Licensing Structured Data with Ease

Yee Wei Law[*], Cheun Ngen Chong[†], Sandro Etalle, Pieter Hartel, and Ricardo Corin

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

August 1, 2003

Abstract

In response to the need of a rights expression language (REL), we have proposed LicenseScript, an REL based on multiset rewriting and Prolog. LicenseScript has advantage over existing RELs, in the sense that it has a well-defined semantics. In fact besides semantics, LicenseScript has a lot of other advantages over other RELs. The mission of this paper is twofold: (1) to put a spotlight on these advantages, (2) at the same time justifying some of our design rationales in LicenseScript. We accomplish this by giving examples of licensing models that are greatly facilitated by the use of Prolog as a component of LicenseScript. At the same time showing how LicenseScript makes these non-trivial models viable, we also make LicenseScript a stronger case than it previously might have occurred to be.

Keywords: DRM, REL, Prolog, multiset rewriting, security.

1 Introduction

DRM is a controversial technology. If not treated carefully, DRM may do more harm than good, and so far DRM has a bad track record of upholding the legal rights of fair use [9, 16]. This state of DRM ('Digital Restrictions Management') more or less stems from the inflexibility of the licenses content providers impose on their contents. We would argue that if a DRM system is excessively restrictive, users would choose not to use it; for instance the reason why Apple's iTunes (`www.apple.com/itunes`) is gaining market share is probably because it is offering an unprecedented amount of freedom.

This paper does not intend to debate further the good or bad of the technology, but sees it as an inevitable development (e.g. Nokia's Content Publishing Toolkit, Microsoft Media DRM etc.) and tries to bring more good than bad, with our rights expression language (REL), LicenseScript.

Motivation The LicenseScript language [5, 4] provides an answer to one of the essential ingredients of DRM systems [12], i.e. REL. However with competition coming from the eXtensible rights Markup Language (XrML)(`www.xrml.com`) and Open Digital Rights Language (ODRL)(`www.odrl.net`), we feel the need to present LicenseScript as an even more convincing case than we have previously done. In this paper, we are going to highlight, elaborate and explain some of our design choices of the LicenseScript language, using application scenarios of structured data. Basically we are going to answer the following questions:

1. LicenseScript is a Prolog-based REL and hence a program-based REL, but why a program-based instead of XML-based REL?
2. Why Prolog?
3. How do you handle the security of executable code?

The main theme of this paper lies in our answer to question (2). Note that for the rest of the paper we will use the term 'data' and 'content' interchangeably, for the reason that 'content' is more in-tuned with our theme of content protection, but 'content' is just anyway a form of 'data'.

Organization This paper is laid out as follows. Section 2 introduces the LicenseScript language. Section 3, 4 and 5 answer question (1), (2) and (3) respectively. Section 6 concludes.

2 The LicenseScript Language

We have already described the basic abstractions and semantics of LicenseScript in our two previous papers [5, 4], so we only provide a summary here. LicenseScript is based on multiset rewriting and Prolog. A multiset is a set with possibly repeated elements. Multiset rewriting allows us to express the *dynamic* evolution of licenses, whereas Prolog allows us to express non-trivial constraints in a declarative manner. The notion of 'constraint' in LicenseScript is general, i.e. as opposed to ODRL [11], it does not distinguish between 'constraint', 'requirement' and 'condition'. Next, we define the building blocks of LicenseScript: *licenses* and *rules*.

2.1 License

A license is a tuple written in the form $lic(N, C, B)$ where N is the *name* or identifier of the data the license refers to, C is the set of *Prolog clauses* that implement the license constraints, and B is the set of $(name, value)$ pairs

[*]PhD thesis submission date: December 2005.
[†]PhD thesis submission date: September 2004.

called *bindings* representing the state of the license. We often express a license as a Prolog term `lic(N,C,B)`, where `C` becomes a *list* of clauses and `B` becomes a list of lists of the form `[name,value]`. In this paper, we often use the abstract and Prolog representation interchangeably. A sample trivial license might be

```
lic(music,
  [canplay(B,Bprime):- get_value(B,paid,Paid),
  Paid==1], [[paid,1]])
```

Here, the `canplay(B,Bprime)` clause retrieves the value of `paid` from the list of bindings `B` and check if it is set to 1. `B` and `Bprime` refer to the list of bindings before and respectively after the license is updated. Since `Bprime` is a singleton, it could have been written as `_` instead. `get_value` is a primitive that reports in variable `Paid`, the value of the binding named `paid` in the list of bindings `B`. `set_value` is the counterpart of `get_value`. The arity of the predicate `canplay` is pre-defined and has to be synchronized with the corresponding `canplay` predicate in the rules, which are discussed next.

2.2 Rules

Now that we have the mechanisms of determining *which* data to protect (N), *what* actions and state variables to monitor (C and B), we should also have a way of expressing *how* the license transits from one state to another, i.e. how the license is updated. For this reason, we need *multiset rewrite rules*. To express these rules, we use an abstract syntax adapted from Gamma [2] instead of Prolog:

$$action(arg) : lms \rightarrow rms \Leftarrow cond \tag{1}$$

Intuitively, *action* represents the action we are regulating, permitting to happen if the condition *cond* is satisfied. *lms* and *rms* are the multisets that represent the state of the data and the license, *before* and respectively *after* the action. For ease of reference, we called *lms* and *rms* the left multiset and the right multiset. Since the data is usually unmodified by the action (e.g. play, render etc.), our examples usually do not include the state of the data in the multisets. *arg* represents the necessary arguments for executing the action. For example,

$$\begin{aligned}\texttt{play(N)} : \texttt{lic(N,C,B)} &\rightarrow \texttt{lic(N,C,Bprime)}\\ &\Leftarrow \texttt{C} \vdash \texttt{canplay(B,Bprime)}\end{aligned}$$

In this example, the left multiset and the right multiset each consists of a single license term. The semantics of this rule is basically that if there is a successful SLD resolution [3] of query `canplay(B,Bprime)` in the set of clauses `C`, the action `play` may be executed on the content identified by `N`, and the state of the corresponding license will be updated from `B` to `Bprime`. Note that the arity of `canplay` here corresponds exactly with the arity of `canplay` in the sample license of Section 2.1.

With licenses and rules, we now have at hand an adequate set of primitives. In the next section, we will explain the benefits of expressing license constraints in a programming language.

3 Why Program-Based REL?

We tackle this question from two fronts: *semantics* and *expressiveness*.

Semantics The strongest argument against XML-based RELs is that these RELs do not have formal semantics [10, 15]. LicenseScript, as a program-based REL, on the other hand has a well-defined semantics, partly inherited from the semantics of Prolog. The importance of semantics lie in that it can be used for verifying useful properties of a license, for example that the license expires exactly after five usage; and for validating a prototype implementation of the language itself [14]. In fact our motivation is inline with current industrial movement of applying formal semantics to open specifications such as XQuery and XPath [17].

Expressiveness By expressing license constraints in programmable clauses (which in our case are Prolog clauses), LicenseScript offers more flexibility than say mere comparing equalities and inequalities. But then again, pragmatism would probably argue that our approach is an overkill, using real-world scenarios such as `BuyMusic.com`'s license restrictions:

1. Songs can be transferred from the primary computer to another computer(s) for at most n_1 times.
2. Songs can be transferred to an approved portable device for at most n_2 times.
3. Songs can be burnt to a CD for at most n_3 times.

It is true that these restrictions can be expressed easily with mere bindings and (in)equalities. But we argue that

1. This is just a snapshot of the current state of the music industry. We are aiming for a wider application of LicenseScript on other kinds of data and in other markets.
2. The fact that these rules are in use does not mean that they are the best. The essence of *fair use* [9] is that as long as you are using the content for your own personal purpose, you should be able to transfer your songs for an unlimited number of times. The reality is that most content providers to date still do not know what non-trivial license restrictions to apply on their products, that would both be fair and appealing to the consumers.

Consequently, it is our mission in this paper to demonstrate a non-trivial application of our seemingly overkill approach that is actually justified, and this is what we are going to do in the next section.

4 Why Prolog?

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. Our notion of digital library does not carry the requirement that all or parts of the library need to be online. The practical requirement of transparent and seamless access still however applies though it does not concern us here. The obvious linkage between these resources provide a rich source of inspiration as to how we can license them. Based on this scenario, we propose three content licensing models that allow us to exploit some 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.

4.1 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. The catch is that some of these parts are locked. A locked content allows only *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 in principle works by arousing the user's interest in the locked parts of the digital library, by cultivating her interest in the free unlocked parts of the digital library.

We continue by giving examples how the licenses can be constructed. First, a locked audio resource can be associated with a license of the form:

```
lic(track1,
[
  canunlock(B,_,Code) :-
    get_value(B,unlock_code,Unlock_code),
    Code==Unlock_code
], [[unlock_code,1234]])
```

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` variable. The clause checks if the input code is valid by comparing it with the 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:

```
lic(track1,
[
  canplay(B,_,Code):-
    get_value(B,unlock_code,Code)
], [[unlock_code,1234]])
```

Notice that the `canplay` clause is in the unlocking license instead of 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:

$$\begin{aligned}
\texttt{play(N)}\ \texttt{:lic(N,C1,B1),lic(N,C2,B2)}& \\
\rightarrow \texttt{lic(N,C1,B1prime),lic(N,C2,B2prime)}& \\
\Leftarrow \texttt{C1} \vdash \texttt{canunlock(B1,B1prime,Code)},& \\
\texttt{C2} \vdash \texttt{canplay(B2,B2prime,Code)}&
\end{aligned}$$

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 (the *cond* in (1)), and additionally 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 [3].

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 only the unlocking license needs to be renewed.

4.2 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 we can say that part of the reason a customer buys these games is due to her antipication 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 breathe 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 a basic set of 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 routes 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 much clearer:

```
lic(tourist_attractions,
[
  canrender(B,_) :-
    get_value(B,acceptAddon,AcceptAddon),
    AcceptAddon==1
], [])

lic(basic_map, [canrender(_,_)],
    [[acceptAddon,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,1]])
```

Unfortunately 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 1. 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 2. 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 3. 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 $b \notin N_2$, $\dot{\exists} b_3$ 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 $b \notin N_1$, $\dot{\exists} b_3$ 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;
- $\forall b \in N_1 \cap N_2$, $\dot{\exists} b_{3(1)}$, $\dot{\exists} b_{3(2)}$ 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 realised.

4.3 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 pronounciation. 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 get:

```
lic(ebook,
[
  (candisplaytext(_,_)),
  (canrenderaudio(_,_)),
  (cansyncrender(_,_):-
    candisplaytext(_,_),
    canrenderaudio(_,_))
],[])
```

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:

$$\mathtt{syncrender(N)} : \mathtt{lic(N,C,B)} \rightarrow \mathtt{lic(N,C,Bprime)} \Leftarrow \mathtt{C} \vdash \mathtt{cansyncrender(B,Bprime)}$$

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

5 Security of Executable Code

Prolog is a high-level programming language. It does not expose any memory manipulation functions, thus buffer overflow-based attacks are impossible. System calls like *shell*() should obviously not be exposed to the license interpreter. However currently denial-of-service attacks are not only possible but also easy to implement, with clauses like `canplay(X) :- canplay(X)` that loop indefinitely. Like any other programming language, there is no termination guarantee in Prolog programs. It is also possible to construct licenses which consume such an amount of resources that overwhelms the system. As we envision deploying LicenseScript in resource-limited consumer devices, this issue is very real and is in dire need of being resolved. To solve this problem one could use a technique similar to *proof carrying code* [13], which is a framework in which untrusted programs are supplied to the user together with a proof of the safety properties they have to comply with. It is worth noting that such a proof is usually a short piece of code and this technique has already been successfully applied to Java programs [6]. In our case, the properties we are interested in are (a) termination, (b) limited use of resources and (c) non-interference with other licenses. We are working on a method to detect rogue licenses. In particular, for (a) we can use well-established techniques from logic programming [1, 8], for (b) we are considering whether it is possible to tackle the problem by extending the techniques used to prove termination, while for (c) we probably have to resort to mode systems or to abstract interpretation [7].

6 Conclusion and Future Work

Summarizing what we have achieved so far: For the content providers, we have provided them with three useful licensing schemes to use with LicenseScript. Some of these schemes are market-proven in the sense that successful examples can be found in the real-world, without sacrificing compliance with the legal rights of fair use. For the REL designers, we have shown not only the viability, but also the advantage of Prolog, justifying our seemingly radical approach.

For this paper, the license and rule code snippets have been successfully simulated with ECL^{i}PS^e (`http://www-icparc.doc.ic.ac.uk/eclipse/`), although there is not yet an implementation of LicenseScript. The implementation, which we call the LicenseScript Engine, is meant to be a software component. During our work in the LicenseScript Engine, we are faced with the interesting problem of bridging the semantic gap between the chemical reaction model (rules) and Prolog (licenses). Parallel with our implementation effort, we are in the process of defining verifiable properties of LicenseScript. For example, as mentioned in the previous section, it is important to detect rogue licenses that violate some of these properties.

Lastly, we believe that by exploiting the DRM technology in such a way that puts the content providers and the consumers in a win-win situation, the state of DRM does not need to be one of a Down-Right Mess as it is today [18].

References

[1] K. R. Apt and D. Pedreschi. Modular termination proofs for logic and pure Prolog programs. In G. Levi, editor, *Advances in Logic Programming Theory*, pages 183–229. Oxford University Press, 1994.

[2] J.-P. Banâtre and D. Le Mètayer. Gamma and the chemical reaction model: ten years after. In J.-M. Andreoli, H. Gallaire, and D. Le Mètayer, editors, *Coordination programming: mechanisms, models and semantics*, pages 1–39, 1996.

[3] P. Brna. Prolog Programming: A First Course. `http://cbl.leeds.ac.uk/~paul/prologbook/`, March 2001.

[4] 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. IEEE Computer Society Press, Los Alamitos, California, Sep 2003.

[5] C. N. Chong, R. Corin, S. Etalle, P. H. Hartel, and Y. W. Law. LicenseScript: A novel digital rights language. In R. Grimm and J. Nitzel, editors, *IFIP TC6 WG6.11 Int. Workshop for Technology, Economy, Social and Legal Aspects of Virtual Goods*, pages 104–115. Technical University Ilmenau, Germany, May 2003.

[6] C. Colby, P. Lee, G.C. Necula, and M. Plesko. A Proof-Carrying Code Architecture for Java. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV00)*, Chicago, July 2000.

[7] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Fourth ACM Symp. Principles of Programming Languages*, pages 238–252, 1977.

[8] S. Etalle, A. Bossi, and N. Cocco. Termination of well-moded programs. *Journal of Logic Programming*, 38(2):243–257, 1999.

[9] E.W. Felten. A skeptical view of DRM and fair use. *Communications of ACM*, 46(4):57–59, April 2003.

[10] C.A. Gunter, S. Weeks, and A. Wright. Models and languages for digital rights. In *34th Annual Hawaii Int. Conf. on System Sciences (HICSS)*, page 9076, Maui, Hawaii, Jan 2001. IEEE Computer Society Press, Los Alamitos, California.

[11] R. Iannella. *Open Digital Rights Language (ODRL) Version 1.1.* IPR Systems, August 2002.

[12] B. LaMacchia. Key Challenges in DRM: An Industry Perspective. In *2002 ACM Workshop on Digital Rights Management, DRM 2002*, volume 2696 of *LNCS*, page to appear. Springer-Verlag, 2002.

[13] G.C̃. Necula. Proof-carrying code. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[14] H.R. Nielson and F. Nielson. *Semantics with applications.* Wiley Professional Computing. Wiley, 1992.

[15] R. Pucella and V. Weissman. A logic for reasoning about digital rights. In *15th IEEE Computer Security Foundations Workshop*, pages 282–294, Cape Breton, Nova Scotia, Canada, Jun 2002. IEEE Computer Society Press, Los Alamitos, California.

[16] P. Samuelson. DRM {and,or,vs.} the Law. *Communications of ACM*, 46(4):41–45, April 2003.

[17] J. Siméon and P. Wadler. The essence of xml. *ACM SIGPLAN Notices*, 38(1):1–13, 2003.

[18] M. Walter, P. Evans, and M. Letts. DRM: 'Down-Right Messy' And Getting Worse. *The Seybold Report: Analyzing Publishing Technologies*, 1(3), May 2001.