An XPath-based Preference Language for P3P

Rakesh Agrawal Jerry Kiernan Ramakrishnan Srikant Yirong Xu

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120

ABSTRACT

The Platform for Privacy Preferences (P3P) is the most significant effort currently underway to enable web users to gain control over their private information. The designers of P3P simultaneously designed a preference language called APPEL to allow users to express their privacy preferences, thus enabling automatic matching of privacy preferences against P3P policies. Unfortunately subtle interactions between P3P and APPEL result in serious problems when using APPEL: Users can only directly specify what is unacceptable in a policy, not what is acceptable; simple preferences are hard to express; and writing APPEL preferences is error prone. We show that these problems follow from a fundamental design choice made by APPEL, and cannot be solved without completely redesigning the language. Therefore we explore alternatives to APPEL that can overcome these problems. In particular, we show that XPath serves quite nicely as a preference language and solves all the above problems. We identify the minimal subset of XPath that is needed, thus allowing matching programs to potentially use a smaller memory footprint. We also give an APPEL to XPath translator that shows that XPath is as expressive as APPEL.

Categories and Subject Descriptors

K.4.1 [**Public Policy Issues**]: Privacy

General Terms

Standardization, Languages

Keywords

P3P, APPEL, XPref, XPath, Preference, Privacy-Aware Data Management, Hippocratic Databases

1. INTRODUCTION

Platform for Privacy Preferences (P3P) is the most significant effort currently underway to enable web users to gain control over their private information. It provides a way for a web site to encode its data-collection and data-use practices in a machine-readable XML format, known as a P3P policy [10]. The designers of P3P simultaneously designed a preference language called APPEL [9] to allow users to express their privacy preferences. The goal was to enable users to programmatically check their privacy preferences against a P3P policy to decide whether to release their data to the web site. The P3P policy language became a W3C Recommendation in April 2002; APPEL is a Working Draft.

Unfortunately, there are subtle interactions between APPEL and the P3P policy language that make APPEL extremely hard to use *correctly*. We present a critique of APPEL that points out several shortcomings of APPEL. These shortcomings arise from a fundamental design choice: to only allow logical operations at nodes corresponding to P3P elements. These shortcomings cannot be overcome without completely redesigning the language.

We propose a small preference language that uses a strict subset of XPath 1.0 [7] for almost all of its functionality. It additionally makes use of the quantified expression *every* from the XPath 2.0 Working Draft [6]. This construct does not increase the expressive power of XPref, but makes the preferences written in XPref easier to understand and hence less error prone.[1] We refer to the proposed language as XPref.

Although XPref is a small language, it subsumes the full functionality of APPEL, while avoiding its pitfalls. In essence, XPref replaces the body of APPEL rules with XPath expressions. XPath was developed to match the structure of an XML document using a compact path notation and it was designed to be integrated as a subsystem within other systems. By reusing XPath, XPref is leveraging all the efforts that have gone into debugging its semantics and developing efficient implementations.

Although XPref is strictly based on XPath, it does not use many of the expensive features of XPath. So, while we expect many XPref implementations to use off-the-shelf XPath implementations, it is possible to build specialized implementations with much smaller memory footprint. This aspect of XPref is quite attractive for enabling preference checking in thin, mobile devices that are likely to dominate Internet access in the future.

Since XPref subsumes APPEL, it should be possible to programmatically translate APPEL into XPref. Indeed, we provide a translation algorithm that should simplify the migration from APPEL to XPref.

1.1 Paper Road Map

The rest of the paper is structured as follows. We start with a brief overview of the P3P policy language as well as the APPEL preference language in Section 2. We critique APPEL in Section 3, pointing out some of its shortcomings and why it will be hard to fix them. We present XPref in Section 4 and show how it avoids the pitfalls of APPEL. We also give an algorithm for translating APPEL rules into XPref. We discuss related work in Section 5 and conclude with a summary and directions for future work in Sections 6.

WWW2003, May 20–24, 2003, Budapest, Hungary.
ACM 1-58113-680-3/03/0005.

[1] It is straightforward to write a small preprocessor to transform an XPref preference into one that strictly uses XPath 1.0 features, or embed this translation in the XPref processor. Thus, XPref can be implemented using any standard XPath 1.0 implementation.

We assume familiarity with the basic concepts of XML [16]. Throughout the paper, we use *element* and *attribute* as in the XML specification. To distinguish XML elements appearing in an APPEL preference from those appearing in a P3P privacy policy, we will sometimes refer to an APPEL element as an *expression*. Its subelements will be correspondingly called *subexpressions*.

2. P3P POLICY AND APPEL

In this section, we briefly review the core features of the P3P policy language as well as APPEL. See [10] and [9] for complete specifications of the policy language and APPEL respectively.

2.1 P3P Policy Language

P3P policies are described in XML format as a sequence of STATEMENT elements that include the following subelements:

- PURPOSE: describes purposes for which information is collected. Multiple purposes can be listed in a STATEMENT if all of them have the same values for RECIPIENT, RETENTION and DATA-GROUPS; otherwise, they are specified in different STATEMENT elements.
- RECIPIENT: describes the intended users of the collected information. Multiple recipients can be specified in one statement.
- RETENTION: defines the duration for which the collected information will be kept.
- DATA-GROUP: provides the list of individual data items (specified using DATA tags) that are collected for stated purposes in the statement.

P3P has predefined values for PURPOSE (12 choices), RECIPIENT (6), and RETENTION (5). Examples of PURPOSE include:

- *current:* completion and support of activity for which data was provided,
- *pseudo-analysis:* inferring habits, interests, and other characteristics of individuals, but not to identify specific individuals,
- *individual-decision:* inferring habits, interests, and other characteristics of individuals, and
- *contact:* contacting visitors for marketing of services or products through a communication channel other than voice telephone.

Examples of RECIPIENT include:

- *ours:* ourselves,
- *same:* legal entities following our practices, and
- *unrelated:* legal entities whose practices are unknown to us.

Examples of RETENTION include:

- *stated-purpose:* discarded at the earliest time possible,
- *business-practice:* long term retention but with a destruction time table, and
- *indefinitely*.

P3P also has predefined types of data items. It is also possible to assign CATEGORIES to data items.

A policy can provide *opt-in* or *opt-out* values for the *required* attribute of PURPOSE and RECIPIENT elements. The opt-in value says that the user must provide explicit consent to the stated purpose/recipient. The opt-out value gives the user flexibility to reject the specified purpose/recipient, but the user needs to take additional action for the opt-out to take effect.

```
<POLICY>
  ... ...
  <STATEMENT>
    <PURPOSE><current/></PURPOSE>
    <RECIPIENT><ours/><same/></RECIPIENT>
    <RETENTION><stated-purpose/></RETENTION>
    <DATA-GROUP>
      <DATA ref="#user.name"/>
      <DATA ref="#user.home-info.postal"/>
      <DATA ref="#dynamic.miscdata">
        <CATEGORIES><purchase/></CATEGORIES>
      </DATA>
    </DATA-GROUP>
  </STATEMENT>

  <STATEMENT>
    <PURPOSE>
      <individual-decision
       required="opt-in"/>
      <contact required="opt-in"/>
    </PURPOSE>
    <RECIPIENT><ours/></RECIPIENT>
    <RETENTION>
      <business-practices/>
    </RETENTION>
    <DATA-GROUP>
      <DATA
        ref="#user.home-info.online.email"/>
      <DATA ref="#dynamic.miscdata">
        <CATEGORIES><purchase/></CATEGORIES>
      </DATA>
    </DATA-GROUP>
  </STATEMENT>
</POLICY>
```

Figure 1: Volga's Privacy Policy in P3P

An Example Policy [4] Volga is a bookseller who needs to obtain certain minimum personal information to complete a purchase transaction. This information includes name, shipping address, and credit card number. Volga also uses the purchase history of customers to offer personalized book recommendations, for which it needs customers' email address.

Figure 1 shows what Volga's policy may look like in the P3P policy language. The first STATEMENT says that the name, postal address, and miscellaneous purchase data (i.e., book titles, credit card number, etc.) will be used for completing the current purchase transaction.

The second STATEMENT allows Volga to use miscellaneous purchase data for creating personalized recommendations and email them to the customer. However, the *opt-in* value of the *required* attribute of the purposes *individual-decision* and *contact* implies that the explicit customer consent is necessary. By default, the value of the *required* attribute is set to *always*, which precludes the possibility of customer opt-in or opt-out.

2.2 APPEL

Privacy preferences are expressed in APPEL as a list of RULEs [9]. These rules are matched against a policy in the order in which they appear. A rule consists of two parts:

- *Rule behavior (Rule head):* Specifies the action to be taken if the rule fires. The behavior can be *request*, implying that the policy conforms to preferences specified in the rule body. It can be *block*, implying that the policy does not respect user's

```
<appel:RULESET>
  <appel:RULE behavior="block">
    <POLICY>
      <STATEMENT>
        <PURPOSE appel:connective="or">
          <contact/>
          <telemarketing/>
        </PURPOSE>
      </STATEMENT>
    </POLICY>
  </appel:RULE>

  <appel:RULE behavior="request"/>
    <appel:OTHERWISE/>
  </appel:RULE>
</appel:RULESET>
```

Figure 2: A preference in APPEL

preferences. See [9] for other behaviors.

- *Rule body:* Provides the pattern that is matched against a policy. The format of a pattern follows the XML structure used in specifying privacy policies described earlier.

An APPEL rule is satisfied by matching its constituent expressions and (recursively) their subexpressions. Every APPEL expression has a *connective* attribute that defines the logical operators between its subexpressions. A connective can be: *or*, *and*, *non-or* (negated or), *non-and* (negated and), *or-exact*, and *and-exact*. The default connective is *and*. The two unusual connectives are *and-exact* and *or-exact*, whose semantics are as follows:

- *and-exact:* A successful match is made if (a) all of the contained expressions can be found in the policy and (b) the policy contains only elements listed in the rule. For the *and* connective, only part (a) needs to be satisfied, not part (b).

- *or-exact:* A successful match is made if (a) one or more of the contained expressions can be found in the policy, and (b) the policy only contains elements listed in the rule. For the *or* connective, only part (a) needs to be satisfied, not part (b).

An Example Preference Figure 2 shows an APPEL rule set that implements the following preference:

> Block sites whose policies indicate that the information collected can be used for *contact* or *telemarketing*.

The rule set consists of two rules. The first rule specifies conditions under which the access to a site must be blocked. The second rule is guaranteed to fire if the first one does not since the rule body contains the *appel:otherwise* element and allows the site to be accessed.

3. A CRITIQUE OF APPEL

At first glance, APPEL comes across as an attractive language. It is small, readable, and uses standard XML for syntax. The user has to simply provide the pattern of the substructure of interest in the rule body and associate the desired behavior in the rule head. Unfortunately, the APPEL constructs interact with the P3P policy language in unintended ways, making it non-trivial to get even simple preferences right. It is easy to write a preference that appears correct and find that it does not accomplish the intended goal. The shortcomings we identify grew out of our experiences with APPEL while testing our implementation of P3P [4]. Concurrently, the implementors of the JRC APPEL engine have also pointed out some deficiencies of APPEL [12].

```
<appel:RULESET>
  <appel:RULE behavior="request">
    <POLICY>
      <STATEMENT>
        <PURPOSE appel:connective="or-exact">
          <current/><pseudo-analysis/>
        </PURPOSE>
      </STATEMENT>
    </POLICY>
  </appel:RULE>

  <appel:RULE behavior="block"/>
    <appel:OTHERWISE/>
  </appel:RULE>
</appel:RULESET>
```

Figure 3: Jack's first attempt. This preference does not block unacceptable websites because other statements in the same policy might violate the preference.

We invite the reader to join in Jack's odyssey with APPEL. Jack wants to write a very simple preference:

PREFERENCE 1. *Only the purposes current and pseudo-analysis are acceptable.*

We will share Jack's tribulations before getting it right in APPEL.

3.1 Cannot Specify What Is Acceptable

Jack thinks that APPEL has provided the or-exact connective precisely for expressing preferences like his, and writes the preference shown in Figure 3. This preference appears to permit access to only those sites whose privacy policies have statements containing strictly the *current* or *pseudo-analysis* purposes. After all, the semantics of or-exact is that if a statement lists purposes other than *current* and *pseudo-analysis*, the rule will not match.

Unfortunately the above preference does not achieve what Jack intended. The source of the problem is that the policy language allows a policy to contain multiple statements and the rule fires if any of the statements satisfies the rule. For example, a site may have a policy containing two statements; the first statement only lists *current* as the purpose, but the second statement includes *telemarketing*. Jack's preference will match the website's policy because of the first statement, and Jack has unknowingly consented to telemarketing.

3.2 Rejects Good Policies

At this point, Jack spends more time with the APPEL manual and adds the *and-exact* connective to the *policy* element in his APPEL rule, as shown in Figure 4. He thinks that *and-exact* will force all statements in the matching policy to strictly contain the purposes he wants.

Unfortunately, this modification does not work. The *policy* element in a P3P policy usually contains other subelements such as *entity* and *access*, apart from *statement*. Thus placing an and-exact connective at *policy* will cause the rule to fail for many acceptable websites whose *policy* element included other subelements, even if all the statements conformed to the rule.

3.3 Convoluted Specifications

```
<appel:RULESET>
  <appel:RULE behavior="request">
    <POLICY appel:connective="and-exact">
      <STATEMENT>
        <PURPOSE appel:connective="or-exact">
          <current/><pseudo-analysis/>
        </PURPOSE>
      </STATEMENT>
    </POLICY>
  </appel:RULE>

  <appel:RULE behavior="block"/>
    <appel:OTHERWISE/>
  </appel:RULE>
</appel:RULESET>
```

Figure 4: Jack's second attempt. This preference blocks many acceptable websites, since the *policy* element in the P3P policy usually has other subelements in addition to *statement*.

```
<appel:RULESET>
  <appel:RULE behavior="block">
    <POLICY>
      <STATEMENT>
        <PURPOSE appel:connective="or">
          <admin/><develop/><tailoring/>
          <pseudo-decision/>
          <individual-analysis/>
          <individual-decision/>
          <contact/>
          <historical/><telemarketing/>
          <other-purpose/>
        </PURPOSE>
      </STATEMENT>
    </POLICY>
  </appel:RULE>

  <appel:RULE behavior="request"/>
    <appel:OTHERWISE/>
  </appel:RULE>
</appel:RULESET>
```

Figure 5: Jack's third attempt. Convoluted specification which does not block unacceptable websites that use extensions.

Jack now starts wondering if he can solve the problem by converting his specification of what is acceptable into what is unacceptable. He notices there is a fixed number of predefined purposes in P3P. So, he rewrites his preference, as shown in Figure 5, enumerating purposes that are unacceptable. Notice that the preference has become harder to understand. As we will see momentarily, this difficulty is exacerbated when we have preferences that involve more than one dimension (e.g., involve both purpose and recipient).

3.4 Lack Of Robustness

Jack's travails are not over yet. The P3P policy language allows websites to define extensions to purpose. Jack has to guard himself against the possibility that a website may define and add, say, *telemarketing-home* as a subelement of *purpose* in its policy. This policy will match Jack's preference as written in Figure 5 and Jack has again inadvertently consented to telemarketing. The solution is to add *extension* as an additional subelement of *purpose* in Jack's preference.

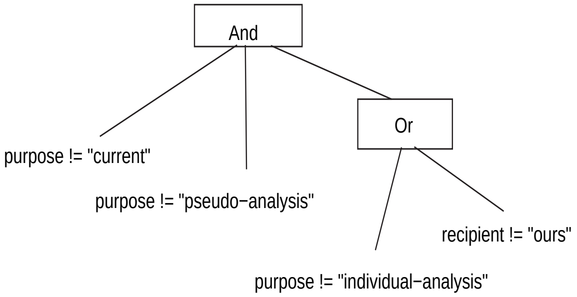
And
Or
purpose != "current"
purpose != "pseudo−analysis"
recipient != "ours"
purpose != "individual−analysis"

Figure 6: Parse tree for logical expression (after double negation) corresponding to Jack's second preference.

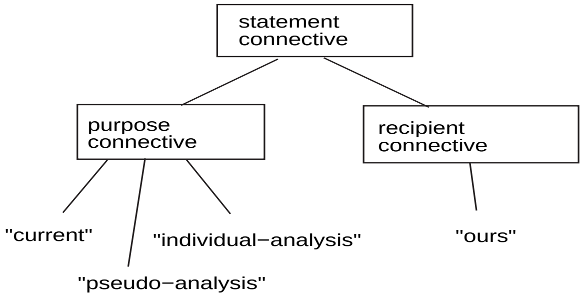

Figure 7: Fragment of XML parse tree for P3P policy: Shows logical expressions involving purpose and recipient that can be expressed in a single rule in APPEL.

3.5 Simple Combinations Are Hard To Express

Jack now decides that he does not mind websites using his personally identifiable information for individual analysis so that they may give him personalized recommendations, as long as they do not give this information to anyone else. Thus, Jack wishes to express the following preference, which is only a slightly extended version of his earlier preference.

PREFERENCE 2. *The purposes* current *and* pseudo-analysis *are acceptable. The purpose* individual-analysis *is also acceptable, as long as the recipient is* ours.

Jack has learned that he must specify preferences by writing rules about what is unacceptable. So, Jack converts

$$\forall \text{ purpose, recipient (purpose = "current"} \vee \text{ purpose = "pseudo-analysis"} \vee \text{(purpose = "individual-analysis"} \wedge \text{ recipient = "ours"))} \implies \text{accept}$$

into

$$\exists \text{ purpose, recipient (purpose} \neq \text{"current"} \wedge \text{ purpose} \neq \text{"pseudo-analysis"} \wedge \text{(purpose} \neq \text{"individual-analysis"} \vee \text{ recipient} \neq \text{"ours")} \implies \text{block}$$

Consider the parse tree shown in Figure 6 corresponding to the latter expression above. In a single APPEL rule, we can only specify those logical expressions whose parse trees are (structurally) subtrees of the XML tree corresponding to the P3P schema. For example, Figure 7 shows the class of logical expressions involving three purpose values and one recipient value that can be specified in a single APPEL rule. Clearly, there is no way to map the tree from Figure 6 into the tree in Figure 7. Therefore, this expression cannot be directly expressed in APPEL.

```
<appel:RULESET>
  <appel:RULE behavior="block">
    <POLICY>
      <STATEMENT>
        <PURPOSE appel:connective="or">
          <admin/><develop/><tailoring/>
          <pseudo-decision/>
          <individual-decision/>
          <contact/>
          <historical/><telemarketing/>
          <other-purpose/>
          <extension/>
        </PURPOSE>
      </STATEMENT>
    </POLICY>
  </appel:RULE>

  <appel:RULE behavior="block">
    <POLICY>
      <STATEMENT>
        <PURPOSE appel:connective="or">
          <individual-analysis/>
        </PURPOSE>
        <RECIPIENT appel:connective="or">
          <delivery/><same/>
          <other-recipient/>
          <unrelated/><public/>
          <extension/>
        </RECIPIENT>
      </STATEMENT>
    </POLICY>
  </appel:RULE>

  <appel:RULE behavior="request"/>
    <appel:OTHERWISE/>
  </appel:RULE>
</appel:RULESET>
```

Figure 8: APPEL rules for the preference: (purpose = current or purpose = pseudo-analysis or (purpose = individual-analysis and recipient = ours)).

Fortunately, Jack realizes that the above expression is the same as evaluating the following two rules:

> *Rule 1:* $\exists$ purpose (purpose $\neq$ "current" $\wedge$ purpose $\neq$ "pseudo-analysis" $\wedge$ purpose $\neq$ "individual-analysis") $\Longrightarrow$ block
>
> *Rule 2:* $\exists$ purpose, recipient (purpose = "individual-analysis" $\wedge$ recipient $\neq$ "ours") $\Longrightarrow$ block

He now has to further convert all the negations into enumerations to get the complex preference shown in Figure 8.[2]

3.6 Is Jack Alone?

We contend that Jack was not alone in being fooled into writing incorrect preferences; those well versed in APPEL have fallen prey to the same traps. For example, the "simple ruleset" example in Figure 3.1 of the APPEL Working Draft [9] makes several mistakes:

- The rule in lines 27–40 does not account for other statements in the same policy violating the preference (same mistake as Jack's first attempt).
- The rule in lines 17–23 does not block extensions (same mistake as Jack's third attempt).

```
<appel:RULESET>
  <appel:RULE behavior="block">
    <POLICY>
      <STATEMENT>
        <PURPOSE appel:connective="any-except">
          <current/><pseudo-analysis/>
        </PURPOSE>
      </STATEMENT>
    </POLICY>
  </appel:RULE>

  <appel:RULE behavior="request"/>
    <appel:OTHERWISE/>
  </appel:RULE>
</appel:RULESET>
```

Figure 9: Jack's first preference with a new "any-except" connective added to APPEL.

The example APPEL preferences provided by JRC P3P Resource Center [13] also do not block extensions.

It is also interesting to observe that the APPEL designers recommend the following about the ordering of rules in a ruleset (Section 5.2 of [9]):

> After starting out with all cases that are deemed acceptable (request rules), append all situations under which only limited request should be made (limited rules). The final set of rules cover all cases that should result in a blocked request (block rules).

However, as we showed with our examples, the correct way to write APPEL preferences generally is to place block rules before the request rules!

3.7 Can We Fix APPEL?

We give below two suggestions that can partially ameliorate some of APPEL's problems.

3.7.1 any-except *connective*

One idea is to add an *any-except* connective to APPEL that returns true if the P3P policy element contains any subelements except those listed in the APPEL preference.[3] Such an operator would have allowed Jack to express his first preference as shown in Figure 9.

However, for writing Preference 2, Jack would still have to transform the relatively simple double negation into a form that could actually be expressed in APPEL (due to the constraint on the operators). If Jack had even a slightly more complex preference that involved combinations of purpose, recipient, and data, the resulting preference would become very convoluted even with an *any-except* connective.

3.7.2 STATEMENTS *element*

Jack's second attempt (Figure 4) would have worked if the P3P policy language had a STATEMENTS element that was the parent

[2] We could also combine the two STATEMENTs into a single <POLICY connective="or">, and thus have only one APPEL rule. However, the combination is just as complex, and in fact the version in Figure 8 is slightly easier to read.

[3] The APPEL connectives for negation – *non-and* and *non-or* – are logical negations, not set negations, and hence their semantics are completely different from the semantics of *any-except*.

```
<appel:RULESET>
 <appel:RULE behavior="request">
   <POLICY>
     <STATEMENTS
       appel:connective="or-exact">
       <STATEMENT>
         <PURPOSE
           appel:connective="or-exact">
           <current/><pseudo-analysis/>
         </PURPOSE>
       </STATEMENT>
       <STATEMENT>
         <PURPOSE
           appel:connective="or-exact">
           <individual-analysis/>
         </PURPOSE>
         <RECIPIENT
           appel:connective="or-exact">
           <ours/>
         </RECIPIENT>
       </STATEMENT>
     </STATEMENTS>
   </POLICY>
 </appel:RULE>

 <appel:RULE behavior="block"/>
   <appel:OTHERWISE/>
 </appel:RULE>
</appel:RULESET>
```

Figure 10: The STATEMENTS tag does not solve the problem with the constraints on the logical expression parse tree: this preference is incorrect.

of all the STATEMENT elements (rather than POLICY being the parent of the STATEMENT elements). In that case, Jack would have associated the *and-exact* connective with the STATEMENTS tag instead of the POLICY tag. Thus it is natural to wonder whether modifying the P3P standard by adding a STATEMENTS tag would cure APPEL's shortcomings.

Consider Jack's second preference. We still have the problem that the expression parse tree has a structure that is not a subtree of the XML tree for the P3P policy. While the preference shown in Figure 10 appears to express this preference, it is incorrect. A P3P policy with a single statement that contains all three purposes, along with the recipient *ours*, would incorrectly be rejected by this preference. The only advantage of the STATEMENTS tag is that we can now break this preference into two rules that are easier to understand (though not easier to get to from the original preference):

> *Rule 1:* $\forall$ purpose, recipient ((purpose = "current" $\vee$ purpose = "pseudo-analysis" $\vee$ purpose = "individual-analysis") $\wedge$ recipient = "ours") $\Longrightarrow$ accept
>
> *Rule 2:* $\forall$ purpose (purpose = "current" $\vee$ purpose = "pseudo-analysis" $\Longrightarrow$ accept

3.8 The Fundamental Limitation Of APPEL

As we discussed in Section 3.5, an APPEL rule can be viewed as a tree rooted at the RULE node. APPEL operators (connectives) can only be placed at the nodes. The fundamental limitation of APPEL arises from the fact that the children of a node can only be combined using the logical connective specified at the node. It is not possible to combine them in any other logical way, except to break them in multiple rules (as we did in Figure 8). APPEL also chose not to include logical operators for combining multiple rules in a ruleset; they are evaluated strictly in order. Taken together, these two limitations box in APPEL.

We are, therefore, not sanguine that band-aiding will fix the problems of APPEL. Fortunately, there is a better way out. We can design an XPath-based language that incorporates the lessons and aspirations of APPEL, subsumes APPEL's functionality, and leverages all the past effort in debugging the semantics of XPath and developing efficient implementations. We discuss this language, which we call XPref, next.

4. XPref

We begin by giving the rationale for the design of XPref in Section 4.1. We give a summary of XPref in Section 4.2, and then give some examples to illustrate its functionality in Section 4.3. We show that XPref subsumes the functionality of APPEL in Section 4.4.

We assume familiarity with XPath [7]. We include a brief overview of XPath in Appendix A for quick reference. Appendix B describes the subset of XPath 1.0 used in XPref. Appendix C gives a complete BNF specification of XPref.

4.1 Design Rationale

The starting point of any language design is the articulation of what the language is supposed to accomplish and what it should be able to express in a natural way. Since the APPEL designers incorporated feedback from potential users when designing APPEL, we treat the goals of APPEL (not the actual implementation) as the desiderata for XPref.

APPEL was designed such that each rule accomplishes one of the following goals:

1. *Specify-Unacceptable:* Identify some combination of P3P policy elements (e.g., purpose and recipient) in the policy which is unacceptable (and then block).
2. *Specify-Acceptable:* Verify that all combinations of P3P elements are acceptable (and then request).
3. *Catch-All:* Provide a "catch-all" placeholder that fires if previous rules fail to fire.

Unfortunately, the implementation of APPEL allows specify-unacceptable rules, but not specify-acceptable rules. In addition, even specify-unacceptable rules are constrained in terms of the logical expressions that could be provided in a single rule. The main design goal of XPref was to remove both these deficiencies.

We decided to base our design on XPath for the following reasons. XPath has been designed to be integrated as a subsystem within other systems. It provides a common syntax and semantics shared across multiple systems and is used to match the structure of an XML document against a compact path notation used for navigating through the hierarchical structure of an XML document. Since P3P policies are XML documents, XPath is a natural contender for expressing the rule conditions that can be pattern matched against policies. In addition, by using XPath, we automatically accrue all the benefits of using a mature standard.

XPath can both identify combinations of P3P elements, and verify that only specified combinations of P3P elements are present. Thus we can use XPath to define both specify-unacceptable and specify-acceptable rules, and express arbitrary logical expressions.

Having replaced APPEL rule bodies with XPath expressions, we retained the APPEL constructs for specifying rule behavior and rulesets. These constructs are specified in an XML format and naturally integrate with the rule bodies written in XPath. We also retained the APPEL semantics of evaluating rules in the order in which they appear in a ruleset.

4.2 Overview of XPref

XPref retains the two outermost XML elements in APPEL: RULESET and RULE. However, rather than using the subelements of RULE to specify acceptable or unacceptable combinations of P3P elements, XPref uses an XPath expression for this purpose. This expression is specified in a new *condition* attribute of the rule. A rule fires if the XPath expression returns a non-empty result. The other attributes of RULE, such as explanation, are retained as such from the APPEL specification.

XPref uses a strict subset of XPath 1.0 [7] for writing rule conditions, with one exception: the quantified expression *every* is borrowed from XPath 2.0 [6]. Having a small subset admits the possibility of smaller footprint implementations of XPref. A smaller footprint is important for preference checking in the mobile clients that are likely to dominate Internet access in the future. Appendix B specifies this subset and also explains our choices.

The *every* expression does not increase the expressive power of XPref, but we chose to include it to simplify preferences that require exact selection of a combination of policy elements. It is straightforward to write a small preprocessor to translate an XPref preference into one that strictly uses XPath 1.0 features, or embed this translation in the XPref processor. Thus, XPref can be implemented using any standard XPath 1.0 implementation.

4.3 Illustrative Examples

4.3.1 Specifying what is unacceptable

A preference that blocks if the P3P policy includes *contact* or *telemarketing* purposes can be written in XPref as follows[4]:

```
<RULESET>
  <RULE behavior="block"
     condition="/POLICY/STATEMENT/PURPOSE/*
       [name(.) = "contact" or
        name(.) = "telemarketing"]" />

  <RULE behavior="request" condition="true"/>
</RULESET>
```

The rule's condition starts at the root node and descends to the POLICY node following the child axis from the root. It then descends further to the STATEMENT node, and then the PURPOSE node. Next, the "*" operator selects all children elements of the context (PURPOSE) node. We then evaluate a predicate – denoted using square brackets – over each selected element. This predicate tests whether the name of the selected element is either *contact* or *telemarketing*. If the predicate is satisfied, the corresponding element is returned. Thus the output of the XPath expression is the set of elements in the P3P policy that are children of some PURPOSE element, and whose name is either *contact* or *telemarketing*. If this set contains at least one element, the rule will fire and return *block*. Otherwise, the evaluator proceeds to the next rule, and returns *request*.

If we wish to modify the above preference not to block websites that require *opt-in* for *contact* and *telemarketing*, we would rewrite the rule as follows:

```
<RULESET>
  <RULE behavior="block"
     condition="/POLICY/STATEMENT/PURPOSE/*
       [(name(.) = "contact" or
         name(.) = "telemarketing") and
         @required != "opt-in"]" />

  <RULE behavior="request" condition="true"/>
</RULESET>
```

We now give an example involving two different elements: purpose and recipient. The following preference blocks all policies where the purpose is *individual-analysis* and the recipient is not *ours* in any of the statements in a policy:

```
<RULESET>
  <RULE behavior="block"
     condition="/POLICY/STATEMENT [
       PURPOSE/* [
         name(.) = "individual-analysis"]
       and RECIPIENT/* [
         name(.) != "ours"]
       ]" />

  <RULE behavior="request" condition="true"/>
</RULESET>
```

4.3.2 Specifying what is acceptable

We now illustrate that the *every* expression makes it easy to express the acceptable combinations in a preference. The following preference ensures that all purposes across all statements are strictly *current* or *pseudo-analysis*.

```
<RULESET>
  <RULE behavior="request"
     condition="/POLICY [
       every $pname in
       STATEMENT/PURPOSE/* satisfies
         (name($pname) = "current" or
          name($pname) = "pseudo-analysis")
       ]" />

  <RULE behavior="block" condition="true"/>
</RULESET>
```

4.3.3 Combinations

Preference 2, which gave Jack so much trouble in Section 3.5, can be expressed using the *every* expression, as shown in Figure 11.

This preference can also be written using only XPath 1.0 as follows, though the preference writer now has to think in terms of what is unacceptable, rather than what is acceptable.

```
<RULESET>
  <RULE behavior="block"
     condition="/POLICY/STATEMENT/PURPOSE/*
       [name(.) != "current" and
        name(.) != "pseudo-analysis" and
        (name(.) != "individual-analysis"
         or ../../RECIPIENT/*
            [name(.) != "ours"])
       ]" />

  <RULE behavior="request" condition="true"/>
</RULESET>
```

[4] The quoted strings in the condition should be escaped since they are themselves contained in quoted strings. However, for clarity of exposition, we don't show the escape sequences in the examples.

```
<RULESET>
  <RULE behavior="request"
     condition="/POLICY [
       every $stmt in STATEMENT satisfies (
         every $purpose in $stmt/PURPOSE/*, every $recip in $stmt/RECIPIENT/* satisfies
           (name($purpose) = "current" or
            name($purpose) = "pseudo-analysis" or
            (name($purpose) = "individual-analysis" and name($recip) = "ours")) )]" />

  <RULE behavior="block" condition="true"/>
</RULESET>
```

Figure 11: Preference 2 in XPref

4.4 Translating APPEL into XPref

Figure 12 gives an algorithm for translating the body of an APPEL rule into an XPref condition. In addition to showing that XPref subsumes the functionality of APPEL, this algorithm can also be used for converting existing APPEL preferences into XPref. To simplify exposition, the algorithm pseudocode omits checks for avoiding generating superfluous parenthesis as well as unneeded trailing OR or AND operators in the query.

An APPEL expression is satisfied by matching its attributes and the constituent subexpressions which are connected through the APPEL logical operators. The algorithm for translating a rule body into XPref works as follows:

- The match() function generates the XPref code for matching an APPEL expression. It first matches any attributes specified in the APPEL expression (lines 6–9). Next, it recursively matches any subexpressions (lines 10), with the appropriate connective, by calling the matchSubexpressions() function. Finally, it combines the conditions for the attributes and the subexpressions and associates them with the current P3P element (line 11).
- The matchSubexpressions() function recursively matches each subexpression (line 17). It then negates the expression if the connective is *non-and* or *non-or* (lines 18–19). If the connective is *and-exact* or *or-exact*, it calls the noOther() function (lines 20–21) to ensure that the P3P element does not contain any additional subelements.
- The noOther() function ensures that each P3P subelement matches at least one of the APPEL subexpressions.
- The main() function calls matchSubexpressions() giving the rule body as the argument and associates the resulting condition with the root P3P element (line 3).

Example The rule body in Figure 13 will be converted by the translation algorithm as:

```
/self::node() [POLICY [STATEMENT [PURPOSE [
     (contact [@required="always"]
      OR telemarketing)
     ]]]]
```

If the connective in line 4 was *and-exact* instead of *or*, the algorithm will instead generate:

```
/self::node() [POLICY [STATEMENT [PURPOSE [
    ((contact [@required="always"]
      AND telemarketing)
     AND every $s in ./* satisfies
       ($s/self::contact [@required="always"]
        OR $s/self::telemarketing))
     ]]]]
```

```
1   String main (RuleBody $r$) {
2       String xpSub = matchSubexpressions($r$);
3       return “/self::node() [” + xpSub + “]”;
4   }

5   String match (Expression $e$) {
        // match attributes of $e$
6       String xpAttr;
7       for each attribute attr of $e$ do
8           xpAttr += “@” + attr.name() + “=" ” + attr.value() + “"”;
9           xpAttr += “AND”;

        // match subexpressions of $e$
10      String xpSub = matchSubexpressions($e$);

11      return $e$.name() + “[” + xpAttr + “AND (” + xpSub + “)]”;
12  }

13  String matchSubexpressions (Expression $e$) {
14      String xpSub;
15      let $\theta$ = $e$.connective();
        // $\theta$.orAnd() returns the “or” or “and” part of $\theta$
16      for each subexpression $s$ of $e$ do
17          xpSub += match($s$) + $\theta$.orAnd();

18      if $\theta$ is “non-or” or “non-and” then
19          xpSub = “NOT (” + xpSub + “)”;
20      else if $\theta$ is “or-exact” or “and-exact” then
21          xpSub = “(” + xpSub + “) AND” + noOther($e$);

22      return xpSub;
23  }

24  String noOther (Expression $e$) {
25      String xp = “every $s in ./* satisfies (”;
26      for each subexpression $s$ of $e$ do
27          xp += “$s/self::” + match($s$) + “OR”;
29      return xp + “)”;
30  }
```

Figure 12: Algorithm for translating an APPEL rule body into an XPref condition.

```
<appel:RULE behavior="block">
  <POLICY>
    <STATEMENT>
      <PURPOSE appel:connective="or">
        <contact required="always"/>
        <telemarketing/>
      </PURPOSE>
    </STATEMENT>
  </POLICY>
</appel:RULE>
```

Figure 13: Example APPEL Preference

5. RELATED WORK

While developing APPEL, the APPEL working group explored the possibility of using an existing database query language rather than designing a new language. They decided against this alternative because of the large implementation burden on user agent implementors and the mismatch for XML processing [8]. However, they did speculate that the XML XQuery work may eventually result in a language suitable for encoding and processing P3P preferences. The APPEL designers also considered the possibility of using database query languages [8]. Our work completes this thought by proposing XPref based on XPath and showing that it serves nicely as a preference language for P3P.

Independent of our work, it has been noted in [12] that constructing preferences in APPEL is complex and error prone. They speculate that XPath might be an alternative.

There have been efforts to develop graphical tools to simplify preference creation. For example, the JRC APPEL Preference Editor [13] provides a Java-based GUI for preparing APPEL preferences. For developers of these tools, a symbolic language that is small and not error prone (such as XPref) would help.

Current APPEL implementations include AT&T Privacy Bird [15], which implements an APPEL engine in a browser extension to Internet Explorer 5. There is also a Java-based APPEL implementation available from JRC [13]. A server-centric architecture for matching preferences against policies is presented in [4]. The server-centric approach lays the groundwork for enforcing privacy policies at the database level. XPref is agnostic to specific implementation architectures and can be used in client as well as server centric architectures.

The context for the work presented in this paper is our effort to design information systems that protect the privacy and ownership of individual information while not impeding the flow of information. Our other related papers include [1, 2, 3, 5, 11].

6. CONCLUSION

The raison d'être of P3P is to provide automated matching of user preferences with privacy policies; thus its success depends on the availability of an associated preference language. While APPEL was designed for that purpose, subtle interactions between APPEL and P3P lead to serious problems:

- Users cannot directly specify what is acceptable in a policy, only what is unacceptable. The resulting specifications are convoluted and verbose.
- Simple preferences are surprisingly hard to express.
- Writing APPEL preferences is error prone.

These shortcomings are symptoms of a fundamental problem with the design of APPEL: only allowing logical operations at nodes corresponding to P3P elements. This design choice cannot be changed without a complete redesign of the language.

Therefore we explored alternatives to APPEL, and in particular, showed that XPath serves quite nicely as a preference language. It solves all the problems mentioned above. We identified the minimal subset of XPath that is needed, thus allowing a matching engine to potentially use a smaller memory footprint. We also gave an XPref to APPEL translator, showing that XPath is as expressive as APPEL.

We conclude by mentioning two interesting future directions for extending XPref. First, how do we specify a preference such that different websites can be automatically ranked with respect to their privacy friendliness? Second, there is recent work on quantifying the value of privacy by formulating the problem as a coalitional game [14]. How can we extend the preference language to incorporate such negotiations?

7. REFERENCES

[1] R. Agrawal, A. Evfimievski, and R. Srikant. Information sharing across private databases. In *Proc. of the 2003 ACM SIGMOD Int'l Conf. on Management of Data*, San Diego, CA, June 2003.

[2] R. Agrawal and J. Kiernan. Watermarking relational databases. In *Proc. of the 28th Int'l Conference on Very Large Databases*, Hong Kong, China, August 2002.

[3] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *Proc. of the 28th Int'l Conference on Very Large Databases*, Hong Kong, China, August 2002.

[4] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Implementing P3P using database technology. In *Proc. of the 19th Int'l Conference on Data Engineering*, Bangalore, India, March 2003.

[5] R. Agrawal and R. Srikant. Privacy preserving data mining. In *ACM SIGMOD Conference on Management of Data*, pages 439–450, Dallas, Texas, May 2000.

[6] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simeon, editors. *XML Path Language (XPath) 2.0*. W3C Working Draft, August 2002.

[7] J. Clark and S. DeRose, editors. *XML Path Language (XPath) Version 1.0*. W3C Recommendation, November 1999.

[8] L. Cranor. *Web Privacy with P3P*. O'Reilly&Associates, September 2002.

[9] L. Cranor, M. Langheinrich, and M. Marchiori. *A P3P Preference Exchange Language 1.0 (APPEL1.0)*. W3C Working Draft, April 2002.

[10] L. Cranor, M. Langheinrich, M. Marchiori, M. Presler-Marshall, and J. Reagle. *The Platform for Privacy Preferences 1.0 (P3P1.0) Specification*. W3C Recommendation, April 2002.

[11] A. Evfimievski, R. Srikant, R. Agrawal, and J. Gehrke. Privacy preserving mining of association rules. In *Proc. of the 8th ACM SIGKDD Int'l Conference on Knowledge Discovery and Data Mining*, Edmonton, Canada, July 2002.

[12] G. Hogben. A technical analysis of problems with P3P 1.0 and possible solutions. In *Position paper, W3C Workshop on the Future of P3P*, Dulles, Virginia USA, November 2002.

[13] JRC P3P Resource Centre. http://p3p.jrc.it.

[14] J. Kleinberg, C. H. Papadimitriou, and P. Raghavan. On the value of private information. In *Proc. 8th Conf. on Theoretical Aspects of Rationality and Knowledge (TARK)*, 2001.

[15] AT&T privacy bird. http://privacybird.com.

[16] The World Wide Web Consortium. *Extensible Markup Language (XML)*. http://www.w3.org/XML.

APPENDIX

A. OVERVIEW OF XPATH

XPath was designed to match the structure of an XML document against a compact path notation used for navigating through the hierarchical structure of an XML document.

An XPath path expression, when applied to a document, returns a sequence of distinct nodes in document order. There are seven different types of nodes in an XML document: root nodes, element nodes, text nodes, attribute nodes, namespace nodes, processing instruction nodes, and comment nodes. A path is made up of steps, each step is an expression that returns nodes. Each step navigates the document from the context node along an axis defined by the step. The context node is an implicit variable used to represent a node in the XML document while processing an XPath.

XPath axes are used to navigate the structure of an XML document in all directions. Given a context node n which is an element node, the child axis, for example, navigates towards the subelements or attributes of n, while the parent axis moves in the direction of the immediate element node that contains n. By default, each step is along the child axis. Thus, the path /POLICY/STATEMENT returns a sequence of distinct STATEMENT nodes in document order. It retrieves, starting from the root node of the document, element nodes having element name POLICY, and passed each node n thus found onto the next step in the path. For the following step, n is called the context node for evaluating the step. The STATEMENT node selects subelements of POLICY which have the element name STATEMENT and returns these as the result of the path.

Predicates can also be specified at each step in the path in order to restrict the set of nodes derived at a step. Predicates are specified using square brackets and appear after the step. If a predicate evaluates to a numeric value instead of a boolean value, the result is converted to true if the number is equal to the context position. For example /POLICY/STATEMENT[3] selects only the third statement in a policy. This is equivalent to the boolean predicate /POLICY/STATEMENT[position() = 3] which uses the position function that returns the context position.

Attribute nodes are accessed using the abbreviated syntax “@name”. For example, current/@require selects the *required* attribute of the *current* element. The @ prefix differentiates an attribute nodes from other types of nodes.

B. CHOICE OF SUBSET OF XPATH 1.0

The following is a summary of the features of XPath 1.0 included in XPref and the rationale for our choices:

- **Axes**: XPath defines 13 axes that can be used to traverse the nested structure of an XML document. In XPref, the axes required to specify preferences are limited to: child, parent, attribute, self.

  We have chosen to include a limited number of axes since the schema of P3P XML policies is fixed and known; axes such as *descendant* and *descendant-or-self* (represented with // when using abbreviated XPath syntax) allow finding an element within the nested structure of an XML document without knowing the full path to that element. For example, the XPath //name(.) = “telemarketing” tests for the presence of an element node called *telemarketing* contained in a P3P policy. While this adds some convenience, the task of evaluating such a predicate would incur a performance penalty since the predicate would have to be evaluated for all nodes in the XML tree, instead of simply at those nodes that are children of a PURPOSE node.

- **Predicates**: Boolean predicates are needed but positional predicates are not necessary.

  Our motivation for excluding positional predicates is that privacy preferences address whether a policy contains specific sets of elements, irrespective of their relative position in the XML document.

- **Expressions**: Relational operators $\{<, >, \leq, \geq\}$ are not required. Similarly, arithmetic operators can be omitted.

  Our motivation for excluding these operators is similar to our motivation for excluding positional predicates; preferences address whether a policy contains specific sets of elements.

- **Functions**: Only a limited number of functions are needed. Math functions such as *ceiling*, *round* are not required.

  We kept XPath string functions that provide services similar to the wildcard “*” operator found in APPEL, which is used, for example, when matching P3P data types in preferences.

Altogether, our reduced XPath BNF has 29 productions (26 productions are taken from XPath 1.0 and 3 productions are taken from XPath 2.0) while XPath 1.0 has a total of 39 productions. XPath 1.0 has a set of 27 core functions, and we retain only eight of them. More importantly, we have omitted some of the more complex constructs of XPath 1.0. These simplifications should result in a smaller footprint implementation when compared to a full XPath implementation.

C. BNF SPECIFICATION OF XPref

XPref replaces APPEL’s rule body with XPath specifications, but retains the portion of APPEL used for specifying the behavior of the rules and the ruleset. It uses a subset of XPath 1.0 functionality and functions, and additionally borrows the *every* expression from XPath 2.0. Accordingly, we give the BNF for XPref in four pieces:

1. Figure 14 gives the BNF for the subset of XPATH 1.0 used in XPref.
2. Figure 15 gives the XPath 1.0 functions included in XPref.
3. Figure 16 gives the subset of XPath 2.0 borrowed in XPref.
4. Figure 17 gives the portion of APPEL retained in XPref.

```
LocationPath ::= RelativeLocationPath
                 | AbsoluteLocationPath
AbsoluteLocationPath ::= '/' RelativeLocationPath?
RelativeLocationPath ::= Step
                 | RelativeLocationPath '/' Step
Step ::= AxisSpecifier NodeTest Predicate*
         | AbbreviatedStep
AxisSpecifier ::= AxisName '::'
                 | AbbreviatedAxisSpecifier
AxisName ::= 'attribute' | 'child' | 'parent' | 'self'
NodeTest ::= NameTest | 'node' '(' ')'
Predicate ::= '[' PredicateExpr ']'
PredicateExpr ::= Expr
AbbreviatedStep ::= '.' | '..'
AbbreviatedAxisSpecifier ::= '@'?
Expr ::= OrExpr
OrExpr ::= AndExpr | OrExpr 'or' AndExpr
AndExpr ::= EqualityExpr | AndExpr 'and' EqualityExpr
EqualityExpr ::= PathExpr
                 | EqualityExpr '=' PathExpr
                 | EqualityExpr '!=' PathExpr
PathExpr ::= LocationPath
             | FilterExpr
             | FilterExpr '/' RelativeLocationPath
FilterExpr ::= PrimaryExpr | FilterExpr Predicate
PrimaryExpr ::= VariableReference | '(' Expr ')'
                | Literal | Number | FunctionCall
FunctionCall ::=
                FunctionName '(' ( Argument ( ',' Argument )* )? ')'
Argument ::= Expr
Literal ::= ' " ' [ˆ "]* ' " ' | " ' " [ˆ ']* " ' "
Number ::= Digits ('.' Digits?)? | '.' Digits
Digits ::= [0-9]+
FunctionName ::= QName
VariableReference ::= '$' QName
NameTest ::= '*' | NCName ':' '*' | QName
```

Figure 14: Subset of XPath 1.0 used in XPref

```
Core Functions:
          string local-name(node-set?)
          string name(node-set?)

String Functions:
          boolean starts-with(string, string)
          boolean contains(string, string)
          string substring(string, number, number?)

Boolean Functions:
          boolean not(boolean)
          boolean true()
          boolean false()
```

Figure 15: Subset of XPath functions included in XPref

```
AndExpr ::= EqualityExprOrQuantifiedExpr
            | AndExpr 'and' EqualityExprOrQuantifiedExpr

EqualityExprOrQuantifiedExpr ::= EqualityExpr | QuantifiedExpr

QuantifiedExpr ::=
    ('every' VariableReference 'in' Expr
    (',' VariableReference 'in' Expr)* 'satisfies')* Expr
```

Figure 16: Quantified expression from XPath 2.0 used in XPref

```
ruleset ::= '<appel:RULESET
              xmlns:appel="http://www.w3.org/2002/04/APPELv2" '
              common-attributes '>'
              rseq
              '</appel:RULESET>'
rseq ::= 1*rule
rule ::= '<appel:RULE behavior="' behavior '"'
           common-attributes
           rule-attributes '>'
           [request-group]
           '</appel:RULE>'
common-attributes ::= [' crtdby=' quoted-string]
                      [' crtdon="' datetime '"']
                      [' description=' quoted-string]
rule-attributes ::= [' prompt ="' ('yes'—'no') '"']
                    [' persona=' quoted-string]
                    [' promptmsg=' quoted-string]
                    [' condition=' quoted-string]
behavior ::= 'request' | 'block' | 'limited'
request-group ::= '<appel:REQUEST-GROUP>'
                  1*request
                  '</appel:REQUEST-GROUP>'
request ::= '<appel:REQUEST uri="' [URI] per RFC 2396'">'
quoted-string ::= ' " ' string ' " '
string ::= <[UTF-8] string (with " and & escaped)>
datetime ::= <date/time as per ISO8601 or sec. 3.3.1 in RFC2616>
```

Figure 17: Portion of APPEL used in XPref