A Reasoning Framework for Heterogeneous XML

Yuh-Pyng Shieh, Chung-Chen Chen, and Jieh Hsiang
Department of Computer Science and Information Engineering,
National Taiwan University, Taipei, Taiwan.
e-mail: {arping, johnson}@turing.csie.ntu.edu.tw, hsiang@csie.ntu.edu.tw

***Abstract*—XML is designed to structure data for exchange. It is also a new trend to use XML to encode knowledge on the Web. An important project for this purpose is the Semantic Web of W3C. They proposed several specifications, including RDF/RDFS and OWL, for knowledge representation. In the architecture of SW, they proposed XML for the syntax layer, RDF/RDFS for the semantics layer, and OWL for the ontology layer. However, languages for the logic layer and proof layer are not yet defined.**

We propose a different approach for knowledge representation and reasoning in this paper. We define a logic-based framework to transform XML documents into logical facts, and to reason about these facts. The transforming and reasoning processes are based on a logic programming language, *Path Inference Language* (PIL), which is specifically designed for tree-structure documents like XML. People may write a PIL program to extract logical facts from XML documents, and the extracted logical facts are imported into a logic-based ontology in PIL for reasoning. Based on PIL, we intend to develop a simple and powerful framework for people to interpret the semantics of XML easily.

***Index Terms*—Automated Reasoning, Logic Programming, Ontology.**

I. INTRODUCTION

After the first release of XML announced by W3C in 1998, thousands of XML specifications have been developed to encode data into XML format. It is easy for people to define their own metadata as their needs using XML. However, the interpretation XML has become more and more difficult as the number of metadata grows. This is because XML focuses on the syntax of data instead of the semantics. Several problems may occur if one wants to deal with several different types at once, as the following simple example illustrates.

```
<private security= “L112481632”>
   <name> Yuh-Pyng Shieh </name>
   <father> Ye Fu Shieh </father>
</private>
```

```
<客戶資料>
   <姓名>謝育平</姓名>
   <電話>02-28825252</電話>
</客戶資料>
```

```
<person sex= “male”>
   <security>L112481632</security>
   <name>
      <first_name>Yuh-Pyng </first_name>
       <last_name> Shieh </last_name>
   </name>
   <father> L186619019</father>
   <brother> An Shieh </brother>
</person>
```

Figure 1. Heterogeneous XML

The three XML documents above are of three different types, and describe information about the same person. When we want to handle them all at once, we may encounter the following problems.

1. Different tag names: The tags <private>, <person>, and <客戶資料> are all about personal data, but have different names.
2. Different locations: In <private> the ID number is recorded as an attribute of <private>, but in <person> the ID number is recorded in a sub-tag.
3. Different languages: Yuh-Pyng Shieh and 謝育平 are the names of the same person (the first author), but is in English and the other in Chinese.
4. Different structures: The tags <name> in <private> and <name> in <person> have different structure. The latter splits a name to first and last.
5. Different semantics: The meaning of <father> in <private> and <father> in <person> are very different. The former is the name of the person's father, while the latter is the father's ID number.

The last problem mentioned above the most serious. A simple

Research supported in part by Grant NSC 91-2213-E-002-039 and NSC 91-2811-E-002-022 of the National Science Council of the Republic of China.

minded solution is to require that all tags have the same meaning. A more reasonable one is to design a mechanism to transform documents of one type into other types.

Several projects were initiated to unify XML tags using ontology. The most well known is the Semantic Web project [1,2] of W3C. The SW working group of W3C proposed a series of languages including XML, XPATH, XSLT, RDF, RDFS and OWL. XML is used to encode data on the Web in a standard markup format. XPATH is to locate nodes in XML documents. XSL is for showing the same XML document in different presentational styles. RDF is used to encode XML documents in standard object-oriented format for machine to read and write. RDFS is used to encode the meaning of XML tags. And OWL is for representing ontology in XML format. More specifically, RDF contains a set of standard tags used to describe data as objects, and RDFS contains a set of tags to describe the meanings of user-defined tags. OWL is a language evolved from OIL and DAML, and contains a set of tags such as "subClassOf" and "onProperty" to define the simple set-theoretic semantics of the ontology. Those who want to understand the relationship among these languages can read the tutorial paper of Decker et al [3].

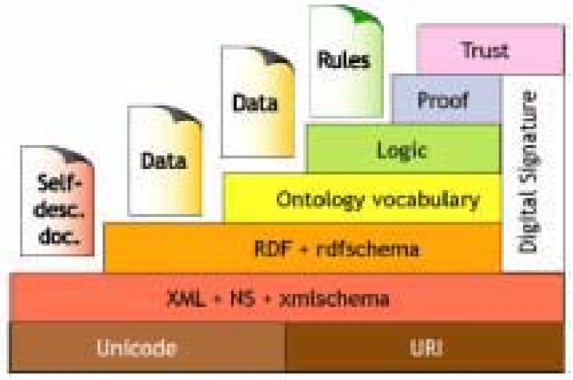


Figure 2. The specification stack of SW [7]

Figure 2 shows the specification stack of the Semantic Web project. In SW, XML is a language for the syntax layer; RDF/RDFS are languages for the semantic layer; and DAML, OIL and OWL are for the ontology layer. Languages for the logic and proof layers are not yet defined. The languages proposed by SW so far have adopted the object-oriented approach that encodes objects in XML.

Another possible way to represent ontology is the logic-based approach. The solid foundation of mathematical logic had attracted people to design programming languages based on logic. One of the most notable is Prolog [6]. Logic-based languages such as Prolog offer a number of advantages. First, it can treat a program and the data that it wants to operate on in a uniform way. Second, Prolog adopts resolution as its execution mechanism. Thus, executing a program and reasoning about a program can be done in the same environment. Third, logic is a very convenient way to specify relations, which is exactly the purpose of ontologies.

For these reasons, we propose a logic-based programming language, the *Path Inference language* (PIL), for reasoning about Web documents. The same language is used for specifying the semantics of XML, describing ontologies, and for reasoning. In other words, in our framework we simplified the SW specification stack into just two layers, as shown in Figure 3.

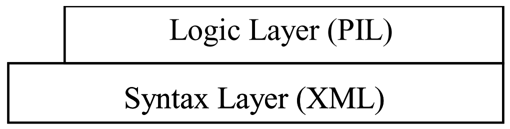


Figure 3. The specification stack of our framework

The scenario goes as follows: instead of using heavy languages such as RDF/S to define relations between tags, we use PIL to write *extractors* (PIL programs) to extract elements from XML documents. The extract elements are logical facts (also PIL programs). The logical facts can then be exported to the world of ontologies for reasoning. But since ontologies are also described as PIL programs, the execution mechanism of PIL can be used for reasoning as well.

This paper is organized as follows. Section 2 shows an overview for our logic-based framework to transform XML into the logic world, including the extraction process and the reasoning process. The framework is based on PIL to extract logical facts from XML documents and to reason using these facts. Section 3 shows the syntax, denotational and operational semantics of PIL. Section 4 shows how to extract logical facts from XML documents by PIL. Section 5 shows how to represent ontology and reason with PIL. We summarize our approach and present a comparison with SW in section 6.

II. Overview

Our logic-based framework is intended to integrate heterogeneous XML documents into a unified semantics structure. Our approach is to transform heterogeneous XML documents into logical facts with uniform semantics for reasoning.

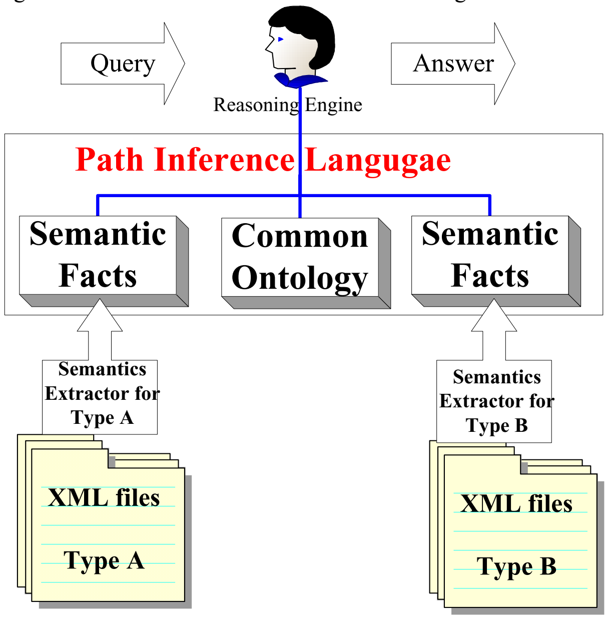


Figure 4. The architecture of our logic-based framework

Figure 4 shows the architecture of our proposed framework.

The framework contains the following parts, a set of XML documents of several different types, a set of semantic extractors in PIL, a common ontology in PIL, and a reasoning engine. In the architecture of Figure 4, logical facts are extracted from XML documents via a semantic extractor, and then a reasoning process based on the extracted logical facts and the common ontology is used to answer questions by reasoning.

In order to illustrate our approach, the following XML document about the customer "Peter" is used as an example.

```
<customer name="Peter">
  <tel>886-2-23445267</tel>
</customer>
```

A semantic extractor contains a set of extraction rules. An extraction rule is a rule in the following form.

```
Matching Part => Transformation Part
```

The matching part and transformation part here are both logical expressions with the extension of operator ".". The operator "." is an object-oriented extension to the logical expression to access the members of the object. The following example shows an extraction rule to extract the value of tag <telephone> in the block enclosed by the "customer" tag, and then to transform it into logical facts "person(P).telephone(#V)".

```
customer(P).tel(T#V) => person(P).telephone(#V)
```

The condition-part working like the XPATH, but it is in the form of object-oriented logical expression. It is designed to be embedded into PIL naturally.

A variable in an extraction rule is an address of a node. For example, the variable "P" binds to nodes with tag "customer" in XML documents, and the variable "T" binds to nodes with tag <tel> in XML documents. The operator "#" is a symbol to separate the node from its value. For example, the "V" in the matching part "customer(P).tel(T#V)" binds to the value of the node "T".

The matching-part is used to locate nodes in the XML documents, and the transformation-part is used to transform the located nodes into logical facts. For example, the "customer(P).tel(T#V)" is used to match nodes in XML documents with tag <tel> associated with the tag <customer>, and the "person(P).telephone(#V)" is used to transform the matching nodes into object-oriented logical facts where the predicate "person(P)" has a member predicate "telephone(#V)". The transformation-part of an extraction rule is a horn-clause with the extension of "." operator. The syntax of transformation-part is similar to Prolog with the following syntax.

```
Conclusion Part :- Premise Part
```

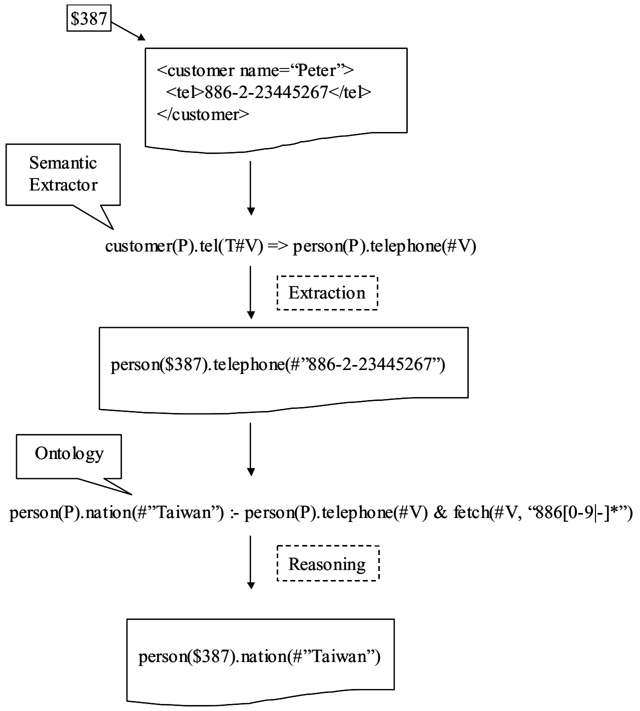
We emphasize that semantic extractors are not used to do syntax transformation, but to do semantic extractions. See the "Different semantics" problem in Section 1. The followings are the corresponding extraction rules with a conclusion part.

```
private(X) => person(X).
private(X).father(Y#Z)=>father_of(W,X):-person(W).name(#Z).
person(X) => person(X).
person(X).father(Y#Z)=>father_of(W,X):-person(W).security(#Z).
```

A rule in common ontology is a horn-clause just the same as the transformation part of an extraction rule. Matching part is no longer needed because all XML documents are transformed into logical facts. The common ontology should encompass the universe of the logical facts.

We emphasize that the left hand side of "=>" is the XML world which has its own syntax and semantics, and the right hand size of "=>" is the logic world which has its own syntax. So "person(X)=>person(X)" is not useless which means if person(X) is evaluated as true in XML world then put person(X) into logic world.

Seeing Figure 4, when we want to do reasoning on heterogeneous XML documents, the first thing is to choose a suitable ontology about the concerned domain, and the second is to write suitable semantic extractors for each type of XML documents.



$387
<customer name="Peter">
<tel>886-2-23445267</tel>
</customer>
Semantic Extractor
customer(P).tel(T#V) => person(P).telephone(#V)
Extraction
person($387).telephone(#"886-2-23445267")
Ontology
person(P).nation(#"Taiwan") :- person(P).telephone(#V) & fetch(#V, "886[0-9|-]*")
Reasoning
person($387).nation(#"Taiwan")

Figure 5. An example of dataflow for reasoning on XML

In Figure 5, we use one ontology and one type of XML documents to show the dataflow of our framework. The same condition holds on different types of XML documents. An XML document about a customer of Citibank named "Peter" is used as the input source. The node with "customer" tag in XML documents is assigned with a unique address "$387". A rule "customer(P).tel(T#V) => person(P).telephone(#V)" is used to locate the related nodes in the document and to transform a found one into the logical fact person($387).telephone(#"886-2-23445267"). This fact can be used in the reasoning process. The rule person(P).nation(#"Taiwan"):-person(P).telephone(#V) & fetch(#V, "886[0-9|-]*") in the common ontology is used to reason with the logical fact. The function "fetch" is an embedded function to match string, where "886[0-9|-]*" is a regular expression. In the reasoning process, the variable "P" is bound to "$387" and the variable "V" is bound to "886-2-23445267". Finally, the conclusion person($387).nation(#"Taiwan") is reached from the reasoning.

III. The Path Inference Language

In this chapter, we talk about the Path Inference Language (PIL). The syntax, denotational semantics and operational semantics will be discussed in detail. PIL is not only used to be a language to write extraction rules, but also a language to write ontologies.

A. *Syntax*

We adopted Prolog-like syntax in the Path Inference Language (PIL). A program is a set of statements. Each statement is a rule or an atom followed by a "." symbol. Each rule is of a form "A:-AS" where A is an atom and AS is a set of atoms. The syntax of an atom is much different from the definition of predicates in Prolog. A predicate in Prolog is a name string followed by parentheses with parameters. For example, person(a) or first_name(a,"Jack") are predicates in Prolog. But an atom in PIL may be a normal predicate with some arguments or a sequence of basic atom connected with "." or ".." operators. Each basic atom is of forms r(n#t), r(n), r, r(#t), n#t, n, or #t where r is a label, n is a node symbol or variable, and t is a term. Generally speaking, a basic atom is a name string followed by parentheses with a node address and a basic datum separated by a "#" symbol. For example, person(a) and first_name(c#"Jack") are basic atoms. For a formal definition of syntax of PIL, see the followings.

Syntax of PIL. A syntax of PIL is constructed by a tuple (VN,VD,L,P,F,C). VN and VD are two disjoint sets of variables. VN is a set of node variables, and VD is a set of data variables. L is a set of labels, P is a set of normal predicates, F is a set of functions, and C is a set of node symbols (node constants). Then, we are going to construct four sets: a term set T, a basic atom set B, an atom set A, and a sentence set S by induction.

1. $x \in T$, if $x \in VD$.
2. $f(t_1,t_2,t_3,\ldots,t_k) \in T$, if $f \in F$, $t_i \in T$, and the arity of f is k.
3. r(n#t), r(n), r(#t), r, n#t, n, #t$\in B$, if $r \in L$, $n \in C \cup VN$, $t \in T$.
4. $b_1 \bullet_1 b_2 \bullet_2 b_3 \bullet_3 \ldots \bullet_{k-1} b_k \in A$ if $b_i \in B$, and each $\bullet_i$ is a "." symbol or a ".." symbol.
5. $p(t_1,t_2,t_3,\ldots,t_k) \in A$, if $p \in P$, $t_i \in T \cup C \cup VN$, and the arity of p is k.
6. a. $\in S$, if $a \in A$.
7. a_0:-$a_1,a_2,a_3,\ldots,a_k$. $\in S$, if $a_i \in A$.
8. $s_1\ s_2\ s_3 \ldots s_k$ is a program, if $s_i \in S$.

Figure 6. The syntax of PIL language

B. *Denotational Semantics*

The universe of a model M in PIL is a directed graph G=(N,E,L,D), where N is a set of nodes, E is a set of directed links between nodes, each node is labeled by a label in L and associated at most one datum in D. The universe is just a part of a model. A model M also has to interpret the variable, function, predicate symbols in syntax definitions. For example, the following directed graph is an universe of a model M where N={na0,na1,…,na23}, E={(na0,na3), (na0,na6), (na0,na7), (na0,na21),…}, L={person, name, first_name, last_name, telephone, nation_code, area_code, number, extension, security, university}, and D={"A128825252", "Jieh Hsiang", "Jieh", "Hsiang",…}.

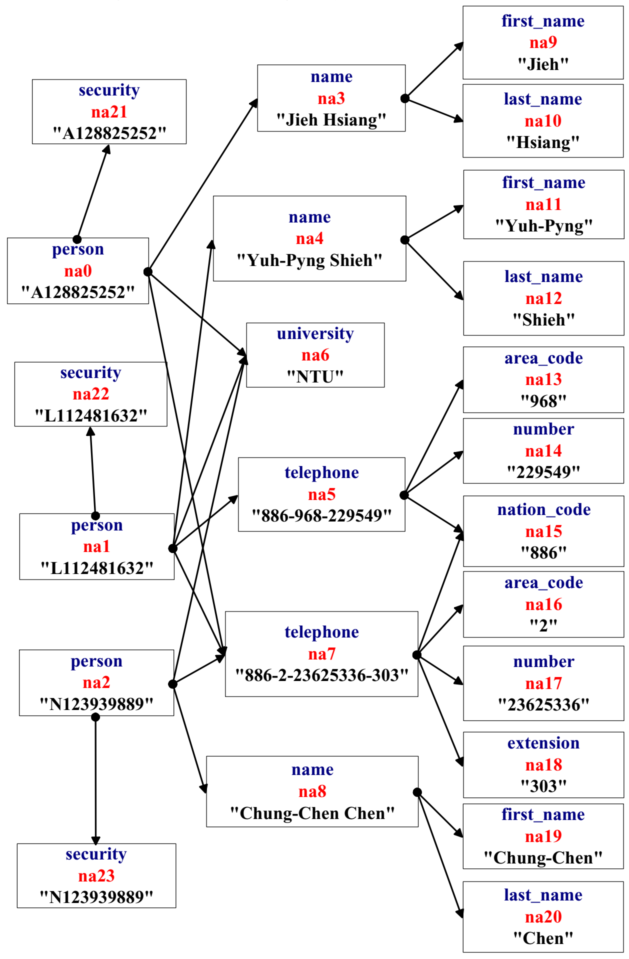


Figure 7. A graph model of some PIL program

In Figure 7, each node has a unique identifier called node address. For example, na0, na1, … and na23 are node addresses. Each node is associated with a label and a datum of some basic data type. For example, the node na6 is labeled by "university" and associated with a datum "NTU". Each label "s" represents two sets. One is the set of nodes labeled by s. The other is the set of data associated with the nodes labeled by s. For example, the label "last_name" represents a set of nodes {na10, na12, na19} and a set of data {"Hsiang", "Shieh", "Chen"}. Since a label can represent two sets, each label can also be taken as two predicates. One is a node predicate with a node argument, and the other is a datum predicate with a datum argument. For example, last_name(na10) and last_name("Hsiang") will be evaluated to be true and last_name(na0) or last_name("Liu") will be evaluated to be false. In order to distinguish between these two predicates, we put a "#" symbol before the datum argument. For example, last_name(X) is a node predicate and last_name(#Y) is a datum predicate. Also the "#" symbol is also a binary predicate. X#Y means Y is the datum associated on the node X. Sometimes, we will compose these predicates to construct a more strong predicate last_name(X#Y), which means last_name(X), last_name(#Y) and

X#Y. For example, last_name(na20#"Chen") is true and last_name(na13#"Shieh") is false.

After describing property of nodes, we will describe the relations between nodes. For each node x and y, x.y means there is a directed link from x to y. So the "." operator represents a directed links, also a binary relation. For example, na0.na3 is true and na0.na20 is false. Also, we will compose the "." operator with node predicates and datum predicates. person(na0).name(na3) means person(na0), name(na3) and na0.na3. There should be some natural association between the two predicates of the operator ".". For instance, in person(X).name(#Y) the natural meaning is that X is person whose name is Y. We further remark that the logical meaning of person(X).name(#Y) is actually that there exists a node z such that person(X).name(Z#Y). Sometimes we will use a sequence of predicates connected with "." operators. For example, person(X).name(Y).last_name(Z) means person(X).name(Y) and name(Y).last_name(Z). Also person(X).name.last_name(Y) means that there exists a node Z such that person(X).name(Z).last_name(Y). Since we use "." operator to represent a link relation, we use ".." operator to represent the path relation. For example, person(X)..last_name(Z) means that person(X), last_name(Z) and there exists a path from X to Z.

So the facts described in the graph can be encoded in the following program, and conversely the model M with universe G in Figure 7 is a model of following program.

example.pil

```
person(na0).security(#"A128825252").
          na0.name(na3).first_name(#"Jieh").
                    na3.last_name(#"Hsiang").
          na0.university(na5#"NTU").
          na0.telephone(na7).nation_code(na15#"886").
                        na7.area_code(#"2").
                        na7.number(#"23625336").
                        na7.extension(#"303").
person(na1).security(#"L112481632").
          na1.name(na4).first_name(#"Yuh-Pyng").
                    na4.last_name(#"Shieh").
          na1.telephone(na5).na15.
                         na5.areacode(#"968").
                         na5.number(#"229549").
          na1.na7.
person(na2).security(#"N123939889").
          na2.na6.
          na2.na7.
          na2.name(na8).first_name(#"Chung-Chen").
                    na8.last_name(#"Chen").
X#(Concatenation(Y,Z)) :- name(X).first_name(#Y),
X.last_name(#Z).
X#(Concatenation(Y,"-",Z,"-",N,"-",E)) :- telephone(X),
X.nation_code(#Y), X.area_code(#Z), X.number(#N),
X.extension(#E).
X#(Concatenation(Y,"-",Z,"-",N)) :- telephone(X),
X.nation_code(#Y), X.area_code(#Z), X.number(#N).
X#Y :- person(X), X.security(#Y).
```

Figure 8. A PIL program

For the formal definition of semantics, see the following definitions.

Denotational Semantics of PIL. Remark that the syntax of a program is constructed by a tuple (VN,VD,L,P,F,C). The universe of a model M is a tuple G of (N, E, L, D) where N is a set of nodes, E is a binary relation to represent the directed links between nodes, and D is a universe of data. Each node in N is labeled by a label in L, and associated by at most one datum in D. A model M also has to interpret each symbol, label, predicate or function in VN, VD, L P, F, and C.

1. $x \in T$, if $x \in VD$.
   M maps VD to D, and x to x^M.
2. $f(t_1,t_2,t_3,\ldots,t_k) \in T$, if $f \in F$, $t_i \in T$, and the arity of f is k.
   M maps f to be a function f^M from D^k to D and $f(t_1,t_2,t_3,\ldots,t_k)$ to $f^M(t_1^M,t_2^M,t_3^M,\ldots,t_k^M)$.
3. For each node symbol n in C or VN, M maps n to n^M in N.
4. r(n#t), r(n), r(#t), r, n#t, n, #t$\in$B, if $r \in L$, $n \in C \cup VN$, $t \in T$.
   $M \models$ r(n#t), if node n^M is labeled by label r and associated with a datum t^M.
   $M \models$ r(n), if node n^M is labeled by label r.
   $M \models$ r(#t), if $M \models$ r(n#t) for some n in VN.
   $M \models$ r, if $M \models$ r(n) for some n in VN.
   $M \models$ n#t, if the node n^M is associated with a datum t^M.
   $M \models$ n always, since M has to map each n in C or VN to n^M in N.
   $M \models$ #t, if $M \models$ n#t for some n in VN.
5. For each symbol n_1, n_2 in C or VN,
   $M \models n_1.n_2$, if there exists a directed link (n_1^M,n_2^M) in E.
   $M \models n_1..n_2$, if there exists a directed path in G=(N,E,L,D) from n_1^M to n_2^M.
6. For each b in B with the form r(n#t), r(n), n#t, r, r(#t), #t, or n, we define N(b) to the set {n} if b=r(n#t), r(n), n#t, n or the set VN if b=r, r(#t), #t. We also define b(x) to be r(x#t),r(x), x#t, r(x), r(x#t), x#t when b is r(n#t), r(n), n#t, r, r(#t), #t respectively. For each b_1, b_2 in B,
   $M \models b_1 \bullet b_2$, if $M \models b_1(n_1)$, $M \models b_2(n_2)$ and $M \models n_1 \bullet n_2$, for some n_1 in $N(b_1)$ and some n_2 in $N(b_2)$ where $\bullet$="." or "..".
7. $b_1\bullet_1 b_2\bullet_2 b_3\bullet_3\ldots\bullet_{k-1}b_k \in A$ if $b_i \in B$, and each $\bullet_i$ may be a "." symbol or a ".." symbol.
   $M \models b_1\bullet_1 b_2\bullet_2 b_3\bullet_3,\ldots,\bullet_{k-1}b_k$, if $M \models b_1\bullet_1 b_2$, $M \models b_2\bullet_2 b_3$, $M \models b_3\bullet_3 b_4,\ldots,M \models b_{k-1}\bullet_{k-1}b_k$.
8. $p(t_1,t_2,t_3,\ldots,t_k) \in A$, if $p \in P$, $t_i \in T$, and the arity of p is k.
   M maps p to be a set $p^M \subseteq D^k$. And $M \models p(t_1,t_2,t_3,\ldots,t_k)$ if $(t_1^M,t_2^M,t_3^M,\ldots,t_k^M)$ is an element of p^M.
9. a. $\in$S, if $a \in A$.
   $M \models$ a. if $M \models$ a.
10. a_0:-$a_1,a_2,a_3,\ldots,a_k$. $\in$S, if $a_i \in A$.
    $M \models a_0$:-$a_1,a_2,a_3,\ldots,a_k$. if $M \models a_0$, or M does not satisfy a_i for some i.
11. $s_1\ s_2\ s_3 \ldots s_k$ is a program if $s_i \in S$.
    $M \models s_1\ s_2\ s_3 \ldots s_k$, if $M \models s_i$ for each i.

Figure 9. The denotational semantics of PIL language

C. Operational Semantics

We use Prolog as the operational semantics by transforming a PIL program into a Prolog program.

1. If $x \in VD$, then x itself is in Prolog.
2. If $f(t_1,t_2,t_3,\ldots,t_k) \in T$, then it is already in Prolog as long as the function f^M is an embedded function in Prolog.
3. For each node symbol n in C or VN, it is already a Prolog statement.
4. If r(n#t), r(n), r(#t), r, n#t, n, #t∈B, we do the following. For each r in L, we construct predicates r_nt(n,t), r_n(n), r_t(t), r to encode r(n#t), r(n), r(#t), and r. Also, we construct predicates _nt(n,t), _t(t) to encode n#t and #t. we have to encode the relations between these predicates in the followings.
   r_nt(n,t):-r_n(n),_nt(n,t).
   r_n(n):-r_nt(n,t).
   _nt(n,t):-r_nt(n,t).
   r_t(t):-r_nt(n,t).
   r:-r(n).
   _t(t):-_nt(n,t).
   _t(t):-r_t(t).
5. For $n_1.n_2$, and $n_1..n_2$, we output $E(n_1,n_2)$ and $Path(n_1,n_2)$. Also, we have to put the relations between E and Path.
   Path(X,Y):-E(X,Y).
   Path(X,Y):-E(X,Z), Path(Z,Y).
6. For each b in B with the form r(n#t), r(n), n#t, r, r(#t), #t, or n, we define b' to be r(n#t), r(n), n#t, r(v), r(v#t), v#t, n, respectively, and b'' to be r(n#t), r(n), n#t, r(c), r(c#t), c#t, n, respectively where v is a unused node variable in VN and c is a unused node symbol in C. We define n(b') to be n or v according the form of b' and define n(b'') to be n or c according the form of b''. Then, for a atom $b_1 \bullet b_2$ in the right hand side of some rule, we output
   b_1', b_2', $n(b_1') \bullet n(b_2')$
   otherwise we output
   b_1'':-rhs.
   b_2'':-rhs.
   $n(b_1'') \bullet n(b_2'')$:-rhs.
   when the rule is $b_1 \bullet b_2$:-rhs.
7. For $b_1 \bullet_1 b_2 \bullet_2 b_3 \bullet_3 \ldots \bullet_{k-1} b_k \in A$, we output $b_1 \bullet_1 b_2$, $b_2 \bullet_2 b_3$, $b_3 \bullet_3 b_4, \ldots, b_{k-1} \bullet_{k-1} b_k$.
8. For $p(t_1,t_2,t_3,\ldots,t_k) \in A$, we transform each t_i into Prolog.
9. For a. , and a_0:-$a_1,a_2,a_3,\ldots,a_k$. $\in S$, we transform each a_i into Prolog.
10. For a program $s_1\, s_2\, s_3 \ldots s_k$, we transform each s_i into Prolog.

Figure 10. The operational semantics of PIL language

We note that the main difference between PIL and Prolog is the addition of three operators, ".", "..", and "#". The first two capture the parent-child node and ancestor-descendant node relationships in an XML document. The "#" operator specifies data in a node. Although these operators can be encoded as Prolog programs, these Prolog programs can be complicated and hard to write. It is therefore convenient to designate special operators for these purposes.

IV. Logic Based Ontology

Ontology in our sense is just background knowledge. Seeing Figure 4, a logic-based ontology is just a program of PIL encoding background knowledge. An ontology together with logical facts extracted from heterogeneous documents can be used to do some reasoning. For example, example.pil in last section can be divided into two parts. One part is directly extracted from XML documents. The other part can be seen as background knowledge. We put them there as an ontology person.pil. We emphasize, however, that an ontology can also include facts.

person.pil

```
X#(Concatenation(Y,Z)) :- name(X).first_name(#Y),
        X.last_name(#Z).
X#(Concatenation(Y,”-“,Z,”-“,N,”-“,E)) :- telephone(X),
         X.nation_code(#Y), X.area_code(#Z), X.number(#N),
         X.extension(#E).
X#(Concatenation(Y,”-“,Z,”-“,N)) :- telephone(X),
        X.nation_code(#Y), X.area_code(#Z), X.number(#N).
        X#Y :- person(X), X.security(#Y).
WorkingTogether(X,Y) :- person(X).telephone(Z),
        person(Y).telephone(Z).
```

Figure 11. A logic based ontology

V. Semantic Extractor For XML

When we want to do reasoning on heterogeneous XML documents, we should first choose a suitable ontology about the concerned domain. We then design suitable semantic extractors for each type of XML documents.

Here we take one ontology and one type of XML documents to demonstrate how the semantic extractor works. The same condition holds on heterogeneous XML documents. We use the ontology person.pil in the Section 4, and the following XML files with a private.dtd to illustrate our point.

private.dtd

```
<?xml version=”1.0”?>
<!DOCTYPE private[
<!ELEMENT private (name,father?,brother?,advisor?,tel+)
>
<!ATTLIST private
   security CDATA #REQUIRED
   sex (male|female)
>
<!ELEMENT name (CDATA)>
<!ELEMENT father (CDATA)>
<!ELEMENT brother (CDATA)>
<!ELEMENT advisor (CDATA)>
<!ELEMENT tel (CDATA)>
]>
```

hsiang.xml

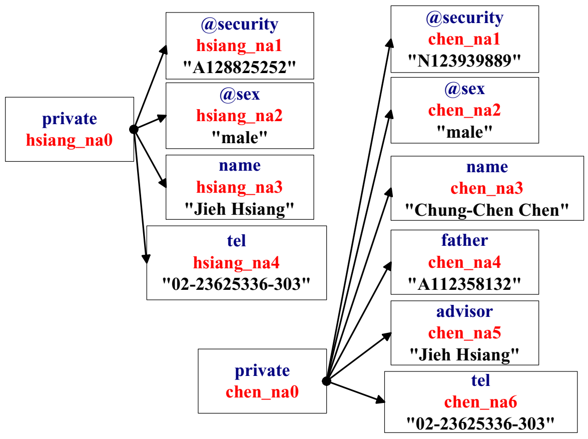
```
<private security=”A128825252” sex=”male”>
   <name>Jieh Hsiang</name>
   <tel>02-23625336-303</tel>
</private>
```

chen.xml

```
<private security=”N123939889” sex=”male”>
    <name>Chung-Chen Chen</name>
    <father>A112358132</father>
    <advisor>Jieh Hsiang</advisor>
    <tel> 02-23625336-303</tel>
</private>
```

The logical facts in the above XML documents can of course be composed manually in PIL. A semantic extractor, on the other hand, can extract them from these XML files automatically.

Given a set of XML documents, each document can be represented as a rooted tree. Each node is labeled by the tag name, or attribute name. We put a "@" symbol in front of the attribute name to distinguish these two types of labels. Each node can also be associated by a datum of type string. For example, we can draw two trees for the above two XML documents.



@security hsiang_na1 "A128825252"
@sex hsiang_na2 "male"
name hsiang_na3 "Jieh Hsiang"
tel hsiang_na4 "02-23625336-303"
private hsiang_na0
@security chen_na1 "N123939889"
@sex chen_na2 "male"
name chen_na3 "Chung-Chen Chen"
father chen_na4 "A112358132"
advisor chen_na5 "Jieh Hsiang"
tel chen_na6 "02-23625336-303"
private chen_na0

Figure 12. The tree structure of XML documents

These rooted trees can also be thought together as a directed graph. So we can use our PIL language to ask which atom is true under our directed graph model. For example, private(chen_na0).@security(#"N123939889") is true.

A semantic extractor (".sem" file) indicates that when the extractor matches some property of an XML file, the extractor will output some instruction into the ontology. The syntax of a .sem file is a set of rules. Each rule is of the form "AS => SS.", where AS is a set of atoms and SS is a set of sentences. On the left hand side of =>, the atoms describe the world of XML files, and on the right hand side, the sentence describes the world of ontology. On the left hand side, the atoms have its own syntax different from the right hand side. See the following sem file as an example.

private.sem

```
private(X).@security(#Y) => person(X).security(#Y).
private(X).@sex(#"male") => male(X).

private(X).name(Z#Y), fetch(Y,"[[a-zA-Z]*/F]
[[a-zA-Z]/A][[ ]*]") =>person(X).name(Z).last_name(#A),
Z.first_name(#F).

private(X).tel(Z#Y), fetch(Y,"0[[0-9]/A]-[[0-9]*/N]-[[0-9]*/E]")
=>person(X).telephone(Z#Y).nation_code(#"886"),
Z.area_code(#A), Z.number(#N), Z.extension(#E).
private(X).advisor(#Y)=>advisor_of(X,Z):-person(Z).name(#Y).
private(X).father(#Y)=>father_of(X,Z):-person(Z).sercurity(#Y).
```

Figure 13. A semantic extractor

The syntax of left hand side is constructed by (VN, VD, L, P, F, C) where VN={X,Z}, VD={Y,A,F,N,E}, L={private, @security, @sex, name, father, advisor, tel}, F={"N123939889", "male", "Chung-Chen Chen", …}, P={fetch} and C={chen_na0, chen_na1,…}. And the syntax of right hand side is constructed by (VN, VD, L, F, C) where VN={X,Z}, VD={Y,A,F,N,E}, L={person, security, name, first_name, telephone, …}, F={"N123939889", "male", "Chung-Chen Chen", …}, P={male} and C={chen_na0, chen_na1,…}.

The behavior of the extractor engine is to evaluate the truth-value of atoms on the left hand side, and (if it is true) then output the right hand side (by substitution) into the ontology. Here we can see there is a special predicate fetch which is used to divide a string into small piece using the regular expression decomposition. For example, the meaning of fetch(Y,"0[[0-9]/A]-[[0-9]*/N]-[[0-9]*/E]") is to check whether Y is a sting of the form "0[[0-9]]-[[0-9]*]-[[0-9]*]", and (if yes) put the part [0-9] to variable A, [0-9]* to variable N and [0-9]* to variable E, respectively. How many data processing function we should provide? We still take them into considerations.

Seeing the last two rules in this semantic extractor, an interesting observation is that the meanings of the contents in advisor and father tags are different. One is the "NAME" of X's advisor, but the other is the "SECURITY" number of X's father. This is why the DTD file cannot provide the semantics but SEM file can.

So when we use private.sem to extract the meanings of hsiang.xml, and chen.xml, will output some statements for the semantic facts in XML files as follows.

private.pil

```
person(hsiang_na0).security(#"A128825252).
male(hsiang_na0).
person(hsiang_na0).name(hsiang_na3#"Jieh
Hsiang").last_name(#"Hsiang").
hsiang_na3.first_name(#"Jieh").
person(hsiang_na0).telephone(hsiang_na4#"02-23625336-303").
nation_code(#"886").
hsiang_na4.area_code(#"2").
hsiang_na4.number(#"2325336").
hsiang_na4.extension(#"303").
… …
```

Figure 14. Semantic facts extracted from XML documents

And then together with the ontology person.pil, we can do some reasoning in the ontology. For example, we can find out whether Chung-Chen works together with Jieh Hsiang by asking whether WorkingTogether(X,Y), X.name("Chung-Chen Chen"), Y.name("Jieh Hsiang ") is true or false.

VI. Summary And Discussion

In this paper we proposed a logic-based framework for reasoning about XML documents. This includes a logic programming language, PIL, for describing ontologies and elements in XML documents as logic programs. We also presented a method to write semantic extractors in PIL extracting logical facts (also PIL programs) from XML documents, and a method for reasoning about these facts using ontologies. We think this framework can be used for integrating heterogeneous XML documents into uniform semantic rules, and for reasoning about them.

Our approach simplifies the specification stack of Semantic Web into just two layers, data and knowledge. XML describes the data. The knowledge part, including the specification of relations, ontologies, logic, and proof, can all be handled by PIL. A semantic extractor is the bridge to connect data and knowledge. Figure 15 shows the relationship between XML documents, semantic extractor and ontology.

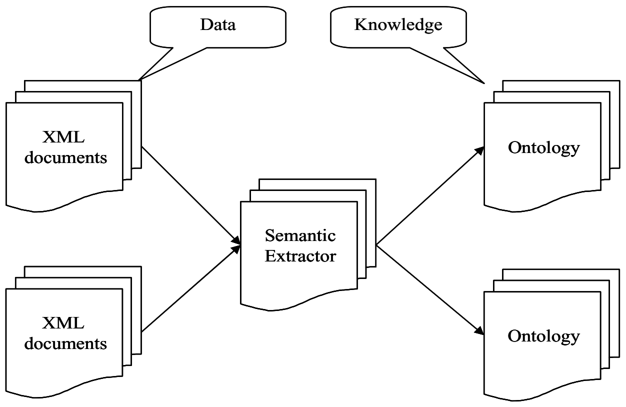


Figure 15. The relationship between data and knowledge

The separation of data and knowledge results in a flexible model. Data creators write XML documents as they like without having to follow standard. Knowledge workers may focus on the problem of knowledge construction for specific domains. Programmers may build applications by write program in PIL to extract logical facts from XML documents into ontology. Different programmers may have different ways to extract logical facts from data and put them into different ontologies for reasoning. This flexibility provides a good solution for knowledge engineering to scale up.

We now give a brief comparison between our approach and Semantic Web. The SW project was built in a bottom-up fashion. The languages for different layers were proposed incrementally (with those for the logic and reasoning layers still in the waiting.) Because the languages were built incrementally, there are heavy in notion and quite restricted. For instance, only binary relations can be expressed in RDF, and even OWL can only deal with simple set-theoretic relations.

In the model of SW, data builders must write documents based on the RDF specifications. Those XML documents that do not follow the RDF specification will be ignored. In our model, data builders may construct data as they like, and define tags as they need. They do not have to follow any specifications other than XML to construct documents. People who want to interpret these documents may write extraction rules to extract logical facts from XML documents, and then write rules of common ontology for reasoning. Logical rules in OWL are written in XML format. For example, the "intersection" operator is encoded as "<owl:intersectionOf>…</owl:intersectionOf>". In PIL, on the other hand, logical rules are described naturally as Horn clauses such as "r(z) :- p(x) & q(y)". So PIL expressions are clearly more readable than RDF or OWL.

Ontologies encoded in RDF and OWL can also be extracted by PIL into logical facts for reasoning. These can be done using the transformation mechanism given earlier.

We plan to extend PIL with object-oriented syntax and semantics, such as inheritance and polymorphism. We will design more embedded functions for string matching and processing. Furthermore, we will try to solve problems in a variety of domains to justify the practicality of our logic-based framework.

Acknowledgment

We would like to thank Szu-Pei Chen and Hou-Ieong Ho for fruitful discussions.

References

[1] T. Berners-Lee. Semantic Web Roadmap. World Wide WebConsortium (W3C), 1998 http://www.w3.org/DesignIssues/Semantic.html

[2] T. Berners-Lee, J. Hendler and O. Lassila. The Semantic Web, Scientific American, May (2001) 34-43.

[3] S. Decker, S. Melnik, F. van Harmelen, D. Fensel, M. Klein, J. Broekstra, M. Erdmann, and I. Horrocks. The semantic web: the roles of XML and RDF. IEEE Internet Computing, 43:2--13, 2000.

[4] M. Minsky (1975). A framework for representing knowledge. Available in Readings in Knowledge Representation, Brachman, R.J. & Levesque, H.J., Eds. (1985), Morgan Kaufman.

[5] R. Ifikes, and J. Kehler, (1985). The role of frame-based representation in reasoning. Communications of the ACM, Volume 28 Number 9, September 1985, pp. 904-920.

[6] R.A. Kowalski, The Early Years of Logic Programming, CACM, January 1988, pages 38-43.

[7] T. Berners-Lee. RDF and the Semantic Web. http://www.gca.org/attend/2000_conferences/XML_2000/knowledge.htm#lee