LicenseScript: A Novel Digital Rights Language and its Semantics*

Cheun Ngen Chong[1], Ricardo Corin[1], Sandro Etalle[1],

Pieter Hartel[1], Willem Jonker[2], and Yee Wei Law[1]

[1]University of Twente, The Netherlands [2]Philips Research, The Netherlands

{chong,corin,etalle,pieter,ywlaw}@cs.utwente.nl, willem.jonker@philips.com

Abstract

We propose LicenseScript as a new multiset rewriting/logic based language for expressing dynamic conditions of use of digital assets such as music, video or private data. LicenseScript differs from other rights expression languages in that it caters for the intentional but legal manipulation of data. We believe this feature is the answer to providing the flexibility needed to support emerging usage paradigms of digital data. We provide the language with a simple semantics based on traces.

1 Introduction

Most information, such as books, music, video, personal data and sensor readings (we generalize this information as *data*), is intended for a specific use. This specific use should conform to particular terms and conditions, which are often governed by *licenses*. To describe a license, a specific language is needed. In fact, the last few years have witnessed a proliferation of *rights expression language*[1] (REL). These are usually based on XML, e.g. XrML [5] (`www.xrml.org`) and ODRL [6] (`www.odrl.net`).

It is now widely acknowledged that the abovementioned XML-based RELs have some important shortcomings: (1) the syntax is complicated and obscure when the conditions of use become complex, (2) these languages lack a formal semantics [3, 9]; the meaning of licenses relies heavily on the human interpretation, and (3) the language cannot express many useful copyright acts [8]. Gunter *et al* [3] overcome some of the drawbacks by introducing an abstract model and language with a corresponding formal semantics. Pucella and Weissman [9] follow up Gunter *et al*'s effort with more rigor.

Licenses play an important role in the electronic distribution of music. With the advent of the Internet, music labels are searching for ways of distributing music over the Internet in a way that respects the rights of the owners and the labels. At the same time, partly due to Napster, users have become used to easy access to music on their devices. As a result, *Electronic Music Distribution* (EMD) will only be successful if these systems provide flexible support for licensing of music and the user gains a broad freedom in accessing the music, which in turn requires flexible and easy to understand licenses will a well defined semantics. In an attempt to cope with the requirement of seamless music access on users devices, the notion of authorized domain [12] has been developed (`http://www.dvb.org`).

An authorized domain can be seen as the collection of devices that belongs to a user or a household. The idea is that music is delivered to the authorized domain, and that it can be accessed seamlessly on any device in that domain. The music access is governed by licenses that are bound to the domain, rather than to individual devices. In addition to licenses that govern the access within the domain, there are licenses that govern the exchange of music between domains. The latter guarantees that unauthorized music exchange can be prohibited. Regrettably, state-of-the-art languages cannot cope with this scenario.

*This work is supported by the Telematica Instituut via the LicenseScript project and we like to thank Ernst-Jan Goedvolk from Telematical Instituut for his valuable help in this paper.

[1]Also known as digital rights language (DRL)

Proceedings of the Third International Conference WEB Delivering of Music (WEDELMUSIC'03)
0-7695-1935-0/03 $17.00 © 2003 IEEE

In this paper, we propose LicenseScript, a language that is able to express conditions of use of dynamic and evolving data in authorized domains. LicenseScript is based on (1) multiset rewriting, which is able to capture the *dynamic* evolution of licenses, (2) logic programming, which captures the static terms and conditions on a license, and (3) 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 precisely and clearly. The formal basis of LicenseScript (multiset rewriting and logic programming) provides for a concise and explicit formal semantics.

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

The organization of the remainder of the paper is as follows: Section 2 explains the LicenseScript language. Section 3 demonstrates examples for Electronic Music Distribution. Section 4 gives some related work. Finally, Section 5 gives the conclusions.

2 Language

In this section we describe the LicenseScript language. We start by introducing some basic concepts that are needed in the sequel.

2.1 Preliminaries

As mentioned earlier, LicenseScript is based on multiset rewriting. By a *multiset* (also known as a *bag*) we mean a set with possibly repeated elements; denoted with *brackets*. For example, $[a, b, b, c]$ is a multiset.

In our approach, licenses are bound to terms that reside in multisets. For the specification of these licenses, we use logic programming; the reader is thus assumed to be familiar with the terminology and the basic results of the semantics of logic programs [7]. We also use Prolog notation: we use words that start with uppercase $(X, Y, ...)$ to denote variables, and lowercase $(music\_piece, video\_track, expires, ...)$ to denote constants. We work with *queries*, that is sequences of atoms. Further, given a syntactic construct E (so for example, a term, an atom or a set of equations) we denote by *Var*(E) the set of the variables appearing in E. Given a substitution $\sigma = \{x_1/t_1, ..., x_n/t_n\}$ we denote by *Dom*(σ) the set of variables $\{x_1, \ldots, x_n\}$. A substitution σ is called a *matching substitution* of terms t and s if $t\sigma = s$, and *Dom*(σ) = *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.

We also borrow the concept of *SLD-resolution* [7] from logic programming:

Definition 1. Given a program P, and a query (i.e., a conjunction of atoms) Q, we write $P \vdash_{SLD} Q$ (or simply $P \vdash Q$) when there is a successful SLD-derivation for query Q in program P. A successful execution of a query may result in a (computed answer) substitution. $P \vdash Q$ basically means that execution of the query Q in the program P yields *success*.

2.2 Licenses

A license defines the terms and conditions of use for music, videos etc. Therefore, a license contains at least two relevant items of information: (i) a reference to the *data* that is being licensed, and (ii) the *conditions of use* on that data.

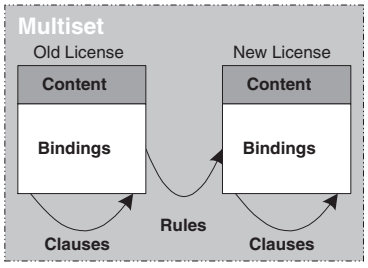

Figure 1. The transformation of licenses with *content* and *bindings* in a multiset caused by rules.

In our formalism, a license is represented by a term of the form $lic(content, \Delta, B)$ (as can be seen in Figure 1 where:

- *content* is a unique identifier representing the data the license refers to.
- Δ is a set of *clauses*, i.e., a Prolog program. This program defines when certain operations (like *play*) are allowed.
- B is a set of *bindings*, i.e., a set containing elements of the form $name \equiv value$. For instance

Proceedings of the Third International Conference WEB Delivering of Music (WEDELMUSIC'03)
0-7695-1935-0/03 $17.00 © 2003 IEEE

$\{expires \equiv 10/10/2003\}$ is a set of bindings with just one element.

Bindings are needed to have 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 an *interface*, i.e., a set of reserved calls that form the "API" of the license. The precise definition of this interface is beyond the scope of this paper. For example, 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 piece a to be played. Notice that we passed the set of bindings B as an argument to the query.

Example 2.

1. The following license allows to play mus:

   $$lic(mus, \{canplay(X, X) : -true.\}, \{\})$$

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

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

   allows to play *a* until the given expiration day.

$today(D)$ and $get\_value(B, n, V)$ are two primitives 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. Notice that there can be many domains in which licenses reside, and probably a domain will have different meanings for the primitives than another domain.

There are situations in which the "execution" of a license should be followed by the creation of a new set of bindings for the next step in the evolution of the license. Consider for instance 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 $set\_value(OldB, name, value, NewB)$. This primitive allows a name from a binding to be associated with a new value, which we use to support the evolution of licences. Consider, for instance, the following license: $lic(a, \Delta, \{played\_times \equiv 3\})$, where Δ consists of the following clause:

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

Here, we first extract the value of variable $played\_times$ into local variable R. Then, if we have not exhausted the possible playing times allowed by the license (in this case, 10), we proceed to increase the value of $played\_times$ from bindings B to $R+1$, into new *output* bindings set B'.

2.3 The Rules

Licenses typically reside inside a device. The modelling of communication between devices and the licenses is done by means of *rewrite rules*. These rules can be thought as the *firmware* of the device; licenses may come and go from a device, but the rules are fixed into the device (however, rules can be 'updated' once in a while). The syntax of rules we adopt is that of *multiset rewriting* (we use, in particular, Gamma notation [1]).

Definition 3. A rewrite rule is a 4-tuple

$$rule(arg) : lms \rightarrow rms \Leftarrow cond$$

where $rule(arg)$ is an atom called *rule label*, lms and rms are two multisets, and $cond$ is a sequence of elements of the form $P_i \vdash Q_i$.

Notice that a substitution σ can be applied in a natural way to a rule: $\sigma(rule(arg) : lms \rightarrow rms \Leftarrow cond) = rule(\sigma(arg)) : \sigma(lms) \rightarrow \sigma(rms) \Leftarrow \sigma(cond)$.

Intuitively, rules can be applied to a "target" multiset *MS*, if the following two conditions hold:

- First, the left multiset lms has to match some sub multiset of *MS*; this sub multiset is to be replaced with (right) multiset rms;
- Second, the conditions $cond$ of the rule must hold; This is done by executing all the queries in $cond$, and checking that the result is *success*.

We formalise the meaning of rule execution in the next section. An example for a rule is the following:

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

Proceedings of the Third International Conference WEB Delivering of Music (WEDELMUSIC'03)
0-7695-1935-0/03 $17.00 © 2003 IEEE

This rule can be applied to a license $lic(X, \Delta, B)$, replacing it with another license $lic(X, \Delta, B')$ if condition $\Delta \vdash canplay(B, B')$ holds.

2.4 LicenseScript Execution Model

As we already mentioned, licenses are represented by terms of the form $lic(content, \Delta, B)$. For the sake of exposure, we assume that all available licenses are stored in a given multiset *MS*.

Definition 4. Rule execution. Given two multisets *MS* and *MS'*, a rule $label : l \rightarrow r \Leftarrow cond$, and an atom a (called the request action), we write *MS* $\xrightarrow{a\sigma}$ *MS'* if:

1. a matches with $label$, with matching σ_1.
2. $l\sigma_1$ matches with T', with matching σ_2, for some sub multiset T' of *MS*.
3. For each $\Delta_i \vdash \phi_i$, $i \in [1, n]$ in $cond$, $(\Delta_i \vdash \phi_i)\sigma_1\sigma_2\delta_1 \cdots \delta_{i-1}$ succeeds, with computed answer substitution δ_i;
4. *MS'* is the result of removing each term in T' from *MS*, and appending $r\sigma_1\sigma_2\delta_1 \cdots \delta_n$ to it, then, σ is the composition of σ_1, σ_2, $\cdots$, i.e. $\sigma_1 \circ \sigma_2 \circ \cdots$.

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. Notice that there may be more than one rule that matches with the request action, so a first source of non-determinism appears here. The request action can also *fail* if no rule matches with the request. After a rule is chosen, *Step 2* finds a set of licenses T' in the multiset *MS*to which the rule can be applied. Again, different choices of T' may produce non-determinism here. This corresponds to the possible situation in which the user possesses more than one license (or set of licenses) that allows her to effectuate the desired action. In this case, we can assume that the system asks the user which license should be used. *Step 3* checks that the conditions of the rule hold, by executing the queries with the carried out substitutions. Finally, *Step 4* transforms the multiset by applying the replacement specified by the rule.

Example 5. Consider the multiset containing the following licenses: *MS* $= [lic(music, \Gamma, C), lic(video, \Sigma, D)]$, where $C = \{played\_times \equiv 2\}$, $D = \{played\_times \equiv 10\}$, and $\Gamma = \Sigma =$

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

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

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

1. Suppose the environment requests the action $play(music)$. This 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 $canplay(C, B')$ 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 $C' = \{played\_times \equiv 3\}$.

Example 6. Consider the same multiset and rules of the previous example. Suppose now request action $play(video)$ is issued. This action, even though has a matching rule and a matching license in the multiset, cannot be carried out completely. This is so since, in the unique matched license (that is, $lic(video, \Sigma, D)$) condition $\Delta \vdash canplay(B, B')$ does not hold, the video has been played too often.

Definition 4 describes how a multiset can evolve to another by means of executing a rule. The precise notion of multiset execution, which can be understood as the semantics of LicenseScript, can be then described as sequences of rule executions.

Definition 7. Given a multiset MS and a set of rules R, an *execution* is the (possibly infinite) sequence of rule applications $MS \xrightarrow{a_1\sigma_1} MS_1 \xrightarrow{a_2\sigma_2} MS_2 \cdots$. The *trace execution* of MS is defined as $a_1\sigma_1 \cdot a_2\sigma_2 \cdots$.

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

Proceedings of the Third International Conference WEB Delivering of Music (WEDELMUSIC'03)
0-7695-1935-0/03 $17.00 © 2003 IEEE

3 Electronic Music Distribution

In this section we provide examples in Electronic Music Distribution, to show the flexibility of LicenseScript.

3.1 Authorized Domains

As explained in the introduction, an authorized domain can be seen as the collection of devices belonging to a household. In this paper we only focus on the relationship between licenses and ADs, and we assume the existence of an AD implementation that deals with domain and content management issues (see [12]), i.e. we assume the presence of compliant devices that are governed by AD management rules. We first show how a license can be bound to a specific domain. Consider the following license:

$$lic(mus, \Delta, \{in\_domain \equiv cert\})$$

where $\Delta =$

$$\{validD(B) \ :- \ get\_value(B, in\_domain, Id_2), identify(Id_1), Id_1 = Id_2.\}$$

Here, $identify(Id_1)$ is a primitive which is used to retrieve the identity of the current domain. Clause $validD(\cdot)$ checks that the current domain is in fact the domain to which the license is bound. A rule like $play(\cdot)$, for instance, can now be defined as only valid if the license is in the allowed domain:

$$play(Mus) \ : \ lic(Mus, \Delta, B) \rightarrow lic(Mus, \Delta, B) \Leftarrow \Delta \vdash validD(B)$$

Now we are ready to illustrate a slightly more complex example: Consider a license that allows a certain piece of music to be played only for a limited time within the domain. This license can be written as follows:

$$lic(mus, \Delta, \{in\_domain \equiv cert, expires \equiv expiration\_date\})$$

where Δ consists of the following two clauses:

$$validD(B) \ :- \ get\_value(B, in\_domain, Id_1), identify(Id_1), Id_1 = Id_2.$$

$$nexpired(B) \ :- \ get\_value(B, expires, Exp), today(D), D \leq Exp.$$

Finally, the corresponding *play(·)* rule can be defined:

$$play(Mus) \ : \ lic(Mus, \Delta, B) \rightarrow play(Mus, \Delta, B) \Leftarrow \Delta \vdash validD(B), \Delta \vdash nexpired(B)$$

3.2 Payment

To address the different forms of payment, we first model a *wallet*. Then, we show how various payment methods are implemented in LicenseScript.

3.2.1 Modelling a Wallet

We model the existence of a *wallet*, in our approach, as a another element of the multiset. The wallet is represented as a term of the form *wallet*(Δ, B), where Δ is a Prolog program, and B are a set of bindings. Similarly to licenses, in the wallet we have clauses that allow rules to perform operations over the wallet. We assume that one binding named m, which represents the amount of money in the wallet, is always in B.

A clause that may reside in Δ is *canload*$(\cdot)$, which is used to load or increase the balance of the wallet, as can be shown as follows:

$$canload(A, B, B') \ :- \ get\_value(B, m, M), set\_value(B, m, M + A, B').$$

where A is the amount of money the user likes to load onto the wallet.

Using clause $canload(\cdot)$, a rule that loads money into the wallet can now be written:

$$load(Amount) \ : \ wallet(\Delta, B) \rightarrow wallet(\Delta, B') \Leftarrow \Delta \vdash canload(\textbf{\textit{Amount}}, B, B')$$

Another useful clause in the wallet is $cantransfer(\cdot)$, used to transfer a certain amount of money to another entity (e.g., a content provider):

$$cantransfer(P, A, B, B') \ :- \ get\_value(B, m, M), A \leq M, set\_value(B, m, M - A, B'), transfer(P, A).$$

where primitive $transfer(P, A)$ models the money transfer to entity P of the amount of money A.

3.2.2 Payment Methods

There are at least three common alternatives payment models, as described in [3]: (1) pay *per*-use, a payment is issued *each time* the content is used; (2) pay *upfront*, the content can be used after the payment has taken place, for a period of time p; and (3) pay *flatrate*: The content is used, and then the payment must be made at the end of the content usage We now illustrate the implementation of the pay per-use and pay upfront methods in LicenseScript. We leave aside pay flatrate, as it is similar to pay per-use.

Proceedings of the Third International Conference WEB Delivering of Music (WEDELMUSIC'03)
0-7695-1935-0/03 $17.00 © 2003 IEEE

Pay Per Use We can model pay per-use in LicenseScript by means of including in a license the following clause $canplay(\cdot)$:

$$\begin{aligned} canplay(P, A, B) \ :- \\ \quad get\_value(B, provider, P), \\ \quad get\_value(B, amount, A). \end{aligned}$$

where $provider$ is the binding representing the content provider, while binding $amount$ represents the cost to play music track. Intuitively, clause $canplay$ returns the price of the content in A and the provider who should receive the payment, in variable P. This allows a rule calling this clause to perform the required payment:

$$\begin{aligned} play(Mus) \quad : \quad & lic(Mus, \Delta, B), wallet(\Gamma, C) \rightarrow \\ & lic(Mus, \Delta, B), wallet(\Gamma, C') \\ \Leftarrow \quad & \Delta \vdash canplay(P, A, B), \\ & \Gamma \vdash cantransfer(P, A, C, C') \end{aligned}$$

Here, clause $canplay(P, A, B)$ retrieves P and A, which clause $cantransfer(P, A, C, C')$ uses to perform the actual money transfer.

Pay Upfront For modelling this payment method, we need to use two different clauses and rules, since the actual payment and content usage may differ in time: the payment is first done, and only later the content is used.

We first define a clause $canpay(\cdot)$ for paying as follows:

$$\begin{aligned} canpay(P, A, B, B') \ :- \\ \quad get\_value(B, paid, Paid), Paid = false, \\ \quad get\_value(B, provider, P), \\ \quad get\_value(B, amount, A), today(D), \\ \quad set\_value(B, period, D + fp, B'), \\ \quad set\_value(B, paid, true, B'). \end{aligned}$$

Here, binding $paid$ is a flag that represents whether the content has already been paid or not. Binding $period$ is used to store the allowed period of time for playing the content, and constant fp represents the period of time in which the content can be accessed after the payment.

Clause $canpay(Provider, Amount, B, B')$ first checks that the payment has not been done yet. It then returns the value of the provider and the amount in variables $Provider$ and $Amount$. After this, the period of allowed use is set appropriately, and finally flag $paid$ is set to $true$, indicating the payment.

Using the above clause we can define the rule for pay upfront:

$$\begin{aligned} pay(Mus) \ : \ & lic(Mus, \Delta, B), wallet(\Gamma, C) \\ & \rightarrow lic(Mus, \Delta, B'), wallet(\Gamma, C') \\ & \Leftarrow \Delta \vdash canpay(Provider, Amount, B, B'), \\ & \Gamma \vdash cantransfer(Provider, Amount, C, C') \end{aligned}$$

Now, we can define the $canplay(\cdot)$ clause for action *play*, which will allow the content to be played only if the payment has been done, and the allowed period of time has not expired:

$$\begin{aligned} canplay(B) \ :- \\ \quad get\_value(B, paid, Paid), Paid = true, \\ \quad today(D), get\_value(B, period, P), D \leq P. \end{aligned}$$

Finally, we define the rule for $play(\cdot)$:

$$\begin{aligned} play(Mus) \quad : \quad & lic(Mus, \Delta, B) \rightarrow lic(Mus, \Delta, B) \\ \Leftarrow \quad & \Delta \vdash canplay(B) \end{aligned}$$

3.3 Clipping Licenses

Suppose a user who has purchased a music track from a content provider requires some comments from other users. In LicenseScript, she can *clip* the license, and then she can send the clipped licenses to other people as a preview or recommendation. Notice that this operation splits the license but not the content.

The license in question looks like this:

$$lic(mus, \Delta, \{start \equiv 0, end \equiv mus\_length\})$$

where Δ contains the following $canclip(\cdot)$ clause:

$$\begin{aligned} canclip(S, E, B, B') \ :- \\ \quad get\_value(B, start, OS), \\ \quad get\_value(B, end, OE), S \geq OS, \\ \quad E \leq OE, set\_value(B, start, S, B'), \\ \quad set\_value(B, end, E, B'). \end{aligned}$$

Bindings $start$ and end are markers that indicate the head and tail of the music track.

The corresponding rule for $clip$ operation can now be defined:

$$\begin{aligned} clip(S, E, Mus) \ : & lic(Mus, \Delta, B) \rightarrow \\ & lic(Mus, \Delta, B), lic(Mus, \Delta, B') \\ & \Leftarrow \Delta \vdash canclip(S, E, B, B') \end{aligned}$$

Note that the content is still the same, full-length here; only the start and end markers are modified. A

Proceedings of the Third International Conference WEB Delivering of Music (WEDELMUSIC'03)
0-7695-1935-0/03 $17.00 © 2003 IEEE

different clip operation that includes the actual production of a new, clipped content, would need the use of a primitive that performs the operation.

A special case of the *clipping* operation occurs when we want to duplicate a license. This operation is often needed. In fact, one of the primary requirements of LicenseScript architecture is that devices do not have to be always on: in particular, we do not want the system to be dependent from the reacheability and the availability of *domain server*. To implement correctly the concept of authorized domain, we then have to be able to duplicate licenses. Consider for instance the situation of a person that has the license for listening to a piece of music within his home and who rightfully wants to listen to it also while driving her car. If devices are not always on, then the car device might be incapable of checking on the home server for the presence of the right license. Therefore, there has to be a license for the music in the car device as well, and this is possible only if we can duplicate licenses. Thus, by allowing users to duplicate the licenses confined in their authorized domain without any restrictions provide a broad freedom for the users to listen to the music *whenever*, *wherever* and *however* they like.

4 Related Work

In this section, we briefly discuss the related work. As mentioned in section 1, there are several XML-based RELs proposed, the most prominent being XrML and ODRL. The crucial difference between LicenseScript and XML-based RELs, is that the former is a (Turing-complete) language, while the latter are only suitable to describe a set of constraints, the semantics of which is given by the implementation algorithm. This make it very difficult to compare the two approaches. A sure problem with XML-based RELs is that their syntax becomes intricated when the scenarios of licensing and content usage patterns becomes complex. As a result, the license may expand drastically in size. We aim to render the LicenseScript lightweight to be fit into small devices, i.e. with limited resources (CPU, memory, etc.). Eventually, we could compile XrML and ODRL into LicenseScript (which can be thought of as an "intermediate code") to be accommodated in small devices. LicenseScript, being executable, allows to formulate complex policies in a succinct manner like: *Pay 1USD to videos.com when you view this video for the first time; each subsequent time that you view this video the cost drops by* 10%. *After ten times, the video becomes free*. To the best of our knowledge, to implement such a policy in XrML one needs to define an extension of the language, and needs to provide an implementation to it.

On the other hand, we found other concepts that are easily expressed in XrML or ODRL but would require much more work in LicenseScript. Consider, for instance, the use of *role based access control* in ODRL. This allows one to specify, for example, "any student of this university may listen to the local university radio." Here, the right to listen to the radio is assigned to the *role* "student", and not to each physical student. Then, there is no need to deal with identification or authorization mechanisms at the specification level. In LicenseScript, on the other hand, this would not be possible because there is no way to "abstract" such high-level descriptions. In this specific case, to support role based access control we would need to implement identification and authorization procedures (possibly using a PKI infrastructure) in the same license.

Gunter et al. [3] from InterTrust Technologies Corporation and Pucella and Weissman [9] from Cornell University present two logics for licenses. By borrowing techniques from programming semantics [4], 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 modelled as a pair of a time value and an action. Note that 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 in order 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. formularize several standard license types, which they call *simple licenses*. They are the same that 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 [9] 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 distinction is what 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 denotational semantics of

Proceedings of the Third International Conference WEB Delivering of Music (WEDELMUSIC'03)
0-7695-1935-0/03 $17.00 © 2003 IEEE

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. To state the obvious, Pucella et al. also have 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.

5 Conclusions and Future Work

We propose LicenseScript, a novel digital rights language based on multiset rewriting and logic programming. We present the design of the language using a scenario that represents an elaborate pattern of content use.

LicenseScript differs from other RELs in that it has an explicit static and dynamic part. The terms and conditions on content form the static part. These terms and conditions usually derive from legal, regulatory and business rules, and are therefore appropriately expressed using Prolog clauses [11]. A license is used in a changing context and must therefore have the ability to evolve. The dynamics are represented by interpreting a license as an element of a multiset to which multiset rewrite rules are applied. These rules represent the way in which the context (devices and systems) act upon licences. The dual nature of a license (static versus dynamic) is thus represented by a two-tier structure of LicenseScript. The two levels are linked by a set of bindings that represents the current state of the evolution.

As future work, we are currently implementing the language, using an existing DRM platform [2]. Furthermore, we plan to study in detail relevant legal, regulatory and business cases to ensure that the language is convenient to use. Licenses evolution is an interesting issue, and we plan to investigate this point further. Formal verification of license properties (e.g., safety) in a given multiset and rule set, could allow us to reason more precisely about what a license is supposed to achieve and what *actually* allow.

References

[1] 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 *Lecture Notes in Computer Science*, pages 17–44. Springer-Verlag, Berlin, August 2001.

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

[3] 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.

[4] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992. ISBN: 0262071436.

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

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

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

[8] D. Mulligan and A. Burstein. Implementing copyright limitations in rights expression languages. In *Proceedings of 2002 ACM Workshop on Digital Rights Management*, volume 2696 of *Lecture Notes in Computer Science*, page To appear. Springer-Verlag, November 2002.

[9] 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.

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

[11] M. J. Sergot, F. Sadri, R. A. Kowalski, F. Kriwaczek, P. Hammond, and H. T. Cory. The british nationality act as a logic program. *Communications ACM*, 29(5):370–386, May 1986.

[12] 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 *The World's Electronic Media Event IBC 2002, Sept. 13-17, Amsterdam RAI, The Netherlands*, pages 467–474, September 2002.

Proceedings of the Third International Conference WEB Delivering of Music (WEDELMUSIC'03)
0-7695-1935-0/03 $17.00 © 2003 IEEE