A Formal Foundation for XrML*

Joseph Y. Halpern
Cornell University
Ithaca, NY 14853
halpern@cs.cornell.edu

Vicky Weissman
Cornell University
Ithaca, NY 14853
vickyw@cs.cornell.edu

Abstract

XrML is becoming a popular language in industry for writing software licenses. The semantics for XrML is implicitly given by an algorithm that determines if a permission follows from a set of licenses. We focus on a representative fragment of the language and use it to highlight some problematic aspects of the algorithm. We then correct the problems, introduce formal semantics, and show that our semantics matches the (corrected) algorithm. Finally, we consider the complexity of determining if a permission is implied by a set of XrML licenses. We show that the general problem is NP-hard, but it is polynomial-time computable for an expressive fragment of the language.

1. Introduction

The eXtensible rights Markup Language (XrML) is becoming an increasingly popular language in which to write software licenses. When first released in 2000, XrML received the support of many technology providers, content owners, distributors, and retailers, including Adobe Systems, Hewlett-Packard Laboratories, Microsoft, Xerox Corp., Barnesandnoble.com, and Time Warner Trade Publishing. Companies including Microsoft, OverDrive, and DMDsecure have publicly announced their agreement to build products and/or services that are XrML compliant. Currently, XrML is being used by international standard committees as the basis for application-specific languages that are designed for use across entire industries. For example, the Moving Picture Experts Group (MPEG) has selected XrML as the foundation for their MPEG-21 Rights Expression Language (MPEG-21 REL). (See http://www.xrml.org for more information.) It is clear that a number of industries are each moving towards a standard language for writing licenses, and that many of these standard languages are likely to be based on XrML. To understand the new standards, we need to understand XrML.

XrML does not have formal semantics. Instead, the XrML specification [3] presents the semantics in two ways. First is an English description of the language. Second is an English description of an algorithm that determines if a permission follows from a set of licenses. Unfortunately, the two versions of the semantics do not agree. To make matters worse, the algorithm has unintuitive consequences that do not seem to reflect the language writers' intent.

As a first step towards addressing these issues, we provide a formal semantics for XrML. To the best of our knowledge, we are the first to do this. We give the language formal semantics by providing a translation from XrML licenses to formulas in modal first-order logic. We verify the translation by proving that the algorithm included in the XrML document, slightly modified to correct the unintuitive behavior, matches our semantics. More precisely, the algorithm says that a permission follows from a set of licenses iff the translated permission is a logical consequence of the translated licenses. We then consider the complexity of determining if a permission is implied by a set of licenses. We show that the general problem is NP-hard, but, for an expressive fragment of the language, it is polynomial-time computable.

The rest of the paper is organized as follows. In the next section we present a fragment of XrML that suffices to illustrate the key issues. In Section 3 we review XrML's algorithm for determining what is permitted given a set of licenses. After considering some examples in which the algorithm's behavior is unintuitive and almost certainly unintended, we propose a revised algorithm that we believe captures the designers' intent. Formal semantics for XrML are given in Section 4, and the revised algorithm is shown to be sound and complete with respect

* Authors supported in part by NSF under grant CTC-0208535, by ONR under grants N00014-00-1-03-41 and N00014-01-10-511, by the DoD Multidisciplinary University Research Initiative (MURI) program administered by the ONR under grant N00014-01-1-0795, and by AFOSR under grant F49620-02-1-0101.

to the semantics. In Section 5 we show that the problem of determining if a permission follows from a set of licenses is NP-hard. We also discuss a fragment of the language that is both tractable and relatively expressive. In Section 6 we outline how our results can be modified to apply to the entire language, including extensions that are within the XrML framework. We conclude in Section 7. Proofs for all of the theorems can be found at www.cs.cornell.edu/home/halpern/papers/xrml.pdf.

2. Syntax

Before we can begin our analysis of XrML, we need to choose a version of the language. ContentGuard froze the language in November 2001 [1] with the intention that standard committees would extend the language to suit the needs of their particular constituents. However, the standard committees are currently modifying the language, as opposed to simply extending it. In particular, the MPEG committee published their version of the language [4] a few weeks before the final submission of this paper. Although we have not studied that release carefully, we have examined their March 2003 Specification [3], which is the version before the current one. In this paper we discuss the March 2003 Specification; in the full paper we discuss the current version.

In this section we present a syntax for a representative fragment of XrML. (The rest of the language is discussed in Section 6.) Although XrML is an XML-based language, our syntax does not follow the XML conventions. Instead, we have chosen a syntax that we believe is more intuitive. There are other differences between the two approaches; these are discussed at the end of the section.

At the heart of XrML is the notion of a *license*. A license is a (grant; principal) pair, where the license (g, p) means p issues (i.e., says) g. For example, the license (Bob is smart, Alice) means "Alice says 'Bob is smart' ".

A grant has the form '$\forall x_1, \ldots, \forall x_n$(condition $\rightarrow$ conclusion)', which intuitively means that the condition implies the conclusion under all appropriate substitutions. Conditions and conclusions are defined as follows.

- A condition has the form $\mathbf{Said}(p_1, e_1) \wedge \ldots \wedge \mathbf{Said}(p_n, e_n)$, where p_i is a principal and e_i is a conclusion. We represent the empty conjunction (when $n = 0$) by $\mathbf{true}$. Roughly speaking, $\mathbf{Said}(p_i, e_i)$ holds if the grants issued by the principal p_i, along with the set of licenses, imply e_i.
- A conclusion has either the form $\mathbf{Perm}(p, r, s)$ or the form $Pr(p)$ where Pr is a property, p is a principal, r is a right (i.e., an action), and s is a resource. The conclusion $\mathbf{Perm}(p, r, s)$ means p may exercise r over s. For example, $\mathbf{Perm}(Bob, edit, budget\ report)$ means Bob may edit the budget report. The conclusion $Pr(p)$ means p has the property Pr. For example, the conclusion $\mathbf{Attractive}(Bob)$ means Bob is attractive.

We abbreviate the grant $\forall x_1, \ldots, \forall x_n(\mathbf{true} \rightarrow e)$ as $\forall x_1, \ldots, \forall x_n e$. Also, we try to consistently use d, possibly subscripted, to denote generic conditions and e, possibly subscripted, to denote generic conclusions.

Consider the following example. Suppose that Alice issues the grant 'Bob is smart' and Amy issues the grant 'if Alice says that Bob is smart, then he is attractive'. We can write the first license in our syntax as $(g_1, Alice)$, where $g_1 = \mathbf{Smart}(Bob)$ (recall that this is an abbreviation for $\mathbf{true} \rightarrow \mathbf{Smart}(Bob)$), and we can write the second as (g_2, Amy), where $g_2 = \mathbf{Said}(Alice, \mathbf{Smart}(Bob)) \rightarrow \mathbf{Attractive}(Bob)$. Because $(g_1, Alice)$ means 'Alice says Bob is smart' and g_2 means 'if Alice says Bob is smart, then he is attractive', the licenses together with the grants issued by Amy imply that Bob is attractive. Thus, the condition $\mathbf{Said}(Amy, \mathbf{Attractive}(Bob))$ holds.

The sets of principals, properties, rights, and resources depend on the particular application. For example, a multimedia application might have a principal for each employee and each customer; properties such as 'hearing impaired' and 'manager'; rights such as 'edit' and 'download'; and a resource for each object such as a movie. We assume the application gives us a finite set $primitivePrin$ of principals and a finite set $primitiveProp$ of properties. We then define the components in our language as follows.

- The set of principals is the result of closing $primitivePrin$ under union. (Here and elsewhere we identify a principal $p \in primitivePrin$ with the singleton $\{p\}$.) Thus, every principal has the form $\{p_1, \ldots, p_n\}$, where each p_i is a primitive principal. The interpretation of a principal $\{p_1, \ldots, p_n\}$ depends on context (i.e., depends on whether the principal appears as the first argument in a $\mathbf{Said}$ condition, as the first argument in a conclusion, or as the second argument in a license). We discuss this later in the paper (primarily in Section 5).
- The only properties considered by XrML are those in $primitiveProp$. Following XrML, we consider only properties that take a single principal as an argument. For example, $primitiveProp$ can include the property $\mathbf{Employee}$, where $\mathbf{Employee}(x)$ means principal x is an employee, but it cannot include the property $\mathbf{MotherOf}$, where $\mathbf{MotherOf}(x, y)$ means x is the mother of y, nor can it include the property $\mathbf{Vehicle}$, where $\mathbf{Vehicle}(x)$ means resource x is a vehicle (e.g., a motorcycle, car, or truck). All of the results of this paper would continue to hold if we extend the fragment to include properties that take multiple arguments of vari-

ous sorts (i.e., principals, rights, and resources). We remark that in XrML the set of properties is not closed under conjunction or negation. It is easy to show that closing the set under conjunction adds no expressive power to the language. Closing under negation does add expressive power; we return to this issue in Section 7.

- The only right is `issue` and the only resources are grants. Intuitively, if a principal p has the right to issue a grant g, and p does issue g, then g is a true statement.

We remark that, although the grants $\forall x Pr(x)$ and $\forall y Pr(y)$ have the same semantic interpretation (all principles have property Pr), XrML treats them as distinct grants syntactically. For example, according to the XrML algorithm, if Alice is permitted to issue the grant $\forall x Pr(x)$, then she is not necessarily permitted to issue $\forall y Pr(y)$.

We formally define the language's syntax according to the following grammar.

$$
\begin{array}{lcl}
\mathit{license} & ::= & (\mathit{grant}, \mathit{prin}) \\
\mathit{grant} & ::= & \forall \mathit{var} \ldots \forall \mathit{var}(\mathit{cond} \rightarrow \mathit{conc}) \\
\mathit{var} & ::= & \mathit{prinVar} \mid \mathit{rsrcVar} \\
\mathit{cond} & ::= & \mathbf{Said}(\mathit{prin}, \mathit{conc}) \mid \mathit{cond} \wedge \mathit{cond} \mid \mathbf{true} \\
\mathit{conc} & ::= & \mathit{prop}(\mathit{prin}) \mid \mathbf{Perm}(\mathit{prin}, \mathit{right}, \mathit{rsrc}) \\
\mathit{prop} & ::= & Pr \\
\mathit{prin} & ::= & \{p\} \mid \{\mathit{prinVar}\} \mid \mathit{prin} \cup \mathit{prin} \\
\mathit{right} & ::= & \texttt{issue} \\
\mathit{rsrc} & ::= & \mathit{grant} \mid \mathit{rsrcVar},
\end{array}
$$

where Pr is a generic element of $primitiveProp$, p is a generic element of $primitivePrin$, and $prinVar$ and $rsrcVar$ are variables ranging over primitive principles and resources, respectively. For the remainder of this paper we assume that the second argument in a license is a singleton. Because the XrML document treats the license $(g, \{p_1, \ldots, p_n\})$ as an abbreviation for the set of licenses $\{(g, p) \mid p \in \{p_1, \ldots, p_n\}\}$, it is easy to modify our discussion to support all of the licenses included in the grammar. We further restrict the language so that all grants are *closed*, that is, for each grant $\forall x_1 \ldots \forall x_n(d \rightarrow e)$, the set of free variables in $d \rightarrow e$ is contained in $\{x_1, \ldots, x_n\}$. We discuss some of the consequences of this restriction in Section 4, and remove it in the full paper.

As mentioned at the beginning of this section, the grammar presented here is not identical to that described in the XrML document. The differences are listed below.

- Instead of assuming that the application provides a set of primitive principals, XrML assumes that the application provides a set K of cryptographic keys; the set of primitive principals is $\{\texttt{KeyHolder}(k) \mid k \in K\}$. We could take $primitivePrin$ to be this set; however, our more general approach leads to a simpler discussion. Moreover, our results do not change if we restrict primitive principals to those of the form $\texttt{KeyHolder}(k)$.
- XrML does not have conditions of the form $\mathbf{Said}(p, e)$. To capture **Said** conditions, XrML uses `PrerequisiteRight` conditions of the form (G, e), where G is a set of grants and e is a conclusion. A condition $\mathbf{Said}(\{p_1, \ldots, p_n\}, e)$ in our syntax (where each p_i is a primitive principal or a variable of sort $Principal$) corresponds to the `PrerequisiteRight` condition $(\{g_1, \ldots, g_n\}, e)$ in XrML, where each g_i is $\forall x \mathbf{Perm}(p_i, \texttt{issue}, x)$ and x is a variable of sort $Resource$.
- XrML does not have conclusions of the form $Pr(p)$. To capture properties, XrML uses a right called `PossessProperty`, and considers the properties given by the application to be resources. The conclusion $Pr(p)$ in our grammar corresponds to the conclusion $\mathbf{Perm}(p, \texttt{PossessProperty}, Pr)$ in XrML. We have two types of conclusions because we believe the grammar should help distinguish the conceptually different notions of permissions and properties, rather than confounding them.
- Instead of writing $\texttt{allConds}(c_1, \ldots, c_n)$, $\texttt{allConds}()$, and $\texttt{AllPrincipals}(p_1, \ldots, p_n)$, we use the more standard notations $c_1 \wedge \ldots \wedge c_n$, **true**, and $p_1 \cup \ldots \cup p_n$, respectively.
- XrML abbreviates a set of licenses $\{(g_i, p_j) \mid i \leq n, j \leq m\}$ as the single license $(\{g_1, \ldots, g_n\}, \{p_1, \ldots, p_m\})$. For ease of exposition, we do not do this. Our discussion can be easily extended to take the abbreviation into account.

3. XrML's Authorization Algorithm

The XrML document includes a procedure that we call **Query** to determine if a conclusion follows from a set of licenses (and some additional input that is discussed below). In this section we present and analyze the parts of the algorithm that pertain to our fragment.

Before describing the algorithm, we note that some aspects of **Query** are inefficient. This is acknowledged in the XrML document, which explains that **Query** was designed with clarity as the primary goal; it is the responsibility of the language implementors to create efficient algorithms with the same input/output behavior as **Query**. (In Section 5, we show that it is highly unlikely that such an efficient algorithm exists.)

3.1. A Description of Query

The input to **Query** is a closed conclusion e (i.e., a conclusion with no free variables), a set L of licenses (g, p) such that p is variable-free, and a set R of grants; **Query** returns $\mathbf{true}$ if e is implied by L and R. To explain the intuition behind L and R, we first note that the procedure treats a predefined set of principals as trusted. If the grant g is issued by a trusted principal, then g is in R and it is assumed to be true. If the license (g, p) is in L, then g is issued by p (i.e., p says g) and p is not an implicitly trusted principal. To clarify the inferences that are drawn from R and L, suppose that the grant g is $\mathbf{QueenOfSiam}(Alice)$, which means Alice is Queen of Siam, and the grant g' is $\mathbf{Perm}(Alice, \mathtt{issue}, g)$, which means Alice may issue g. If $g \in R$, then we assume that Alice really is queen. If $(g, Alice)$ is in L, then Alice says that she is the queen, but we cannot conclude that she is royalty from this statement alone. If $(g, Alice)$ is in L and g' is in R, then we assume that Alice has the authority to declare herself queen, because $g' \in R$; we assume that she exercises that authority, because $(g, Alice) \in L$; and we conclude that Alice is queen, because this follows from the two assumptions.

Query begins by calling the **Auth** algorithm. **Auth** takes e, L, and R as input; it returns a set D of closed conditions (i.e., conditions with no free variables). Roughly speaking, a closed condition d is in D if d, L, and R together imply e. To determine if a condition in D holds, **Query** relies on the **Holds** algorithm. The input to **Holds** is a closed condition d and a set L of licenses; **Holds**(d, L) returns true only if the licenses in L imply d. (Notice that **Holds** does not take R as input. We examine the consequences of this omission in Section 3.2.) If **Holds**(d, L) returns $\mathbf{true}$ for some d in D, then **Query** returns $\mathbf{true}$, indicating that L and R imply e. **Query** is summarized in Figure 1.

Query(e, L, R):

let $D =$ **Auth**(e, L, R)
for each $d \in D$
  if Holds(d, L) returns $\mathbf{true}$
  then Return $\mathbf{true}$
Return $\mathbf{false}$

Figure 1. The Query Algorithm

We now discuss **Auth** and **Holds** in some detail. To define **Auth**, we first consider the case where $L = \emptyset$ and every grant in R has the form $d_g \rightarrow e_g$. In this case **Auth**$(e, \emptyset, R)$ returns the set D of closed conditions such that each condition, together with R, implies e. More precisely, a condition d is in D if there is a grant $g = d \rightarrow e_g$ in R such that e_g implies e. In determining whether or not the implication holds, **Auth** makes the *subset assumption*. The subset assumption says that any property or permission attributed to a principal p is attributed to every principal that includes p. In other words, if $p \subseteq p'$, then $Pr(p)$ implies $Pr(p')$ and $\mathbf{Perm}(p, r, s)$ implies $\mathbf{Perm}(p', r, s)$. Thus, **Auth**$(Pr(p), \emptyset, R)$ returns

$$\{d \mid d \rightarrow Pr(p_g) \in R \text{ and } p_g \subseteq p\}$$

and **Auth**$(\mathbf{Perm}(p, r, s), \emptyset, R)$ returns

$$\{d \mid d \rightarrow \mathbf{Perm}(p_g, r, s) \in R \text{ and } p_g \subseteq p\}.$$

Suppose that there is at least one grant $\forall x_1 \ldots \forall x_n (d_g \rightarrow e_g)$ in R such that $d_g \rightarrow e_g$ is open (i.e., has free variables). Then we reduce this case to the previous one by considering all the substitution instances of grants in R. Define a *closed substitution* to be a mapping from variables to closed expressions of the appropriate type. Given a closed substitution σ and an expression t, let $t\sigma$ be the expression that arises after all free variables x in t are replaced by $\sigma(x)$. Let $R_\Sigma = \{d_g\sigma \rightarrow e_g\sigma \mid \forall x_1 \ldots \forall x_n (d_g \rightarrow e_g) \in R, \sigma \text{ is a closed substitution}\}$. We then define **Query**$(e, \emptyset, R)$ to be **Query**$(e, \emptyset, R_\Sigma)$. As we show in Section 3.2, R_Σ may be infinite, in which case **Query** is not well-defined. For ease of exposition, we assume that R_Σ is finite for the rest of this section. In Section 3.3 we propose a restriction on the language that guarantees that R_Σ is finite.

Finally, suppose that L is not necessarily $\emptyset$. Then we reduce this case to the previous one by taking **Auth**$(e, L, R) =$ **Auth**$(e, \emptyset, R')$, where

$$R' = R \cup R''$$
$$R'' = \{g \mid \text{for some principal } p \text{ and condition } d,\ (g, p) \in L,\ d \in \textbf{Auth}(\mathbf{Perm}(p, \mathtt{issue}, g), L - \{(g, p)\}, R),\ \text{and } \textbf{Holds}(d, L) \text{ returns } \mathbf{true}\}.$$

R' consists of the grants in R and the grants that are issued by someone who has the authority to do so. Intuitively, R' is the set of grants that hold, since the grants in R implicitly hold and, as discussed in Section 2, a grant g holds if it is issued by a principal p and p is permitted (i.e., has authority) to issue g. To determine if a principal p is permitted to issue a grant g we should be able to call **Query**$(\mathbf{Perm}(p, \mathtt{issue}, g), L, R)$, which returns $\mathbf{true}$ if **Holds**(d, L) returns $\mathbf{true}$ for some d in the set returned by **Auth**$(\mathbf{Perm}(p, \mathtt{issue}, g), L, R)$. This is indeed how we determine if $g \in R'$, with one minor change: the second argument to **Auth** is $L - \{(g, p)\}$ rather than L. We discuss the consequences of this choice in Section 3.2. (As an aside, we suspect that the design decision was made because it is easy to see that the algorithm will not terminate if we replace $L - \{(g, p)\}$ by L.) Pseudocode for **Auth** is given in Figure 2.

Auth(e, L, R):

let $D = \emptyset$
if $L = \emptyset$
then
  % Find D, the conditions under which R implies e
  for each grant $\forall x_1 \ldots \forall x_n (d_g \rightarrow e_g) \in R$
    and each closed substitution σ
    if $e = Pr(p)$ and $e_g\sigma = Pr(p_g)$ or
      $e = \mathbf{Perm}(p, r, s)$ and $e_g\sigma = \mathbf{Perm}(p_g, r, s)$,
      and $p_g \subseteq p$
    then let $D = D \cup \{d_g\sigma\}$
else
  % Find R'
  let $R' = R$
  for each $(g, p) \in L$
    let $L' = L - \{(g, p)\}$
    let $D =$ **Auth**$(\mathbf{Perm}(p, \texttt{issue}, g), L', R)$
    for each $d \in D$
      if Holds(d, L) returns $\mathbf{true}$
      then let $R' = R' \cup \{g\}$
  % Find D, the conditions under which R' implies e
  let $D =$ **Auth**$(e, \emptyset, R')$
Return D

Figure 2. The Auth Algorithm

Holds(d, L):

if $d = \mathbf{true}$ **then** Return $\mathbf{true}$
if $d = d_1 \wedge d_2$ **then** Return **Holds**$(d_1, L) \wedge$ **Holds**(d_2, L)

if $d = \mathbf{Said}(p, e)$
then
  % Find R_p
  let $R_p = \emptyset$
  let $R'_p = \{\forall x \mathbf{Perm}(p', \texttt{issue}, x) \mid p' \in p\}$
  for each $(g, p') \in L$
    let $L' = L - \{(g, p')\}$
    let $D =$ **Auth**$(\mathbf{Perm}(p', \texttt{issue}, g), L', R'_p)$
    for each $d' \in D$
      if Holds(d', L) returns $\mathbf{true}$
      then let $R_p = R_p \cup \{g\}$

  % Return $\mathbf{true}$ if R_p implies e
  for each grant $\forall x_1 \ldots \forall x_n (d_g \rightarrow e_g) \in R_p$ and
      closed substitution σ such that $e_g\sigma = e$
    if Holds$(d_g\sigma, L)$ returns $\mathbf{true}$
      Return $\mathbf{true}$
  Return $\mathbf{false}$

Figure 3. The Holds Algorithm

We define **Holds**(d, L) by induction on the structure of d. If d is **true**, then **Holds**(d, L) returns $\mathbf{true}$. If $d = d_1 \wedge d_2$, then **Holds**(d, L) returns $\mathbf{true}$ iff both **Holds**(d_1, L) and **Holds**(d_2, L) return $\mathbf{true}$. If $d = \mathbf{Said}(p, e)$, then **Holds** sets R_p to the grants issued by a principal who has the authority to do so, under the assumption that p may issue any grant. That is,

$$R_p = \{g \mid \text{for some license } (g, p') \in L \text{ and condition } d \in \textbf{Auth}(\mathbf{Perm}(p', \texttt{issue}, g), L - (g, p'), R'_p) \text{ such that } \textbf{Holds}(d, L) \text{ returns } \mathbf{true}\},$$

where

$$R'_p = \{\forall x \mathbf{Perm}(p', \texttt{issue}, x) \mid p' \in p\}.$$

Holds$(\mathbf{Said}(p, e), L)$ returns $\mathbf{true}$ if there is a grant g in R_p and a condition d such that g and d together imply e (without making use of the subset assumption). Pseudocode for **Holds** is given in Figure 3.

Example 3.1: In Section 2, we argued informally that Amy says Bob is attractive, if the set of licenses is $L = \{(g_1, Alice), (g_2, Amy)\}$, where $g_1 = \mathbf{Smart}(Bob)$ and $g_2 = \mathbf{Said}(Alice, \mathbf{Smart}(Bob)) \rightarrow \mathbf{Attractive}(Bob)$. The formal algorithm confirms this conclusion, since **Holds**$(\mathbf{Said}(Amy, \mathbf{Attractive}(Bob)), L)$ returns $\mathbf{true}$. To see this note that **Holds**$(\mathbf{Said}(Amy, \mathbf{Attractive}(Bob)), L)$ first determines R'_{Amy}, which is $\{g_2\}$. Intuitively, this indicates that g_2 holds, if we assume that the grants issued by Amy hold. The algorithm then calls **Holds**$(\mathbf{Said}(Alice, \mathbf{Smart}(Bob)), L)$ to determine if the condition $\mathbf{Said}(Alice; \mathbf{Smart}(Bob))$ holds. Intuitively, this is done because g_2 and $\mathbf{Said}(Alice, \mathbf{Smart}(Bob))$ together imply $\mathbf{Said}(Amy, \mathbf{Attractive}(Bob))$. **Holds**$(\mathbf{Said}(Alice, \mathbf{Smart}(Bob)), L)$ first calculates R'_{Alice} to be $\{g_1\}$ and then calls **Holds**$(\mathbf{true}, L)$, because g_1 holds if we assume that the grants issued by Alice hold and that g_1 and $\mathbf{true}$ together imply $\mathbf{Smart}(Bob)$. **Holds**$(\mathbf{true}, L)$ returns $\mathbf{true}$, so **Holds**$(\mathbf{Said}(Alice, \mathbf{Smart}(Bob)), L)$ returns true, and then **Holds**$(\mathbf{Said}(Amy, \mathbf{Attractive}(Bob)), L)$ does as well.

Suppose that a trusted principal says that Amy has the authority to issue g_2 (i.e., if Amy says g_2, then g_2 holds). Then we can conclude that Bob really is attractive, because **Query**$(\mathbf{Attractive}(Bob), L, R)$ returns $\mathbf{true}$, where $R = \{\mathbf{Perm}(Amy, \texttt{issue}, g_2)\}$. Specifically, **Query** begins by calling **Auth**$(\mathbf{Attractive}(Bob), L, R)$. **Auth**$(\mathbf{Attractive}(Bob), L, R)$, in turn, calls **Auth**$(\mathbf{Attractive}(Bob), \emptyset, R')$, where $R' = \{g_2, \mathbf{Perm}(Amy, \texttt{issue}, g_2)\}$.

$\mathbf{Auth}(\mathbf{Attractive}(Bob), \emptyset, R')$ returns $\{\mathbf{Said}(Alice; \mathbf{Smart}(Bob))\}$. So, Bob is attractive if the condition $\mathbf{Said}(Alice; \mathbf{Smart}(Bob))$ holds. To determine if the condition holds, **Query** calls $\mathbf{Holds}(\mathbf{Said}(Alice; \mathbf{Smart}(Bob)), L)$. As observed above, $\mathbf{Holds}(\mathbf{Said}(Alice; \mathbf{Smart}(Bob)), L)$ returns $\mathbf{true}$, so $\mathbf{Query}(\mathbf{Attractive}(Bob), L, R)$ does as well. ∎

3.2. An Analysis of Query

In this section we present seven examples in which **Query** gives unexpected results. Example 3.2 reveals a mismatch between **Query** and the informal language description; the discrepancy exists because **Auth** makes the subset assumption and the informal language description does not. Examples 3.3 and 3.4 illustrate the consequences of not including assumptions in the input to **Holds**; in Example 3.3 the grants issued by implicitly trusted principals (i.e., those in R) are disregarded, and in Example 3.4 the assumption that p may issue any grant is disregarded when evaluating $\mathbf{Said}(p, e)$. Examples 3.5 and 3.6 show that **Query** does not terminate on all inputs, for two quite different reasons: Example 3.5 shows that on certain input **Auth** returns an infinite set, and Example 3.6 shows that on certain input **Holds** makes infinitely many recursive calls. Example 3.7 demonstrates that a license (g, p) should not be removed from the set of licenses when determining if p is permitted to issue g. Finally, Example 3.8 uncovers a minor discrepancy between the description of R and its use in **Query**.

Example 3.2: Suppose that Alice is quietly walking beside her two giggling daughters, Betty and Bonnie. Are the three of them a quiet group? Intuitively, the answer is no, since Betty and Bonnie are giggling. According to **Query**, however, the answer is yes. Because Alice is quiet and **Auth** makes the subset assumption, **Query** concludes that the principal $\{Alice, Betty, Bonnie\}$ is quiet. That is,

$$\mathbf{Query}(\mathbf{Quiet}(\{Alice, Betty, Bonnie\}), \emptyset, \{\mathbf{Quiet}(Alice)\})$$

returns $\mathbf{true}$, indicating that the group is quiet. ∎

Example 3.3: Suppose that Alice may issue a grant g and, if it follows from Bob's statements that Alice may issue g, then Charlie may issue g. May Charlie issue g? The answer should be yes. If Alice may issue g, then this right follows from any statement (or no statement at all). However, according to the XrML algorithm, Charlie may not issue g. To see why, let

$$\begin{array}{lcl} g_1 & = & \mathbf{Perm}(Alice, \mathtt{issue}, g), \\ g_2 & = & d \rightarrow \mathbf{Perm}(Charlie, \mathtt{issue}, g), \text{ and} \\ d & = & \mathbf{Said}(Bob, \mathbf{Perm}(Alice, \mathtt{issue}, g)). \end{array}$$

We are interested in

$$\mathbf{Query}(\mathbf{Perm}(Charlie, \mathtt{issue}, g), \emptyset, \{g_1, g_2\}).$$

Query begins by calling

$$\mathbf{Auth}(\mathbf{Perm}(Charlie, \mathtt{issue}, g), \emptyset, \{g_1, g_2\}),$$

which returns $\{d\}$. Intuitively, Charlie may issue g if the condition d holds. The algorithm then executes $\mathbf{Holds}(d, \emptyset)$ (since the set of licenses is $\emptyset$), and this call returns $\mathbf{false}$. Roughly speaking, $\mathbf{Holds}(d, \emptyset)$ returns $\mathbf{false}$ because d does not follow from the (empty) set of licenses. To get the intuitively correct answer that d holds, **Holds** would need to take R into account. But it cannot possibly do this, since R is not part of **Holds**'s input. Because none of the conditions returned by **Auth** are met according to **Holds**, **Query** returns $\mathbf{false}$ and we (wrongly) conclude that Charlie does not have permission to issue g. ∎

Example 3.4: Let d be the condition

$$\mathbf{Said}(\{Alice, Amy\}, \mathbf{Perm}(Alice, \mathtt{issue}, \mathbf{Smart}(Alice))).$$

Intuitively, d holds, because Alice is permitted to issue the grant $\mathbf{Smart}(Alice)$, given that Amy and Alice are permitted to issue every grant. However, $\mathbf{Holds}(d, \emptyset)$ returns $\mathbf{false}$, because R_p includes only grants that are mentioned in L, which is $\emptyset$ in this case.

Suppose that instead of being $\emptyset$, $L = \{(g_1, Alice), (g_2, Amy)\}$, where $g_1 = \mathbf{Smart}(Alice)$ and $g_2 = \mathbf{Said}(Bob, \mathbf{Smart}(Alice)) \rightarrow \mathbf{Perm}(Alice, \mathtt{issue}, \mathbf{Smart}(Alice))$. Now, $\mathbf{Holds}(d, L)$ should return $\mathbf{true}$ for a reason quite different from that above. To see this, notice that if Alice and Amy may issue any grant, then g_1 and g_2 hold. Therefore, Alice is smart and, as in Example 3.3, it follows that $\mathbf{Said}(Bob, \mathbf{Smart}(Alice))$ holds (since Alice is smart, this fact follows from Bob's statements). Because the condition in g_2 holds and g_2 holds, the conclusion of g_2 holds and, thus, Alice may issue the grant $\mathbf{Smart}(Alice)$. Unfortunately, $\mathbf{Holds}(d, L)$ still returns $\mathbf{false}$. Specifically, $\mathbf{Holds}(d, L)$ sets $R_p = \{g_1, g_2\}$. It then calls $\mathbf{Holds}(\mathbf{Said}(Bob, \mathbf{Smart}(Alice)), L)$. This call returns $\mathbf{false}$, because it is not passed the assumption that Alice's statement holds. ∎

Example 3.5: Suppose that Alice is trusted, if Amy says that she may issue some grant (any grant at all). Is Alice trusted? To answer this query, we must compute

$$\mathbf{Query}(\mathbf{Trusted}(Alice), \emptyset, \{\forall x(d \rightarrow \mathbf{Trusted}(Alice))\}),$$

where $d = \mathbf{Said}(Amy, \mathbf{Perm}(Alice, \mathtt{issue}, x))$. **Query** begins by calling

$$\mathbf{Auth}(\mathbf{Trusted}(Alice), \emptyset, \{\forall x(d \rightarrow \mathbf{Trusted}(Alice))\}),$$

which returns $D = \{\mathbf{Said}(Amy, \mathbf{Perm}(Alice, \mathtt{issue}, g)) \mid g \text{ is a grant}\}$. **Query** then calls $\mathbf{Holds}(d', L)$ for each $d' \in D$. However, the language includes infinitely-many

grants, even if there is only one property Pr and one principal p in the language. To see this, define grants g_n, $n \geq 1$, inductively by taking $g_1 = \mathbf{true} \rightarrow Pr(p)$ and $g_{n+1} = \mathbf{Said}(p, \mathbf{Perm}(p, \mathtt{issue}, g_n)) \rightarrow Pr(p)$ for all $n > 0$. Clearly each of these grants is distinct. Since D is an infinite set, **Query** does not terminate. ∎

Example 3.6: Suppose that Alice and Amy may issue any grant, Alice says that Bob is trustworthy if Amy says that he is, and Amy says that Bob is trustworthy if Alice says that he is. Should we conclude that Bob is trustworthy? The intuitive answer is no, since neither Alice nor Amy says that he is.

To answer this query using **Query**, let $e = \mathbf{Trustworthy}(Bob)$, $L = \{(g_1, Alice), (g_2, Amy)\}$, $g_1 = \mathbf{Said}(Amy, e) \rightarrow e$, $g_2 = \mathbf{Said}(Alice, e) \rightarrow e$, and $R = \{\forall x \mathbf{Perm}(Alice, \mathtt{issue}, x), \forall x \mathbf{Perm}(Amy, \mathtt{issue}, x)\}$, where x is a variable of sort *Resource*. We are interested in **Query**(e, L, R). **Query**(e, L, R) begins by calling **Auth**(e, L, R), which returns the set $D = \{\mathbf{Said}(Amy, e), \mathbf{Said}(Alice, e)\}$. **Query** then calls **Holds** on each of the conditions in D. During the execution of **Holds**$(\mathbf{Said}(Amy, e), L)$, **Holds** sets $R_{Amy} = \{g_2\}$, and then calls **Holds**$(\mathbf{Said}(Alice, e), L)$. **Holds**$(\mathbf{Said}(Alice, e), L)$ sets $R_{Alice} = \{g_1\}$, and then calls **Holds**$(\mathbf{Said}(Amy, e), L)$. Notice that this is just the original call. It is not hard to see that an infinite number of calls to **Holds**$(\mathbf{Said}(Amy, e), L)$ will be made during the execution of **Holds**$(\mathbf{Said}(Amy, e), L)$ and, thus, the algorithm does not terminate.

It is tempting to conclude that a conclusion e follows from a set L of licenses and a set R of grants only if **Query**(e, L, R) terminates and returns $\mathbf{true}$. Unfortunately, this might not lead to the intuitively correct answer. To see why, consider a slight modification of our previous example where we add $\{\mathbf{Trustworthy}(Bob)\}$ to the grant set R. Intuitively, this means that an implicitly trusted principal says that Bob is trustworthy. It now seems reasonable to expect **Query**(e, L, R') to return $\mathbf{true}$, where $R' = R \cup \{e\}$, and e, L, and R are as defined in the original example. Surely the issued grants imply that Bob is trustworthy, since a grant issued by a trusted principal says just that! However, the execution of **Query**(e, L, R') does not terminate for the same reason that **Query**(e, L, R) does not terminate. ∎

Example 3.7: Suppose that Alice says that she is smart, and if Alice says that she is smart, then she is permitted to say that she is smart. Is Alice smart? Intuitively, the answer is yes, because Alice is permitted to say that she is smart and she does so. But consider **Query**$(\mathbf{Smart}(Alice), L, R)$, where $L = \{(g, Alice)\}$, $R = \{\mathbf{Said}(Alice, \mathbf{Smart}(Alice)) \rightarrow \mathbf{Perm}(Alice, \mathtt{issue}, g)\}$, and $g = \mathbf{Smart}(Alice)$. **Query**$(\mathbf{Smart}(Alice), L, R)$ begins by calling **Auth**$(\mathbf{Smart}(Alice), L, R)$. This algorithm checks whether or not Alice is permitted to issue g. It determines that Alice may not issue g, because the permission does not follow from R and $L - \{(g, Alice)\}$. Since Alice is not permitted to issue g, **Auth** sets $R' = R$ and returns $\emptyset$. Because **Auth** returns $\emptyset$, **Query** returns **false**. ∎

Example 3.8: Suppose that a trusted principal issues the grant g_1, where g_1 says 'if Amy says that some principal may issue g_2, then Alice may issue it'. May Alice issue g_2? At first glance, the answer should be no. Alice may issue g_2 only if Amy says that some principal may issue g_2; we cannot conclude this from g_1 alone. However, g_1 was issued by a trusted principal, who we will call p_t, p_t may issue any grant, and, thus, p_t's right to issue g_2 follows from Amy's statements, because it is true. So, the condition in g_1 is met and Alice may issue g_2. Unfortunately, **Query**$(\mathbf{Perm}(Alice, \mathtt{issue}, g_2), \emptyset, R)$ returns **false**, where $R = \{\forall x(\mathbf{Said}(Amy, \mathbf{Perm}(x, \mathtt{issue}, g_2)) \rightarrow \mathbf{Perm}(Alice, \mathtt{issue}, g_2))\}$. Thus, **Query** does not always treat the grants in R as having been issued by a trusted principal (who may issue every grant). Perhaps a better interpretation is that **Query** treats the grants in R as being those that are accepted, without assuming that they have been issued by some trusted principal. While this is a perfectly reasonable interpretation, it is not consistent with the discussion in the XrML document. ∎

3.3. A Corrected Version of Query

In this section we revise **Query** and the informal language description to avoid the problems observed in Section 3.2. Three of the corrections are fairly straightforward.

- We resolve the mismatch illustrated in Example 3.2 by removing the subset assumption from **Auth**. We note that the language is sufficiently expressive to force the subset assumption, if desired, by including the following grants in R:

  $$g = \forall x_1 \ldots \forall x_3(\mathbf{Perm}(x_1, \mathtt{issue}, x_2) \rightarrow \mathbf{Perm}(x_1 \cup x_3, \mathtt{issue}, x_2))$$
  $$g_i = \forall x_1 \forall x_2(Pr_i(x_1) \rightarrow Pr_1(x_1 \cup x_2)),$$
  $$\text{for } i = 1, \ldots, n,$$

  where $x_1, \ldots, x_3$ are variables of the appropriate sorts and $Pr_1, \ldots, Pr_n$ are the properties in the language.

- We address the problem exhibited in Example 3.3 by modifying **Query** so that R (the set of grants issued by a trusted principal) is passed to and used appropriately

by **Holds**. In addition, we add R'_p to R during the evaluation of $\mathbf{Said}(p, e)$. This solves the problems highlighted in Example 3.4.

- We do not revise **Query** in any way to respond to the discussion in Example 3.8. Instead, we change the intuitive meaning of R; we assume R is the set of grants that implicitly hold, although they are not necessarily issued by a principal (trusted or otherwise). We remark that we could force the algorithm to treat a principal p as trusted by including the grant $\forall x \mathbf{Perm}(p, \mathtt{issue}, x)$ in R, where x is a variable of the sort $Resource$.

There are three remaining issues, corresponding to Examples 3.5, 3.6, and 3.7. We consider each of these in turn.

We restrict the language to avoid the type of nontermination that occurs in Example 3.5. There are various restrictions that could accomplish this. To understand ours, recall that **Auth**(e, L, R) first extends R to R' by adding all the grants that are issued by someone who has the authority to do so. Since all the grants in $R' - R$ are in L, the set R' must be finite. Then, **Auth** creates the possibly-infinite set R_Σ consisting of all substitution instances of grants in R', and returns $\{d \mid d \rightarrow e \in R_\Sigma\}$. (For simplicity here, we are assuming that **Auth** does not use the subset assumption; this assumption does not affect our discussion.) Since **Auth** considers only the grants in R_Σ whose conclusion matches the first input to **Auth**, we could certainly replace R_Σ by R'_Σ, where

$$R'_\Sigma = \{d_g\sigma \rightarrow e \mid \forall x_1 \ldots \forall x_n (d_g \rightarrow e_g) \in R', \\ \sigma \text{ is a closed substitution, and } e_g\sigma = e\}.$$

Because e is closed, R'_Σ is finite if for every grant g in R', if g's condition mentions a free variable x, then either x ranges over a finite set or x appears in g's conclusion. Our solution is simply to restrict the language so that every grant has this property. Since, in our fragment, there are infinitely-many resources (grants), and only finitely many principles, this amounts to restricting the language so that if $\forall x_1 \ldots \forall x_n (d_g \rightarrow e_g)$ is a grant, then every free variable of sort $Resource$ that appears in d_g also appears in e_g. We call a grant *acceptable* if it has this property; we call a license (g, p) acceptable if g is acceptable. Thus, for example, $\forall x \forall y (\mathbf{Said}(\emptyset, \mathbf{Perm}(x, \mathtt{issue}, y)) \rightarrow \mathbf{Perm}(Alice, \mathtt{issue}, y))$ is acceptable, but

$$\forall y \forall z (\mathbf{Said}(\emptyset, \mathbf{Perm}(Alice, \mathtt{issue}, y)) \rightarrow \\ \mathbf{Perm}(Alice, \mathtt{issue}, z))$$

is not. It is not hard to see that if a grant $g = \forall x_1 \ldots \forall x_n (d_g \rightarrow e_g)$ is acceptable, then for any closed conclusion e, there are at most $2^{|g||primitivePrin|}$ grants of the form $d_g\sigma \rightarrow e_g\sigma$ such that σ is a closed substitution and $e_g\sigma = e$. Thus, by restricting to acceptable grants and licenses, we solve the problem raised in Example 3.5. Note that, this restriction is, in a sense, necessary to deal with the problem: If g is not acceptable, then there is a variable of sort $Resource$ in d_g, say x, that does not appear in e. If we can find a substitution σ such that $e_g\sigma = e$, then there must be infinitely many distinct grants $d_g\sigma \rightarrow e_g\sigma$ where $e_g\sigma = e$, since there are infinitely many distinct substitutions possible for x. For example, if $e = \mathbf{Perm}(Alice, \mathtt{issue}, g)$, where g is a closed grant, then the set of substitutions σ such that

$$\mathbf{Perm}(Alice, \mathtt{issue}, z)\sigma = \mathbf{Perm}(Alice, \mathtt{issue}, g)$$

and $\mathbf{Said}(\emptyset, \mathbf{Perm}(Alice, \mathtt{issue}, y))\sigma$ is closed is the infinite set

$$\{[y/t\ z/g] \mid t \text{ is a closed term of sort } Resource\}.$$

We now consider the type of nontermination exhibited in Example 3.6. This behavior occurs because **Query** tries to verify that a **Said** condition d holds by checking if d holds. We see this in Example 3.6 when the algorithm tries to verify that the condition **Holds**$(\mathbf{Said}(Amy, e), L)$ holds by repeatedly calling **Holds**$(\mathbf{Said}(Amy, e), L)$. To correct the problem, we modify **Holds** so that no call is evaluated twice. Specifically, the revised **Holds** takes a fourth argument S that is the set of **Said** conditions that have been the first argument to a previous call; **Holds**(d, L, R, S) returns **false** if $d \in S$. In addition, **Holds** sets S to $\emptyset$ if grants are added to R, because no condition has been evaluated under the new set of assumptions. Because **Holds2** calls the other algorithms, they also take the additional argument.

Finally, we resolve the problem illustrated in Example 3.7. As we hinted in Section 3.1, the problem lies in the definition of R'. Recall that we define **Auth**(e, L, R) = **Auth**$(e, \emptyset, R')$. Roughly speaking, R' should consist of the set of grants in R together with those issued by someone who has the authority to do so. In other words, R' should be $R \cup \{g \mid \text{for a principal } p\,, (g, p) \in L \text{ and } \mathbf{Query}(\mathbf{Perm}(p, \mathtt{issue}, g), L, R) \text{ returns } \mathbf{true}\}$. However, when computing $\mathbf{Query}(\mathbf{Perm}(p, \mathtt{issue}, g), L, R)$, **Auth** is given the argument $L - \{(g, p)\}$ rather than L. Our solution is to do the "right" thing here, and compute $\mathbf{Query}(\mathbf{Perm}(p, \mathtt{issue}, g), L, R)$. But now we have to deal with the problem of termination, since a consequence of our change is that $\mathbf{Query}(e, L, R)$ terminates only if the set $L = \emptyset$. To ensure termination, we modify **Auth** so that it does not call itself to evaluate a conclusion that has already been considered. We remark that this is essentially the same approach that we take to ensuring that **Holds** terminates.

Pseudocode for the modified **Query2** is given in Figure 4. **Query2** calls **Auth2** and **Holds2**, which are the modified versions of **Auth** and **Holds**, respectively. Pseudocode

for **Auth2** is given in Figure 5 and pseudocode for **Holds2** is given in Figure 6.

To summarize, the main differences between **Query2**, **Auth2**, and **Holds2** and their analogues as defined in the XrML document are:

- **Auth2** does not make use of the subset assumption.
- **Holds2**'s output depends on R, which is the set of grants that implicitly hold.
- **Holds2**$(\mathbf{Said}(p, e), L, R, S, E)$ returns $\mathbf{true}$ if e follows from L, R, and R'_p, which says that every principal in p may issue every grant.
- For any $\mathbf{Said}$ condition d, **Holds2**(d, L, R, S, E) returns $\mathbf{false}$ if $d \in S$.
- **Auth2** does not remove (g, p) from the set of licenses when determining if p may issue g.
- For any conclusion e, **Auth2**(e, L, R, S, E) returns $\mathbf{false}$ if $e \in E$.

Query2(e, L, R, S, E):

let $D =$ **Auth2**(e, L, R, S, E)
for each $d \in D$
  if Holds2(d, L, R, S, E) returns $\mathbf{true}$
  then Return $\mathbf{true}$
Return $\mathbf{false}$

Figure 4. The Query2 Algorithm

Auth2$(e, L, R, S.E)$:

if $e \in E$
then Return $\emptyset$
else
  let $E' = E \cup \{e\}$
  let $R' = R$
  for each $(g, p) \in L$
    if Query2$(\mathbf{Perm}(p, \mathtt{issue}, g), L, R, S, E')$
      returns $\mathbf{true}$
    then let $R' = R' \cup \{g\}$
  let $D = \emptyset$
  for each grant $\forall x_1 \ldots \forall x_n (d_g \rightarrow e_g) \in R'$ and
    closed substitution σ such that $e_g\sigma = e$
    let $D = D \cup \{d_g\sigma\}$
  Return D

Figure 5. The Auth2 Algorithm

Holds2(d, L, R, S, E):

if $d = \mathbf{true}$ **then** Return $\mathbf{true}$
if $d = d_1 \wedge d_2$
then Return $\bigwedge_{i=1,2}$ **Holds2**(d_i, L, R, S, E)
if $d = \mathbf{Said}(p, e)$
then
  if $d \in S$
  then return $\mathbf{false}$
  else
    let $S' = S \cup \{d\}$
    let $R' = R \cup \{\forall x \mathbf{Perm}(p', \mathtt{issue}, x) \mid p' \in p\}$
    if $R' = R$
      return **Query2**(e, L, R, S', E)
    else
      % Reset S' and E; they might follow from R'.
      return **Query2**$(e, L, R', \emptyset, \emptyset)$

Figure 6. The Holds2 Algorithm

We observe that **Query2** terminates on all input that is in our restricted language. Recall that a grant $g = \forall x_1 \ldots \forall x_n (d_g \rightarrow e_g)$ is *acceptable* if every free variable of sort $Resource$ that is mentioned in d_g is also mentioned in e_g; the license (g, p) is acceptable if g is acceptable.

Theorem 3.9: *Suppose that e is a closed conclusion, L is a set of acceptable licenses, R is a set of acceptable grants, S is a set of* $\mathbf{Said}$ *conditions, and E is a set of conclusions. Then* ***Query2****(e, L, R, S, E) terminates.*

Query2 has the intuitively correct input/output behavior for the examples in Section 3.2. In Example 3.2, **Query2**$(\mathbf{Quiet}(p'), \emptyset, \mathbf{Quiet}(p), \emptyset, \emptyset)$ returns $\mathbf{false}$, when $p' = \{Alice, Betty, Bonnie\}$ and $p = Alice$, because **Auth2** does not rely on the subset assumption. In Example 3.3, **Query2**$(\mathbf{Perm}(Charlie, \mathtt{issue}, g), \emptyset, \{g_1, g_2\}, \emptyset, \emptyset)$ returns $\mathbf{true}$, because **Holds2** correctly handles the set R of grants that implicitly hold. In Example 3.4, **Holds2**$(d, \emptyset, \emptyset, \emptyset, \emptyset)$ and **Holds2**$(d, L, \emptyset, \emptyset, \emptyset)$ both return the intuitively correct answer $\mathbf{true}$. We avoid the problem in Example 3.5, since the grant $\forall x(\mathbf{Said}(Amy, \mathbf{Perm}(Alice, \mathtt{issue}, x)) \rightarrow \mathbf{Trusted}(Alice))$ is not acceptable. In Example 3.6, **Query2**$(\mathbf{Trustworthy}(Bob), L, R, \emptyset, \emptyset)$ returns $\mathbf{false}$ and **Query2**$(\mathbf{Trustworthy}(Bob), L, R', \emptyset, \emptyset)$ returns $\mathbf{true}$, as it should. In Example 3.7, **Auth2**$(\mathbf{Smart}(Alice), L, R, \emptyset, \emptyset)$ returns the intuitively correct answer, namely $\mathbf{true}$, because Alice's issuance of g is taken into account when determining if Alice may issue g. Finally, in Example 3.8, **Query2**$(\mathbf{Perm}(Alice, \mathtt{issue}, g_2), \emptyset, R, \emptyset, \emptyset)$

returns **false**, where $R = \{\forall x(\mathbf{Said}(Amy, \mathbf{Perm}(x, \texttt{issue}, g_2)) \rightarrow \mathbf{Perm}(Alice, \texttt{issue}, g_2))\}$. This is the intuitively correct answer, because we no longer assume that some principals are implicitly trusted.

We have discussed the examples in Section 3.2 with members of the MPEG-21 working group that are developing XrML. They agreed with our observations and have apparently dealt with many of them in the final version of the specification. Based on these discussions, we believe that the technical results in the rest of the paper should apply with very little change to the final XrML specification.

4. Formal Semantics

In this section we provide a formal semantics for the XrML fragment described in the previous section, by translating licenses in the grammar to formulas in a modal many-sorted first-order logic. The logic has three sorts: $Principal$, $Right$, and $Resource$. The vocabulary includes the following symbols:

- a constant p of sort $Principal$ for every principal $p \in primitivePrin$;
- a constant `issue` of sort $Right$;
- a ternary predicate $\mathbf{Perm}$ that takes arguments of sort $Principal$, $Right$, and $Resource$;
- a unary predicate Pr that takes an argument of sort $Principal$ for each property $Pr \in primitiveProp$;
- a function $\cup : Principal \times Principal \longrightarrow Principal$;
- a constant c_g of sort $Resource$ for each grant g in the language.
- a modal operator Val that takes a formula as an argument.

Intuitively, $Pr(p)$ means principal p has property Pr, and $\mathsf{Val}(\varphi)$ means formula φ is valid. Notice that every principal in the grammar corresponds to a term in the language, because $\cup$ is a function symbol.

The semantics of our language is just the standard semantics for first-order logic, extended to deal with Val. We restrict attention to models for which $\cup$ satisfies the following properties:

U1. $\forall x((x \cup x) = x)$

U2. $\forall x_1 \forall x_2((x_1 \cup x_2) = (x_2 \cup x_1))$

U3. $\forall x_1 \forall x_2 \forall x_3(x_1 \cup (x_2 \cup x_3)) = ((x_1 \cup x_2) \cup x_3))$

U4. $\forall x((x \cup \emptyset) = x)$

In addition, we are interested only in Herbrand models, where the only elements of sort $Principal$, $Right$, and $Resource$ are interpretations of syntatic terms. We call such models *acceptable models*. $\mathsf{Val}(\varphi)$ is true in an acceptable model m if φ is true in all acceptable models. If a formula φ is true in all acceptable models, then we say φ is valid and write $\models \varphi$. Thus, $\mathsf{Val}(\varphi)$ is true in an acceptable model iff $\models \varphi$.

The translation takes a set L of licenses, a set R of grants, and a set S of **Said** conditions as parameters. Intuitively, L is the set of licenses that have been issued, R is the set of grants that are assumed to be true, and S is the set of **Said** conditions that have been considered when determining if a condition holds. (The role of S should become clearer in the course of defining the translation.) Finally, we assume that if $(g, p) \in L$ then p is variable-free. (We do this because the assumption is built into **Query**.) The translation is defined below, where $t^{L,R,S}$ is the translation of the string t given input L, R, and S.

- If $(g, p) \in L$, $(g, p)^{L,R,S} = \mathbf{Perm}(p, \texttt{issue}, c_g) \Rightarrow g^{L,R,S}$.
- If $(g, p) \notin L$, $(g, p)^{L,R,S} = \mathbf{true}$.
- $\forall x_1 \ldots \forall x_n(d \rightarrow e)^{L,R,S} = \forall x_1 \ldots \forall x_n(d^{L,R,S} \Rightarrow e^{L,R,S})$.
- $\mathbf{true}^{L,R,S} = \mathbf{true}$.
- If $\mathbf{Said}(p, e) \in S$, then $\mathbf{Said}(p, e)^{L,R,S} = \mathbf{false}$.
- If $\mathbf{Said}(p, e) \notin S$, then $\mathbf{Said}(p, e)^{L,R,S} =$

$$\mathsf{Val}(\bigwedge_{\ell \in L} \ell^{L,R',S'} \wedge \bigwedge_{g \in R'} g^{L,R',S'} \Rightarrow e^{L,R',S'}),$$

  where $S' = S \cup \{\mathbf{Said}(p, e)\}$ and $R' = R \cup \{\forall x \mathbf{Perm}(p', \texttt{issue}, x) \mid p' \in p \text{ and } x \text{ is a variable of sort } Resource\}$.
- $(d_1 \wedge d_2)^{L,R,S} = d_1^{L,R,S} \wedge d_2^{L,R,S}$.
- $\mathbf{Perm}(p, r, s)^{L,R,S} = \mathbf{Perm}(p, r, s^*)$, where $s^* = s$ if s is a variable of sort $Resource$ and $s^* = c_s$ if s is a grant.
- $Pr(p)^{L,R,S} = Pr(p)$.

The following theorem shows that our semantics match the procedure given in the XrML document (corrected as described in Section 3).

Theorem 4.1: *For every closed conclusion e, set L of acceptable licenses, and set R of acceptable grants,* ***Query2***$(e, L, R, \emptyset, \emptyset)$ *returns* **true** *iff* $\models \bigwedge_{\ell \in L} \ell^{L,R,\emptyset} \wedge \bigwedge_{g \in R} g^{L,R,\emptyset} \Rightarrow e^{L,R,\emptyset}$.

Before leaving this section, we remark that, in the translation of $\mathbf{Perm}(p, \texttt{issue}, g)$, we can replace g by the constant c_g because of our assumption that g is closed. If we had allowed g to be open (i.e., to include free variables), then we would need to translate every conclusion $\mathbf{Perm}(p, \texttt{issue}, g)$ as $\mathbf{Perm}(p, \texttt{issue}, f_g(x_1, \ldots, x_n))$, where f_g is a function symbol and $x_1, \ldots, x_n$ are the free

variables in g. Although this extension is straightforward, it might have a substantial impact on tractability; see Section 6.

5. Complexity

In this section, we examine the complexity of deciding if the revised **Query2** returns $\mathbf{true}$ on a given input. As we now show, the problem is NP hard. The real problem turns out to be that, if there are n primitive principals, we can construct 2^n principals using the $\cup$ operator.

Theorem 5.1: *Deciding if* ***Query2***$(e, L, R, \emptyset, \emptyset)$ *returns* $\mathbf{true}$*, where e is a closed conclusion, L is a set of acceptable licenses, and R is a set of acceptable grants, is NP-hard.*

Proof: (Sketch:) We show that we can reduce the Hamiltonian path problem to the problem of determining whether **Query2**$(e, L, R, \emptyset, \emptyset)$ returns $\mathbf{true}$. Given a graph $G(V, E)$, where $V = \{v_1, \ldots, v_n\}$, we take $v_1, \ldots, v_n$ to be primitive principles. We also assume that the language has primitive properties **Node**, **Edge**, and **Path**. For each node $v \in V$, we consider a grant $g_v = \mathbf{Node}(v)$ (recall that this is an abbreviation for $\mathbf{true} \rightarrow \mathbf{Node}(v)$); for each edge $e = (v, v') \in E$, we consider the grant $g_{(v,v')} = \mathbf{Edge}(\{v, v'\})$. (We are taking advantage of the fact here that $\{v, v'\}$ is a principal if v and v' are primitive principals.) Finally, let g be the grant $\forall x_1 \ldots \forall x_n (d_1 \wedge d_2 \rightarrow \mathbf{Path}(\{x_1, \ldots, x_n\}))$, where

$$d_1 = \bigwedge_{1 \leq i \leq n} \mathbf{Said}(Alice, \mathbf{Node}(x_i)) \text{ and}$$
$$d_2 = \bigwedge_{1 \leq i \leq n-1} \mathbf{Said}(Alice, \mathbf{Edge}(\{x_i, x_{i+1}\})).$$

Let $R = \{g_v \mid v \in V\} \cup \{g_e \mid e \in E\} \cup \{g\}$. It is not hard to show that **Query2**$(\mathbf{Path}(\{v_1, \ldots, v_n\}), \emptyset, R, \emptyset, \emptyset)$ returns $\mathbf{true}$ iff G has a Hamiltonian path. To see this, observe that **Auth2**$(\mathbf{Path}(\{v_1, \ldots, v_n\}), \emptyset, R, \emptyset, \emptyset)$ returns $\{d_1\sigma \wedge d_2\sigma \mid \sigma(x_i) = v_\pi(i), i = 1, \ldots, n,$ where π is some permutation of $\{1, \ldots, \mathrm{n}\}\}$. The condition $d_2\sigma$ holds iff there is a path $x_1\sigma, \ldots, x_n\sigma$. Thus, **Query2**$(\mathbf{Path}(\{v_1, \ldots, v_n\}), \emptyset, R, \emptyset, \emptyset)$ returns $\mathbf{true}$ iff there is a Hamiltonian path in G. ∎

Theorem 5.1 shows that deciding the consequences of even simple grants can be hard. The real culprit here, as we hinted before, is the ability to form more complex principals from primitive principals by taking union. If we prohibit the use of the union operator (so that the only principals that can appear in grants are primitive principals, and variables of sort *Principal* are taken to range over primitive principals), then the problem becomes tractable. The key insight is that, without union, the fragment is quite similar to function-free negation-free Horn clauses. (The only difference is that our translation includes the Val operator, which does not affect the complexity.) Determining if a literal follows from a set of function-free negation-free Horn clauses is a well-known polynomial time problem [5]. Using the same techniques, we can answer our queries in polynomial time.

Theorem 5.2: *The problem of deciding whether* ***Query2***$(e, L, R, \emptyset, \emptyset)$ *returns* $\mathbf{true}$*, where e is a closed conclusion, L is a set of acceptable licenses, and R is a set of acceptable grants, and $\cup$ does not appear in e, L, or R, is in polynomial time.*

How serious a restriction is it to disallow the $\cup$ operator? Principals appear as the second argument in a license, the first argument in a $\mathbf{Said}$ condition, and the first argument in a conclusion.

- According to the XrML documentation, the license $(g, \{p_1, \ldots, p_n\})$ is an abbreviation for the set of licenses $\{(g, p) \mid p \in \{p_1, \ldots, p_n\}\}$. It follows that we can restrict the second argument of licenses to primitive principals and variables without sacrificing any expressive power. (In fact, we can restrict the second argument of licenses to only primitive principals, because **Query** assumes that if (g, p) is a license in L, then p is variable-free.)
- As for the $\mathbf{Said}$ condition, if we disallow $\cup$, then we do lose some expressive power. However, the loss is not as serious as it may appear. We can replace all grants of the form $\mathbf{Said}(\{p1, \ldots, p_n\}, e)$, where $p_1, \ldots, p_n$ are primitive principals, by a grant $\mathbf{Said}(\{p_1, \ldots, p_n\}^*, e)$, where $\{p_1, \ldots, p_n\}^*$ is a new primitive principal, and then expand the set L of issued licenses by adding a new license $(g, \{p_1, \ldots, p_n\}^*)$ for every license (g, p) already in L, where $p \in \{p_1, \ldots, p_n\}$. It is not hard to show that this results in at most a quadratic increase in the number of grants. Thus, as long as the first argument to $\mathbf{Said}$ is variable-free, we can express it without using $\cup$. However, it seems that we do lose some expressive power in not being able to express $\mathbf{Said}$ conditions where the first argument is a set that involves a variable.
- To understand the impact of our restriction on conclusions, we need to consider the meaning of statements such as $\mathbf{Trust}(\{Alice, Bob\})$ and $\mathbf{Perm}(\{Alice, Bob\}, issue, g)$. According to the XrML document,$\mathbf{Trust}(\{Alice, Bob\})$ means Alice and Bob together (i.e., when viewed as a single entity) is trusted; $\mathbf{Perm}(\{Alice, Bob\}, issue, g)$ means Alice and Bob is permitted to issue g. However, the XrML document does not explain precisely what it means for Alice and Bob to be viewed as a single entity Indeed, it seems to treat this notion somewhat inconsistently (recall the inconsistent

use of the subset assumption). There are other difficulties with sets. Notice that if $\{Alice, Bob\}$ is permitted to issue a grant, then presumably g holds if $\{Alice, Bob\}$ issues g. However, according to the XrML documentation, the license $(g, \{Alice, Bob\})$ is simply an *abbreviation* for the set of licenses $\{(g, \{Alice\}), (g, \{Bob\})\}$. So, it is unclear whether a principal that is not a singleton can issue a license. Furthermore, if principals that are not singletons can issue grants and $\{Alice, Bob\}$ is permitted to issue a grant g, then it seems reasonable to conclude that g holds if g is issued by both Alice and Bob, but it is not clear whether or not g holds if it is issued by only Alice (or by only Bob).

There may well be applications where there is an obvious and clear semantics for these notions. But we suspect that, in these applications, there are typically only relatively few groups of interest. In that case, it may be possible to simply take these groups to be new primitive principals, and express the relationship between the group and its elements in the language. (This approach has the added advantage of forcing license writers to be clear about the semantics of groups.)

In short, we are optimistic that many applications do not need the union function.

6. The Entire XrML Language

XrML has several components that are not in our fragment. Most have been excluded simply for ease of exposition. In this section we list the main omissions, briefly discussing each one.

- XrML supports *patterns*, where a pattern restricts the terms over which a variable ranges. For example, if the variable x is restricted to the pattern 'ends in Simpson', then x ranges over the terms that meet this syntactic constraint (e.g., x ranges over $\{HomerSimpson, MargeSimpson, \ldots\}$). Patterns in XrML correspond to properties in our fragment. We could represent the example in our fragment by having the property **Simpson** in the language and having the set of grants determine which terms have the property. XrML also allows a pattern to be a set of patterns. We can express a set of patterns as a conjunction of patterns. Since we can express conjunctions of properties in our fragment, we can also capture sets of patterns.
- XrML supports *delegable grants*. A delegable grant g can be viewed as a conjunction of a grant g' in our fragment and a set G of grants that, essentially, allow other principals to issue g'. For example, the delegable grant 'Doctor Alice may view Charlie's medical file and she may also give the right to view the file to her colleague, Doctor Bob' can be viewed as the conjunction of the grant 'Doctor Alice may view Charlie's medical file' and the grant 'Alice is permitted to issue the grant 'Doctor Bob may view Charlie's medical file". Thus, we can express delegable grants in our framework.
- XrML supports *grantGroups*, where a grantGroup is a set of grants. We can extend our syntax to support grantGroups by closing the set of grants (as currently defined) under the union operator. Note that our proposed treatment of grantGroups is quite similar to our current treatment of principals.
- XrML supports additional types of rights, resources, and conditions, beyond what we have in our fragment. There seems to be no difficulty in extending our translation to handle these new features, and proving an analogue of Theorem 4.1. However, full XrML allows resource terms to be formed by applying functions other than $\cup$. For example, the Standard Extension [3][1] refers to a *container* resource that is a sequence of resources. This naturally translates to a function container:$Resource \times Resource \longrightarrow Resource$, so that the container $\langle s_1, s_2, s_3 \rangle$ is translated as container$(s_1, \text{container}(s_2, s_3))$. Allowing such functions makes the problem of deciding if a conclusion follows from a set of XrML licenses and grants undecidable, for much the same reason that the validity problem for negation-free Datalog with function symbols is undecidable [5].
- XrML allows an application to define additional principals, rights, resources, and conditions within the XrML framework. Obviously, we cannot analyze terms that have yet to be defined; however, we do not anticipate any difficulty in extending the translation to deal with these terms and getting an analogue of Theorem 4.1.
- XrML allows a grant g to include free variables if g appears in the scope of a closed grant. As we mention in Section 4, there is no problem dealing with this extra expressive power semantically; we simply replace each constant c_g by a function f_g. However, there might be a problem with the extended language's tractability. As we have just shown, adding function symbols in general leads to undecidability. We are currently investigating whether adding the symbols in this particular way has similar consequences.

1 XrML has three parts: the core language that we discuss here; extensions to the core that are provided by the XrML language developers, which includes the Standard Extension; and extensions made by applications to suit their particular needs.

- XrML allows licenses to be encrypted and supports abbreviations via the *Inventory* component. However, the XrML procedure for determining if a permission follows from a set of licenses assumes that all licenses are unencrypted and all abbreviations have been replaced by the statements for which they stood. In other words, these features are engineering conveniences that are not part of understanding or reasoning about licenses.

7. Concluding Remarks

We have examined XrML carefully, showing that the XrML algorithm does not seem to capture the designers' intentions in a number of ways. Since no formal semantics for XrML is given in the XrML documentation, we cannot argue that the XrML algorithm is incorrect, although its behavior does not always seem reasonable. To address the problem, we provided formal semantics for XrML in a way that we believe captures the designers' intent, modified the algorithm, and showed that the modified algorithm corresponds to our semantics in a precise sense. Our work emphasizes the need for license languages to have formal semantics. Without formal semantics, even carefully crafted languages are prone to ambiguities and inconsistencies.

We have examined only a fragment of XrML. A key reason for XrML's popularity is that the framework is extensible; applications can define new components (i.e., principals, rights, resources, and conditions) to suit their needs. We do not believe there should be any difficulty in giving semantics to the extended language. The real question of interest though is whether we can find useful *tractable* extensions. As we have already seen, although functions pose no semantic difficulties, adding them makes determining what follows from XrML licenses and grants undecidable. Another obvious and desirable feature to add is negation. Currently, XrML does not support negation in either the condition or conclusion of grants. This is a significant expressive weakness. Without negation, policy makers cannot distinguish actions they would like to forbid from the actions that they do not care to regulate. This makes merging two sets of policies essentially impossible; the merger will be inconsistent unless the polices are identical.

While it is easy to extend XrML to support negation, doing so without placing further restrictions on the language quickly leads to intractability. We believe that, using ideas from our earlier work [2], we will be able to identify useful tractable fragments of XrML extended with negation. However, we leave this to future work.

Acknowledgements

Many thanks to Xin Wang (an editor for the XrML document), who answered our questions about the intended meaning of various XrML components.

References

[1] ContentGuard. XrML: The digital rights language for trusted content and services. http://www.xrml.org/, 2001.

[2] J. Halpern and V. Weissman. Using first-order logic to reason about policies. In *Proc. 16th IEEE Computer Security Foundations Workshop*, pages 187–201, 2003.

[3] MPEG. MPEG-21 rights expression language FCD. http://www.chiariglione.org/mpeg/working_documents.htm, 2003.

[4] MPEG. Information technology—Multimedia framework (MPEG-21) – Part 5: Rights expression language (ISO/IEC 21000-5:2004). http://www.iso.ch/iso/en/, 2004.

[5] A. Nerode and R. Shore. *Logic for Applications*. Springer-Verlag, New York, 2nd edition, 1997.