Algorithmic Design Using Object-Z for Twig XML Queries Evaluation

Yang Liu¹ and Jun Sun²

School of Computing
National University of Singapore
Singapore

Key words: XML, Twig Query Evaluation, Object-Z, Specification

Abstract
Web technologies based on XML, e.g. Semantic Web and Web Services, facilitate Web automation and universally accessible content. One of the key factors for the success of XML-based technologies is of finding an efficient query evaluation algorithm for XML-based data models. An XML twig query is a complex selection predicate on both structure and content of a labelled XML document. Several novel twig query evaluation algorithms have been proposed recently. However, these algorithms are difficult to understand and hence implement due to high complexity. In this work, we present an algorithmic design for XML queries evaluation system using Object-Z. An Object-Z specification is developed to give a concise and logical description of the XML data model and the twig queries. It makes the twig query evaluation straight-forward, and allows different evaluation algorithms to be constructed easily and independently.

1 Introduction

XML is currently emerging as a de facto standard for information exchange over the Internet. XML-based ontology languages like RDF, DMAL+OIL/OWL [3] are the building blocks of Semantic Web [23] as they provide basic vocabularies for data markups: the ontologies. Therefore, the capability of processing XML documents efficiently is vital to the success of SW-aware applications because software agents are to perform autonomous discovery, querying, processing of data marked-up by XML documents.

Although XML documents could have rather complex internal structures, they can generally be modelled as ordered trees. Queries in XML query languages typically specify patterns of selection predicates on multiple elements which have some

¹ Email: liuyang@comp.nus.edu.sg
² Email: sunj@comp.nus.edu.sg

This is a preliminary version. The final version will be published in
Électronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
specified structural relationships (see, e.g., [4,5]). For example, to retrieve all paragraphs that are nested inside sections and have at least one figure and one table can be expressed as:

\[ Q_1 = \text{//section//paragraph[figure AND table]} \]

Such a query can be represented as a node-labelled twig pattern (or a small tree) with elements and string values as node labels [6]. Finding all occurrences of a twig pattern is a core operation in XML query processing [14,21,19,22]. Currently, one of the most efficient algorithms is holistic twig join algorithm \( TSGeneric \) [16]. It makes use of stacks to cache XML elements and a cursor interface that provides standard methods to return elements with possible matches. With different implementations of the cursor interface [15], efficient algorithms [16] have been developed to process twig joins based on available access methods.

However, due to the high complexity of the holistic approach, it is very difficult to understand and hence implement the evaluation process. Formal specification languages like Object-Z are precise and highly reusable, which offer a non-ambiguous specification of the problems and hence allow efficient algorithmic design. In this paper, Object-Z [13] is used to give an object-oriented specification of the tree-structured XML data model and twig query structure. In particular, generic classes are combined with class inheritance to structure the specifications, which leads to a specification that is highly extensible and a clear structure that is useful for efficient algorithmic designs for XML query evaluation. We show that various evaluation algorithms can be formalized straightforwardly using our structure. In general, we believe that our approach is applicable to other situations where intensive algorithmic design based on complicated data structure is required, e.g. task scheduling, security protocols and etc.

Our work is related to studies on applying Formal Methods to XML-related aspects of the Web domain. In [25], a formal object-oriented semantic model of XSLT in Object-Z is presented, which provides a formal understanding of the language and helps the standardization effort for XSLT. In [26], Object-Z specification is used to model common semantic constructs of XSLT2.0, XPath2.0 and XQuery1.0. The purpose is to reuse these semantic constructs to specify the semantics of XML family languages and to understand the commonality and difference between them. Our work focuses on algorithmic design using Object-Z for XML queries evaluation. We show that our Object-Z specification eases the design of the complicated algorithms for the twig query evaluation problem. We believe that our approach is applicable to systems where complicated algorithmic design based on complex data structures are required. Our work is also related to works on applying Formal Methods to other Web technologies, some of which are evidenced in [20,12,28,17,11]. Another remotely related work is the work on connections between Z schemas and relational database by Martin and Simpson [18].

The rest of the paper is organized as follows: Section 2 briefly introduces the Object-Z specification language. Section 3 describes positional representation of XML data model and its Object-Z specification. Section 4 describes the XML twig
query and related data structures. Section 5 presents the whole system of XML twig query evaluation. Section 6 shows how the specification facilitates the design of different evaluation algorithms. Section 7 concludes the paper.

2 Object-Z

Z [24] is a state-based formal specification language based on the established mathematics of set theory and first-order logic. It has been used to specify a wide range of systems including transaction processing systems and communication protocols. A specification in Z typically consists of a number of state and operation schemas. A state schema groups together variables and defines the relationship that holds between their values. An operation schema defines the relationship between the ‘before’ and ‘after’ valuations of one or more state schemas. Object-Z [13] is an object-oriented extension of the Z language. It improves the clarity of large Z specifications through enhanced structuring. The main Object-Z construct is class definitions, which captures the object-oriented notion of a class by encapsulating a single state schema with all the operations which may affect its variables.

An Object-Z class is represented syntactically as a named box with zero or more generic parameters. There may be local types and constant definitions, at most one state schema and one associated initial state schema and zero or more operations. The declarations of the state schema are referred to as the state variables and the predicate as the class invariant. The class invariant restricts the possible valuations of the state variables. The initial schema identifies the possible initial valuations of the state schema. An operation is either an operation schema or a schema expression involving existing class operations and schema operators.

To indicate that an object is (physically) contained by another class, rather than just referenced by it, its type is annotated with the symbol \[\in\] [9]. For example, the declaration \(a : A\in B\) in class \(B\) indicates that an object of class \(B\) has a reference \(a\) to an object of class \(A\) and this object is contained by the object of class \(B\). A single object cannot (directly) be contained by more than one object. Classes may be specified incrementally using inheritance. A class which inherits another may extend its definition with new state variables, new invariant and initial constraints, and new operations. It may also add new constraints to existing operations. In a specification including one or more inheritance hierarchies, variables may be declared polymorphically to have the type of any class within a hierarchy.

3 XML Data Model

In this section, we explore the commonly used representation of XML data in twig query evaluations. This representation gives some interesting relations between the XML elements. The formal definition of XML document using Object-Z class definition is presented in the end of this section.

Most existing XML query processing algorithms rely on a positional representation of XML elements [27,2,8,15], where XML documents are modelled as
Fig. 1. A Sample XML Document

ordered trees and each tree node (XML element) is represented as a tuple of the form: \((docID, start, end, level)\). \(docID\) is the identifier of the XML document in XML database. \(start\) (end) refers to the number of elements traversed in pre-order from the root \(r\) of the document till the start (end) of the element. \(level\) indicates the nesting depth of the element. A sample XML document containing root \(r\) and elements \(a, b\) and \(c\) is shown in Figure 1. Pre-order traverse is performed starting from the root \(r\). Once an element is visited, its \(start\) value is set to current counter. When an element is visited in second time, its \(end\) value is set to current counter.

In order to represent the XML document elements completely in Object-Z, first we define XML element tag names set \(TAG\) using free type \(Char\). \(Char\) includes all valid characters for tag names. \(TAG\) is the set of all possible XML element node tag names, i.e. a sequence of valid characters starting with a letter or two special characters. This definition strictly follows the definition in the World Wide Web Consortium. Detailed definitions can be found at [1].

\[
\begin{align*}
[Letter, Digit, CombiningChar, Extender] \\
Char &::= letter\langle Letter \rangle | digit\langle Digit \rangle | ', '* | ' . ' | ' - ' | ': ' \\
& \quad | comChar\langle CombiningChar \rangle | extender\langle Extender \rangle \\
TAG &::= \{ t : seq Char | head(t) \in Letter \lor \\
& \quad (head(t) = ' . ' \lor t.head(t) = ': ') \}
\end{align*}
\]

The Object-Z modelling of XML elements is defined as the following \textit{Element} class, where two simple class invariants are defined:

\[
\begin{array}{l}
\text{Element} \\
\quad \mid (docID, start, end, level, tag) \\
\quad \quad docID, start, end, level : \mathbb{N} \\
\quad \quad tag : TAG \\
\quad \quad start < end \land level < start
\end{array}
\]
With positional representation, the structural relationships between tree nodes can be determined easily as: (1) ancestor-descendant (A-D) relationship: element $v$ is a descendant of element $u$ if and only if $u.start < v.start < u.end$; (2) parent-child (P-C) relationship: element $v$ is a child of element $u$ iff $u.start < v.start < u.end$ and $u.level = v.level - 1$. Two relations $\text{IsAncestorOf}$ and $\text{IsParentOf}$ are defined accordingly:

\[
\begin{align*}
\text{IsAncestorOf} : & \text{Element} \rightarrow \text{Element} \\
\text{IsParentOf} : & \text{Element} \rightarrow \text{Element} \\
\forall x, y : & \text{Element} \bullet \\
& x \text{IsAncestorOf} y \iff x.start < y.start \land y.start < x.end \land \\
& x.level < y.level \land x.docID = y.docID \\
\forall x, y : & \text{Element} \bullet x \text{IsParentOf} y \iff \\
& x \text{IsAncestorOf} y \land x.level = y.level - 1
\end{align*}
\]

For any two elements $x$ and $y$ in a XML document, there are four possible positional relationships between them, which are illustrated in Figure 2. Case 2 and case 3 in Figure 2 are captured by the previous relation $\text{IsAncestorOf}$. A new relation is needed to describe the case 1 and case 4: $x.end < y.start$ and $x$ and $y$ have a common ancestor. These two constraints are summarized in the following relation $\text{IsLeftKinOf}$ (and it’s inverse):

\[
\begin{align*}
\text{IsLeftKinOf} : & \text{Element} \leftrightarrow \text{Element} \\
\forall x, y : & \text{Element} \bullet x \text{IsLeftKinOf} y \iff x.end < y.start \land \\
& \exists z : \text{Element} \bullet z \text{IsAncestorOf} x \land z \text{IsAncestorOf} y
\end{align*}
\]

Before we give the full Object-Z definition of XML documents, we need to introduce one more concept: stream. A stream contains all the positional representations (tuples) of the XML nodes that have the same tag. The tuples in one stream are sorted by their $\text{start}$ values ascendingly. For example, a stream of tag $a$ (denoted as $T_a$) in the XML document shown in Figure 1 is a sequence (assume $\text{docID}$ of the XML is 1): $\{(1, 2, 17, 1), (1, 3, 8, 2), (1, 5, 6, 4), (1, 11, 12, 4), (1, 36, 45, 1), (1, 37, 42, 2), (1, 47, 90, 1)\}$.
The Object-Z class definitions for `Stream` and `XMLDocument` are defined above. Each XML document is a single rooted tree, which can be accessed from
the root element. An XML document has a collection of elements with positional representation, which have the same docID as the XML document. The distinct tag names of the elements form the tag set of this XML document. Any two XML elements are in one of the four possible positional relationships. The streams partition all the elements, which is described using a secondary attribute [10]. Operation GetStream retrieves the stream of the given tag name. The use of secondary attribute gives an elegant way of describing the operation.

4 XML Twig Query and Related Data Structure

In this section, we present the XML twig query specification and two related data structures: cursor interface and element stack.

Twig queries are represented as trees. Each node in the tree stands for one query node in the original twig query. A query node has the content /tag or //tag, where ‘/’ denotes a child relationship, ‘//’ denotes a descendant relationship, and ‘tag’ is the corresponding label in the twig query. For example, query Q1 in Section 1 can be represented using tree in Figure 3 above.

Matching a twig query against an XML database is of finding all occurrences of the pattern in the database. It is a core operation in XML query processing [6]. Formally, given a twig query Q and an XML database D, a match of Q in D is identified with a mapping from nodes in Q to nodes in D, such that query node predicates are satisfied by the corresponding database nodes (the images under the mapping), and the structural (P-C and A-D) relationships between query nodes are satisfied by the corresponding database nodes. The answer to query Q with n nodes is represented as an n-ary relation where each tuple (d₁, d₂, ..., dₙ) consists of the database nodes that identify a distinct match of Q in D.

A cursor interface C_q is defined for each query node q as the interface to access its stream T_q. Each cursor C_q points to some element in the stream. Initially, the stream of the cursor is empty. It is assigned when the stream data is retrieved from the XML document successfully. The cursor can move to the next element (if any) by invoking operation C_q.Advance. The current stream element can be accessed by operation C_q.GetCurrentElement. The class of Cursor is shown below. There is a variable pointer to remember the current cursor position. The pointer can be advanced before reaching the end of the stream.
We also associate each query node $q$ with a stack $S_q$. Initially, all stacks are empty. During query execution, each stack $S_q$ may cache some stream elements before current cursor $C_q$ and these elements are strictly nested in the stack from bottom to top, i.e. each element is a descendant of the element below it. Each element $e$ in $S_q$ is associated with a pointer pointing to the lowest ancestor in $S_{\text{parent}(q)}$ so as to efficiently access all $e$’s ancestors in $S_{\text{parent}(q)}$. In fact, cached elements in stacks represent the partial solutions that could be further extended to full solutions as the evaluation continues. As described above, first we extend the `Element` class to form `StackElement` class by adding the `ParentStackPointer`. The `ElementStack` is essentially a stack to store stack elements. One extra operation that `ElementStack` supports is `ClearStack`, which removes all elements whose end value is smaller than the input value.
Twig queries are tree structures with multiple branches. Therefore, we can use the recursive tree structure to represent the twig queries. Object containment \[9\] is used to model tree nodes to get rid of cyclic tree structures. Each query node has a tag, which is the same as the tags of elements in its cursor and stack. It has a property \textit{ADJoin}, which specifies whether the structural relationship is child or descendant. The secondary Boolean variable \textit{hasValidMatching} tests whether the elements in the stack forms the match (See Section 4) of the query against the database.

\textit{QueryNode} provides various operations for query evaluation. Child nodes can be added into and retrieved from the \textit{children} variable. Evaluation algorithms can be applied to the query node to produce matching in the element stacks. Before the evaluation, query’s cursor is pointing to the first element in the stream and stack is empty. After the evaluation, the cursor moved to the end of the stream and matched elements are stored in the element stack. In addition, most of the operations in \textit{Cursor} and \textit{ElementStack} are promoted in \textit{QueryNode} class to support the query evaluation. The Object-Z class definition of query node is presented on the next page. Two auxiliary functions \textit{IsRoot} and \textit{IsLeaf} are defined below to test whether the given query node is a root node or leaf node respectively.

<table>
<thead>
<tr>
<th>\textbf{ElementStack}</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (\text{stack}, \text{INIT}, \text{Push}, \text{Pop}, \text{ClearStack}) )</td>
</tr>
</tbody>
</table>

| stack : seq StackElement@ |
| \( \forall i, j : 1..\#\text{stack} \bullet \text{stack}(i).\text{tag} = \text{stack}(j).\text{tag} \) |
| \( i < j \Rightarrow \text{stack}(i) \text{IsAncestorOf} \text{stack}(j) \) |

[Strictly nested relation]

| INIT |
| stack = \langle \rangle |

| ClearStack |
| \( \Delta(\text{stack}) \) |
| position' : \( \mathbb{N}_1 \) |
| stack' \subseteq \text{stack} |
| \( \forall e : \text{ran} \text{stack} \bullet \) |
| \( e \not\in \text{ran} \text{stack'} \Rightarrow \) |
| e.end < position? |

| Push |
| \( \Delta(\text{stack}) \) |
| e? : StackElement |
| stack' = stack \wedge \langle e? \rangle |

| Pop |
| \( \Delta(\text{stack}) \) |
| e! : StackElement |
| stack \neq \langle \rangle |
| stack = stack' \wedge \langle e! \rangle |

\text{IsRoot} : \text{QueryNode} \to \mathbb{B}

\( \forall q : \text{QueryNode} \bullet \text{IsRoot}(q) = (\exists p : \text{QueryNode} \bullet q.\text{parent} = p) \)

\text{IsLeaf} : \text{QueryNode} \to \mathbb{B}

\( \forall q : \text{QueryNode} \bullet \text{IsLeaf}(q) = (q.\text{children} = \langle \rangle) \)
QueryNode

\[\begin{align*}
\text{tag} & : \text{TAG} \\
\text{tags} & : \mathbb{P} \text{TAG} \\
\text{stack} & : \text{ElementStack} \odot \\
\text{cursor} & : \text{Cursor} \odot \\
\text{children} & : \text{seq} \text{QueryNode} \odot \\
\text{parent} & : \text{QueryNode} \\
\text{ADJoin} & : \mathbb{B} \\
\end{align*}\]

[Tags of the whole subtree]

\[\begin{align*}
\text{hasValidMatching}, \text{endOfStream} & : \mathbb{B} \\
\Delta \text{hasValidMatching} & \land \text{endOfStream} \\
\forall c : \text{children} \cdot \forall e : \text{dom} c.\text{stack} \cdot \forall \text{index} : 1..\text{ParentStackPoint} \cdot \forall q_i : \text{children} \cdot \forall e : \text{dom} q_i.\text{stack} \cdot \forall \text{index} : 1..\text{ParentStackPoint} \cdot \\
\text{tags} = \bigcup \{ i : 1..\#\text{children} \cdot \text{children}(i).\text{tags} \} \cup \text{tag} \\
\forall i : 1..\#\text{children} \cdot \text{children}(i).\text{tag} = \text{cursor}.\text{stream}.\text{tag} \land \text{stack}(i).\text{tag} = \text{tag} \\
\forall q_i : \text{children} \cdot q_i.\text{hasValidMatching} \land \forall e : \text{dom} q_i.\text{stack} \cdot \forall \text{index} : 1..\text{ParentStackPoint} \cdot q_i.\text{ADJoin} \Rightarrow q_i.\text{stack}(\text{index}) \text{IsAncestorOf} e \land \\
- q_i.\text{ADJoin} \Rightarrow q_i.\text{stack}(\text{index}) \text{IsParentOf} e \\
\text{endOfStream} = \text{cursor}.\text{endOfStream} \\
\end{align*}\]

\[\begin{align*}
\text{INIT} & \\
\text{children} & = \langle \rangle \\
\text{stack}.\text{INIT} & \land \text{cursor}.\text{INIT} \\
\text{GetTag} & \\
\text{tl} & : \text{TAG} \\
\text{tl} & = \text{tag} \\
\text{AddChild} & \\
\Delta(\text{children}) \\
\text{child} & : \text{QueryNode} \\
\text{child} & \not\in \text{dom} \text{children} \\
\text{children}' & = \text{children} \setminus \{ \text{child} \} \\
\text{EvaluateQuery} & \\
\Delta(\text{cursor}, \text{stack}) \\
\#(\text{stack}.\text{stack}) & = 0 \land \text{cursor}.\text{pointer} = 1 \land \text{endOfStream}' \land \text{hasValidMatching}' \\
\end{align*}\]

GetCursorElement \(\doteq\) cursor.GetCurrentElement
AdvanceCursor \(\doteq\) cursor.Advance
AssignStream \(\doteq\) cursor.AssignStream
PushToStack \(\doteq\) stack.Push
PopFromStack \(\doteq\) stack.Pop
ClearStack \(\doteq\) stack.ClearStack


5 Query Evaluation System

In this section, the whole system for XML twig query evaluation is presented based on the XML document and query node classes described in section 3 and 4.

XML document and query node are independent classes. The query system is built to evaluate twig queries against XML documents. The Object-Z class describing the query system is defined as the following. The query system consists of an XML database which is a set of XML document, and an input query. XML documents can be added and deleted from the system via AddXML and DeleteXML operations respectively. New query can be input into the system via InputQuery operation. The most important operation QueryEvaluation gives the formal requirement of twig query evaluation, which is promoted from QueryNode class.

<table>
<thead>
<tr>
<th>QuerySystem</th>
</tr>
</thead>
<tbody>
<tr>
<td>database : DBXMLDocument</td>
</tr>
<tr>
<td>query : QueryNode</td>
</tr>
<tr>
<td>∃ xml : database • query.tags ∈ xml.tags</td>
</tr>
</tbody>
</table>

| INIT |
| database ≠ ∅ |
| ∀ xml : database • xml.INIT |

| InputQuery |
| Δ(query) |
| q? : QueryNode |
| query' = q? |

| AddXML |
| Δ(database) |
| db? : XMLDocument |
| db? ∉ database |
| database' = database ∪ {db?} |

| RemoveXML |
| Δ(database) |
| db? : XMLDocument |
| db? ∈ database |
| database' = database - {db?} |

| RetrieveStream | (query.GetTag ∧ query.AssignStream) |
| [xml? : Database] • xml?.GetStream |

| EvaluateQuery | query.Evaluation |

The query evaluation process is described like following: first of all, a new query is input into the system. At this stage, all cursors’ streams of query nodes are not assigned and stacks of query nodes are empty. Then operation RetrieveStream is applied to build up the cursors of query by retrieving the stream from the destination XML document to query streams according to the tag name. After that, we can apply a particular query evaluation algorithm [6,16] to evaluate the query by invoking QueryEvaluation operation. After the evaluation, cursor of the query
Liu and Sun

root reaches the end and all valid matchings are stored in the stacks of the query nodes. Finally stack elements are merged and output to form the final solutions. Merging multiple lists of sorted path solutions is a simple practice of a multi-way merge join [27].

6 Query Evaluation Algorithms

The Object-Z specification shown in the last section is well designed with high extensibility and reusability. Different evaluation algorithms can be built based on the general interface and data structures defined in class QueryNode. In this section, we present two evaluation algorithms PathStack [6] and TSGeneric [16], which produce all matchings of a twig query against an XML document. PathStack is a decomposition method, while TSGeneric uses a holistic approach. We show that both of them can be expressed precisely based on the data structures and operations defined in the Object-Z specification.

PathStack is a straightforward way of computing answers to a twig query pattern by decomposing the twig query into multiple path queries. Each path query is evaluated, and then the solutions of the path queries are merged to form the answers of the twig query. For simplicity, we ignore the decomposition and merge of PathStack. The main algorithm that finds solutions of a path query is presented in the following:

Algorithm PathStackMain(root)
1: while (not root.endOfStream)
2:  q_min = GetMinSource(root);
3:  for each q_i in root.children
4:      while (#(q_i.stack) != 0 &&
       tail(q_i.stack).start < q_i.cursor.end)
5:         q_i.PopFromStack();
6:  q_min.PushToStack(q_min.GetCursorElement(), #(q_min.parent.stack));
7:  q_min.AdvanceCursor();
8:  if (IsLeaf(q_min))
9:     OutputPathSolutions(q_min);
10: q_min.PopFromStack();

The key idea of Algorithm PathStack is to repeatedly construct element stack of partial and total answers to the query path pattern, by traversing the stream. Thus, the path query nodes will be matched from the query root down to the query leaf. In line 2, GetMinSource method returns the descendant query node of q along the path query, whose current cursor element has the minimum start value. The details of GetMinSource function is ignored for the simplicity. Interested readers could construct this function based on the above description. Lines 3-5 remove partial answers from the stacks that cannot be extended to total answers, given knowledge of the next stream node to be processed. Line 6 augments the partial answers encoded in the stacks with the new stream node. Line 7 advances the Cursor to next element. OutputPathSolutions is invoked by Algorithm PathStack (lines 8-10) to output these answers.

PathStack has a significant disadvantage: many intermediate results may not contribute to the final answers. In order to solve this problem, TSGeneric is pro-
posed by [6,16]. *TSGeneric* operates in two phases. Firstly, it repeatedly calls the *GetQNode* algorithm to get the next “good” query node for processing. Secondly, selected query node is stored in the element stacks or discarded.

*GetQNode* is similar to *GetMinSource*, which returns a subtree node of root who has the minimal start value. Because the twig query is considered as a whole, this gives idea of the holistic approach. The detail of the *GetQNode* algorithm is shown below:

```
Algorithm GetQNode(q)
1: if (IsLeaf(q))
2:   return q;
3: for each q_i in q.children
4:   n_i = GetQNode(q_i);
5: if (n_i != q_i)
6:   return n_i;
7: n_min = getMinSource(q);
8: if (q.cursor.start < n_min.cursor.start)
9:   return q;
10: else
11:   return q_min;
```

The following is the *TSGeneric* algorithm. After getting the next good node using *GetQNode* (line 2), elements are popped from the q’s parent stack and stack S_q (lines 4-5), because they do not contribute to new outputs. C_q is then processed if q is either a root node or C_q has ancestor elements in the parent stack S_parent(q), which means it is possible to contribute to the final matches. If q is a leaf output node, all path solutions for element C_q (line 10) are outputted.

```
Algorithm TSGeneric
1: while (not root.endOfStream)
2:   q = GetQNode(root);
3: if (not IsRoot(q))
4:   q.CleanStack(q.cursor.start);
5:   q.CleanStack(q.cursor.start);
6: if (IsRoot(q) or #(q.parent.stack) != 0)
7: if (not IsLeaf(q))
8:   q.PushToStack(q.GetCursorElement(), #(q.parent.stack));
9: else
10:   OutputPathSolutions(q);
11: q.AdvanceCursor();
```

With the Object-Z specification of the XML data structure and twig query, the twig query evaluation becomes straightforward by using the operations provided by the Object-Z classes. Different evaluation algorithms can be constructed using different methods to access the data stream. An implementation of a query evaluation algorithm is correct as long as it fulfills the operation *QueryNode* defined in class *QuerySystem*. Nevertheless, the design of the algorithm is independent of the implementation of the XML data model and hence focuses only on the algorithmic aspects. For example, *Cursor* class can be implemented using either simple file cursor, XB-tree cursor [6] or XR-tree cursor [15]. Efficient index access methods can skip the stream quickly, hence significantly improve the evaluation.
7 Conclusion

In this paper, we presented an Object-Z specification of the XML twig query evaluation system. A reusable formal specification of the general XML data model is constructed, based on which the XML twig query evaluation is formalized incrementally. This specification gives a concise and logical description of relationships between XML data and twig queries. Having this specification, the twig query evaluation becomes straightforward. Two highly complicated twig query evaluation algorithms are presented based on the data model.

Our work in this paper serves as an example to show that extensible formal specification languages like Object-Z may help complicated algorithmic designs for systems with complex data structures.

References


