyalice, U), U == User))\}), \\
\quad (onlyalice \equiv alice), (copies \equiv \{(alice, 1/8/2003)\}), \\
\quad (trace \equiv \{(alice, 2/8/03, charles)\}), (user \equiv charles)\})
\end{array}
$$

The distribution is logged in the license binding $trace$. The value of the binding $user$ is assigned to $charles$ indicating the transfer of the ownership of this license.

We have demonstrated how rules can transform the objects in the multiset by using the example as illustrated in Figure 5. This concludes the detailed description of the approach to approximating fair use in LicenseScript.

6 Full Language

So far we have ignored to discuss how multiple devices are modeled in LicenseScript. Such an abstraction of devices is important because the execution of actions depends also on the environment in which they are executed among other things. Consistent with intuition, in our model, devices have two different storage sources: a *private* memory and a *public* one. The former can be thought as the device's own memory, which can be accessed only by the device itself. On the other hand, the public memory is the "collective" memory, consisting of all the shared memories of each device. An *authorized domain* is then defined by the group of devices that (have permission to) form such a public shared memory.

6.1 From Devices to Authorized Domains

More formally, a *device* is represented by a term of the form $dev(id,$*MSpub*$,$*MSpriv*$,$*Ruleset*$)$, where:

- id is the device identity.
- *MSpub* is the *public* multiset of the device, representing the AD.
- *MSpriv* is the *private* multiset of the device, representing the local memory.
- *Ruleset* is the set of *license* rules.

Rules can operate over both *MSpub* and *MSpriv* multisets, and thus the syntax of simple play rule must be extended accordingly. We also need to distinguish in which device we are executing the rule.

The new syntax is as follows:

$$name(args)@id \quad : \quad \begin{array}{c} \textit{MSpub} : lhs \rightarrow rhs \\ \textit{MSpriv} : lhs' \rightarrow rhs' \end{array} \quad \Leftarrow \Phi \tag{13}$$

The differences are twofold:

- $name(args)@id$ means "fire rule $name$ with arguments $args$ at device id". To prevent ambiguities, we assume that device ids are unique in a given domain (analogously to ethernet addresses in a LAN).
- $\begin{array}{c} \textit{MSpub} : lhs \rightarrow rhs \\ \textit{MSpriv} : lhs' \rightarrow rhs' \end{array} \quad \Leftarrow \Phi$ means:
  1. Choose lhs from the public multiset *MSpub* located at device id,
  2. Similarly choose lhs' from the private multiset *MSpriv*.
  3. Evaluate Φ, if false abort.
  4. Otherwise, rewrite lhs to rhs in *MSpub* and lhs' to rhs' in *MSpriv*.

Example 17. For example, a *play* rule that plays from the public multiset of device *pda* could look like:

$$\begin{array}{l} play(X)@pda \ : \\ \quad \textit{MSpub} : lic(X, \Delta, B) \rightarrow lic(X, \Delta, B') \\ \qquad \Leftarrow \Delta \vdash canplay(B, B') \end{array}$$

The private multiset is not affected by the above rule. This notation is just syntactic sugar for:

$$\begin{array}{l} play(X)@pda \ : \\ \quad \textit{MSpub} : lic(X, \Delta, B) \rightarrow lic(X, \Delta, B') \\ \qquad \textit{MSpriv} : rhs \rightarrow rhs \\ \\ \qquad \Leftarrow \Delta \vdash canplay(B, B'), \end{array}$$

where rhs can be any multiset.

6.2 The model of communication

Consider the last *play* rule (described in Example 17 above). There, the rewriting of elements happens from one multiset (say *MSpriv*) to the *same* multiset (again, *MSpriv*). This is modeling the movement of data *within* the private memory of the device. To model the *communication* of data, we simply need to allow the transfer of objects from one multiset to the other one. To illustrate this point, consider the following rule, a *pub2priv* rule that *copies* a license from the public multiset to the private multiset:

$$\begin{array}{c} pub2priv(X)@pda \ : \\ \textit{MSpub} : lic(X,\Delta,B) \rightarrow lic(X,\Delta,B') \\ \textit{MSpriv} : \_ \rightarrow lic(X,\Delta,B'') \\ \Leftarrow \Delta \vdash canmovetopriv(B,B',B'') \end{array}$$

This rule, when executed at device pda, chooses a license $lic(X,\Delta,B)$ from the public multiset *MSpub*, and executes a query called $canmovetopriv(B,B',B'')$. Intuitively, this query will evaluate whether the license can be copied to the private memory of a device (in this case, device pda). If this is so, the query should return *two* new sets of bindings: one for the license that will continue residing on the public multiset, and another for the new license that will reside on the private multiset.

Suppose that license $lic(X,\Delta,B)$, in the bindings B has a counter $playtimes \equiv n$, allowing n times of playing. Then we can give a specification for $canmovetopriv(B,B',B'')$ that provides a *one-time-play* license to the private multiset, and keeps a $n-1$-*times-play* in the public one. This can be specified as follows:

$$\begin{array}{c} canmovetopriv(B,B',B'') \ :- \\ get\_value(B,playtimes,N), N>0, set\_value(B',playtimes,N-1), \\ set\_value(B'',playtimes,1). \end{array}$$

In more general terms, the public multiset of each device represents the "global" database of the authorized domain. This is depicted in Figure 6. Each device can either "contribute" to the public multiset, the Authorized Domain, or it can "extract" licenses from it (e.g. an application of the $pub2priv$ rule).

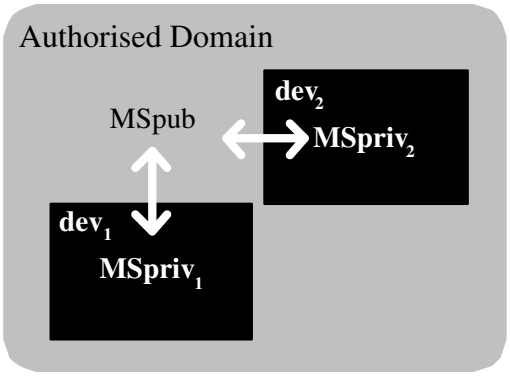

Figure 6: Authorized domain.

6.3 Gateway

A gateway acts as the "connector" between two authorized domains. This is depicted in Figure 7.

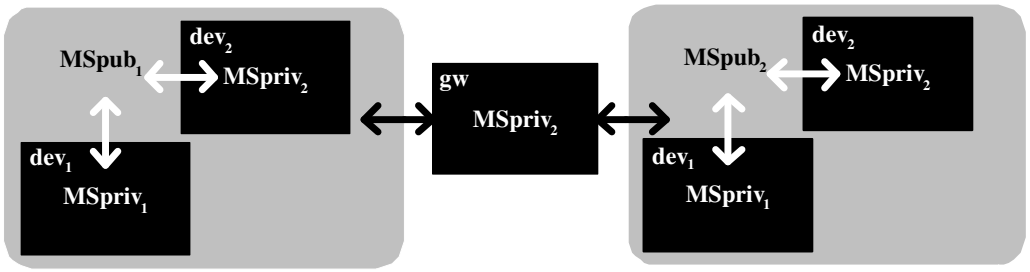

Figure 7: Gateway in action.

More formally, in our framework a gateway is a special device with access to two public multisets: $gw(id, \mathit{MSpub}_1, \mathit{MSpub}_2, \mathit{MSpriv}, \mathit{Ruleset})$.

As an illustrating example, below we specify a gateway that copies a license X from MSpub_1 to MSpub_2.

Example 18. The gateway $gw(gateway, [lic(mus, \Gamma, C)], [], [], \{copy(X) : code\})$
With:

$$
\begin{aligned}
&copy(X)@gateway\ : \\
&\quad \mathit{MSpub}_1 : lic(X, \Delta, B) \rightarrow lic(X, \Delta, B') \\
&\quad \mathit{MSpub}_2 : \_ \rightarrow lic(X, \Delta, B') \\
&\quad \mathit{MSpriv} : \_ \rightarrow \_ \\
&\qquad\quad \Leftarrow \Delta \vdash cancopy(B, B')
\end{aligned}
\tag{14}
$$

LicenseScript is designed to cope well with the notion of authorized domains and the dynamic evolution of licenses, as we have shown above.

7 Implementation

In this section, we discuss a high level architecture for supporting LicenseScript. As illustrated in Figure 8, the architecture is composed of three main components, namely the License Interpreter, the Content Renderer, and the Cryptographic Engine. We have implemented a prototype that serves as a proof-of-concept.

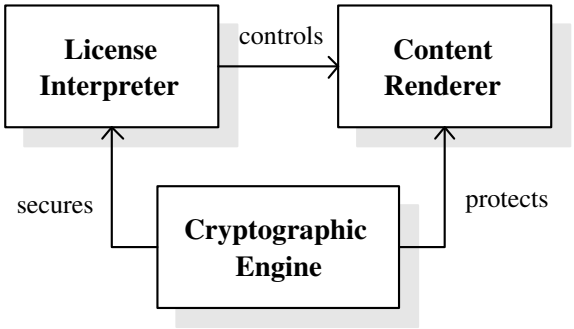

Figure 8: The LicenseScript architecture.

The License Interpreter interprets and maintains LicenseScript licenses. It determines if the actions performed by a user is allowed or forbidden as dictated by the license, and controls the functionality of the

Content Renderer accordingly. For instance, if a license does not allow a user to print an ebook, the print function of the Content Renderer is disabled.

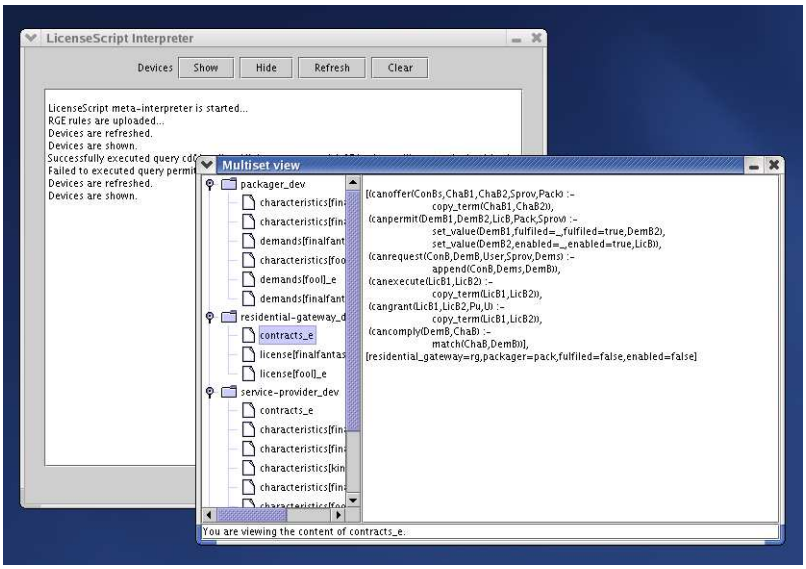

Figure 9: The LicenseScript Interpreter user interface.

The core of the License Interpreter is a simple extension of the vanilla meta-interpreter presented in [Sterling and Shapiro, 1994], implemented on top of the Eclipse engine:

```
solve([],_).
solve([Query|Queries],Program) :-
    copy_term(Program,Temps),
    member((Query:-Body),Temps),
    solve_body(Body),
    solve(Queries,Program).
solve_body(true).
solve_body((Goal1,Goal2)) :-
    solve_body(Goal1),solve_body(Goal2).
solve_body(Goal) :-
    call(Goal).
```

The only difference lies in the fact that LicenseScript runs a query in a specific `Program`. Multiset rewriting has also been implemented in Prolog [Chong et al., 2003a].

Figure 9 shows a screenshot of the (Java) user interface for the LicenseScript components. Various buttons allow the user to view the LicenseScript objects in the various devices. The text area underneath the buttons logs and displays the status of the execution of the meta-interpreter and the ECL^{i}PS^e engine. The multiset viewer allows the users to monitor the status of the LicenseScript objects before and after the execution.

The Content Renderer is either a customized general-purpose or special-purpose application capable of rendering the content. We have implemented two types of Content Renderer, namely a plug-in to Adobe Acrobat [Chong et al., 2002] and a streaming audio player based on a tamper-resistant hardware token [Cheng et al., 2004].

The Cryptographic Engine performs the necessary cryptographic operations for the License Interpreter and the Content Renderer, for instance to identify, authenticate and authorize users [Chong et al., 2002]; to

decrypt licenses and content if they are encrypted [Chong et al., 2004]; and to encrypt audit logs that record the user's actions on the content [Chong et al., 2003c].

8 Related Work

We now discuss related work in three categories, namely authorized domains, the MPEG-21 REL, and other rights expression languages.

8.1 Authorized Domains

LicenseScript has an explicit representation for authorized domains. This allows content providers to specify customized rules to regulate the flow of content between devices according to their business model. Intra-domain flow is usually unconstrained, while inter-domain flow is typically limited. The enforcement of such rules relies on a number of *trusted* components, such as the LicenseScript Interpreter discussed earlier. Other components, such as the implementation of the authorized domain itself are beyond the scope of this paper.

Authorized domains serve the sole purpose of separating compliant devices from non-compliant devices. This involves the authentication of devices to ensure that only compliant devices enter the domain, as well as key revocation to ensure that unscrupulous devices can be excluded from the domain. We provide a brief survey of the three most relevant industry-lead projects on authorized domains:

Philips van den Heuvel et al. [2002] present the functional requirements and design options for an implementation of authorized domains for home A/V networks. This work is elaborated further by Popescu et al. [2004], who present a security architecture for authorized domains focusing on protocols for device compliance checking. The main advantages of the protocols are (1) public-key cryptography only for non-time critical operations, (2) the space-efficient encodings of revocation lists.

IBM xCP [Lotspiech et al., 2004] uses broadcast encryption [Fiat and Naor, 1994] to implement the compliance checking necessary for authorized domains. The main advantages of this scheme is (1) no time consuming public-key cryptography at all, and (2) suitable for off-line operation due to the lack of handshakes. However revocation lists are potentially of unbounded length.

Thomson SmartRight [SmartRight.org, 2003] uses smart cards to store sensitive keys. Global Revocation is realized by issuing new cards as is customary in conditional access; revocation of devices within a personal private network (i.e. and authorized domain) is handled locally, by generating a new network-wide key. The main advantages of SmartRight are (1) public-key cryptography is only used for non-time-critical operations, and (2) it is consistent with existing business models for conditional access.

8.2 MPEG-21 REL

Unlike the MPEG-1, 2 and 4 standards that deal primarily with compression coding methods, MPEG-21 contains a number of parts that together, provide building blocks for the description of content (in combination with MPEG-7), the reporting on content usage (Event Reporting, ISO/IEC 21000-15), and the expression of usage rights for digital media items (ISO/IEC 21000-5 and -6).

Part 5 of the MPEG-21 standard, (ISO/IEC 21000-5), concerns the specification of an extensible Rights Expression Language (REL) that is based upon XrML. The MPEG-21 REL provides a Core Schema that focuses on defining the relationship between granted Rights, Principals, Resources and associated Conditions. Its Standard Extension provides new Conditions that relate to metering and charging for content, while its Multimedia Extension provides a means to specify Rights, on Resources under certain conditions that is particularly applicable in a Multimedia context.

Part 6 of the MPEG-21 standard, (ISO/IEC 21000-6), concerns the specification of an extensible Rights Data Dictionary. Much of the MPEG-21 RDD is based upon the <indecs>2rdd work [Rust and Bide, 2000] which defines a set of verbs that can be used with the REL. As the meaning of the terms is unambiguously defined in the RDD, it is possible to create "sentences" in the REL, using the terms defined in the RDD. It is interesting to note that the use of an RDD by MPEG allows the creation of "translation" services that can convert REL expressions between different languages according to well-defined and agreed-upon mappings.

8.3 Other languages

Gunter et al. [2001] from InterTrust Technologies Corporation and Pucella and Weissman [2002] from Cornell University present two logics for licenses. Firmly based on programming language semantics [Gunter, 1992], Gunter et al. develop a model and a language for describing licenses. Their logic consists of a domain of sequences of events called *realities*. An event is modeled as a pair of a time value and an action. Only one event is allowed at a time. A finite set of events is embodied in a reality. A license, then, is a set of realities. Most licenses consist of infinitely many realities to allow the use of a work at one or more of infinitely many times during some period. Using the proposed model, Gunter et al. formulate several standard license types, which they call *simple licenses*. They are similar to what we have treated in this paper: simple licenses are "Up Front", "Flat Rate" and "Per Use". Simple licenses can be used as the building blocks of more complex licenses.

Pucella and Weissman [2002] follow up Gunter et al.'s effort. They propose 3 syntactic categories: (1) action expression, (2) license, and (3) formula. The action expressions are either *permitted* or *obligatory*. In other words, they reason about the licenses and the user's actions with respect to the licenses; this is done by means of a temporal *deontic* logic. This makes their logic more accessible and complete than Gunter et al.'s. A license is an action sequence (not to be confused with an action expression). A formula designates an action sequence that is valid for a license. At most, one action per time per license can occur.

LicenseScript uses multiset rewriting which is more expressive than the pure logic based language of Gunter et al. LicenseScript is also readily subject to logical parallelism. Pucella et al.'s logic is only a starting point, with the assumption of one client and one provider and therefore definitely does not cater for concurrency, like LicenseScript does. Pucella et al. have also not yet taken into account the malleability of licenses and contents (e.g. as a result of "clipping" and "mixing"), and the concepts of authorized domains.

9 Conclusions and Future Work

We present LicenseScript, a logic-based language for Digital Rights management. LicenseScript is strictly more powerful than existing languages such as ODRL and XrML because LicenseScript supports both the *static* aspects of rights control and the *dynamic* aspects of rights evolution.

We show by translating many examples from the ORDL and XrML documentation that LicenseScript does not lack essential functionality offered by the main existing languages. We also show by presenting several advanced examples that LicenseScript offers more functionality than existing languages. We conclude that LicenseScript is expressive, flexible and extensible and thus able to express complex and novel usage scenarios.

We present a succinct formal semantics of LicenseScript, thus providing a solid basis for the construction of implementations as well as providing the foundation for techniques of reasoning about the meaning of licenses. We believe that given the complexity of the licensing process in any but the most trivial applications, the ability to reason about licenses and their evolution will prove to be crucial.

We present an abstract architecture for a LicenseScript implementation and briefly discuss the implementation of the key components of such architecture. We conclude that tamper resistant hardware is indispensable to enforce the control specified by LicenseScript. We regard an integrated implementation of all the components as future work.

Bibliography

K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

K. R. Apt. *From Logic Programming to Prolog*. Prentice Hall, Hertfordshire, United Kingdom, 1997.

K. R. Apt and D. Pedreschi. Reasoning about termination of pure Prolog programs. *Information and Computation*, 106(1):109–157, 1993.

J-P. Banâtre, P. Fradet, and D. L. Métayer. Gamma and the chemical reaction model: Fifteen years after. In C. Calude, G. Paun, G. Rozenberg, and A. Salomaa, editors, *Workshop on Multiset Processing (WMP)*, volume 2235 of *LNCS*, pages 17–44. Springer-Verlag, Berlin, August 2001.

B. Birney and T. Gill. *Microsoft Windows Media Resource Kit*. Microsoft Press, February 2003.

H. Boley. Relationships between logic programming and RDF. In *Advances in Artificial Intelligence. PRICAI 2000 Workshop Reader*, volume 2112 of *LNCS*, pages 201–218. Springer-Verlag, 2001.

A. Bossi, N. Cocco, and M. Fabris. Proving termination of logic programs by exploiting term properties. In S. Abramsky and T.S.E. Maibaum, editors, *Theory and Practice of Software Development (TAPSOFT 91)*, volume 494 of *LNCS*, pages 153–180, Brighton, United Kingdom, April 1991, 1991. Springer-Verlag.

L. J. Camp. DRM: doesn't really mean digital copyright management. In *Proceedings of the 9th ACM conference on Computer and Communications Security*, pages 78–87. ACM Press, 2002.

J. Cheng, C. N. Chong, J. Doumen, S. Etalle, P. H. Hartel, and S. Nikolaus. StreamTo: Streaming content using tamper-resistant tokens. Technical Report TR-CTIT-04-47, Centre for Telematics and Information Technology, Univ. of Twente, The Netherlands, November 2004.

C. N. Chong, R. Corin, S. Etalle, P. H. Hartel, W. Jonker, and Y. W. Law. LicenseScript: A novel digital rights language and its semantics. In K. Ng, C. Busch, and P. Nesi, editors, *3rd International Conference on Web Delivering of Music (WEDELMUSIC)*, pages 122–129, Los Alamitos, California, United States, September 2003a. IEEE Computer Society Press.

C. N. Chong, S. Etalle, and P. H. Hartel. Comparing Logic-based and XML-based Rights Expression Languages. In R. Meersman and Z. Tari, editors, *Proceedings of On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops*, volume 2889 of *LNCS*, pages 779–792, Berlin, Germany, November 2003b. Springer-Verlag.

C. N. Chong, Z. Peng, and P. H. Hartel. Secure audit logging with tamper-resistant hardware. In D. Gritzalis, S. D. C. di Vimercati, P. Samarati, and S. K. Katsikas, editors, *18th IFIP International Information Security Conference (IFIPSEC)*, volume 250 of *IFIP Conference Proceedings*, pages 73–84. Kluwer Academic Publishers, May 2003c.

C. N. Chong, B. Ren, J. Doumen, S. Etalle, P. H. Hartel, and R. Corin. License protection with a tamper-resistant token. In C. H. Lim and M. Yung, editors, *5th Workshop on Information Security Applications (WISA 2004)*, volume 3325 of *LNCS*, pages 224–238. Springer-Verlag, August 2004.

C. N. Chong, R. van Buuren, P. H. Hartel, and G. Kleinhuis. Security attribute based digital rights management (SABDRM). In F. Boavida, E. Monteiro, and J. Orvalho, editors, *Joint Int. Workshop on Interactive Distributed Multimedia Systems/Protocols for Multimedia Systems (IDMS/PROMS)*, volume 2515 of *LNCS*, pages 339–352. Springer-Verlag, November 2002.

D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *Journal of Logic Programming*, 19–20:199–260, 1994.

J. DeTreville. Binder, a Logic-Based security language. In *23rd Symp. on Security and Privacy (S&P)*, pages 105–113, Berkeley, California, Mar 2002. IEEE Computer Society Press, Los Alamitos, California. URL `http://research.microsoft.com/research/pubs/view.aspx?tr_id=545`.

S. Etalle and M. Gabbrieli. Layered modes. *The Journal of Logic Programming*, 39(1–3):225–244, 1999.

D. Eyers, J. Shepherd, and R. Wong. Merging Prolog and XML databases. In J. Thom and J. Kay, editors, *Proceeding of the 7th Australasian Document Computing Symposium (ADCS 2002)*, pages 57–62, 2002.

A. Fiat and M. Naor. Broadcast encryption. In *Advances in Cryptology (CRYPTO'03) Proceedings*, volume 773 of *LNCS*, pages 480–491, Santa Barbara, California, 1994. Springer-Verlag.

C. Gunter, S. Weeks, and A. Wright. Models and languages for digital rights. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)*, pages 4034–4038, Maui, Hawaii, United States, January 2001. IEEE Computer Society Press.

C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.

H. Guo. Digital rights management (DRM) using XrML. In *T-110.501 Seminar on Network Security 2001*, page Poster paper 4, 2001. URL `http://www.tml.hut.fi/Studies/T-110.501/2001/papers/`.

S. Guth. Rights expression languages. In E. Becker, W. Buhse, D. Günnewig, and N. Rump, editors, *Digital Rights Management: Technological, Economic, Legal and Political Aspects*, volume 2770 of *LNCS*, pages 101–112. Springer-Verlag, November 2003.

M. H. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, 1974.

F. Hartung and F. Ramme. Digital rights management and watermarking of multimedia content for m-commerce applications. *IEEE Communications Magazine*, 38(11):78–84, November 2000.

R. Iannella. Open digital rights management. In *World Wide Web Consortium (W3C) DRM Workshop*, page Position paper 23, January 2001. URL `http://www.w3.org/2000/12/drm-ws/pp/`.

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

N. Li, J. Mitchell, and W. Winsborough. Beyond proof of compliance: Security analysis in trust management. *Journal of ACM*, 2004. To appear.

J. W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation – Artificial Intelligence. Springer-Verlag, 1987. Second edition.

J. Lotspiech, S. Nusser, and F. Pestoni. Anonymous trust: Digital rights management using broadcast encryption. *Proceedings of the IEEE Special Issue on Digital Rights Management*, 92(6):898–909, June 2004. URL `http://ieeexplore.ieee.org/xpl/abs_free.jsp?arNumber=1299165`.

D. Mulligan and A. Burstein. Implementing copyright limitations in rights expression languages. In J. Feigenbaum, editor, *Proceedings of 2002 ACM CCS-9 Workshop on Security and Privacy in Digital Rights Management*, volume 2696 of *LNCS*, pages 137–154. Springer-Verlag, November 2002.

D. Parrott. Requirements for a rights data dictionary and rights expression language. Technical Report version 1.0, Reuters Ltd., 85 Fleet St., London EC4P 4AJ, June 2001. In response to ISO/IEC JTC1/SC29/WG11 N4044: "Reissue of the Call for Requirements for a Rights Data Dictionary and a Rights Expression Language" – MPEG-21.

B. C. Popescu, B. Crispo, A. S. Tanenbaum, and F. L. A. J. Kamperman. A DRM security architecture for home networks. In *4th ACM workshop on Digital rights management (DRM)*, pages 1–10, Washington DC, USA, October 2004. ACM Press. URL `http://doi.acm.org/10.1145/1029146.1029150`.

R. Pucella and V. Weissman. A logic for reasoning about digital rights. In *IEEE Proceedings of the Computer Security Foundations Workshop*, pages 282–294, Cape Breton, Nova Scotia, Canada, June 2002. IEEE Computer Society Press.

B. Rosenblatt, B. Trippe, and S. Mooney. *Digital Rights Management: Business and Technology*. John Wiley & Sons, New York, United States, November 2002.

Godfrey Rust and Mark Bide. The <indecs> metadata framework, June 2000. URL `http://www.indecs.org/pdf/framework.pdf`.

P. Samuelson. Digital rights management {and,or,vs.} the law. *Communications of ACM*, 46(4):41–45, April 2003.

SmartRight.org. *SmartRight Technical white paper Version 1.7*. Thompson, Paris, France, January 2003. URL `http://www.smartright.org/images/SMR/content/SmartRight_tech_whitepaper%_jan28.pdf`.

L. Sterling and E. Shapiro. *The Art of Prolog (Second Edition)*. The MIT Press, Cambridge, Massachusetts 02142, Uniter States, 1994.

S.A.F.A. van den Heuvel, W. Jonker, F.L.A.J. Kamperman, and P.J. Lenoir. Secure content management in authorised domains. In *Int. Broadcasting Convention (IBC)*, pages 467–474, Amsterdam, The Netherlands, September 2002. Broadcastpapers Pty Ltd, PO Box 259, Darlinghurst, NSW, 1300, AUSTRALIA.