1. INTRODUCTION

Being an example to the use of denotational semantics this text should be an overview on the paper 'A Prototype of a Schema-Based XPath Satisfiability Tester' by Groppe and Groppe [3]. It will show all the basic functions needed to compute the examples at the end of these paper.

For better understanding the reader should read before 'Denotational semantics' by Klemens Muthmann. Also the concepts of XML [5], XML Schema [2] and XPath [1] should not be new to the reader.

In this paper the concepts of the satisfiability tester will be shown. Therefore we will compute some examples. For that we need some knowledge about XML Schema, which will be introduced in section 3. Because of the bad readability of XML Schema for humans a XML example file has been written. This file will be used to explain some XPath basics. We then also need some additional functions to be able to compute the semantic rules. After that we can analyse the three examples. At the end there will be some points of criticism and a conclusion.

Because it should only be an overview on the paper the loop detection and the computation of predicate expressions will not be described.

1.1 A satisfiability test for XPath

Examples for the use of this satisfiability tester can’t be given (due to the fact that no implementations could be found), but the intent of this tester is the analysis of XPath queries.

XPath itself is a query language for XML data. So the satisfiability tester should help to avoid unnecessary submission and evaluation of unsatisfiable queries. Therefore it evaluates XPath queries in presence of XML Schema files. The key benefits of testing with the presence of schemas are faster testing (because we only need the XML Schema file and not a XML instance), less processing time and less query costs.

Thus testing only with a schema the tester can only return unsatisfiable, if the XPath query evaluates as false, but can not return satisfiable, if the query returns true on the schema. On the XML instance the query can even be unsatisfiable. So in this fact the tester could only return maybe satisfiable.

2. XML AND XML FEATURES

2.1 XML Schema

As you can see XML Schema is a XML language which describes XML itself with specified rules. In this schema file we have a group with name 'pages' that has one child. The child element 'page' has 3 children: 'title', 'link' and a reference to 'pages'. This reference is a loop. Thus we can infinitely repeat this construct. But the reference points only to the style definition in the schema file and not to the specific tag in the XML instance.

The second definition is an element 'web' which has the children id as an attribute and the element page due to the reference on pages. So like the first example it refers not to the element itself but its children.

So in this example we have a path to the title element with '/web/page/title'.
You should remember that the semantic rules (they will be introduced in section 3.3) work on the XML Schema and not on this XML instance.

```xml
<?xml version='1.0' encoding='UTF-8'?>
<web id='1'
xsi:schemaLocation='http://www.example.org/web.xsd'>
  <page>
    <title>Outer</title>
    <link>
      <page>
        <title>Inner</title>
      </page>
    </link>
  </page>
</web>
```

Here we can match the rules of the XML Schema file against the XML tags and see clearly the reference, which is described before.

### 2.3 XPath

Because we need some basic XPath queries in the examples a quick overview of the syntax will be presented.

The '/' represents an absolute path beginning from the root node. If we want to return a specific node, we can reference to it by building a path from the root node to the node itself by separating each node with a slash.

Because to the fact, that 'child' is the standard axis in XPath, we can rewrite it.

**XPath: */web**

```xml
<?xml version='1.0'?><web id='1'>
  <page>
    <title>Outer</title>
    <link>
      <page>
        <title>Inner</title>
      </page>
    </link>
  </page>
</web>
```

**XPath: //page**

```xml
<?xml version='1.0'?><web id='1'>
  <page>
    <title>Outer</title>
    <link>
      <page>
        <title>Inner</title>
      </page>
    </link>
  </page>
</web>
```

With the '@' prefix we refer the attributes. In the case of '//@attribute' it will return all appearances of the attribute with the name 'attribute'. Due to the rewriting rules we can replace 'attr::' with '@'.

**XPath: //[@id]**

```xml
<?xml version='1.0'?><web id='1'>
  <page>
    <title>Outer</title>
    <link>
      <page>
        <title>Inner</title>
      </page>
    </link>
  </page>
</web>
```

With the square brackets we can make a node test. If we write '//@[attribute]', we test all nodes for the attribute 'attribute'. Only if the node contains the attribute, the node will be returned. We can also write '1' in square brackets to determine the first element of a set. The function 'last()' will then return the last element of a set.

**XPath: //[@id]**

```xml
<?xml version='1.0'?><web id='1'>
  <page>
    <title>Outer</title>
    <link>
      <page>
        <title>Inner</title>
      </page>
    </link>
  </page>
</web>
```

Out of the original paper the tester supports the following expressions:

```plaintext
e ::= e | e / e | e/e | e[q] [axis::nodetest]
q ::= e | e = C | e = e | qandq | qorq | not(q) | (q) | true() | false()
axis ::= child | attr | desc | self | following | preceding | parent | ances | DoS | AoS | FS | PS
nodetest ::= label | * | node() | text()
```

To solve the examples and to make it simpler for this paper we only need a subset:

```plaintext
e ::= /e | e/e | axis::nodetest
axis ::= child | attr
nodetest ::= label
```
3. XPATH SATISFIABILITY TESTER

3.1 Prerequisites

We also need some additional functions.

The function $\text{NT}(x, \text{label})$ is a node test. It checks, if a schema node and a given label are identical.

The function $\text{iAttr}(x)$ returns a set of all attributes of a given schema node.

The function $\text{iChild}(x)$ computes all children of a given schema node, returns a set and is defined as following:

$$\text{iChild}(x) = \{z \mid y \in S(x) \land z \in \text{succ}(y) \land (\text{isElem}(z) \lor \text{isText}(z))\}$$

The auxiliary function $S(x)$ relates the node $x$ to the self node and all the descendant nodes of $x$, which occur before the instance child nodes of $x$ in the document order. Now all successors of $y$ will be calculated and returned to $z$. Due to the definition of XML $z$ needs to be checked to be an element or a text, because also attributes are children of a node.

In the paper by Groppe and Groppe [3] some functions like $\text{iAttr}$ are only described in prose. This makes it hard to determine the output of these functions. Thus we can only try to interpret and hope to get the right output. You also can come to the conclusion, that there can be some inaccuracies. For this academically paper the authors should have described all functions with mathematical terms.

3.2 Schema Path

A schema path 'p' is a sequence of pointers to the schema path records $< XP', N, z, lp, f >$

- $XP'$ is an XPath expression,
- $N$ is a node in an XML Schema definition,
- $z$ is a set of pointers to schema path records,
- $lp$ is a set of schema paths,
- $f$ is a schema path list or a predicate expression $q'$

$\vartheta(r, g):$ The function generates a new schema path record $e = < x'r, g, -, - >$, adds a pointer to $e$ at the end of the given schema path $p$ and returns a new schema path.

For the examples we only need the first three entries of a schema path. 'lp' holds the information about detected loops and $f$ will only be computed if there is a predicate expression $q'$. The loop detection will not be used, so it will also not be explained here.

Because also $\vartheta$ is only described in prose, it is really hard to determine its output. Especially the pointer to prior entries makes it difficult to show on paper, but a line based method with identifiers in the front will be used to refer to these identifiers for the results.

$\vartheta$ will always generate a new schema path record, if all prerequisites are fulfilled. It will generate $xp'$, which is the actual part of the XPath query we process, $r$, which is the actual node in the XML Schema and $g$, which is set of pointers to prior schema path records.

3.3 Denotational Semantics

The denotational semantics in the paper are not in $\lambda$-Notation, although it is the most common method in denotational semantics. The functions in this paper get a syntactic expression and a parameter. The syntactic expression is in this case an XPath expression and the parameter is a set of schema paths. They use a set of schema paths because through the evaluation of the XPath expressions it can happen, that we get two different schema paths, which must be evaluated separately.

We also need only a subset of the given semantic expressions to solve the examples:

$$L : \text{XPath expression} \times \text{schema path} \rightarrow \text{set}(\text{schema path})$$

$$L_{/}[-]/[e]/[p] = L_{/}[-]/[p1] \land p1 = (\langle /, /, -, -, - >)$$

$$L_{/}[-]/[e]/[p] = \{p2|p2 \in L_{/}[-]/[p1] \land p1 \in L_{/}[-]/[e][p]\}$$

$$L_{[\text{child} :: \text{n}]}(p) = \{\vartheta(r,p(S))| r \in \text{iChild}(p(S)).N \land \text{NT}(r, n)\}$$

$$L_{[\text{attr} :: \text{n}]}(p) = \{\vartheta(r,p(S))| r \in \text{iAttr}(p(S).N) \land \text{NT}(r, n)\}$$

The first rule tells us, that we can compute 'e' when we substitute $\varepsilon$ with $\langle /, /, -, -, - >$. This schema path record is the root schema path record, because the third parameter 'e' points to no other schema path record.

The second rule splits the XPath expression and computes each component of the expression beginning on the right side, due to the fact, that we need 'p1' to compute the left function.

The third rule computes the standard axis of XPath. Before we can calculate $\vartheta$ we have to compute the prerequisites. First $\text{iChild}$ and then the node test. The auxiliary function 'p(S)' gets the last schema path record out of the schema path. The letter 'S' represents the size of the schema path, thus 'p(S)' means the last one, 'p(S-1)' means the pre-last and so on.

'p(S).N' refers to the specific entry in the schema path record and gets us the node of the XML Schema.

The last rule is similar to the third one, only $\text{iChild}$ is substituted by $\text{iAttr}$.

In the original paper [3] the authors could also in this case try to improve these expressions. In an article [4] (cited also by the authors) is described how XPath expressions can be rewritten to a simpler form by eliminating some axes. This improvement could save two third of the expressions and would make it in some ways much easier to understand.
4. EXAMPLES

4.1 Example 1
We will start with a simple XPath expression '/web':

\[ L[\text{/web}](\emptyset) = L[\text{web}](\emptyset) \]
\[ = L[\text{child} :: \text{web}](\emptyset) \]
\[ = \{ r \in \text{iChild}(\emptyset) \land NT(r, \text{web}) \} \]
\[ \text{iChild}(\emptyset) = \{ z \in S(\emptyset) : z \in \text{succe}(y) \land (\text{isElem}(z) \lor \text{isText}(z)) \} \]
\[ L[\text{/web}](\emptyset) = \{ r \in \{ \text{web} \} : r \in \text{iChild}(\emptyset) \land NT(r, \text{web}) \} \]
\[ = \emptyset \]

Result:
(R1) \{ (\emptyset) \}
(R2) \{ (\emptyset, \emptyset) \}

As you can see, we can use the first rule of our semantic expressions to start the process. 'p1' will always be \(< /, /, \ldots , /, \ldots >/\) for that rule. Due to the standard axis in XPath we then can rewrite 'web' to 'child::web'. We now can use the third rule. 'r' is the next part that needs to be calculated for the use in NT and \(\emptyset\). For that we execute the function iChild with '/-' as parameter, because \(p(S)N\) matches '/-' in our last schema path entry.

In iChild we get all successors of the root node '/-'. There we only get '/web' as a result. This result is also an element so we can return '/web'.

Now we can process NT(web, web) and get back 'true'. The next step is to calculate \(\emptyset\). For 'r' we can write '/web' and \(p(S)\) is our latest schema path entry.

As you can see this XPath expressions is satisfiable due to the XML Schema.

4.2 Example 2
The next example will show an unsatisfiable XPath expression '/page':

\[ L[\text{/page}](\emptyset) = L[\text{page}](\emptyset) \]
\[ = L[\text{child} :: \text{page}](\emptyset) \]
\[ = \{ r \in \text{iChild}(\emptyset) \land NT(r, \text{page}) \} \]
\[ \text{iChild}(\emptyset) = \{ z \in S(\emptyset) : z \in \text{succe}(y) \land (\text{isElem}(z) \lor \text{isText}(z)) \} \]
\[ L[\text{/page}](\emptyset) = \{ r \in \{ \text{web} \} : r \in \text{iChild}(\emptyset) \land NT(r, \text{page}) \} \]
\[ = \emptyset \]

In this case most of the example is identically to the prior one. Only the node test will fail. As result, we get back an empty set.

4.3 Example 3
A little bit longer XPath expression '/web/@id':

\[ L[\text{/web}/@id](\emptyset) = \{ p2 | p2 \in L[@id](p1) \land p1 \in L[\text{/web}](\emptyset) \} \]

\[ L[\text{/web}](\emptyset) = \{ (R1) \{ (\emptyset) \}, (R2) \{ (\emptyset, \emptyset) \} \}
\]
\[ p1 = \{ (R1) \{ (\emptyset) \}, (R2) \{ (\emptyset, \emptyset) \} \}
\]
\[ L[\text{/web}/@id](\emptyset) = \{ p2 | p2 \in L[@id](p1) \} \]
\[ L[@id](p1) = \{ (\emptyset) \}
\]
\[ \text{Result:}
(R1) \{ (\emptyset) \}
(R2) \{ (\emptyset, \emptyset) \}
\]

As you can see we use the second semantic rule to split the calculation. The calculation of \(L[text](\emptyset)\) is identically to the first example. We then can match 'p1' and our schema path the result of the calculation. Due to that we compute \(L[@id](p1)\). We rewrite '@id' to 'attr::id' and can use the fourth expression. iAttr(web) will return all attributes of 'web'. In this case we get back 'id'. Now our node test will succeed and we can calculate \(\emptyset\).

5. CRITICISM
In previous sections I have mentioned some points that could have been written better.

The most annoying part is the description of functions only in prose. Because of that for this paper I can't give the assurance that all output is correct. These functions were tried to interpret and the right conclusions have been reached hopefully. And although we used only a small part, functions like iAttr, \(\emptyset\) or \(p(S)\) are some examples.

Furthermore there are sometimes functions that were double defined, for example iAttr, isiAttr and iAttribute, isRoot and root. Here the authors could have written some better methods.

Another point is the definition of the semantic rules. Overall there exists 25 rules, but due to [4] it is possible to cut away at least two third of them. Perhaps they don't use that paper because they complain that they have written
an incomplete (but fast) satisfiability test.

The semantic rules are another factor. If you don’t know the specification of XPath and don’t know the standard axis and the rewriting you can’t compute any XPath query. But that fact is nowhere to read in the paper. Another problematic fact is the enormous complexity of some rules. Sometimes it can’t be avoided but then a good explanation would help. This paper sometimes lacks of explanations in general.

6. CONCLUSION

In the end one can’t recommend the paper by Groppe and Groppe. All in all also no area of application for this tester is seen. It could perhaps be used for testing XPath queries during the development, but in every day usage the need is probable marginal, due to the fact the developer should use only queries that are satisfiable.

The main fact for this little need is mainly because of the fact, that the tester could only return the result not satisfiable definitely. If the tester would test also a XML instance it possibly could be a good alternative for actual XPath query parser.

In the paper is also mentioned a performance analysis between the prototype, Saxon Evaluator and Qizx Evaluator. But that analysis compares only not satisfiable queries and due to that one can assume that in every day usage such speedup factors can’t be achieved.

Overall one have to say that they showed an interesting way of processing XPath queries and this approach could return a good way for processing XPath in general.

7. REFERENCES


