TREX-Q: A query language based on XML Schema

Brad Penoff
Sun Microsystems Ireland
Hamilton House
East Point Business Park
Dublin 3 ,Ireland

Chris Brew
Department of Linguistics
The Ohio State University
1712 Neil Avenue
Columbus, Ohio, USA

Abstract

James Clark's TREX is a clean, simple and powerful schema language for XML. On this foundation implemented a query language (called TREX-Q). The purpose of the present paper is to report our experiences in doing this, and to contribute to understanding of the issues that arise when a validator is extended into a query language. Aside from TREX, the strongest influences on our work are McKelvie's XMLQUERY and a similar algorithm described by Mark Hopkins in UseNet postings.

1 Introduction

1.1 Query languages and XML

Much of the work presented at this workshop focuses on the design and use of query systems for accessing large scale text and speech corpora. The promise of XML is that it will facilitate the creation of standard tools and techniques for the corpus-related tasks that arises in linguistics and language technology.

One of the strengths of XML is that it encourages us to make explicit our assumptions about the structure of data. We do this by preparing a DTD or a schema that encodes these assumptions. The most important tool in this enterprise is a *validator*: a program that checks whether the structure of a particular piece of data matches the assertions made in its schema. If so, we say that the data is valid according to the schema.

Whether the data in question is linguistic or not downstream processing is much easier if schema validity can be assumed, so good validation tools exist, are widely used and tend to be well-maintained by the community.

This is all good news for the developer of corpus tools, who can pull standard components off the shelf and use them as the front end to processes of linguistic or technological interest. But for corpus exploration mere validation is obviously insufficient: we need results. In other words, we require a query language. If the query language is well-designed many corpus-related tasks may be accomplished by means of the query language alone, without recourse to custom tool building. Thw `sggrep` tool described below is a prime example.

What then are the desiderata for corpus exploration? We certainly need to search large corpora and examine the results. It is unlikely that the format in which the data is delivered to us will be convenient or appropriate for every search that we will need to carry out. We may therefore need to conduct a systematic transformation of a large corpus, with or without substantial human intervention. We should expect that data will be delivered to us with irregular, inconsistent, poorly documented or peculiar structures. A powerful query language will make all these processes easier. But we cannot ignore efficiency, because our corpora are very likely to be big. In particular, we cannot assume that we will be able to afford to read our corpus into main memory. We therefore need a query language implementation that handles memory efficiently, at least for the common case where the corpus is large but the search result is of manageable size.

Given these considerations, the starting point of this paper is the observation that validation technology is currently better understood than query language technology (at least for the kind of messy and irregular documents that arise in language technology) The leading idea of this paper is to present a methodology for turning an XML validator into a query engine. We work up to this by easy stages. The plan of the paper is as follows.

- We will describe the query language used in the LT XML toolkit. This provides an efficient implementation of a useful set of baseline capabilities, and is presented primarily as background.

- We then introduce an unconventional and very concise regular expression matching algorithm due to Mark Hopkins. This algorithm the nice property that it is highly incremental, processing its input in a single pass. We call this algorithm REGEX. A key property of this algorithm is that it explicitly represents the non-determinism that is inherent in the query matching process.

- We then argue, following a suggestion by English (English) that the key ideas of REGEX are a viable approach to the validation of XML documents. We'll dignify this approach with a name and call it XMLX.

- We further explain how to convert REGEX into a query engine, by threading context through the operations of the matcher. This introduces extra non-determinism, but preserves the outline of the original. We call the result REGEX-Q.

- We next show how McKelvie's XMLQUERY (a non-deterministic extension of the LT XML query language) can be seen as an implementation of what we could call XMLX-Q. This system combines the two previous extensions of REGEX, producing a clean query language for XML.

- We next introduce TREX, the schema language on which our own system is based. This is one of several languages that have been designed with the ultimate intention of replacing DTDs. Part of the purpose of TREX seems to have been to clarify a few well-chosen issues in schema language design, without attempting the breadth of coverage that is present in the official W3C proposal.

- Finally we introduce our own TREX-Q, which is intended to stand in the same relation to TREX as the REGEX-Q does to REGEX.

2 LT XML

We argued above that we need a query language that can process large corpora with decent efficiency. One such is built in to LT XML(LTXML). Here "A query is a sequence of terms separated by /, where each term describes an [XML] element. It is no accident that they resemble Posix pathnames."(LTXML). Abstractly an XML document a decorated tree structure. Queries simply pick out subsets of the nodes present in that tree. Example queries include:

- `CORPUS/DOC/TITLE/s` – This matches all the `s` nodes whose ancestors are exactly `CORPUS/DOC/TITLE`. The `CORPUS` node in question has to be the root node of the hierarchy. In figure 1 the query matches the node `<s>Hamster plays with dog</s>`.

- `CORPUS/DOC/./s` – This relaxes the requirement that the element within which the `s` appears is a `TITLE`, but still requires that it be the grandchild of a `DOC`. This query matches all three `<s>` nodes in figure 1.

- `CORPUS/DOC/.*/s` – Now the `*` postfix operator, playing a role analogous to that which it plays in Posix regular expressions, allows sentences to be found anywhere among the descendants of the `DOC` element. In the example this matches the same set of sentences as the previous query.

- `./[0]/[1]/[1]/[5]` – the query language can also reference objects by position, using zero-based indexing. This query refers to 6th word of the second sentence of the second element of the first (and only) document. We could have specified more information, giving `./DOC[0]/BODY[1]/s[1]/w[5]`, and achieved the same result. Note however that `BODY[1]` does not mean "the second BODY element", but rather "any element that is both a `BODY` and in second position." An unwary user could run into trouble with this.

- `.*/BODY/s/[0]` – all the first elements of sentences.

- `.*/BODY/s/w[0]` – all the first elements of sentences, provided that they are words. In the example words are the only things that can appear in the position, but if the content model for `s` had been `(#PCDATA|w|c)*`, then there might be sentences whose first element is a `c` rather than a `w` and the query will return only a subset of the result which the user probably expects.

  The British National Corpus uses a `c` element in this way to code punctuation. In a subset of the corpus there were 263,918 sentence-initial `w` elements, but 20,699 sentence-initial `c` elements. There is therefore some risk that an

```
<?xml version="1.0"?>
<!DOCTYPE CORPUS [
<!ELEMENT CORPUS (DOC+)>
<!ELEMENT DOC    (TITLE,BODY) >
<!ELEMENT TITLE  (s+) >
<!ELEMENT BODY   (s+) >
<!ELEMENT s       (#PCDATA|w)* >
<!ELEMENT w      (#PCDATA) >
<!ATTLIST w   type CDATA #IMPLIED>
]>
<CORPUS>
<DOC>
<TITLE><s>Hamster plays with dog</s></TITLE>
<BODY>
<s>
<w>A</w> <w>hamster</w>
<w>played</w> <w>with</w><w>a</w><w>dog</w>
</s>
<s>
Update:<w>Unfortunately</w>,<w type="1">the</w> <w>dog</w>
<w>played</w> <w>with</w><w type="2">the</w><w>hamster</w>.
</s>
</BODY>
</DOC>

</CORPUS>
```

Figure 1: A sample XML input file

uninformed or forgetful user will miss nearly 10% of the relevant data.

- `.*/BODY/s/[type]` – all the elements below `s` that are specified for `type`. In figure 1 we have distinguished between instances of "the" that occur before a consonant from those that occur before a vowel or "h". The distinction might be important in a study of the way in which context affects the pronunciation of this article. The query returns all the appropriately annotated data.

- `.*/BODY/s/[type="2"]` – all the elements below `s` that have "2" as their value for `type`.

- `.*/BODY/s/w[type="2"]` – all the `w` elements below `s` that have "2" as their value for `type`.

- `.*/BODY/s/w[0 type="2"]` – the domain initial `w` elements that have "2" as their value for `type`.

From the outset, the LT XML query language was designed to support corpus annotation. As we have already seen, and will continue to see, it is far from trivial to define a usable query language. At least three constraints operate:

- The query language should be associated with a precise interpretation, in order that users have a chance of telling the difference between a bug in the implementation and a bug in query formulation. This is a *sine qua non*.

- There should be a good "story" (Pain and Bundy,1987) about how the query language works. Motivated users should be able to understand the query language with sufficient accuracy that they are able to use it effectively. The story might be a useful but sketchy version of the real truth about what happens, or might be the same as the interpretation given above. If casual users can use it too, so much the better.

- The final constraint is acceptable efficiency. In our view the primary factor here is document size rather than query complexity, but that is strongly conditioned by the particular tasks for which we have needed query languages. Presumably there are as many justified opinions on this as there are significantly different use cases for query languages.

All the above queries can be run using `sggrep`. This is one of the LT XML tools. Each is expressible as a single path terminating in the item of interest. A sample command line is:

```
sggrep -q '.*/s/w[type]'
```

The tool prints out all the satisfying instances, but does not show the surrounding context. To see context it is necessary to split `sggrep`'s query into two parts. As before, the first part of the query (specified with `-q`) selects a subset of the nodes in the input document. These nodes are then tested with a second query (the subquery, specified on the command line with `-s`). If a node satisfies the subquery, it is printed out by the tool. Thus, to see all the sentences containing a word specified for type, an appropriate command line is:

```
sggrep -q '.*/s' -s './w[type]'
```

Finally, `sggrep` allows a third argument, specified with `-t`, by means of which a user may provide a Posix-style regular expression which is to match the textual content of the leaf node at the foot of the search path specified by `-q` and `-s`. Thus, if we want to search for sentences containing a particular word in third position we can do this as follows.

```
sggrep -q '.*/s' -s './w[2]' \
 -t '[Hh]amster'
```

This mirrors the behaviour of the standard `grep` utility. For `grep` the input is a series of records, the extent of each record is defined by a newline character, and the output is a filtered subset of these records. Similarly, for `sggrep` the input is a series of records, the extent of each record is defined by the `-q` argument, and the output is a filtered subset of these records. For many purposes this is just what is required.

`Sggrep` is a simple tool, and it sometimes produces results significantly different from those which would be most useful. `Sggrep` satisfies queries in the following way. It first searches for a node satisfying the main query. The search is top-down, and ceases as soon as a matching node is found, so any ambiguity is resolved in favor of the highest node that matches the query. This node is now read into memory, and tested against the predicates from the `-s` and `-t` arguments. If it passes all tests, the node will be printed, otherwise not. Either way, the node is now discarded, and search recommences immediately after its right boundary. This is not necessarily desirable.

For example, one might wish to examine the prepositional phrases occurring in a syntactically analysed corpus. If the corpus is marked up in the obvious way, we will find PPs nested within other PPs. If we nevertheless attempt to use `sggrep` it is natural to use the query `-q .*/pp`. The good news is that this will print out every PP in the corpus, but the corresponding bad news is that it does *not* print out a separate record for each `pp`. Instead it prints out a separate record only for those PPs that are not enclosed by another PP [1]. If one wants one record per PP, postprocessing is needed. This is a limitation of `sggrep` rather than a deficiency in the query language itself. One solution is to move to a generic transformation langage such as XSLT or Xduce: another, pursued here, is to investigate the possibility of adjusting the behaviour of the query language to more closely match user expectations.

3 An incremental algorithm for regular expression matching

Figure 2 is an implementation of a regular expression matching algorithm described by Joe English(English), which he encountered in Usenet postings by Mark Hopkins. This algorithm is based on the insight that given a regular expression e over A that defines a regular language L and a symbol x in A, we can compute a new regular expression deltaex which represents the (possibly empty) language that is formed stripping the first symbol from the front of any strings in L that start with e. This new regular expression is referred to as the derivative of the first with respect to x. The code in the figure (a transcription into Objective Caml of Haskell code by English) shows that this can be done extremely concisely. `Delta` is nondeterministic. When we are computing the derivative of a sequence (`ef`)whose first element is nullable, it is uncertain whether x will be consumed by e or by f, so the algorithm explores both possibilities.

The salient implementation points are

[1]By analogy with the non-recursive base NPs studied in the chunking literature, one might want to call these *antibase* PPs

```
(* regular expressions over some base type ’a *)

type ’a re = Zero | Unit | Sym of ’a | Rep of ’a re | Plus of ’a re
  | Opt of ’a re | Seq of (’a re) * (’a re) | Alt of (’a re) * (’a re)

(* operator identities *)

let rep x   = match x with
  | Unit    -> Unit (* repeating [] gives []      *)
  | Zero    -> Unit (* repeating 0 no times is [] *)
  |e -> Rep e
let plus x  = match x with
  | Unit    -> Unit (* repeating [] gives [] *)
  | Zero    -> Zero (* repeating 0  at least once gives 0  *)
  | e -> Plus e
let opt x   = match x with
  |Unit              -> Unit (* optional [] same as [] *)
  |Zero              -> Unit (* optional 0 matches []  *)
  |e                 -> Opt e
let seq x y = match x,y with
   | Zero,_ | _,Zero -> Zero  (* sequences with 0 are 0 *)
   | Unit,f          -> f     (* Unit is the left identity for sequences *)
   | e,Unit          -> e     (* Unit is also the right identity  *)
   | (e,f) -> Seq(e,f)
let alt x y = match x,y with Zero,f ->f | e,Zero-> e | e,f -> Alt(e,f)

(* nullable e --  true if e could match the empty sequence *)

let rec nullable x = match x with
  | Sym _ | Zero        -> false
  | Opt _ |Rep _ | Unit -> true
  | Plus e              -> (nullable e)
  | Seq (e,f)           -> (nullable e) && (nullable f)
  | Alt (e,f)           -> (nullable e) ||  (nullable f)

(* delta e x -- regular expression derivative *)

let rec delta e x = match e with
  | Zero| Unit                  ->  Zero
  | Sym sym when x = sym        ->  Unit
  | Sym sym                     ->  Zero
  | Rep e                       ->  seq (delta e x)  (rep e)
  | Plus e                      ->  seq (delta e x)  (rep e)
  | Opt e                       ->  delta e x
  | Seq (e,f) when (nullable e) ->  alt (seq (delta e x) f) (delta f x) (*1*)
  | Seq (e,f)                   ->  seq (delta e x) f
  | Alt (e,f)                   ->  alt (delta e x) (delta f x)

let matches e items = nullable (List.fold_left delta e items)
```

Figure 2: A regular expression validator

- The familiar repertoire of regular expression components for symbols, sequencing,alternation and repetition are represented by constructors:`Sym,Seq,Alt,Plus,Rep`.
- There are separate constructors for the empty sequence (`Unit`) and for an empty alternation (`Zero`) that denotes the empty language. This last is not something that any user is likely to want, but is very useful as a representation of failure.
- `delta` does not construct regular expressions directly, but goes through a layer of constructor functions that apply operator identities.

In effect, the algorithm is creating and exploring the deterministic automaton that is implicit in the input regular expression. It does not explore branches that are not required by the input. Hopkins puts it like this

> The method used is to incrementally construct a DFA for the given regular expression one state at a time as the input is being processed. For regular expressions with small DFA's construction of new states will be complete early on with the result that most of the input file will be processed relatively fast without having to calculate new states. For regular expressions with large DFA's but smaller inputs, the construction of new states will be kept to a minimum.

The key idea here (one that is *not* reflected in the Objective Caml implementation given above) is that `delta` should be memoized. Several of its clauses may generate regular expressions that are larger than their inputs, copying subparts of the input to more than one location. Under these circumstances, even though application of the operator identities may mitigate the increase in size, memoization is necessary for efficiency. In a lazy functional language, such as the original Haskell, memoization would be the responsibility of the underlying implementation, but in Objective Caml, which is a strict language, no such convenience is available. We need not for the moment be too concerned about this, since our main interest in the algorithm comes from the insight that it gives into the problem of parlaying a validator into a query language.

In the following two sections we describe two extensions of REGEX. The first extends the string matcher to produce a tree validator, while the second produces a query language over strings. In combination, the two extensions yield a query language over trees, as required. This query language is very close in spirit to McKelvie's XMLQUERY (McKelvie).

4 XMLX: making REGEX into an XML validator

We now address the extensions needed to make REGEX into a validator for a tree language similar to XML The code in figure 2 already exploits the type system of Objective Caml to produce a regular expression matcher that is non-committal about the base type of the sequences that are matched. This is reflected in the presence of a type variable in the definition of `'a re`. The same code can be used either to match a regular expression whose leaves are integers against a sequence of integers or a regular expression whose leaves are strings against a sequence of strings. This is useful, but not enough, for the code still requires identity between the base types of the regular expressions and the sequences. Fortunately, there is only one point in the code at which this identity is required. This crucial fragment is repeated below:

```
let rec delta e x = match e with
  ...
  | Sym sym when x = sym -> Unit
  | Sym sym              -> Zero
  ...
```

It is straightforward to relax this constraint by changing the fragment to

```
let rec delta e x = match e with
  ...
  | Sym sym when
compatible x sym -> Unit
  | Sym sym             ->  Zero
  ...
```

and providing an appropriate definition of `compatible` such as:

```
let compatible s q =
s = int_of_string q
```

. This setup requires the base type of the query to be `string` while the base elements of the sequence searched is `int`. This is not very useful in itself, but provides some of the flexibility that a query language will need. An alternative definition, shown below, provides a simple wild-card facility.

```
let compatible s q =
  match q with
    | "." -> true
    |  _  -> s = q
```

Clearly this idea can be extended. The base type of the searched sequences could be a discriminated union of XML start tags, end tags and textual content, and the base type of queries can be enriched to allow not only wild cards but also the attribute tests used by LT XML. While crude, this would clearly be an XML query language.

This is not sufficient, because REGEX is based on deterministic finite state automata, while XML validation requires the power of a push down automaton. We therefore need a more elaborate extension to the REGEX framework. Two things must change:

- The query now has to provide a regular expression for each XML element type that might arise in the input. This is the content model aspect of DTDs and other schema languages.
- At run time we need a push-down stack that can hold partially matched elements.

In the original `delta` regular expressions alone were sufficient to represent all necessary information each branch of the non-deterministic matching process that is required. Non-determininism is represented in the `alt` statements of the regular expressions. For the extension, all that has to happen is that `delta` provides each process not only with a residual regular expression reflecting the consumption of a symbol but also with a push-down stack of return incomplete items. As we encounter start tags with a we push the appropriate content model expression for onto the stack, and when we encounter end tags we pop the stack and resume matching of the content model for the parent. This is of course old technology, that of a recursive transition network.

This extension preserves the architecture of the original REGEX, and keeps the nice property that the whole thing works incrementally. We are now getting close to the capabilities of the LT XML query language.

5 REGEX-Q: making REGEX into a query language

A very similar extension to REGEX can convert REGEX into a query language which we dub REGEX-Q. REGEX uses `Zero` to represent a match failure and `Unit` to represent expressions that match the empty string. A regular expression matches if, by the time the input string has been consumed, there exists at least one branch of the non-deterministic execution that is equivalent to `Unit`. This is checked by the `nullable` function.

If we want to keep essentially the the same algorithm, but to return a set of strings, we can adjust `delta` so that whenever it matches a symbol (which will happen when we get to a `Sym` element of the query expression) it incorporates information about that symbol in the expressions that it passes to the next stage. Things are arranged so that every branch yields a set of strings. Correspondingly, `nullable` will need to be adapted to combine the sets in an appropriate way, replacing the boolean operation that pulls the results from different branches of an `Alt` with a set-union

This is very straightforward to do (though so mechanical that it is tempting to complicate the task by implementing the as a source-to-source transformation on the original code). It is not especially useful, because all expressions that match the string will record the same information, namely that they managed to match the whole string, and we will be going to great lengths to form sets whose eventual composition we can be sure of from the outset.

But if we augment the regular expression language with a `save` operator that can wrap parts of the query (and perhaps also name them), then more interesting things can happen. As we walk through the symbol string, state is built up in a set of different accumulators, and the way that this happens may differ in interesting ways dependent on the execution path taken. At this point the whole non-deterministic apparatus is motivated and the result is a set of possibly different query results, expressed as bindings for the marked expressions.

This too is not difficult to do, and preserves the skeleton of the REGEX algorithm. Although more contextual information is retained, the whole thing remains incremental and is essentially little more than the lazy exploration of an automaton. There are some issues which arise: in particular it is not obvious what it should mean when a save operator is nested within a repetition operator (see later for our provisional solution to this).

6 XMLQUERY

McKelvie's XMLQUERY (McKelvie) combines the two extensions to REGEX described above. It uses essentially the same non-deterministic execution model, working with multiple processes, which McKelvie describes in the following way

> he xmlquery program works by reading and parsing the query string. If this parse succeeds then, the parsed query (in the form of a parse tree) is translated into a program for an abstract machine. The abstract machine is a non-deterministic stack automaton.
>
> There are a number (initially one) processes that execute this program. . . . Each process has:
>
> - a program counter
> - a depth stack
> - a save stack
>
> Some of the instructions are non-deterministic, in these cases a process that executes them splits into two processes, one for each branch.

Notice that the above description, modulo terminology, is identical to what would be obtained by implementing both of the extensions to REGEX that are introduced above. XMLQUERY has both a return stack for managing tree context and a save stack for keeping track of the material that has been matched to particular parts of the regular expression.

The syntax of XMLQUERY is similar to that of the LT XML query language introduced above. It uses `,` as an explicit sequencing operator, so that

```
xmlquery -q ".*/(a,.*/b,a)" test.xml
```

matches any sequence of siblings consisting of an `a`, then any element that contains a `b` anywhere inside it, followed by another `a`. XMLQUERY uses `!` as postfix operator to indicate what should be printed out.

```
xmlquery -q ".*/s/phr/w!" test.xml
```

matches and returns any `w` inside a `phr` inside an `s` anywhere. while

```
xmlquery -q ".*/s/phr!/w" test.xml
```

matches and returns any `phr` inside an `s` anywhere, so long as it contains a `w`. It also allows variables

```
xmlquery -q ".*/%X!/%X!" test5.xml
```

Match and return any element which has a child of the same element name.

XMLQUERY is also efficient, though not as fast as the much simpler `sggrep`. The performance penalty is on the order of a factor of two for some simple queries.

Because XMLQUERY is non-deterministic, it avoids the difficulties that we noted when we considered using `sggrep` as a tool for treating a corpus as a database of tree fragments. If a query matches several ways, it may be that the same part of an XML document will appear several times in the output, albeit in different contexts. Both behaviours are useful.

7 TREX

James Clark's TREX(Clark) is a clean, simple and powerful schema language for XML. It is implemented in Java, and the source is available. We chose it as the basis for extension partly because we were familiar with Java, and partly because it is well-documented and easily available, and because it seems to be designed with a concern for simplicity. TREX has since been folded into a larger community effort called RELAX-NG (Relax) TREX and RELAX-NG have strong similarities. Our work was based on the original TREX. We wanted to test the hypothesis that the approach to query language contruction outlined above is viable when applied to a validator which uses a language more conventional than Objective Caml or Haskell.

8 TREX-Q

The TREX-Q implementation effort began the first author's senior thesis

but has spilled over into further work conducted after graduation.

TREX-Q allows parts of a document to be saved and returned upon successful validation. Parts to be saved are in between `<save>` tags.

It is possible to save complete elements.

```
<element name="a">
  <save>
    <element name="b">
      <empty/>
    </element>
  </save>
</element>
```

When running the TREX-Q processor with the above pattern on the following document:

```
<a>
  <b/>
</a>
```

We obtain the following output:

```
<matches total="1">
  <match id="1">
    <b/>
  </match>
</matches>
```

We can also save elements and attribute names This facility will be more useful given the regular expression string matching described below.

8.1 Repetition

We had to decide how save patterns should interact with repetition operators. In the case where the `<save>` pattern outscopes the repetition operator we (uncontroversially) return the sequence matched

```
<root>
  <a/>
  <a/>
  <a/>
</root>

<element name="root">
 <save>
  <oneOrMore>
    <element name="a">
      <empty/>
    </element>
  </oneOrMore>
 </save>
</element>

<matches total="1">
  <match id="1">
    <a/>
    <a/>
    <a/>
  </match>
</matches>
```

More controversially, we implement the following semantics for `<save>` patterns within repetion operators

```
<root>
  <a/>
  <b/>
  <a/>
  <b/>
</root>

<element name="root">
  <oneOrMore>
    <save>
      <element name="a">
        <empty/>
      </element>
    </save>
    <element name="b">
      <empty/>
    </element>
  </oneOrMore>
</element>

<matches total="1">
  <match id="1">
    <a/>
  </match>
  <match id="2">
    <a/>
  </match>
</matches>
```

8.2 Regular expressions

TREX-Q allows queries to specify any string by a Perl 5 regular expression. This is done by setting a regexp attribute within a ¡string¿ tag to ”true”.

Say you wanted to make sure a particular string started with the letter ”t”. This can be used in the following way.

```
<root>twinkiesaregood</root>

<element name="root">
  <save>
    <string regexp="true">ˆt.*</string>
  </save>
</element>

<matches total="1" pattern="regexp.trex">
  <match id="1" filename="regexp.xml">
    twinkiesaregood
  </match>
</matches>
```

8.3 Variables

Finally we implement variables, essentially s in XMLQUERY but with different syntax.

```
<element name="root">
  <element name="a">
    <variable name="x"/>
  </element>
  <variable name="x"/>
</element>

<root>
  <a>
    <b/>
  </a>
  <b/>
</root>
```

The same query would also match

```
<root>
  <a>
    <b/>
    <c/>
  </a>
  <b/>
  <c/>
</root>
```

9 Conclusions

We have shown one way of making an XML validator into a query language. Building on similar previous work by Hopkins, English and McKelvie we have shown how a decision to explicitly represent run-time non-determinism can lead to a design that is not only easy to understand but also potentially efficient in its handling of large corpora.

We think that the non-determinism that arises is inherent in the task, and that any query language will need to handle it in one way or another. This is not specific to XML, but arises even in query languages for string search. While one approach is to use devices such as the longest match heuristic in order to mitigate the impact of non-determinism, we have preferred to search for a solution that is as clean as we know how to make it. Early indications, especially from McKelvie's XMLQUERY(McKelvie), are that this approach can succeed.

Our implementation of these ideas is directly based on James Clark's TREX language, which the author describes as

> "basically the type system of XDuce with an XML syntax and with a bunch of additional features (like support for attributes and namespaces) needed to make it a practical language for structure validation."

What has been presented here is an interim report on our efforts to extend TREX into a useful query language, as well as a commentary on the lessons learned in this endeavour.

References

Clark J. TREX - "Tree Regular Exoressions for XML" ms. available from `www.thaiopensource.com/trex`

Clark J. and Murata M. RELAX NG specification `www.oasis-open.org/committees/relax-ng`

English, J "How to validate XML" ms. available from `www.flightlab.com/˜joe/sgml/validate.html`

McKelvie D. XMLQUERY 1.5 manual. ms. available from `www.ltg.ed.ac.uk/˜dmck/xmlstuff`

McKelvie, D., Isard, A. Mengel, A. Moeller,M.B. Grosse, M. and Klein, M. (2001) "The MATE Workbench - an annotation tool for XML coded speech corpora", Speech Communication 33 (1-2) (2001) pp 97-112. Special issue, "Speech Annotation and Corpus Tools"

Pain, H. and Bundy, A. (1987). What Stories Should We Tell Novice Prolog Programmers? In: R. Hawley (Ed.), Artificial Intelligence Programming Environments. Ellis Horwood: Chichester, U.K.

Penoff B. and Brew C.H. TREX-Q tutorial (2001) available at `ling.osu.edu/˜penoff`

Thompson, H.S. McKelvie,D. Brew, C.H. Mikheev, A. and Finch, S. (2000) The LTXML Toolkit available at `www.ltg.ed.ac.uk/software`