How to Pay in LicenseScript*

Ricardo Corin, Cheun Ngen Chong, Sandro Etalle, and Pieter Hartel

University of Twente, The Netherlands
{corin,chong,etalle,pieter}@cs.utwente.nl

July 28, 2003

Abstract

Current DRM systems do not provide flexible payment methods, requiring the user to handle the payment by hand. For instance, when the user needs to pay for watching a movie, she needs to decide which available payment method is the most optimal and suitable. This is a rather cumbersome process for the user that can be avoided by the specification of *payment policies*. A payment policy automates the payment process of each content usage, choosing the best alternative among the possible payment methods.

In this paper, we show how LicenseScript is able to model payment policies, allowing the user to precisely specify how a payment of content usage should be performed.

1 Introduction

Most information, such as books, music, video, personal data and sensor readings 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 this paper we use LicenseScript [1], a language that is able to express conditions of use of dynamic and evolving data. LicenseScript is based on multiset rewriting, which is able to capture the *dynamic* evolution of licenses, and logic programming, which captures the static terms and conditions on a license.

Besides LicenseScript, other rights expression languages (RELs) have been proposed. Two XML-based RELs are amongst the most popular: XrML [2] and ODRL [3]. Unfortunately, these languages only focus on rights management description and specification. Thus, they do not handle payment methods flexibly. These RELs, for instance, merely facilitate the bank information and content provider's bank account specification (to where the content users may transfer the money.) In other words, the user is not provided with other alternatives of payment methods.

On the other hand, we show in this paper how the flexible design of LicenseScript allows users to specify easily payment policies. As Chong et al. [1] proposed, payment can be modelled in LicenseScript by a *wallet*, which contains the user money. As an example, consider a piece of music that is licensed on a pay-per-view basis. Everytime we play this music, we need to pay first. To do so, we look for our wallet, and make the payment transfer. This payment method is a very simple one; in fact, it was intentionally oversimplified. Two deficiencies of Chong et al.'s model [1] are:

1. It is unrealistic: A user usually has many sources of money, for instance, the user's real wallet, her bank account and probably also her savings account. Furthermore, the user may have other special values such as money coupons or *air miles*. Each of these sources has different characteristics. A user may like to use these special values in a different situation.

2. It is inflexible: The user does not have the oppor-

*This work is sponsored by Telematica Instituut, The Netherlands.

tunity to decide or choose how to perform the payment.

In this paper we will consider an extension of the above model to address the above mentioned deficiencies. In particular, to tackle the deficiency 1 we allow the existence of many wallets, each one with different attributes. To solve the deficiency 2, we allow the user to specify a *payment policy* that can precisely decide how to perform a payment.

2 The LicenseScript Language

In this section we briefly describe the LicenseScript language. We refer the interested reader to previous work [1] for a more detailed treatment.

2.1 Preliminaries

As mentioned earlier, LicenseScript is based on multiset rewriting. By a *multiset* we mean a set with possibly repeated elements. For example, $[a, b, b, c]$ is a multiset.

In LicenseScript, 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 [4]. 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.

2.2 Licenses

A license contains two relevant items of information: (i) *Content*: a reference to the data that is being licensed, and (ii) *Clauses + Bindings*: the *conditions of use* on that data.

A license is represented by 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 and B is a set of *bindings*, i.e., a set containing elements of the form $name \equiv value$. Intuitively, Δ contains queries that will answer, according to the conditions of use, whether a certain operation is applicable. The bindings B represent the state of the license, where the clauses Δ can access.

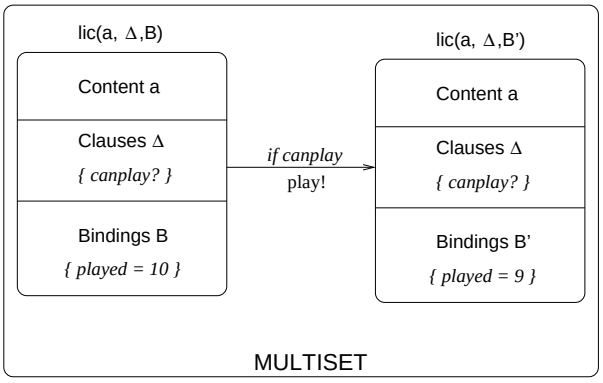

Figure 1: The transformation of license $lic(a, \Delta, B)$.

Example 1.

1. *Play n times license*.
   The following license allows a to be played at most 10 times (see Figure 1).

   $$lic(a, \Delta, \{played\_times \equiv 10\})$$

   where Δ is

   $$\{canplay(B, B') :- \\ get\_value(B, played\_times, N), \\ N > 0, \\ set\_value(B, played\_times, N - 1, B').\}$$

   where primitive $get\_value(B, n, V)$ returns in V the value of binding n according to B, and primitive $set\_value(B, n, m, B')$ sets value m in binding n of B, into B'.

2. *Play until expiration date license.*
   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.

3. *Pay-per-view license.*
   The license $lic(a, \Delta, \{cost\_per\_view \equiv 10\$, provider \equiv \text{“}Records\ \ Inc''\})$ where Δ is

$$\{canplay(B, B, C, P) :- \\ get\_value(B, cost\_per\_view, C), \\ get\_value(B, provider, P).\}$$

   allows to play a, but each time returns the cost C of the playing, along with information about the provider P needed for the payment.

2.3 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' securely once in a while). The syntax of rules we adopt is that of *multiset rewriting*. An example for a rule is the following:

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

where $lic(X, \Delta, B)$ is like in Example 1.1. Here, $\Delta \vdash canplay(B, B')$ means that clauses Δ make query $canplay(B', B')$ be succesful. Intuitively, this means that license $lic(X, \Delta, B)$ can be played. Thus, the above 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.

3 Payment

In this section, we introduce our wallets representation in LicenseScript. Then we provide a simple method to specify the payment policies. To elaborate these payment policies, we provide several viable examples.

3.1 Wallets

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

Example 2. Consider wallet $wallet(\Gamma, B)$ where $B = \{type = bank\_account, money \equiv 1500\$, interest \equiv 0.5\%, bank\_charges \equiv 1, bank\_address \equiv bc\}$ and Γ contains clauses to load and transfer money:

$$canload(B, B', A) \ :- \\ \quad get\_value(B, money, M), \\ \quad get\_value(B, bank\_address, C), \\ \quad set\_value(B, money, M + A, B'), \\ \quad transfer(C, A). \\ cantransfer(B, B', P, A) \ :- \\ \quad get\_value(B, money, M), \\ \quad get\_value(B, bank\_charges, C), \\ \quad C + A \leq M, \\ \quad set\_value(B, money, M - (A + C), B'), \\ \quad transfer(P, A).$$

Here, A is the amount of money the user likes to load onto the wallet and the primitive $transfer(P, A)$ models the money transfer to entity P of the amount of money A; if P is the bank itself, it will not charge any *bank_charges*, otherwise it will.

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

In the multiset of LicenseScript, all the wallets of a user will be gathered in a special term, namely the *wallet manager*, denoted $wm(\Psi, L)$. Here, L is the list of wallets, while Ψ will contain 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*. Because of space constraints we will not specify these clauses here.)

3.2 Payment Policies

Now that we have the wallets and the wallet manager, we need to specify a payment policy that will

decide 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 greater weight assigned by p will be preferred for payment than a wallet with smaller 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 $W = max(\{W_i \mid w \in L \wedge p(w, W_i)\})$.

3.2.1 Implementing payment policies in LicenseScript

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

- First, the weight predicate $p(Wallet, Weight)$ that assigns weight $Weight$ to wallet $Wallet$.
- Second, clause *select*. Intuitively, *select*(L, C, W) will select wallet W from L, the wallet with maximum weight according to p, that has enough money to pay C:

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

where $map(L, L')$ returns L', a list of pairs (*wallet*,*weight*) s.t. p(*wallet*,*weight*) for each *wallet* in L. Furthermore, clause $sort(L', L'')$ sorts list L' in L'': $L'' = [(wallet_1, weight_1), ..., (wallet_n, weight_n)]$ 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 skip 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:

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

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

3.2.2 Rules

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 a policy.

Consider the following rule:

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

where Δ and B are as in Example 1.3. This *play* rule first selects a license $lic(X, \Delta, B)$, a policy $policy(\Psi)$ and the wallet manager $wm(\Gamma, L)$. Execution of this rule will be carried out succesfully 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).

3.3 Examples

1. We first model the simplest policy of Chong et al. [1]: 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 modelling the free choice. Notice that, alternatively, defining $p(\mathit{Wallet}, c)$ 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 (as in Example 2) to perform the payment. Suppose binding $\mathit{bank\_charges} \equiv C$ is in B for wallet $wallet(\Gamma, B)$, and $C > 0$. We set

$$p(wallet(\Gamma, B), \mathit{Weight}) :- \\ \quad get\_value(B, \mathit{bank\_charges}, C), \\ \quad \mathit{Weight} = -C.$$

   Thus, *p(Wallet,Weight)* will set as *Weight* of wallet *Wallet* the negative 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:

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

4 Conclusions & Future Work

In this paper we propose a technique to specify payment policies to optimise payment strategies for content usage in LicenseScript [1]. We believe this technique blends smoothly with the LicenseScript framework thanks to the flexibility of the LicenseScript design. In particular, since LicenseScript allows the use of Prolog as description language for licenses, it allows us to write complex payment policies in simple and short Prolog programs.

One way to extend our work would be to modify the *select* clause of a policy, to return not only *one* wallet but a set of wallets to perform a payment. This would allow us to address "expensive" payments that can be split amongst several wallets. For example, we could specify a payment policy that first uses all the "airmiles" wallets, then "moneycoupons" wallets and so on.

Another possible extension would be to specify a *selection policy*, that is a policy that selects the most appropiate payment policy to perform the payment of a given operation. This selection policy would help into selecting the right payment policy for each payment. For example, suppose we have two payment policies A and B in the multiset, and we want to pay for "watching movies" operations according to policy A, and "listen to music" operations according to policy B.

References

[1] 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 *3rd Int. Conf. on Web Delivering of Music (WEDELMUSIC)*, page to appear, Leeds, UK, Sep 2003. IEEE Computer Society Press, Los Alamitos, California. `http://www.ub.utwente.nl/webdocs/ctit/1/000000bf.pdf`.

[2] H. Guo. Digital rights management (drm) using xrml. In *Telecommunications Software and Multimedai TML-C7*, 2001. http://www.tml.hut.fi/Studies/T-110.501/2001/papers/.

[3] 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/.

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