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 \((g, p)\) issues (i.e., says) \(g\). For example, the license (Bob is smart, Alice) means “Alice says ‘Bob is smart’”.

A grant has the form \(\forall x_1, \ldots, \forall x_n(\text{condition} \rightarrow \text{conclusion})\), 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 \(\text{Said}(p_1, e_1) \land \ldots \land \text{Said}(p_n, e_n)\), where \(p_i\) is a principal and \(e_i\) is a conclusion. We represent the empty conjunction (when \(n = 0\)) by \(\text{true}\). Roughly speaking, \(\text{Said}(p_i, e_i)\) 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 \(\text{Perm}(p, r, s)\) or the form \(\text{Pr}(p)\) where \(Pr\) is a property, \(p\) is a principal, \(r\) is a right (i.e., an action), and \(s\) is a resource. The conclusion \(\text{Perm}(p, r, s)\) means \(p\) may exercise \(r\) over \(s\).

For example, \(\text{Perm}(\text{Bob, edit, budget report})\) means Bob may edit the budget report. The conclusion \(\text{Pr}(p)\) means \(p\) has the property \(Pr\). For example, the conclusion \(\text{Attractive}(\text{Bob})\) means Bob is attractive.

We abbreviate the grant \(\forall x_1, \ldots, \forall x_n(\text{true} \rightarrow e)\) 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 \((g_1, \text{Alice})\), where \(g_1 = \text{Smart}(\text{Bob})\) (recall that this is an abbreviation for \(\text{true} \rightarrow \text{Smart}(\text{Bob})\)), and we can write the second as \((g_2, \text{Amy})\), where \(g_2 = \text{Said}(\text{Alice}) \land \text{Smart}(\text{Bob}) \rightarrow \text{Attractive}(\text{Bob})\). Because \((g_1, \text{Alice})\) 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 \(\text{Said}(\text{Amy}) \land \text{Attractive}(\text{Bob})\) 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 \(\text{primitivePrin}\) of principals and a finite set \(\text{primitiveProp}\) of properties. We then define the components in our language as follows.

- The set of principals is the result of closing \(\text{primitivePrin}\) under union. (Here and elsewhere we identify a principal \(p \in \text{primitivePrin}\) 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 \(\text{Said}\) 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 \(\text{primitiveProp}\). Following XrML, we consider only properties that take a single principal as an argument. For example, \(\text{primitiveProp}\) can include the property \(\text{Employee}\), where \(\text{Employee}(x)\) means principal \(x\) is an employee, but it cannot include the property \(\text{MotherOf}\), where \(\text{MotherOf}(x, y)\) means \(x\) is the mother of \(y\), nor can it include the property \(\text{Vehicle}\), where \(\text{Vehicle}(x)\) 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-
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 \( \forall x \text{Pr}(x) \) and \( \forall y \text{Pr}(y) \) have the same semantic interpretation (all principles have property \( \text{Pr} \)), XrML treats them as distinct grants syntactically. For example, according to the XrML algorithm, if Alice is permitted to issue the grant \( \forall x \text{Pr}(x) \), then she is not necessarily permitted to issue \( \forall y \text{Pr}(y) \).

We formally define the language’s syntax according to the following grammar:

\[
\begin{align*}
\text{license} & ::= \ (\text{grant}, \text{prin}) \\
\text{grant} & ::= \ \forall \text{var} \ldots \forall \text{var}(\text{cond} \rightarrow \text{conc}) \\
\text{var} & ::= \ \text{prinVar} \mid \text{rsrcVar} \\
\text{cond} & ::= \ \text{Said}(\text{prin}, \text{conc}) \mid \text{cond} \land \text{cond} \mid \text{true} \\
\text{conc} & ::= \ \text{prop}(\text{prin}) \mid \text{Perm}(\text{prin}, \text{right}, \text{rsrcc}) \\
\text{prop} & ::= \ \text{Pr} \\
\text{prin} & ::= \ \{p\} \mid \{\text{prinVar}\} \mid \text{prin} \cup \text{prin} \\
\text{right} & ::= \ \text{issue} \\
\text{rsrcc} & ::= \ \text{grant} \mid \text{rsrcVar},
\end{align*}
\]

where \( \text{Pr} \) is a generic element of \( \text{primitiveProp} \), \( p \) is a generic element of \( \text{primitivePrin} \), and \( \text{prinVar} \) and \( \text{rsrcVar} \) are variables ranging over primitive principals 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 \( \{\text{KeyHolder}(k) \mid k \in K\} \).

We could take \( \text{primitivePrin} \) 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 \( \text{KeyHolder}(k) \).

- XrML does not have conditions of the form \( \text{Said}(p, e) \). To capture \( \text{Said} \) conditions, XrML uses \( \text{PrerequisiteRight} \) conditions of the form \( (G, e) \), where \( G \) is a set of grants and \( e \) is a conclusion. A condition \( \text{Said}(\{p_1, \ldots, p_n\}, e) \) in our syntax (where each \( p_i \) is a primitive principal or a variable of sort \( \text{Principal} \)) corresponds to the \( \text{PrerequisiteRight} \) condition \( \{\{g_1, \ldots, g_n\}, e\} \) in XrML, where each \( g_i \) is \( \forall x \text{Perm}(p_i, \text{issue}, x) \) and \( x \) is a variable of sort \( \text{Resource} \).

- XrML does not have conclusions of the form \( \text{Pr}(p) \). To capture properties, XrML uses a right called \( \text{PossessProperty} \), and considers the properties given by the application to be resources. The conclusion \( \text{Pr}(p) \) in our grammar corresponds to the conclusion \( \text{Perm}(p, \text{PossessProperty}, \text{Pr}) \) 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 confusing them.

- Instead of writing \( \text{allConds}(c_1, \ldots, c_n) \), \( \text{allConds}() \), and \( \text{AllPrincipals}(p_1, \ldots, p_n) \), we use the more standard notations \( c_1 \land \ldots \land c_n, \text{true} \), 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 \( \{(g_i, p_j) \mid i \leq n, j \leq m\}\). 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 \text{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 \text{Query} are inefficient. This is acknowledged in the XrML document, which explains that \text{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 \text{Query}. (In Section 5, we show that it is highly unlikely that such an efficient algorithm exists.)
Section 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 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 \(L\) and \(R\), suppose that the grant \(g\) is QueenOfSiam\((Alice)\), which means Alice is Queen of Siam, and the grant \(g'\) is Perm\((Alice, issue, g)\), 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 true for some \(d\) in \(D\), then Query returns true, indicating that \(L\) and \(R\) imply \(e\).

Query is summarized in Figure 1.

![Query Algorithm](image_url)

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 Perm\((p, r, s)\) implies 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\((Perm(p, r, s), \emptyset, R)\) returns

\[\{d \mid d \rightarrow Perm(p_g, r, s) \in R \text{ and } p_g \subseteq p\}\]

Suppose that there is at least one grant \(\forall x_1 \ldots \forall x_n(g_d \rightarrow e_g)\) 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 \(\sigma\) 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 \(R'\) be the closed substitution \(\{d_g \rightarrow e_g\} \subseteq R\), \(\sigma\) is a closed substitution.

When we show in Section 3.2, \(R_S\) may be infinite, in which case Query is not well-defined. For ease of exposition, we assume that \(R_S\) is finite for the rest of this section. In Section 3.3 we propose a restriction on the language that guarantees that \(R_S\) 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

\[R' = R \cup R''\]
\[R'' = \{g \mid \text{for some principal } p \text{ and condition } d, (g, p) \in L, d \in Auth(Perm(p, issue, g), L - \{(g, p)\}, R)\}
\]

and Holds\((d, L)\) returns 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\((Perm(p, issue, g), L, R)\), which returns true if Holds\((d, L)\) returns true for some d in the set returned by Auth\((Perm(p, issue, g), L, R)\). 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.
since

The formal algorithm confirms this conclusion, Amy says Bob is attractive, if the set of licenses is $L$

then

$\text{Auth}(e, L, R)$:

- 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 $\sigma$
- if $e = Pr(p)$ and $e_g\sigma = Pr(p_g)$ or
- $e = \text{Perm}(p, r, s)$ and $e_g\sigma = \text{Perm}(p_g, r, s)$,
- 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 $D = \text{Auth}(\text{Perm}(p, \text{issue}, g), L', R)$
- for each $d \in D$
- if $\text{Holds}(d, L)$ returns true
- then let $R' = R' \cup \{g\}$
- % Find $D$, the conditions under which $R'$ implies $e$
- let $D = \text{Auth}(e, \emptyset, R')$
- Return $D$

Figure 2. The Auth Algorithm

$\text{Holds}(d, L)$:

- if $d = \text{true}$ then Return true
- if $d = d_1 \land d_2$ then Return $\text{Holds}(d_1, L) \land \text{Holds}(d_2, L)$
- if $d = \text{Said}(p, e)$
- then
- % Find $R_p$
- let $R_p = \emptyset$
- let $R'_p = \{\forall x \text{Perm}(p', \text{issue}, x) \mid p' \in p\}$
- for each $(g, p') \in L$
- let $L' = L - \{(g, p')\}$
- let $D = \text{Auth}(\text{Perm}(p', \text{issue}, g), L', R'_p)$
- for each $d' \in D$
- if $\text{Holds}(d', L)$ returns true
- then let $R_p = R_p \cup \{g\}$
- % Return 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 each closed substitution $\sigma$ such that $e_g\sigma = e$
- if $\text{Holds}(d_g\sigma, L)$ returns true
- Return true
- Return false

Figure 3. The Holds Algorithm

We define $\text{Holds}(d, L)$ by induction on the structure of $d$. If $d$ is true, then $\text{Holds}(d, L)$ returns true. If $d = d_1 \land d_2$, then $\text{Holds}(d, L)$ returns true iff both $\text{Holds}(d_1, L)$ and $\text{Holds}(d_2, L)$ return true. If $d = \text{Said}(p, e)$, then $\text{Holds}$ 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,

$$R_p = \{g \mid \text{for some license } (g, p') \in L \text{ and condition } \}$$

where

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

$\text{Holds}(\text{Said}(p, e), L)$ returns 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 $\text{Holds}$ 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 $L = \{(g_1, \text{Alice}), (g_2, \text{Amy})\}$, where $g_1 = \text{Smart}(\text{Bob})$ and $g_2 = \text{Said}(\text{Alice}, \text{Smart}(\text{Bob})) \rightarrow \text{Attractive}(\text{Bob})$. The formal algorithm confirms this conclusion, since $\text{Holds}(\text{Said}(\text{Amy}, \text{Attractive}(\text{Bob})), L)$ returns true. To see this note that $\text{Holds}(\text{Said}((\text{Amy, Attractive(Bob)}), L))$ first determines $R'_{\text{Amy}}$, 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 $\text{Holds}(\text{Said}((\text{Alice, Smart(Bob)}), L))$ to determine if the condition $\text{Said}(\text{Alice, Smart(Bob)})$ holds. Intuitively, this is done because $g_2$ and $\text{Said}(\text{Alice}, \text{Smart}(\text{Bob}))$ together imply $\text{Said}(\text{Amy, Attractive(Bob)})$. $\text{Holds}(\text{Said}((\text{Alice, Smart(Bob)}), L))$ first calculates $R'_{\text{Alice}}$ to be $\{g_1\}$ and then calls $\text{Holds}(\text{true, L})$, because $g_1$ holds if we assume that the grants issued by Alice hold and that $g_1$ and true together imply $\text{Smart(Bob)}$. $\text{Holds}(\text{true, L})$ returns true, so $\text{Holds}(\text{Said}((\text{Alice, Smart(Bob)}), L))$ returns true, and then $\text{Holds}(\text{Said}(\text{Amy, Attractive(Bob)}), L)$ 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 $\text{Query}((\text{Attractive(Bob)}, L, R))$ returns true, where $R = \{\text{Perm}(\text{Amy, issue, g}_2)\}$. Specifically, $\text{Query}$ begins by calling $\text{Auth}(\text{Attractive}(\text{Bob}), L, R)$, $\text{Auth}(\text{Attractive}(\text{Bob}), L, R)$, in turn, calls $\text{Auth}(\text{Attractive}(\text{Bob}), \emptyset, R')$, where $R' = \{g_2, \text{Perm}(\text{Amy, issue, g}_2)\}$.
We are interested in returns true Query. So, Bob is attractive if the condition Said(Alice; Smart(Bob)) holds. To determine if the condition holds, Query calls Holds(Said(Alice; Smart(Bob)), L). As observed above, Holds(Said(Alice; Smart(Bob)), L) returns true, so Query(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 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, 

\[
\text{Query(Quiet(\{Alice, Betty, Bonnie\}), \emptyset, \{Quiet(Alice)\})}
\]

returns 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 

\[
\begin{align*}
g_1 &= \text{Perm}(Alice, issue, g), \\
g_2 &= d \rightarrow \text{Perm}(Charlie, issue, g), \text{ and} \\
d &= \text{Said}(Bob, \text{Perm}(Alice, issue, g)).
\end{align*}
\]

We are interested in 

\[
\text{Query(Perm(Charlie, issue, g), \emptyset, \{g_1, g_2\})}.
\]

Query begins by calling 

\[
\text{Auth(Perm(Charlie, issue, g), \emptyset, \{g_1, g_2\}),}
\]

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

Example 3.4: Let \(d\) be the condition 

\[
\text{Said(}\{Alice, Amy\}; \text{Perm}(Alice, issue, Smart(Alice)))\].

Intuitively, \(d\) holds, because Alice is permitted to issue the grant Smart(Alice), given that Amy and Alice are permitted to issue every grant. However, \text{Holds}(d, \emptyset) returns 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 \(g_1 = \text{Smart}(Alice)\) and \(g_2 = \text{Said}(Bob, Smart(Alice))\) \rightarrow \text{Perm}(Alice, issue, Smart(Alice)). Now, \text{Holds}(d, L) should return 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 \text{Said}(Bob, 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 Smart(Alice). Unfortunately, \text{Holds}(d, L) still returns false. Specifically, \text{Holds}(d, L) sets \(R_p = \{g_1, g_2\}\). It then calls \text{Holds}(\text{Said}(Bob, Smart(Alice)), L). This call returns 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 

\[
\text{Query(Trusted(Alice), \emptyset, \{\forall x(d \rightarrow \text{Trusted}(Alice))\})},
\]

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

\[
\text{Auth(Trusted(Alice), \emptyset, \{\forall x(d \rightarrow \text{Trusted}(Alice))\})},
\]

which returns \(D = \{\text{Said}(Amy, \text{Perm}(Alice, issue, g)) | g \text{ is a grant}\}\). Query then calls \text{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 $g_1 = \text{true} \rightarrow Pr(p)$ and $g_{n+1} = \text{Said}(p, Perm(p, \text{issue}, g_n)) \rightarrow Pr(p)$ 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 $e = \text{Trustworthy}(Bob)$, $L = \{(g_1, Alice), (g_2, Amy)\}$, $g_1 = \text{Said}(Amy, e) \rightarrow e$, $g_2 = \text{Said}(Alice, e) \rightarrow e$, and $R = \{\forall x \text{Perm}(Amy, \text{issue}, x), \forall x \text{Perm}(Amy, \text{issue}, x)\}$, 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 $D = \{\text{Said}(Amy, e), \text{Said}(Alice, e)\}$. Query then calls Holds on each of the conditions in $D$. During the execution of Holds$(\text{Said}(Amy, e), L)$, Holds sets $R_{Amy} = \{g_2\}$, and then calls Holds$(\text{Said}(Alice, e), L)$. Holds$(\text{Said}(Alice, e), L)$ sets $R_{Alice} = \{g_1\}$, and then calls Holds$(\text{Said}(Amy, e), L)$. Notice that this is just the original call. It is not hard to see that an infinite number of calls to Holds$(\text{Said}(Amy, e), L)$ will be made during the execution of Holds$(\text{Said}(Amy, e), L)$ 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 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 $\{\text{Trustworthy}(Bob)\}$ 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 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 do. But consider Query$(\text{Smart}(Alice), L, R)$, where $L = \{(g, Alice)\}$, $R = \{\text{Said}(Alice, \text{Smart}(Alice)) \rightarrow \text{Perm}(Alice, \text{issue}, g), g = \text{Smart}(Alice)\}$, Query$(\text{Smart}(Alice), L, R)$ begins by calling Auth$(\text{Smart}(Alice), L, R)$. 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 $p_1$. $p_1$ may issue any grant, and, thus, $p_1$’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$(\text{Perm}(Alice, \text{issue}, g_2), \emptyset, R)$ returns false, where $R = \{\forall x (\text{Said}(Amy, \text{issue}, g_2)) \rightarrow \text{Perm}(Alice, \text{issue}, g_2)\}$. 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 $R$:

$$g = \forall x_1 \ldots \forall x_3 (\text{Perm}(x_1, \text{issue}, x_2) \rightarrow \text{Perm}(x_1 \cup x_3, \text{issue}, x_2))$$

$$g_i = \forall x_1 \forall x_2 (\text{Pr}_i(x_1) \rightarrow \text{Pr}_i(x_1 \cup x_2)),$$

for $i = 1, \ldots, n$,

where $x_1, \ldots, x_3$ are variables of the appropriate sorts and $\text{Pr}_1, \ldots, \text{Pr}_n$ 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 \textit{Holds}. In addition, we add \( R' \) to \( R \) during the evaluation of \textit{Said}(\( p, e \)). This solves the problems highlighted in Example 3.4.

- We do not revise \textit{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

There are various restrictions that could accomplish this. To understand ours, recall that \( \text{Auth}(e, L, R) \) 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, \( \text{Auth} \) creates the possibly-infinite set \( R_S \) consisting of all substitution instances of grants in \( R' \), and returns \( \{ d \mid d \rightarrow e \in R_S \} \). (For simplicity here, we are assuming that \( \text{Auth} \) does not use the subset assumption; this assumption does not affect our discussion.) Since \( \text{Auth} \) considers only the grants in \( R_S \) whose conclusion matches the first input to \( \text{Auth} \), we could certainly replace \( R_S \) by \( R_S' \), where

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

Because \( e \) is closed, \( R_S' \) 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 \( \text{Resource} \) that appears in \( d_g \) also appears in \( e_g \). We call a grant \textit{acceptable} if it has this property; we call a license \((g, p)\) acceptable if \( g \) is acceptable.

Thus, for example, \( \forall x \forall y \forall z (\text{Said}(\emptyset, \text{Perm}(x, \text{issue}, y)) \rightarrow \text{Perm}(Alice, \text{issue}, y)) \) is acceptable, but

\[
\forall y \forall z (\text{Said}(\emptyset, \text{Perm}(Alice, \text{issue}, y)) \rightarrow \text{Perm}(Alice, \text{issue}, z))
\]

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 \( 2^{\lvert g \rvert \lvert \text{primitivePerm} \rvert} \) grants of the form \( d_g \sigma \rightarrow e_g \sigma \) such that \( e_g \sigma = e \). 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 \( \text{Resource} \) in \( d_g \), say \( x \), that does not appear in \( e \). If we can find a substitution \( \sigma \) 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 \( e = \text{Perm}(Alice, \text{issue}, g) \), where \( g \) is a closed grant, then the set of substitutions \( \sigma \) such that

\[
\text{Perm}(Alice, \text{issue}, z) \sigma = \text{Perm}(Alice, \text{issue}, g)
\]

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

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

We now consider the type of nontermination exhibited in Example 3.6. This behavior occurs because \textit{Query} tries to verify that a \textit{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 \( \text{Holds}(\text{Said}(Amy, e), L) \) holds by repeatedly calling \( \text{Holds}(\text{Said}(Amy, e), L) \). To correct the problem, we modify \textit{Holds} so that no call is evaluated twice. Specifically, the revised \textit{Holds} takes a fourth argument \( S \) that is the set of \textit{Said} conditions that have been the first argument to a previous call; \( \text{Holds}(d, L, R, S) \) returns false if \( d \in S \). In addition, \( \text{Holds} \) sets \( S \) to \( \emptyset \) if grants are added to \( R \), because no condition has been evaluated under the new set of assumptions. Because \textit{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 \( \text{Auth}(e, L, R) = \text{Auth}(e, \emptyset, R') \). 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 \( R \cup \{ g \mid \text{for a principal } p, (g, p) \in L \text{ and } \text{Query}(\text{Perm}(p, \text{issue}, g), L, R) \text{ returns true} \} \). However, when computing \( \text{Query}(\text{Perm}(p, \text{issue}, g), L, R) \), \textbf{Auth} is given the argument \( L - \{ (g, p) \} \) rather than \( L \).

Our solution is to do the “right” thing here, and compute \( \text{Query}(\text{Perm}(p, \text{issue}, g), L, R) \). But now we have to deal with the problem of termination, since a consequence of our change is that \( \text{Query}(e, L, R) \) terminates only if the set \( L = \emptyset \). To ensure termination, we modify \textbf{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 \textit{Holds} terminates.

Pseudocode for the modified \textit{Query2} is given in Figure 4. \textit{Query2} calls \textbf{Auth2} and \textbf{Holds2}, which are the modified versions of \textbf{Auth} and \textbf{Holds}, respectively. Pseudocode
for $\text{Auth2}$ is given in Figure 5 and pseudocode for $\text{Holds2}$ is given in Figure 6.

To summarize, the main differences between $\text{Query2}$, $\text{Auth2}$, and $\text{Holds2}$ and their analogues as defined in the XrML document are:

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

\begin{figure}[h]
\centering
\begin{minipage}{0.45\textwidth}
\begin{align*}
\text{Query2}(e, L, R, S, E): \\
&\text{let } D = \text{Auth2}(e, L, R, S, E) \\
&\text{for each } d \in D \\
&\quad \text{if } \text{Holds2}(d, L, R, S, E) \text{ returns true} \\
&\quad \text{then } \text{Return true} \\
&\text{Return false}
\end{align*}
\caption{The Query2 Algorithm}
\end{minipage}
\begin{minipage}{0.45\textwidth}
\begin{align*}
\text{Auth2}(e, L, R, S, E): \\
&\text{if } e \in E \\
&\quad \text{then } \text{Return } \emptyset \\
&\text{else} \\
&\quad \text{let } E' = E \cup \{e\} \\
&\quad \text{let } R' = R \\
&\quad \text{for each } (g, p) \in L \\
&\quad \quad \text{if } \text{Query2}(\text{Perm}(p, \text{issue}, g), L, R, S, E') \text{ returns true} \\
&\quad \quad \text{then let } R'' = R' \cup \{g\} \\
&\quad \text{let } D = \emptyset \\
&\quad \text{for each grant } \forall x_1 \ldots \forall x_n (d_g \rightarrow e_g) \in R'' \text{ and} \\
&\quad \text{closed substitution } \sigma \text{ such that } e_n \sigma = e \\
&\quad \text{let } D = D \cup \{d_g \sigma\} \\
&\text{Return } D
\end{align*}
\caption{The Auth2 Algorithm}
\end{minipage}
\end{figure}

\begin{figure}[h]
\begin{align*}
\text{Holds2}(d, L, R, S, E): \\
&\text{if } d = \text{true} \text{ then } \text{Return true} \\
&\text{if } d = d_1 \land d_2 \\
&\quad \text{then } \text{Return } \bigwedge_{i=1, 2} \text{Holds2}(d_i, L, R, S, E) \\
&\text{if } d = \text{Said}(p, e) \\
&\quad \text{then} \\
&\quad \quad \text{if } d \in S \\
&\quad \quad \quad \text{Return false} \\
&\quad \text{else} \\
&\quad \quad \quad \text{let } S' = S \cup \{d\} \\
&\quad \quad \quad \text{let } R' = R \cup \{\exists x \text{Perm}(p', \text{issue}, x) \mid p' \in p\} \\
&\quad \quad \text{if } R' = R \\
&\quad \quad \quad \text{return } \text{Query2}(e, L, R, S', E) \\
&\quad \text{else} \\
&\quad \quad \quad \% \text{Reset } S' \text{ and } E; \text{ they might follow from } R'. \\
&\quad \quad \text{return } \text{Query2}(e, L, R', \emptyset, \emptyset)
\end{align*}
\caption{The Holds2 Algorithm}
\end{figure}

We observe that $\text{Query2}$ 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 $\text{Resource}$ 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 Said conditions, and $E$ is a set of conclusions. Then $\text{Query2}(e, L, R, S, E)$ terminates.

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

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

Intuitively, \( \text{Pr}(p) \) means principal \( p \) has property \( \text{Pr} \), and \( \text{Val}(\varphi) \) means formula \( \varphi \) 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 \( \text{Val} \). We restrict attention to models for which \( \cup \) satisfies the following properties:

- \( U_1 \): \( \forall x ((x \cup x) = x) \)
- \( U_2 \): \( \forall x_1 \forall x_2 ((x_1 \cup x_2) = (x_2 \cup x_1)) \)
- \( U_3 \): \( \forall x_1 \forall x_2 \forall x_3 ((x_1 \cup (x_2 \cup x_3)) = ((x_1 \cup x_2) \cup x_3)) \)
- \( U_4 \): \( \forall x ((x \cup \emptyset) = x) \)

In addition, we are interested only in Herbrand models, where the only elements of sort \( \text{Principal, Right, and Resource} \) are interpretations of syntactic terms. We call such models acceptable models. \( \text{Val}(\varphi) \) is true in an acceptable model \( m \) if \( \varphi \) is true in all acceptable models. If a formula \( \varphi \) is true in all acceptable models, then we say \( \varphi \) is valid and write \( \models \varphi \). Thus, \( \text{Val}(\varphi) \) 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 \( \text{Said} \) 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 \( \text{Said} \) 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 \( \text{Query} \).) The translation is defined below, where \( t^{L,R,S} \) is the translation of the string \( t \) given input \( L, R, \) and \( S \).

- If \( (g, p) \in L \), \( (g, p)^{L,R,S} = \text{Perm}(p, \text{issue}, c_g) \rightarrow g^{L,R,S} \).
- If \( (g, p) \notin L \), \( (g, p)^{L,R,S} = \text{true} \).
- \( \forall x_1 . . . \forall x_n (d \rightarrow e)^{L,R,S} = \forall x_1 . . . \forall x_n (d^{L,R,S} \Rightarrow e^{L,R,S}) \).
- \( \text{true}^{L,R,S} = \text{true} \).
- If \( \text{Said}(p, e) \in S \), then \( \text{Said}(p, e)^{L,R,S} = \text{false} \).
- If \( \text{Said}(p, e) \notin S \), then \( \text{Said}(p, e)^{L,R,S} = \text{Val}(\bigwedge_{t \in L} t^{L,R,S'} \land \bigwedge_{g \in R'} g^{L,R,S'} \Rightarrow e^{L,R,S'}) \),

where \( S' = S \cup \{ \text{Said}(p, e) \} \) and \( R' = R \cup \{ \forall x \text{Perm}(p', \text{issue}, x) \mid p' \in p \} \).

- \( (d_1 \land d_2)^{L,R,S} = d_1^{L,R,S} \land d_2^{L,R,S} \).
- \( \text{Perm}(p, r, s)^{L,R,S} = \text{Perm}(p, r, s^*) \), where \( s^* = s \) if \( s \) is a variable of sort \( \text{Resource} \) and \( s^* = c_g \) if \( s \) is a grant.
- \( \text{Pr}(p)^{L,R,S} = \text{Pr}(p) \).

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, \( \text{Query2}(e, L, R, \emptyset, \emptyset) \) returns true iff \( \models \bigwedge_{t \in L} t^{L,R,\emptyset} \land \bigwedge_{g \in R} g^{L,R,\emptyset} \Rightarrow e^{L,R,\emptyset} \).

Before leaving this section, we remark that, in the translation of \( \text{Perm}(p, \text{issue}, g) \), 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 \( \text{Perm}(p, \text{issue}, g) \) as \( \text{Perm}(p, \text{issue}, f_g(x_1, \ldots, x_n)) \), 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 \texttt{Query2} returns \texttt{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 \( n \) principal principals, we can construct \( 2^n \) principals using the \( \cup \) operator.

**Theorem 5.1:** Deciding if \texttt{Query2}(\( e, L, R, \emptyset, \emptyset \)) returns \texttt{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 \texttt{Query2}(\( e, L, R, \emptyset, \emptyset \)) returns \texttt{true}. Given a graph \( G(V, E) \), where \( V = \{v_1, \ldots, v_n\} \), we take \( v_1, \ldots, v_n \) to be primitive principals. We also assume that the language has primitive properties \texttt{Node}, \texttt{Edge}, and \texttt{Path}. For each node \( v \in V \), we consider a grant \( g_v = \texttt{Node}(v) \) (recall that this is an abbreviation for \( \texttt{true} \rightarrow \texttt{Node}(v) \)); for each edge \( e = (v, v') \in E \), we consider the grant \( g_{v,v'} = \texttt{Edge}(v, v') \). (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 \( \forall x_1 \ldots \forall x_n (d_1 \land d_2 \rightarrow \texttt{Path}(x_1, \ldots, x_n)) \), where

\[
\begin{align*}
  d_1 &= \bigwedge_{1 \leq i \leq n} \texttt{Said}(\texttt{Alice}, \texttt{Node}(x_i)) \\
  d_2 &= \bigwedge_{1 \leq i \leq n-1} \texttt{Said}(\texttt{Alice}, \texttt{Edge}(\{x_i, x_{i+1}\})).
\end{align*}
\]

Let \( R = \{g_v \mid v \in V\} \cup \{g_e \mid e \in E\} \cup \{g\} \). It is not hard to show that \texttt{Query2}(\texttt{Path}(\{v_1, \ldots, v_n\}), \emptyset, R, \emptyset, \emptyset) returns \texttt{true} iff \( G \) has a Hamiltonian path. To see this, observe that \texttt{Auth2}(\texttt{Path}(\{v_1, \ldots, v_n\}), \emptyset, R, \emptyset, \emptyset) returns \( \{d_1 \sigma \land d_2 \sigma \mid \sigma(x_i) = v_n(i), i = 1, \ldots, n\} \), where \( \pi \) is some permutation of \( \{1, \ldots, n\} \).

The condition \( d_2 \sigma \) holds iff there is a path \( x_1 \sigma, \ldots, x_n \sigma \). Thus, \texttt{Query2}(\texttt{Path}(\{v_1, \ldots, v_n\}), \emptyset, R, \emptyset, \emptyset) returns \texttt{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 \texttt{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 \texttt{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 \texttt{Query2}(\( e, L, R, \emptyset, \emptyset \)) returns \texttt{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? Principal appear as the second argument in a license, the first argument in a \texttt{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 \texttt{Query} assumes that if \( (g, p) \) is a license in \( L \), then \( p \) is variable-free.)

- As for the \texttt{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 \texttt{Said}(\( \{p_1, \ldots, p_n\}, e \)), where \( p_1, \ldots, p_n \) are primitive principals, by a grant \texttt{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 \texttt{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 \texttt{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 \texttt{Trust}(\{\texttt{Alice}, \texttt{Bob}\}) and \texttt{Perm}(\{\texttt{Alice}, \texttt{Bob}, \texttt{issue}, g\}). According to the XrML document, \texttt{Trust}(\{\texttt{Alice}, \texttt{Bob}\}) means Alice and Bob together (i.e., when viewed as a single entity) is trusted; \texttt{Perm}(\{\texttt{Alice}, \texttt{Bob}, \texttt{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
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 \{Homer Simpson, Marge Simpson, . . . \}). 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] refers to a container resource that is a sequence of resources. This naturally translates to a function $\text{container}:\text{Resource} \times \text{Resource} \rightarrow \text{Resource}$, so that the container $(s_1, s_2, s_3)$ is translated as $\text{container}(s_1, \text{container}(s_2, s_3))$. 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 policies 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