ABSTRACT
We propose an operational model for formalising and enforcing rights expression languages based on the concept of a rights expression compiler. Our compiler transforms an XML-based rights expression into a programme for a virtual machine. This approach provides a formal way of defining semantics for rights expression languages that can be directly used in practice to enforce the expressions while ensuring their consistency and correctness. We further argue that our model eliminates a number of limitations in previous attempts to associate rights expression languages with formal semantics. We demonstrate the power and practicality of our model by using it to develop operational semantics for the OMA Rights Expression Language, from which a real interpreter can be derived with relatively little effort.

Categories and Subject Descriptors
K.6.5 [Management of Computing and Information Systems]: Security and Protection—Digital Rights Management; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Operational Semantics

General Terms
Algorithms, Languages, Security

Keywords
digital rights management, rights expression languages, operational semantics

1. INTRODUCTION
A rights expression language is a fundamental component of a digital rights management system. A rights expression language allows information owners to describe the actions that a user may take with their information, and to what constraints these actions are subject, such that these constraints can be automatically enforced by the digital rights management system.

The well-known rights expression languages ODRL [14] and XrML [5] are both XML-based languages defined in extensive English-language specification documents. A number of authors have argued, however, that the existing specifications are open to inconsistent interpretation by users and implementers [4, 9, 17], and that XML is relatively expensive for computers to interpret [4].

Several authors have attempted to analyse and disambiguate rights expression languages by deriving formal semantics for existing languages using some logical framework [2, 9, 10, 17]; by defining new languages with a formal basis [4, 7, 21]; or by proposing over-arching ontologies to which actual rights expression languages can be mapped [1, 8, 12, 18]. Section 2 gives an overview of these approaches and describes their limitations in expressing and enforcing rights of the kind contemplated by the designers of XrML and ODRL.

Section 3 proposes a new approach to defining the semantics of rights expression languages, in which XML-based rights expressions are defined in terms of the operations required to enforce them on a virtual machine. Semantics of this kind allow rights holders to construct rights expressions in a relatively high-level language, and have them automatically and unambiguously “compiled” into an efficient procedure for enforcing them.

This approach to formalisation is known as operational semantics in the theory of programming languages. While it may seem unusual to define the semantics of declarative languages like ODRL and XrML using a procedural mechanism, we argue that our semantics capture several significant aspects of rights expression languages and real interpreters that are not captured by existing semantics. In particular, our semantics do not require rendering actions to be instantaneous and infallible, and support constraints that modify the effect of an action rather than prohibit it outright.

We demonstrate the power and practicality of our approach by describing operational semantics for the OMA Rights Expression Language in Section 4. The OMA Rights Expression Language is a subset of ODRL, and is in many respects simpler than either ODRL or XrML, but it nonetheless contains several elements whose semantics do not seem...
to be captured by previous approaches to formalisation. Our semantics are complete, and can be used to develop an actual interpreter for the language with relatively little effort.

Finally, Section 5 discusses some observations on rights expression languages from the operational perspective. We will see that the common division of access control rules into “constraints” and “obligations” is somewhat arbitrary in the presence of stateful constraints. We also suggest a new approach to interoperability of rights expression languages based on our virtual machine, and consider the effect of the existence of multiple methods for enforcing a single constraint.

2. FORMAL RIGHTS EXPRESSIONS

Rights expression languages such as ODRL and XrML describe the rights offered by a rights-holder at a relatively high level. This high-level description is enforced by a digital rights management system according to an interpretation provided by the developers of the enforcement engine, with a risk that the developers of different enforcement engines may make different interpretations of the high-level description. Introducing formal semantics to rights expression languages eliminates this risk by associating every rights expression with an unambiguous intended meaning, and a number of authors have consequently proposed a variety of methods of formalising rights expressions.

2.1 Trace-based Semantics

Before XrML and ODRL became widely available, Gunter, et al. proposed a formal rights expression language in which the world is interpreted as a sequence of “payment” and “render” events that occur at particular times [7]. A licence is expressed as a set of permissible sequences, and a sequence of events is deemed to be acceptable if and only if it exists in the licence. Pucella and Weissman subsequently defined a logic for reasoning about such “trace-based” licences [16]. More recently, Xiang, et al. have proposed a more sophisticated trace-based formal rights expression language based on observational transition systems [21].

2.2 Logic-based Semantics

Weissman and colleagues have also proposed formalisations for both ODRL [17] and XrML [9] based on first-order logic, and used these formalisations to argue that the decision problem is NP-hard for ODRL, and undecidable for XrML. Weissman, et al. translate very restricted subsets of both languages, which include only a few permissions and constraints. Barth and Mitchell have also proposed a formalisation for the OMA Rights Expression Language based on linear logic [2], but exclude several time-based constraints that we will see are problematic for all of the formal languages noted in this section.

2.3 Automata-based Semantics

Holzer, et al. present an alternative formalisation for (a fragment of) ODRL based on finite automata [10]. Their fragment is somewhat larger than that used by Weissman, et al., and includes the most common permissions as well as device-based, time-based and count constraints. They do not describe a method by which automata can be constructed, however, and the examples in their paper are presumably constructed by hand. In this paper, we propose a compiler that can transform an ODRL or XrML expression automatically.

2.4 Abstract Rights Models

Several authors have proposed formal abstract rights models with the intention that the semantics of ODRL and XrML be defined in terms of these abstract models. Relatively early in the history of rights expression languages, Guth, et al. [8] proposed to model an ODRL-based licence in a contract schema that could be interpreted by an access control system. Later authors have proposed broadly similar, but more detailed models [1, 12, 18].

Guth, et al. use their model to construct a real interpreter by building a contract schema that can be processed by the policy decision point. This process has some similarities to the compiler-based method proposed in this paper. Our compiler, however, provides detailed semantics for the individual elements of a rights expression language, and avoids some limitations in both the logic-based and ontology-based formalisations that we will describe below.

2.5 LicenseScript

Finally, the approach described in this paper is significantly influenced by LicenseScript, a logic-based rights expression language proposed by Chong, et al. [4]. A LicenseScript licence associates each permission with a Prolog-like programme, such that the programme returns “true” if and only if that permission is granted. The digital rights management regime proposed by the Marlin Developer Community adopts a similar approach, using a language called “Plankton” [13]. We adopt LicenseScript’s idea of associating permissions with programmes, and show how these programmes can be automatically derived from conventional XML-based rights expressions.

2.6 Limitations

The foregoing rights expression languages, with the exception of LicenseScript and contract schema, are primarily designed to make statements about the properties of the languages themselves rather than to implement an actual interpreter. They seem adequate to meet the particular needs of their creators in this respect, but they do not seem adequate to express all of the rights expressions envisaged by the creators of non-formal languages like ODRL and XrML. Nor do they capture all of the tasks faced by the implementers of real rights expression interpreters. We will discuss the missing pieces presently.

Firstly, existing formal rights expression languages treat all permissions as if they are executed instantaneously. This assumption, however, is inadequate to capture the semantics of various constraints that measure the amount of time taken to complete an action, such as the accumulated constraint of ODRL and equivalent ValidityTimeMetered condition of XrML. Furthermore, it fails to account for constraints whose satisfaction may change over time, such as time-based and location-based constraints.

Secondly, actions are treated as if they are infallible, which may lead to incorrect treatment of state variables in a real interpreter. In LicenseScript, for example, a programme may reduce the value of a counter in the act of testing permission to perform an action. If the action fails to proceed due to an I/O error or the like, however, the user pays the cost of advancing that counter, without exercising the permission.
Thirdly, both ODRL and XrML contain constraints that modify the form of an action rather than permit or prohibit it outright. The quality and watermark constraints of ODRL, for example, modify an action by altering the resolution of the output and inserting a watermark, respectively. Existing formal rights expression languages, however, only allow for rights expressions that return a binary truth value. While it may be possible to express such constraints in a binary form as “the action may proceed if the resource is rendered at bitrate b,” and so on, this supposes that the user has selected bitrate b before consulting the licence. We think users and programmers would find it more straightforward to interpret these constraints as “the action may proceed at bitrate b,” and so on. We will also see that non-binary rights expressions are useful for expressing constraints whose values change over time.

Finally, the foregoing rights expression languages often transform constraints and obligations into opaque predicates or functions such as \( \text{count}() \), \( \text{paid}() \), and so on. These predicates and functions may themselves represent quite complex operations, and in Section 3 we will look more carefully at what kinds of functions can or should be treated as “primitives” of our rights expression interpreter.

3. OPERATIONAL RIGHTS EXPRESSIONS

The creators of LicenseScript suggest that their language could be used as a “compiled” form of the XML-based languages. In this view, ODRL or XrML is viewed as a “high-level” language in which it is relatively easy for humans to express their intent, while LicenseScript is viewed as a “low-level” language that can be efficiently executed by machine. Such a compiler would associate the elements of an XML-based language with meanings in terms of their execution on some hypothetical computing device. This approach is known as operational semantics in the theory of programming languages.

Of course, any real ODRL or XrML interpreter must ultimately transform an ODRL or XrML expression into code to be executed on an actual machine, and thus must perform the functions of such a compiler in some sense. The creators of LicenseScript point out that this transformation, when carried out by several independent developers, allows each developer to introduce his or her own particular semantics to the original language. Defining operational semantics for the high-level language eliminates this possibility by standardising the operation of the compiler.

So far as we are aware, however, no compiler of this kind has ever been constructed, and, as we discussed earlier, neither LicenseScript nor any other existing language appears to be able to express all of the concepts envisaged by the creators of ODRL and XrML. We will describe how such a compiler might be developed in this section, and use our model to derive operational semantics for a particular XML-based language in Section 4.

Our compiler can also be interpreted in terms of the layered model of digital rights management systems proposed by Jamkhedkar and Heileman [11]. They propose that the functions of a digital rights management system be divided into a series of layers, such that the interface to each layer can be standardised and the implementation of each layer made opaque to the other layers.

Our compiler divides Jamkhedkar and Heileman’s “rights expression and interpretation layer” into two layers: a rights expression layer in which rights are described using some XML-based rights expression language, and a rights interpretation layer in which rights are described by their implementation on a virtual machine. The primitive functions of the virtual machine represent (part of) the interface to Jamkhedkar and Heileman’s “digital rights enforcement layer”, which is responsible for providing the security primitives required to enforce a licence.

In this section, we first describe a model for rights expressions in which an expression consists of a set of executable functions that test for permission to perform an action, and carry out any obligations incurred by performing the action. We then describe an example C-like virtual machine on which such functions might be executed, and discuss the properties of primitive functions provided by this machine. Finally, we outline the architecture of a hypothetical digital rights management system that employs this model.

3.1 Rights Expressions

A variety of models have been proposed for rights expression languages, as we discussed in Section 2 above. They vary in their organisation and terminology, but in all cases we can say that the task of a rights expression interpreter is to:

- determine whether or not permission is granted to perform an action in a given context; and
- ensure that any obligations incurred by exercising the permission are carried out.

We will not consider the semantics of the actions themselves in this paper. We assume that the authors of licences and the implementers of renderers have agreed upon some meaning for actions such as “play”, “print”, etc., but that this meaning is opaque to the rights expression interpreter.

From an operational perspective, then, we can view a rights expression as being a collection of potential actions on an object, each associated with

- a test function that determines whether or not the action is permitted on that object; and
- a duty function that ensures that any obligations incurred by exercising the permission are carried out.

The test function must obviously be executed prior to exercising the permission, while the duty function must be executed after exercising the permission, since any obligations incurred may depend on the outcome of the exercise. Note that, throughout this paper, we use the term function in the sense that it is used in computer programming rather than in its formal mathematical sense.

As the rights expressions contemplated by ODRL and XrML are stateful, our rights expressions must also be associated with some state information that can be checked and altered by test and duty functions. We will represent the state of a rights expressions as an array \( \sigma \) of variables \( \sigma[s] \), where \( s \) is a variable name (an arbitrary string). Every variable may take on a range of values defined by the rights expression language under consideration.

As we discussed in Section 2, it is not sufficient for the test function to return a binary truth value that either grants or denies the permission. In general, the test function may return some permission modifier that alters the nature of the permission in some way.
The set of modifiers available in a particular rights expression language may vary from language to language depending on what kinds of renderers are contemplated by that language. As for actions, the semantics of modifiers are defined by the renderer, and are opaque to the interpreter (though the interpreter may pass some parameters to a parameterised modifier without concerning itself with what those parameters mean to the renderer).

We can, however, define two fundamental modifiers that correspond to the conventional binary truth values:

- the empty modifier \(\epsilon\) that permits the action to proceed in its unmodified form; and
- the failure modifier \(\bot\) that denies permission for the action to proceed at all.

We will consider non-binary modifiers when we consider a particular rights expression language in Section 4.

We will also represent the outcome of an action as a modifier. An action that proceeds successfully will be represented by the empty modifier, while an action that fails to proceed at all will be represented by the failure modifier. An action that proceeds in a modified form will be represented by the appropriate modifier drawn from the set of modifiers used in the language under consideration.

Formally, then, we can view an operational rights expression \(e\) as a tuple \((\sigma, p, t, d)\) where

- \(\sigma\) is the state of the rights expression, being an array of variable names with values as described above;
- \(p\) is a pair \((a, r)\) of an action \(a\) and a resource \(r\) such that the rights expression (potentially) permits \(a\) to be carried out on \(r\);
- \(t\) is the test function of the rights expression; and
- \(d\) is the duty function of the rights expression.

Both test and duty functions may read and modify the state \(\sigma\). The test function returns a permission modifier that describes the form in which the action may proceed. The duty function then accepts a modifier describing the actual outcome of the action, and may alter its behaviour accordingly. Both functions are expressed in the language of a virtual machine such as the one defined in the next section.

### 3.2 Virtual Machine

There are a variety of ways in which we could specify the virtual machine on which the test function and duty function must be executed. For the purposes of this paper, we will define a C-like machine that supports

- variables and variable assignment;
- an if-then-else statement;
- function calls and a return statement;
- compound statements formed by concatenating other statements;
- the integers, and the usual arithmetic and comparative operators;
- the boolean values true and false, and the usual boolean operators;
- character strings; and
- a special value null to which all variables are initialised.

All statements will be terminated by a semi-colon, as in C-like languages.

Programmes may use local variables, denoted by single letters, that only exist for the execution of the function that encloses them. The array \(\sigma\), however, is part of the rights expression and exists throughout the life of the rights expression. Both kinds of variables are initialised to the special value null. Variable assignment will be denoted by a left arrow, so that \(\sigma[X] \leftarrow 3\) denotes assignment of the value 3 to the state variable \(X\), for example.

Conditional statements will be denoted using the usual if-then-else form. Thus, “if \(b\) then \(s_1\); else \(s_2\)” means execute the statement \(s_1\) if the boolean \(b\) is true, and \(s_2\) otherwise. For brevity, we will omit the else branch if it does not contain any statements.

Function calls will be denoted by a bold-face identifier, followed by their arguments in parentheses, as in the usual mathematical notation. The statement “return \(v\)” causes the function to terminate and return the value \(v\), as in C-like languages.

Compound statements will be denoted by braces, as in C-like languages. Thus, “\{ \(s_1\); \(s_2\); \}” denotes execution of statement \(s_1\) followed by execution of statement \(s_2\).

Integers will be expressed in decimal notation, and our machine will support the usual integer operations of addition (+) and subtraction (−). Our machine will also support the usual comparison operators less-than (<), greater-than (>) and equal-to (=). (The machine could also support multiplication, division and real numbers, but there is no need for these in any of the rights expressions considered in this paper).

The boolean values will be denoted true and false, and our machine will support the usual boolean operators of conjunction (\&\&), disjunction (\|\|) and negation (\neg).

### 3.3 Primitive Functions

Practical rights expressions frequently refer to environment information that is not contained within the rights expression itself, such as the current time, the identity of the user, and so on. The formal languages discussed in Section 2 typically represent this information as opaque predicates and functions.

In order to perform operations on environmental information, our virtual machine will require access to a set of primitive functions whose semantics and operation are not described in the language of the virtual machine itself. Primitive functions can be seen as part of the interface between the rights interpretation layer and the digital rights enforcement layer in Jankhedkar and Heileman’s model.

The particular set of primitive functions necessary to implement a particular rights expression language may vary from language to language. Languages that support time-based constraints, for example, might require the digital rights enforcement layer to provide a time() primitive function, while languages that support location-based constraints might require the digital rights enforcement layer to provide a location() function, and so on.

We can, however, make several fairly straightforward observations here that apply to all languages that are intended for implementation on a real computer. Similar observations
4. OPERATIONAL SEMANTICS OF OMA REL

The rights expression language used in the Open Mobile Alliance’s digital rights management specification [15] employs a very restricted subset of ODRL, together with a few additional elements specially devised for mobile phones. This section is based on OMA REL Version 2.1, which is itself based on ODRL Version 1.1.

Even though OMA REL has a very small vocabulary compared to ODRL and XrML, it contains nearly all of the essential features of those languages. The main feature missing from OMA REL is delegation, which is not used in any real digital rights management systems of which we are aware. In particular, OMA REL serves to illustrate all of the limitations of the formalisations that we described in Section 2, and our solution to them.

An OMA REL rights object (licence) is an XML document rooted in an element called rights with the structure shown diagrammatically in Figure 2. The rights element contains a context element that provides some version information and a unique identifier for the rights object, together with an agreement element that describes the rights granted by the rights object.

The agreement element contains the rights expression itself, consisting of

- one or more asset elements that identify the resource(s) to which the rights object refers; and
- zero or more permission elements that describe the permissions that have been granted over those assets.

The permission element contains one or more elements that identify a particular action (play, display, execute, print and/or export). The permission element may also contain a constraint element and/or a requirement element\(^1\) that applies to all of the actions. Each individual action may also be associated with its own particular constraint or requirement. Finally, each action may be linked to particular assets of the agreement by describing them in an asset child. (Otherwise the permission applies to all of the assets in the agreement.)

\(^1\)The specification sometimes has requirements (plural) for this element, but we will use the singular, as in ODRL. The specification also omits the requirement element from the list of possible children of a permission element, but it is clear from the description of the requirement element that it should be listed there, as it is in ODRL.
An asset element may also inherit permissions from another rights object by including an inherit element as one of its children. The inherit element contains the identifier of an asset in the parent rights object, and indicates that all of the permissions granted over the asset of the parent object are also granted over the asset of the child object.

We are not interested in the semantics of particular actions here, since these are properties of the renderer and not the rights interpretation layer. The main work for this section, however, is to associate operational semantics with all of the constraints and requirements.

Table 1 lists all of the constraints defined by OMA REL together with an informal description of their meaning. Section 4.4 will describe their semantics in more detail.

OMA REL supports only one requirement, called tracked. It requires the device to count the number of times that the permission is exercised, and to meter the amount of time for which the permission is exercised.

The technical definition of some constraints and the tracked requirement is made complicated by the fact that some devices may not support timers or meters. A permission associated with a constraint or requirement that requires an operation not supported by a device may be prohibited. This could be incorporated into our semantics by including primitive functions that test the capabilities of a particular device before attempting to check the constraint or requirement proper. For ease of exposition, however, we will suppose that our devices have all of the capabilities imagined by OMA REL.

4.1 Primitive Functions

It is straightforward to extract the set of primitive functions shown in Table 2 by inspecting the OMA REL specification. The “tests” return true if the input string s matches the corresponding identifier, while the time() function returns the current time as an integer (the number of seconds since the beginning of an epoch, for example). Of course these functions could be defined in a variety of other ways; the identifier tests, for example, could be replaced by functions that returned the appropriate identifier.

The designers of OMA REL presumably believe that all of the functions listed in Table 2 are easy to implement on...
the kinds of devices for which the OMA DRM system is intended. Furthermore, they can presumably be implemented in a secure fashion, so that it is not feasible, for example, for a malicious user to fake the identifier of his or her device. The specifics of their implementation, however, are beyond the scope of the present discussion.

4.2 Modifiers

OMA REL does not include any of ODRL’s modifying constraints, such as quality and watermark. However, we can capture the semantics of time-based constraints using a parameterised modifier \( \tau(t) \) that modifies an action by limiting the amount of time for which it continues. This modifier allows us to express both constraints whose truth values change over time, and actions that are not exercised instantaneously, as follows:

- if a test function returns \( \tau(t) \), the action is permitted to continue for no longer than the time period \( t \); and
- if an action is carried out for a time period \( t \), the modifier \( \tau(t) \) will be passed to the duty function.

Given that the devices contemplated by OMA already contain a timer, this modifier seems easy to implement.

Of course we will also need to use the fundamental modifiers introduced in Section 3. For convenience, however, we will write \( \tau(e) \) for the empty modifier and \( \tau(\bot) \) for the failure modifier. That is, \( e \) represents a time longer than any finite time, and \( \bot \) represents a time shorter than any finite time. This will make many of our duty functions significantly more compact than they might be otherwise.

4.3 Compiler

Our compiler transforms an OMA REL agreement into a series of programmes for the virtual machine described in Section 3 by associating every element of the agreement with a function, then linking these functions according to the structure of the licence. The compiler proceeds in four steps that will be described presently: element identification, canonicalisation, compilation and linking.

4.3.1 Element Identification

The first step is to associate every permission, asset, constraint and requirement in the rights object with a unique identifier. This can be done (for example) by concatenating the rights object’s own unique identifier with an XPath expression [20] that describes the element’s location in the rights object document. Thus the first asset child of the agreement element of a rights object might be identified as ro1:/rights/agreement/asset[1], for example, if the rights object’s identifier is ro1.

Note that the same element may be referred to in several places within the same document using the id and idref attributes. These links must be resolved during identification. An asset element with an idref attribute that occurs as the child of a permission element, for example, should be referred to by the XPath expression of the asset element with the specified id, and not its own XPath expression.

4.3.2 Canonicalisation

The second step is to identify a series of pairs \((p, C)\) where

- \( p \) is a permission \((a, r)\) consisting of an action \( a \) and a resource \( r \); and
- \( C \) is the set of constraints and requirements that apply to the permission \( p \).

Note that our compiler considers “constraints” and “requirements” to have the same type; they both imply test functions and duty functions without any distinction as to whether OMA REL calls them “constraints” or “requirements”.

These pairs can be identified by enumerating the children of the permission element of the agreement. For every child element that describes an action, the canonicaliser must enumerate every asset to which that element refers, using the rules described in the OMA REL specification. Finally, the canonicaliser must similarly enumerate all of the constraints and requirements for every \((a, r)\) pair thus generated.

Canonicalisation must also resolve any inherit elements in the rights object. If an asset contains an inherit element, the canonicaliser must retrieve the permissions to be inherited from the parent rights object, and canonicalise them as for permissions that exist in the child rights object. Alternatively, if the parent rights object has already been compiled, the compiled permissions can be simply copied during the linking stage described below.

Note that a constraint or requirement child of a permission element applies to all of the permissions that are children of the same element, and many child rights objects may inherit from the same parent rights object. Thus the same constraint or requirement may appear in more than one set \( C \).

4.3.3 Compilation

The third step is to transform each tuple \((p, C)\) into a pair of a test function and a duty function, such that these functions apply all of the tests and duties, respectively, implied by \( C \).

The test function will return the most restrictive modifier returned by any of the test functions in \( C \), that is, it will have the form

\[
p_{\text{test}}() \{ 
\begin{align*}
0 & \leftarrow \text{length}(c_1^p(id_1)); \\
1 & \leftarrow \text{length}(c_2^p(id_2)); \\
& \quad \text{if } t_1 < t_0 \text{ then } t_0 \leftarrow t_1; \\
1 & \leftarrow \text{length}(c_3^p(id_3)); \\
& \quad \text{if } t_1 < t_0 \text{ then } t_0 \leftarrow t_1; \\
\vdots \\
& \quad \text{return } \tau(t_0); 
\end{align*}
\}
\]

where the \( c_i^p \) is the test function of \( c_i \in C \), \( id_i \) is the unique identifier of \( c_i \in C \), and a \( \text{length}(\tau(t)) \) is a function that returns \( t \). Though we have shown them here with only their identifiers as arguments, the \( c_i^p \) functions may take arguments according to their definitions given in Section 4.4 below.

The duty function simply executes all of the duty functions in \( C \), that is, it will have the form

\[
p_{\text{duty}}(\tau(t)) \{ 
\begin{align*}
c_1^p(id_1, \tau(t)); & \ldots; c_n^p(id_n, \tau(t)); 
\end{align*}
\}
\]

where \( c_i^p \) is the duty function of \( c_i \in C \). Again, each \( c_i^p \)
4.3.4 Linking

Finally, the compiled test and duty functions must be packaged in some format understood by the target virtual machine. The compiled functions must be “linked” with the necessary standard test and duty functions described in Section 4.4 below, either by including the latter functions within the package itself, or by employing some shared library. We will not describe a particular format for virtual machine programmes here, but we can imagine some format similar to those used for shared libraries in an ordinary operating system.

4.4 Semantics

Every test and duty function takes the unique identifier of the element to which it corresponds as its first argument. This identifier is used to form variable names within the functions, so that every unique instance of an element is associated own set of state variables in $\sigma$, prefixed by the element’s identifier. Thus, if a rights object contains two count constraints identified as $c_1$ and $c_2$, for example, there will exist two distinct counters $\sigma[c_1, \text{counter}]$ and $\sigma[c_2, \text{counter}]$. Both counters, however, will be manipulated by the same piece of code.

In addition to the element identifier, every test and duty function will take a set of arguments that represent the parameters given in the rights object. The test function for the interval constraint, for example, will accept the length of the interval as its second argument. The duty function also accepts the outcome of the action as its last argument.

We use the fact that all variables are initialised to the special value null to initialise state variables with values taken from the rights object. The first time that the test function for the count constraint is called, for example, the counter will evaluate to null. The function can therefore test for this, and initialise the counter with the value from the rights object (which it receives as one of its arguments).

4.4.1 Count

The count constraint simply imposes a limit on the number of times that the associated permission may be exercised. It is easy to see that its semantics consist of a test for the value of the counter before the permission can be exercised, and a duty to update the counter after the permission has been (successfully) exercised. The formal test and duty are as follows, where the $id.counter$ variable is used to hold the state of the counter:

```plaintext
count_test(id, count) {
  if $\sigma[id.counter] = \text{null}$ then $\sigma[id.counter] \leftarrow \text{count}$;
  if $\sigma[id.counter] > 0$ then return $\tau(\epsilon)$; else return $\tau(\bot)$;
}
```

```plaintext
count_duty(id, count, $\tau(t)$) {
  if $t \neq \bot$ then $\sigma[id.counter] \leftarrow \sigma[id.counter] - 1$;
}
```

4.4.2 Timed-count

The timed-count constraint is similar to the count constraint, except that the value of the counter is not changed if the permission was not exercised for longer than a period specified by the timer attribute. In the following semantics, we have supposed that a permission may be exercised for up to timer seconds even if the $id.counter$ variable has already been reduced to zero. This may or may not be the intent of the designers of OMA REL, but it serves to illustrate the use of the $\tau$ modifier to express a constraint whose satisfaction changes over time:

```plaintext
timed_count_test(id, count, timer) {
  if $\sigma[id.counter] = \text{null}$ then $\sigma[id.counter] \leftarrow \text{count}$;
  if $\sigma[id.counter] > 0$ then return $\tau(\epsilon)$; else return $\tau(\text{timer})$;
}
```

```plaintext
timed_count_duty(id, count, timer, $\tau(t)$) {
  if $t > \text{timer}$ then $\sigma[id.counter] \leftarrow \sigma[id.counter] - 1$;
}
```

4.4.3 Datetime

The datetime constraint simply restricts exercise of the associated permission to a time interval specified by the begin and end attributes. Since there is no state associated with the constraint, its semantics consist only of a test as follows:

```plaintext
datetime_test(id, begin, end) {
  $t \leftarrow \text{time}()$;
  if $t \geq \text{begin} \land t \leq \text{end}$ then return $\tau(\text{end} - t)$;
  else return $\tau(\bot)$;
}
```

4.4.4 Interval

The interval constraint requires that the associated permission only be exercised for a given period (the length attribute) after it was first exercised. Like the datetime constraint, it consists only of a test, as follows:

```plaintext
interval_test(id, length) {
  $t \leftarrow \text{time}()$;
  if $\sigma[id.start] = \text{null}$ then $\sigma[id.start] \leftarrow t$;
  if $t \leq \sigma[id.start] + \text{length}$ then return $\tau(\sigma[id.start] + \text{length} - t)$;
  else return $\tau(\bot)$;
}
```

Note that the foregoing semantics start the interval at the first attempt to exercise the permission, even if the action fails to start. If the interval is to start only after the first successful attempt to exercise the permission, the semantics must include a duty function to test for the success or otherwise of the attempt before setting the id.start variable.

4.4.5 Accumulated

The accumulated constraint limits the total amount of time for which a permission may be exercised. Its semantics are therefore similar to those of the count constraint, except that the state is updated with the amount of time for which the permission was exercised, instead of by simply adding 1 every time:

```plaintext
accumulated_test(id, length) {
  if $\sigma[id.meter] = \text{null}$ then $\sigma[id.meter] \leftarrow \text{length}$;
  if $\sigma[id.meter] > 0$ then return $\tau(\text{meter})$; else return $\tau(\bot)$;
}
```
accumulated_duty(id, length, τ(t)) {  
  if t ≤ τ then σ[id.meter] ← σ[id.meter] − τ;  
}

4.4.6 Individual and System

The individual constraint states that the permission may only be used on a device with a particular SIM card (identified by its International Mobile Subscriber Identifier, “IMSI”) or wireless identity module (“WIM”, identified by its public key). Depending on the kind of identifier present in a particular rights object, the compiler should compile the constraint to a test using either the imsi() or wim() function as follows (for the IMSI case):

individual_test(id, sim) {  
  if imsi(sim) then return τ(τ); else return τ(⊥);  
}

The system constraint similarly states that the permission may only be exercised on a device with a particular identifier defined by the Open Mobile Naming Authority (“OMNA”). Its semantics are therefore identical to those of the individual constraint, except that the omna() function is used instead of the imsi() and wim() functions.

4.4.7 Tracked

The tracked requirement states that the device must record the number of times that the associated permission is exercised, along with the total amount of time for which the permission has been exercised. Its semantics therefore consist of duties similar to those of the count and accumulated constraints, as follows:

tracked_duty(id, τ(t)) {  
  if σ[id.count] = null then σ[id.count] ← 0;  
  if σ[id.meter] = null then σ[id.meter] ← 0;  
  if t ≤ τ then {  
    σ[id.count] ← σ[id.count] + 1;  
    σ[id.meter] ← σ[id.meter] + t;  
  }  
}

OMA DRM requires that the information recorded by the tracked constraint (that is, the values of the id.count and id.meter variables above) be transmitted on request to the rights issuer from which the rights object was obtained. How this is done, however, is beyond the scope of the present discussion.

4.5 Example

Figure 3 shows an OMA rights object that grants the display and print rights over an image identified as Image:1. The image may be displayed for up to seven days after it is first viewed (according to the interval constraint), and printed up to ten times (according to the count constraint). While such a licence may not necessarily represent a realistic business model, it serves to illustrate the working of our compiler.

Following the procedure described in Section 4.3, the compiler first associates every element of the rights object with its XPath expression. The rights object contains two permissions:

- the (display, Image:1) permission, which is subject to the rol:/rights/agreement/permission/display/interval constraint; and
- the (print, Image:1) permission, which is subject to the rol:/rights/agreement/permission/print/count constraint.

The compiler then outputs the three functions shown in Figure 4 (note that parts of the XPath expressions have been omitted so that they fit on a single line of text). The three functions are a display test that must be called to test for permission to display the image; a print test that must be called to test for permission to print the image; and a print duty that must be executed upon printing the image.

All of the functions called by Figure 4 are identical to those described in Section 4.4 above, and, for brevity, are not repeated here. In a real system, these functions must be linked to the output programme as described in Section 4.3.

5. DISCUSSION

5.1 Constraints vs. Obligations

One of the most striking features of the operational semantics of constraints such as count is that there is no particular distinction between what conventional policy models
call “constraints” and “obligations”. Stateful “constraints” can viewed as much as an obligation to maintain a state value as they can be viewed as a constraint on that value.

Policy authors may nonetheless find it more intuitive and convenient to express a count constraint (say) as such, rather than as a pair of a test and a duty. Some models, notably XrML, do capture this idea by representing both tests and duties by a single “condition” element that may contain elements of both. Using a compiler allows policy authors to express their intention in the manner proposed by XrML, with the precise algorithm for carrying out this intention being supplied by the compiler.

5.2 Interoperability

Some authors have proposed that abstract rights models be used to promote interoperability by translating expressions in one language to expressions in another language [12, 18, 19], using the abstract model as an intermediate representation. It seems unlikely that the semantics proposed in this paper could be used in such a fashion, since it would require the output of the compiler on the first language to be de-compiled into the second language, and we cannot see any easy way of doing this.

The semantics proposed in this paper, however, allow for interoperability at the rights interpretation level: rights expressions may be written in any rights expression language so long as a suitable compiler exists for that language. This is more like the early approach of Guth, et al. [8] – who also use their model as the basis of a real interpreter – than the later approaches noted above.

5.3 Multiple Interpretations

Of course there are several ways in which the same constraint or obligation might be implemented in a virtual machine programme. In some cases, differing implementations may be of little consequence: implementations of the count constraint may count either up or down, for example, with identical effect. In principle, it is possible to prove that two such programmes – provided by different compiler vendors, for example – meet a common logical specification by using axiomatic semantics or other techniques for verifying the correctness of computer programmes.

On the other hand, some rights expressions languages – notably ODRL and the Creative Commons digital code – contain elements that could be implemented in wildly different manners. ODRL’s purpose constraint, for example, could be implemented by checking that the user is a member of a role permitted to access data for that purpose [3], or by checking that the user is engaged in a task associated with that purpose [6], or by other methods yet to be invented. All of these methods may be valid and acceptable methods of checking for the purpose of an action, but they are not formally equivalent.

In our model, the precise semantics of a rights expression could be determined either by the designer of the rights expression language, or by the designer of the compiler. In the former view, every element has exactly one implementation, and all other implementations are “incorrect”. In the latter view, there may exist several differing implementations of the same element that achieve the same broad end while differing in detail.

While previous authors have generally regarded the existence of several differing interpretations of an element as being a negative – and we do not doubt that there is potential for disputes to arise in the presence of multiple interpretations – there may also be cases in which differing technical interpretations of an element are actually useful. In the case of the purpose constraint, for example, the compiler may choose to use a role-based or task-based implementation depending on whether the target system supports roles or tasks, assuming that either implementation is satisfactory to the rights owner.

Alternatively, we could incorporate the choice of implementation into the semantics of the element itself, as follows:

```
purposeTest(id, purpose) { 
  if use_roles() then 
    return role_purposeTest(id, purpose); 
  else 
    return task_purposeTest(id, purpose); 
}
```

In this view, the semantics of the element are identical in theory, though we might say that they differ in practice.

5.4 Other Semantics

It may be possible to adapt some of the concepts proposed in this paper to other kinds of semantics. It seems possible to incorporate permission modifiers into trace-based semantics, for example, by parameterising the events that correspond to rendering actions. We leave such adaptation as future work.

6. CONCLUSION

The operational view of rights expression languages provides both a practical approach to the enforcement of such languages, and new insights into the properties of such languages and their interpreters. Our semantics, in particular, provide for

- actions whose effects are neither instantaneous nor infallible;
- constraints whose satisfaction changes over time; and
- constraints that modify the form of an action rather than permit or prohibit it outright.

Our semantics seem adequate to capture the OMA Rights Expression Language, which cannot be completely captured by previous systems.

We have not, however, explored the operational semantics of larger languages. In particular, we have not attempted to address delegation, and we have not attempted to associate semantics with very high-level constraints such as ODRL’s purpose constraint.

Our proposal also takes practical steps towards the definition and construction of a layered digital rights management system. We have identified the role of a compiler in transforming human-readable rights expressions into machine-enforceable interpretations, and identified the role of “primitive functions” in communicating between the rights interpretation later and rights enforcement layer.

7. ACKNOWLEDGEMENTS

The authors would like to thank Robin Crockett for guidance on the theory of programming languages.
8. REFERENCES


