The XTATIC Experience

Vladimir Gapeyev   Michael Y. Levin   Benjamin C. Pierce   Alan Schmitt

Abstract

XTATIC is a lightweight extension of $C^\sharp$ with native support for statically typed XML processing. It features XML trees as built-in values, a refined type system based on *regular types* in the style of XDUCE, and "tree grep"-style *regular patterns* for traversing and manipulating XML.

Previous papers on XTATIC have reported results on a number of specific technical issues: basic theoretical properties of an idealized core language, novel compilation algorithms for regular pattern matching, and efficient runtime support for XML processing in the style encouraged by XTATIC. The aim of the present paper is to discuss XTATIC—less formally and more holistically—from the perspective of language design. We survey the most significant issues we faced in the design process and evaluate the choices we have made in addressing them.

```
    <person>                              [[ <person>
      <name>Haruo Hosoya</name>              <name>'Haruo Hosoya'</name>
      <email>hahasoya</email>                <email>'hahasoya'</email>
    </person>                              </person>
    <person>                              <person>
      <name>Jerome Vouillon</name>          <name>'Jerome Vouillon'</name>
      <tel>123</tel>                         <tel>'123'</tel>
    </person>                              </person> ]]
```

Figure 1: XML and XTATIC concrete syntax for an address book

1   Introduction

The fundamental goal of XTATIC is to add native support for statically typed XML processing to a mainstream object-oriented language. Its guiding design principles are *simplicity* (the extension should be lightweight and easy for programmers to understand); *integration* (it should fit cleanly and inter-operate smoothly with the host language at all levels—type system, control structures, run-time system, VM, libraries, etc.); and *flexibility* (its mechanisms for manipulating and typing XML should support a full spectrum of processing styles, from dynamic investigation of documents of unknown or partially known types to fully checked processing of documents for which complete type information is known, and should be robust in the face of program and schema evolution).

XTATIC adds to $C^\sharp$ just two critical mechanisms, both adapted from XDUCE [12, 14]: *regular types* for XML [15] and *regular patterns* for matching and destructing XML data in the style of "grep for trees" [13].

In previous papers, we have reported on a number of specific aspects of XTATIC's design and implementation: basic theoretical properties of an idealized core language [7], high-performance compilation algorithms for regular pattern matching [18, 19], and efficient runtime support for XML processing in the style encouraged by XTATIC [6]. Our aim here is to draw together our experience with the language design as a whole. After a small example (Section 2) showing the key features of the language—regular types and regular pattern matching—we discuss in detail the most important issues in the design of the type system (Section 3), the run-time values used to represent XML data (Section 4), and the mechanisms for pattern matching (Section 5). Section 6 sketches some language extensions currently being considered, and Section 7 briefly surveys a number of related language designs.

We hope that this high-level evaluation of XTATIC will be useful to designers of future languages with similar goals.

2   A Taste of Xtatic

Consider the document fragment of Figure 1—a sequence of two entries from an address book—given here side-by side in XML and XTATIC concrete syntax. XTATIC's syntax for this document is very close to XML, the only differences being the outer double brackets used to segregate the world of XML values and types from the regular syntax of $C^\sharp$, and backquotes, which distinguish PCDATA (XML textual data) from arbitrary XTATIC expressions yielding XML elements.[1]

One possible type for the above value is a list of persons, each containing a name, an optional phone number, and a list of emails:

---

[1]Our concrete syntax differs from XQuery's at this point: we take "computed content" as the default and explicitly mark constant strings in XML values, while XQuery takes constant pcdata as the default and indicates computed content by wrapping it in curly braces—e.g., `<a>3</a>` vs. `<a>{1 + 2}</a>`. At first glance, XQuery's design is attractive, since it allows XML values to be cut and pasted into XQuery programs verbatim. However, it turns out not to work well in XTATIC, where we need concrete syntax not only for values but also for types. (XQuery does not: complex type expressions are supposed to be written down in separate files, in the completely different syntax of W3C Schema.) Making the opposite choice allows us to write `<a>B</>`, rather than `<a>{B}</>`, for the type of `<a>` elements whose content is described by the type named B.

```
<person> <name>pcdata</> <tel>pcdata</>? <email>pcdata</>* </person>*
```

The type constructor "?" marks optional components, and "*" repeated sub-sequences. XTATIC also includes the type constructor "|" for non-disjoint unions of types. The shorthand `</>` is a closing bracket matching an arbitrarily named opening bracket. Every regular type in XTATIC denotes a set of sequences. Concatenation of sequences (and sequence types) is written either as simple juxtaposition or (for readability) with a comma. The constructors "*" and "?" bind more strongly than ",", which is stronger than "|". The type "pcdata" describes sequences of characters.

Types can be given names that may be referred to in other types. For example, in the presence of these definitions

```
regtype Name  [[ <name>pcdata</> ]]
regtype Tel   [[ <tel>pcdata</>  ]]
regtype Email [[ <email>pcdata</> ]]
regtype TPers [[ <person> Name Tel </> ]]
regtype APers [[ <person> Name Tel? Email* </> ]]
```

our address book can be given the type `APers*`.

Between XML types, subtyping is precisely regular tree language inclusion. For example, every value of type `TPers` can also be described by the type `APers`, so we have `TPers <: APers`. Between object types, subtyping follows the rules of $C^\sharp$. (We will have more to say about subtyping in the following section.)

Types and subtyping are also the foundation of *regular pattern matching*, which generalizes both the `switch` statement of $C^\sharp$ and the algebraic pattern matching popularized by functional languages such as ML. For instance, the following method extracts a sequence of type `TPers` from a sequence of type `APers`, removing persons that do not have a phone number and eliding emails.

```
static [[ TPers* ]] addrbook ([[ APers* ]] ps) {
  [[ TPers* ]] res = [[ ]];
  bool cont = true;
  while (cont) {
    match (ps) {
      case [[ <person> <name>any</> n, <tel>any</> t, any </>, any rest ]]:
        res = [[ res, <person> n, t </> ]];
        ps = rest;
      case [[ <person> any </person>, any rest ]]:
        ps = rest;
      case [[ ]]:
        cont = false;
  } }
  return res; }
```

3 Types

Many proposals for XML processing extensions in mainstream languages adopt a *data binding* approach, in which XML types are encoded in terms of the type structures already available in the host language. Others—notably XEN [20, 21]—generalize the host language's object and sequence types so that types for XML become just a special case. XTATIC steers a middle path, embodying a nearly orthogonal integration of XML types and object types that offers maximum flexibility and expressiveness for the former with minimal impact on the latter.

3.1 Regular Tree Types

The tree data model of XML gives rise to a natural notion of *regular tree grammars* generalizing familiar regular expressions on words. Murata, Lee, and Mani [23] identify four increasingly expressive classes of regular tree grammars:

- *Local tree grammars* adopt the restriction that, wherever a given tag occurs in a tree grammar, its *content model* (the sequence of types of its subtrees) must always be the same. This class corresponds roughly to DTDs. It is sufficiently expressive for many of the simpler uses of XML and has been used for static analysis in the programming language XACT [17].

- *Single-type tree grammars* enforce a weaker restriction, requiring that multiple occurrences of a tag *as immediate children of the same parent node* must have identical content models. The W3C's XML Schema [24] standard and, consequently, the type systems of XQuery [26] and XJ [9] are based on this class.

- *Restrained-competition tree grammars* relax the restriction yet further, allowing two trees with equal tags under the same parent to have different content models, as long as the earlier part of the content unambiguously determines which is expected at each point. (For example, `<a> <b>S</> <b>T</> </>` is OK, but `<a> (<b>S</> | <b>T</>) </>` is not.)

- General *regular tree grammars* allow arbitrary combinations of elements with different content models. This class forms the basis of the RELAXNG schema standard [3] and of the type systems of a number of programming languages, including XDUCE [13] and CDUCE [1].

The advantage of staying low in this hierarchy is simplicity and efficiency of implementation: as expressiveness increases, so do the complexity of language membership (validation) and subtyping algorithms. The first two classes can be served by simple adaptations of ordinary *word automata*, while the latter two classes require more complicated *tree automata*. Fortunately, experience with RELAXNG, XDUCE, and CDUCE shows that the algorithms available for unrestricted regular grammars, though more complex, are still implementable and practical in usage. Moreover, the more powerful grammar classes have some significant advantages of their own—in particular, better closure properties. While all of the classes are closed under intersection (which is useful for inferring types of pattern variables; see Section 5.3), only unrestricted regular tree grammars are closed under union (useful for conditionals and `match` expressions), difference (which improves the precision of type inference, as explained in 5.3), and concatenation.

On balance, we feel that both single-type tree grammars and general regular tree grammars offer reasonable foundations for new programming language designs. (Local tree grammars are too limited, and restrained competition grammars are harder to understand without being much more tractable than full regular tree grammars.) In designing XTATIC, we have chosen unrestricted tree grammars for their power and simplicity, accepting the additional implementation burden that they impose. We believe that a similar design based on single-type tree grammars could also be a viable alternative.

3.2   Regular Object Types

As we saw in Section 2, regular tree types and sequence values integrate seamlessly in XTATIC into the world of proper $C^\sharp$: wherever a $C^\sharp$ type or value is expected, a regular type or a sequence value can appear, enclosed in `[[ ... ]]` brackets. We could stop there and disallow integration in the opposite direction, requiring that the contents of the `[[ ... ]]` brackets be pure XML not containing general $C^\sharp$ values. It turns out, however, that it is easy to allow $C^\sharp$ values inside XML sequences. A significant side-benefit of this generality is that we can use regular patterns over trees of objects to emulate the elegant and concise "datatypes and algebraic pattern matching" programming idioms found in modern functional languages such as ML.

At the beginning of the XTATIC design, we set out to allow $C^\sharp$ objects directly as members of XML sequence values. However, attempts to formalize this scheme encountered a number of problems. To see why this is so, notice that it is essential for sequence values to be of some $C^\sharp$ object type so that they can be used with standard generic $C^\sharp$ libraries such as `Stack` and `Hashtable`. To satisfy this requirement, we designate a special class `Seq` extending `object` to denote all sequence values. Semantically, `Seq` is equivalent to the regular type `[[any]]`. Now, let `o` be a `Seq` object and consider a sequence value `[[o]]`. Does it denote a singleton sequence containing `o` or the underlying sequence represented by `o`? Sorting these issues out consistently is not straightforward.

4

```
A   ::=                      XTATIC type          T   ::=                      tree type
    |   C                    class type               |   ()                   empty sequence
    |   [[T]]                tree type                |   <(A)> T </>          tree
                                                      |   T,T                  concatenation
d   ::=                      regular type declaration |   T|T                  alternative
    |   regtype X [[T]]                               |   T*                   type repetition
                                                      |   X                    type name
```

Figure 2: Syntax of regular object types

Instead, we employ a much simpler integration strategy by allowing $C^\sharp$ values only as *labels* of sequence elements. Now, assuming that o_1 and o_2 are `Seq` objects, `[[<(o₁)/> <(o₂)/>]]` is a two-element sequence containing empty elements labeled by o_1 and o_2; whereas `[[o₁, o₂]]` is a sequence containing the concatenation of o_1's and o_2's sequences. Note, that the latter expression would be ill-typed if either of the objects were not of class `Seq`.

Figure 2 shows the syntax of regular object types following the intuition discussed above. Any object can appear as the label of a sequence element, and hence, the type `[[any]]` describing all sequence values can be defined recursively as `[[<(object)>any</>*]]`.

Not all sequence values are representations of XML documents. For instance, `<(1)/>` is a sequence value that is not XML, since XML elements cannot be labeled by integers. To characterize the set of XML sequences, we introduce a special class `Tag` extending `object` that describes precisely the set of proper XML element labels. Conceptually, every XML label corresponds to a distinct subclass of `Tag`. For example, the XML fragment `<author/>` corresponds to to the sequence value `<(new Tag_author())/>` and is classified by the type `<(Tag_author)/>`.

With this in mind, the type of all XML fragments and the type of purely textual XML fragments can be defined by regular types `xml` and `pcdata` as follows:

```
regtype xml [[( <(char)/> | <(Tag)>xml</> )*]]
regtype pcdata [[<(char)/>*]]
```

As shown in Section 2, the concrete syntax of XTATIC provides lightweight XML-like notations for proper XML sequences and text and regular types describing them. Such values and types are treated specially in the implementation. For instance, subclasses of `Tag` are not created physically, but rather a more compact and efficient representation is used. (See [6] for details.)

3.3 Structural vs. Nominal

One of the most significant design choices in XTATIC was between structural and nominal treatments of XML data. The structural vs. nominal distinction is usually discussed in terms of the subtype relation: in the nominal case, the typechecker deals principally with *names* of types and subtyping is allowed only when a subsumption relation between two names has been explicitly declared by the programmer, while, in the structural case, type names are ignored and subsumption relations between types are determined automatically. But the rubber really meets the road at the level of *values*: in a nominal language, each run-time value is marked with a type name, which is then also used as the name of the type of this value during static type-checking. Typically, constructs are provided in the programming language that define and/or rely on an ordering on the type names (e.g., subclass definitions in conventional object-oriented languages), and this ordering is lifted to named types in the form of a subsumption relation.

For a language with XTATIC's goals, treating XML data in a nominal style—marking each tree node with a type name—is an attractive option. For one thing, it makes it easy to construct efficient type-checking algorithms, since there is never any need to examine the "right hand sides" of defined types to determine

whether one is a subtype of another. Similarly, the type names can be put to good use for efficiently checking that a value belongs to a type at run time. This can be used in the implementation of many useful language features, including XDuce-style regular pattern matching, downcasting, re-validation of XML trees for which type information has been lost (e.g., by storing and later retrieving them from a generic collection), and—obviously—XPath 2.0's primitive for matching a node if it is marked with a particular type name. Another argument for nominal subtyping is that the industry standard "type system" for XML data, W3C XML Schema, is (mostly) nominal, offering various mechanisms for declaring subtypes explicitly. XQuery, whose type system is closely based on W3C XML Schema, has adopted the nominal approach for these reasons. A final argument for using nominal subtyping in Xtatic is that this would yield a pleasing similarity to C$^\sharp$'s nominal treatment of objects and their types.

Nonetheless, we have chosen in Xtatic to treat XML trees and their types structurally. Doing so makes the implementation of the typechecker, runtime system, and pattern match compiler somewhat harder (our experience with addressing these challenges is reported in [15, 6, 18, 19]), but yields some significant benefits. Most importantly, it avoids what Michael Kay and others have called the "THE Schema Fallacy"—i.e., the presumption that a given XML value will always be thought of as belonging to one specific type. If types are thought of as just *descriptions* of data, rather than being embedded in the data itself, then it makes perfect sense to think of the same data structure as satisfying multiple descriptions at different points in a program, each specifying just what is needed for the task at hand. This ability to adopt multiple views of the same data can play a useful role in software reusability. (Instead of writing a method that extracts the `<name>` elements from all the entries in an address book, we can write a generic method that extracts `<name>` elements from *any* sequence of data items, each containing at least a `<name>` element. This generic method can be given a type (namely `<(Tag)/>any, Name, any</>*`) that precisely describes the requirement on its input, and the validity of passing it an address book can be checked automatically, with no need for a "re-validation" step—i.e., no unsafe typecast).

Inferring subtyping automatically can also help sidestep some well-known software engineering traps. For example, if we have two existing data structures with similar schemas, it may be useful to write a program that can work over both of them—i.e., that accepts a common supertype as input; but doing this in a nominal setting may in general involve adding supertype declarations to both existing schemas, which may not even be possible if, for example, they are controlled by another organization. Similarly, avoiding the requirement of writing explicit subtype declarations removes a potential source of friction as programs and schemas evolve.

3.4   Attributes

Static typing for attributes is an area where the current Xtatic design falls short: despite several attempts, we have not yet been able to find a treatment that satisfies all of our requirements. For the moment, we have therefore adopted a simple *un*typed scheme for building values with attributes and for pattern matching against attributes. We briefly describe the difficulties we have encountered and sketch our current stopgap solution.

A type system for XML attributes should support a way of specifying both closed and open sets of attribute/value pairs, optional and required attributes, and well-defined boolean operations such as union and intersection of attribute types.

Hosoya and Murata's work on *attribute-element constraints* [11] describes a powerful attribute type system that has all of the above features and also allows the programmer to express dependencies between element and attribute children of a parent element. Their algorithms, however, are rather complex.

Alternatively, one may begin from a conventional record type system and extend it with the features listed above. We have experimented with this approach and found that it does not seem to lead to anything much simpler than Hosoya and Murata's solution: because of arbitrary unions, the subtyping algorithms in both approaches turn out to be surprisingly similar.

Certain use cases, however, require such dynamic traversal and reconstruction of attributes, that even these powerful type systems may not be able to provide sufficient static guarantees. Consider a program that uppercases the value of every attribute. Such anonymous style of processing is not supported by the

attribute-element constraint or the record-based type systems. To make it possible, these type systems may need to provide a dynamically-typed mechanism for converting attribute sets into conventional sequences and back, similar to the XTATIC's current solution described here.

As a temporary measure, XTATIC adapts a simple dynamic approach: attributes are part of values and can be written in patterns, but they are ignored in types; as far as the type system is concerned, any element may have any collection of attributes. XTATIC attribute patterns provide a way of extracting values of specified attributes as well as the remaining attribute/value pairs. The latter are packaged into a conventional sequence value of the form `<attr`$_1$`>val`$_1$`</>` `<attr`$_2$`>val`$_2$`</>` ... `<attr`$_n$`>val`$_n$`</>` that can be dynamically examined using conventional element pattern matching. Sequence values of this form can also be used to specify attributes of a newly created element. To ensure well-formedness, the run-time system must perform a dynamic test verifying that the given sequence does not contain repeating element names.

3.5 Overloading

Naturally XTATIC extends $C^\sharp$ method overloading to support regular types in method signatures. For example, if a class contains two methods `void f([[Person*]] x)` and `void f([[Person+]] x)`, then a call `f(x)` is resolved to one or the other depending on the static type of `x` according to the standard overloading resolution rules modified to use XTATIC subtyping instead of $C^\sharp$ subclassing.

Such extension of overloading requires some additional work on the part of the compiler. Since XTATIC is compiled into $C^\sharp$ homogeneously—every regular type is translated into a single $C^\sharp$ type `Seq`, and every sequence value is translated into an object of this class—it is possible that XTATIC methods with different signatures will be compiled into $C^\sharp$ methods with the same signature, resulting in an illegal $C^\sharp$ program. For instance, both of the above signatures map to `void f(Seq x)`.

We resolve this problem by generating new names for all methods that have arguments of types more precise than `[[any]]`. The purpose of renaming is to encode the regular types information from the original XTATIC method signature into the name of the compiled $C^\sharp$ method in such a way that $C^\sharp$ overloading resolution on renamed methods behaves the same as would XTATIC overloading resolution. In particular, we ensure that a method that overrides or hides a method from a base class receives the same mangled name as the base method. To make this arrangement work with independent type-checking of separate assemblies, our compiler generates for each assembly an auxiliary structure containing, along with the mangled names, the original method signatures with regular types. This structure is used when typechecking an XTATIC program that references the assembly.

A $C^\sharp$ program compiled against an XTATIC library is expected to refer only to non-mangled method names—that is, method names with regular type arguments at most as precise as `[[any]]`. Consequently, if an XTATIC library wants to export a method operating on values of a regular type that is more precise than `[[any]]` to pure $C^\sharp$ code, it can provide a wrapper method accepting values of type `[[any]]`, explicitly casting them to the appropriate regular type and then calling the original method. This approach ensures safety by eliminating the possibility of unchecked invocation of XTATIC methods with ill-typed parameters from arbitrary $C^\sharp$ code.

4 Values

This section describes and motivates XTATIC's design of sequence values. We justify our decision to use immutable values and discuss how this choice interacts with various XML inspection styles and $C^\sharp$ programming idioms. We also cover two enhancements to the basic model of values that facilitate efficient downcasting and on-demand translation from legacy XML representations such as DOM.

4.1 Immutability

A fundamental design goal of XTATIC is ensuring type-safe manipulation of XML data. This implies that either XML structures be immutable, or that updatable parts of XML structure be marked with "ref types"

to prevent subtyping. For the sake of simplicity we have chosen to pursue the first approach and make every XML value immutable. Other languages, such as XJ [9], relax the static type safety requirement and do dynamic checks when mutation of XML data occurs.

Immutability, in turn, demands that we choose a representation that supports a great deal of sharing in order to retain acceptable memory requirements. This sharing prevents the use of doubly-linked trees for a representation of XML values, in particular back-pointers in the style of DOM, thus hindering some value inspection mechanisms in XTATIC: in order to support XPath's backward axes, it would be necessary to maintain a context representing the path from the root of the value to the current position, as is done in XACT [17].

The design choice of immutability also needs to be reconciled with the natural imperative programming style of $C^\sharp$ in ways that can still be implemented efficiently. We now describe how we handle some typical cases, contrasting a purely imperative approach with XTATIC's approach.

In a first example of imperative style, the author of a book is modified from "John" to "Bob." Using mutable XML values, one could do the following:

```
[[book]] doc = LoadXml("file.xml");
XmlNode n = doc.FindFirstNode("book/chapter/author = 'John'");
n.text = "Bob";
```

In XTATIC, one needs to capture the context where the change occurs, and recreate it:

```
[[book]] doc = LoadXml("file.xml");
match (doc) {
  case [[<book>any c, <chapter>any a, <author>'John'</>, any b</chapter>, any d</book>]]:
    return [[<book>c, <chapter>a, <author>'Bob'</>, b</chapter>, d</book>]];
}
```

Because of sharing, only the path from the root of the value to the point where it is modified needs to be copied: XML values c, a, b, and d are not copied.

Another imperative programming idiom consists of creating a sequence of XML values by repeatedly appending XML elements to some accumulator using a `while` loop. For instance, one may create a list of persons the following way:

```
[[ Person* ]] p = [[ ]];
while (some_condition) {
  p = p, <person><Name>create_name ()</></>;
}
```

An efficient way of implementing such a construction when mutation is available is by keeping a pointer to the last element of the output sequence and updating its contents whenever a new element must be appended. In functional programming, an efficient idiom for the same task first builds a reverse sequence by prepending elements, and then reverses it.

In order to retain the intuitive concatenation order as shown in the code above while avoiding turning linear algorithms into quadratic ones, XTATIC uses carefully designed lazy data structures and algorithms that delay creation of concrete values until they are inspected elsewhere. Consult our previous paper [6] for a more detailed account of run-time data structures in XTATIC.

4.2   Fast Downcasting

Part of the appeal of XTATIC is that it allows programmers to use familiar $C^\sharp$ libraries to store and manipulate XML values; in particular, XML values can be stored in generic collections such as `Hashtable` and `Stack`. However, values extracted from such containers have type `object`, and generally need to be cast down to the intended type. In pure $C^\sharp$ this operation incurs only a constant time overhead, but in XTATIC, downcasting to a regular type may involve an expensive structural traversal of the entire value (cf. Section 3.3). One way to avoid this overhead is to *stamp* a sequence value with a representation of its type (upon putting the value

into a collection) and perform a run-time stamp comparison rather than full re-validation during downcasting (upon receiving the value from the collection). Our design places this stamping under programmer's control.

We extend the source language with a stamping construct, written `<[[T]]>e` ("stamp `e` with regular type `T`"). An expression of this form is well typed if `e` has static type `T`; in this case, the result type of the whole expression is `object`.[2] Casting a stamped value can then be done in constant time as long as the type used in the cast expression is syntactically identical to the type used in the stamping expression. Casting to any other type, however, must fall back to the general pattern-matching algorithm, which dynamically re-validates the value.

In our design, the burden of type stamping is placed on the programmer. We have experimented with alternative designs in which stamping is performed silently—either by adding a stamp whenever a sequence value is upcast to type `object` or by including a type stamp in every sequence object. However, we have not found a design in which the performance costs of stamping and stamp checking are acceptably predictable.

4.3   Legacy Representations

XTATIC modules are expected to be useful in applications built in other .NET languages with the use of extensive .NET libraries. The latter already contain support for XML, collected in the `System.Xml` namespace. It is essential for XTATIC's XML manipulation facilities to interoperate smoothly with native .NET XML representations and, conversely, XTATIC XML data to be accessible from XTATIC-agnostic C$^\sharp$ code. We have explored the former direction of this two-sided interoperability problem by designing support for DOM, one popular XML representation available in .NET.

A straightforward solution for accessing DOM from XTATIC would be to first translate any DOM data of interest into our representation in its entirety and then proceed to working with XTATIC's native representation. This is wasteful, however, if an XTATIC program ends up accessing only a small portion of the document. A better idea is to perform the translation from DOM lazily as more and more fragments of the DOM value are accessed by XTATIC code. XTATIC provides the method `[[any]] ImportDOM(XmlNode x)` that takes a raw DOM node and wraps it into a datastructure that is visible to a program as an XTATIC sequence value. Matching a wrapped value against a regular pattern results in copying its matched fragments into the native (non-wrapped) XTATIC representation thus abandoning aliasing with its mutable DOM predecessor. Consider the following example:

```
match (ImportDOM(...)) {
  case [[ <book>any</book> b, any rest ]]: ...;
  case [[ any ]]: ...;
}
```

In the right-hand side of the first clause, the type for `b` is `<book>any</>`. If `b` was still represented by the original DOM fragment, this type assumption could be violated by a program that followed the pattern match by modifying the element's tag (via DOM interface) from `<book>` to, say, `<journal>`. Thus replicating the fragments of DOM that have been pattern-matched by XTATIC is necessary. The fragments that have not been inspected yet, however, can stay in their original DOM representation. For example, modifying the DOM fragment corresponding to `rest` cannot violate any static assumptions since the variable has type `[[any]]`, which accurately describes the corresponding DOM fragment regardless of whether any modification happens. The same applies to the fragment matched by `any` in `b`. Even though the original DOM structure may be destructively updated at any time, the only modifications that are visible to XTATIC are those that happen before pattern matching inspects the fragments in question.

5   Pattern Matching

This section discusses XTATIC's *regular pattern matching* facilities for inspecting sequence and tree values, determining their shape, and extracting their parts.

---

[2]Giving stamped values type `object` ensures that, at run-time, such values will never appear as part of other sequences. This makes implementation of sequence operations simpler and more efficient.

Q	::=		XTATIC pattern	P	::=		regular pattern
	|	C	class pattern		|	X	type name
	|	[[P]]	regular pattern		|	<(Q)> P </>	tree
	|	Q x	$C^\sharp$ var binding		|	()	empty sequence
					|	P,P	concatenation
d	::=		pattern declaration		|	P|P	alternative
	|	regpat X [[P]]			|	T*	type repetition
					|	P x	regular var binding

Figure 3: Syntax of regular patterns

Regular patterns are both powerful and pleasingly simple to formalize: they are just regular types decorated with binders. Our experience using XTATIC has shown them to be very convenient for a broad range of tasks. For some jobs, however, regular patterns can be rather inconvenient and another style of value inspection—embodied in the popular XML XPATH standard [25]—works better. To accommodate paths-based processing in XTATIC, we currently study a desugaring technique that converts an important fragment of XPATH ("downward axis" paths) into equivalent regular patterns, thus giving us the best of both worlds. We sketch our progress in this area.

We next describe type inference for XTATIC's regular patterns, concentrating in particular on the issue of *precision* of the inference algorithm—an important point where XTATIC differs from XDUCE. We conclude the section by addressing the issue of *schema evolution*, i.e., the question of how robust programs are when the type of the data they manipulate changes.

5.1 Regular Patterns and Paths

The syntax of regular patterns (Figure 3) mimics the syntax of regular object types, additionally providing constructs for binding sequence values and objects within element labels. For example, the pattern [[<a/>*, <b/>* x, <c/>*]] matches sequences composed of zero or more a-elements followed by zero or more b-elements followed by zero or more c-elements. The middle sub-sequence containing all the b-elements is extracted into x. For an example of binding within labels consider the pattern [[<(B x)/>]]. It matches singleton sequences whose element is labeled by an object o of class B and binds o to x.

Regular patterns are especially convenient for "horizontal" inspection of XML sequences. This kind of inspection can be exemplified by the following program fragment. Consider an HTML table that contains two sets of rows with a distinctive separator row between them, i.e. the table's contents has type row*, separator, row* where row and separator are defined as follows:

```
regtype separator [[ <tr> <td>pcdata</> <td>pcdata</>, any </> ]]
regtype row [[ <tr> <td>pcdata</> <td>pcdata, <a>pcdata</></>, any </> ]]
```

Observe that there is only a slight difference between a row and a separator—the former has a hyperlink in its second cell while the latter does not. Let table contain an HTML table whose contents satisfies the above type. Using regular patterns, we can extract the two sets of rows in one line of XTATIC (this statement is desugared into a match expression with one clause whose right hand side is the rest of the program):

```
[[<table> row* x, separator, row* y </>]] = table;
```

Paths, as in XPATH language, are inspired by a file-system-like style of hierarchical navigation. We sketch here a fragment of XPATH (the "downward axes" fragment) that can be supported by XTATIC data model and, coincidentally, consists of the more frequently used XPATH features. Here is a typical XPATH expression: table/tr/td. It finds all the table elements at the top level of the current document, locates their tr subelements, and extracts all of their td children. In addition to parent-to-child navigation, XPATH

provides a way of reaching descendants of the current element. For instance, the query `table//a` finds all hyperlinks occurring somewhere inside top-level tables.

XPATH *predicates* allow the programmer filter potential solutions against conditions that can be expressed as paths. The XPATH query `table/tr[td//a]`, for example, locates all the rows of the top-level tables such that some of their cells have a hyperlink descendant. The predicate is delimited by brackets; it is an additional condition on the rows that are returned by the query.

As the above examples indicate, paths is particularly suitable for "vertical" inspection of XML, providing a natural mechanism for specifying parent/child and parent/descendant constraints on the input values. The two styles of value inspection—regular patterns and paths—are complementary: a natural pattern matching task is cumbersome to accomplish by paths and vice versa. The following section shows how subset of XPATH sketched here can be desugared into regular patterns.

5.2   Implementing Paths Using Regular Patterns

The main difference between the semantics of paths and regular patterns is that an path query returns multiple answers while a regular pattern is used to match a value and return at most one set of bindings associating variables with fragments of the input. For further clarification, consider the pattern `[[any x, <b/>, any]]` that matches sequences containing a `b` element and extracts the preceding prefix into `x`. We say that this pattern is *ambiguous*: given a sequence with multiple `b` elements, there are multiple possible bindings for `x`. Nevertheless, when used in a `match` statement in XTATIC, the above pattern will only compute at most one binding. (The particular binding that is computed is left unspecified in the current implementation of XTATIC; in the future we plan to investigate the formalization and implementation of several disambiguation policies.)

To get us closer to paths semantics, we propose an additional pattern matching construct `iterate` that, rather than choosing one way of matching an ambiguous pattern, explores all possible matchings and computes all possible variable bindings. Here is an example of `iterate` with the ambiguous pattern mentioned above:

```
iterate ([[<a/><b/><a/><b/>]]) matching [[any x, <b/>, any]] {
  System.Console.WriteLine(x);
}
```

An `iterate` statement consists of an expression that evaluates to the input value, a pattern, and a body that is executed for every possible match of the value against the pattern. The above fragment will print `<a/>` and `<a/><b/><a/>`.

A variant of `iterate` statement uses path queries instead of ambiguous regular patterns. For instance, the following statement prints all the `td` elements that are children of the `tr` elements that are children of the top-level `table` elements found in the document fragment stored in `table`:

```
iterate (table) matching x at path [[table/tr/td]] {
  System.Console.WriteLine(x);
}
```

Notice that since a path by itself does not define a results-bound variable that can be referred to in the body of `iterate`, this variant of the statement uses special syntax to introduce the variable separately (variable `x` in the example). Such a variable binds to the results of the path query (`td` elements in the example).

The downward-axes subset of XPATH supported by XTATIC is described by the following grammar where `p`, `s`, `q`, `a`, `L` and `l` range over path queries, query steps, predicates, axes, label tests and label names respectively and `*` is a label wildcard:

```
p ::= s | s/p
s ::= a::L | s[q]
L ::= l | *
q ::= p | q and q | q or q
a ::= self | child | descendant | descendant-or-self
```

In the previous examples, we used an abbreviated notation for `child` and `descendant-or-self` axes: `a/b` and `a//b` stand for `child :: a/child :: b` and `child :: a/descendant-or-self :: */child :: b` respectively in the above syntax.

We have defined a semantics-preserving translation from this fragment of XPATH into ambiguous regular patterns [8]. Consider, for example, the path query `table/tr//a`. It is converted into regular pattern equivalent to the pattern `<(Tag)> any, <table> X </>, any </>` assuming the following pattern declarations (notice how x is bound to the `a` elements that are addressed by the right-most step of the query):

```
regpat X [[ any, <tr> any, Y, any </>, any ]]
regpat Y [[ ((<a> any </>) x) | Z ]]
regpat Z [[ <(Tag)> any, Y, any </> ]]
```

In [8], we argue that the translation is correct in the sense that the nodes returned by the path query are precisely the values bound to `x` in all possible matchings of the input value against the regular pattern obtained as the result of translation.

5.3   Type Inference

The goal of type inference is to assign types to the variables appearing in regular patterns. Consider the following example taken from a program that processes and converts BibTeX files into HTML:

```
regtype entry [[ ... ]] // complex type
regtype doc [[<doc> entry* </>]]

void do_xml ([[doc]] doc) {
  match (doc) {
    case [[<doc>any items</doc>]]:
      match (items) {
        case [[anyone item, any rest]]: ...
        ... } } }
```

If the type checker only used the annotations provided by the programmer, it would operate with a pretty limited knowledge that variables `items` and `rest` may contain any arbitrary sequences and variable `item` may contain an arbitrary singleton sequence. This may be insufficient to type-check the right hand side of the first `match` clause. Taking into consideration the type of the input value, however, the type checker can be much more exact and infer that `items` and `rest` are of type `entry*` and `item` is of type `entry`.

So how exact can the type checker be? In fact, type inference can be be fully exact. We say that a type inference algorithm is *precise* if given an input value of type `T` and a variable `x` occurring in a pattern `p`, it infers type `S` for `x` iff for every value v' of `S`, there is a value `v` of `T` such that matching `v` against `p` results in binding `x` to v'. In other words, the inferred type denotes *precisely* the values that may be bound to the variable. XTATIC's parent XDUCE [13] features precise type inference. So does XTATIC's sibling CDUCE [1].

Because regular object types are not closed under boolean operations—union, intersection, and difference—type inference in XTATIC is *not* precise. The core of the problem is in element labels which can be arbitrary $C^\sharp$ types. Suppose we want to compute the following difference: `<(D)/> \ <(C)/>`. This is equivalent to `<(D\C)/>`, but if `C` is a subclass of `D`, in general, there is no $C^\sharp$ type that corresponds to `D\C`. Because difference is not always defined on regular object types, XTATIC's type inference does not compute type difference at all, and, therefore, does not take into account the order of clauses in a `match` expression losing some precision as a result. Consider this example:

```
[[<a/>+]] id([[<a/>+]] arg) {
  match (arg) {
    case [[<a/>]]: return([[<a/>]]);
    case [[<a/>, any rest]]: return [[<a/>, id(rest)]];
  } }
```

Intuitively, the recursive call to `id` cannot go wrong since `rest` can only be bound to values of type `<a/>+`. The type checker, however, can only conclude this by taking the difference between the input type `<a/>+` and the first pattern `<a/>` and then analyzing the second pattern with respect to the resulting type. If it considers the second clause independently of the first, it can only infer that the type of `rest` is `<a/>*` which is insufficient for type-checking the rest of the method.

The fact that object types are not closed under union is also a problem when we have binding inside labels. Consider the pattern `<(C x)/> | <(D x)/>` that binds `x` to objects of classes `C` or `D`. Again, in general, there is no $C^{\sharp}$ type that corresponds to $C \cup D$. XTATIC uses the smallest common superclass of `C` and `D` as the inferred type of `x` straying even further from full precision.

Finally, intersection is problematic as well. XTATIC can correctly compute $C \cap D$ if one is a superclass (superinterface) of the other or if `C` and `D` are unrelated classes. (In the former case, the result is the smaller type; in the latter, the result is empty.) If `C` and `D` are unrelated interfaces, however, there is no $C^{\sharp}$ type that corresponds to their intersection.

Nevertheless, as the opening example of this section illustrates, type inference in XTATIC is still very useful. Examples such as this are common in practice—they are characterized by variables annotated with "wildcard" types such as `any` and `anyone`. The type inferred for these variables is usually precise. More generally, type inference in XTATIC is precise as long as patterns of a `match` statement are mutually disjoint and all of the occurrences of each variable inside labels have the same type annotation.

In order to regain precision in type inference, we are currently exploring an extension of the $C^{\sharp}$ type system with boolean operations, called *boolean host types*, that allows taking the union, difference, and intersection of classes and interfaces. We are able to decide if a boolean host type is empty (i.e. there is no extension of the class and interface hierarchy that may result in a class having the boolean host type), hence we can decide subtyping (using difference and emptiness checking). These new types would only be used in type annotations and declarations, but not in `new` expressions.

5.4 Patterns and Schema Evolution

We close the discussion of patterns with some remarks about schema evolution—in particular, the robustness of programs with regular patterns and vs. programs with paths with respect to changing data formats. How well does the type system help a programmer in locating potential pitfalls?

To illustrate our observations, we will use the following instance of schema evolution in which a new optional element is added. (This real-life example is taken from a program that generates the online Caml Weekly News.)

```
Old = [[ <cwn><cwn_title>pcdata</> <cwn_content>Content</> </cwn> ]]
New = [[ <cwn><cwn_title>pcdata</> <cwn_url>Url</>?  <cwn_content>Content</> </cwn>]]
```

Programs using path-like navigation are pretty robust with respect to such format changes: most path queries that work for values of type `Old` would also work for values of type `New`. (One exception involves paths that use position-based primitives such as `next`.) However, this flexibility has a significant down-side: since such schema evolutions are transparent for most path-based programs, the path-based type system does not automatically indicate where the program should be modified to take newly introduced elements into account.

Regular patterns, on the other hand, can be chosen by the programmer to be either "permissive" like paths or more detailed ones that will be flagged by the type checker when a potentially sensitive format change occurs. The following program is an example of the former:

```
match (input) {
  case [[ <cwn>any, <cwn_title>pcdata x</>, any</cwn> ]]: ...
}
```

The only piece of data that is important to the above fragment is the contents of the `cwn_title` element; hence this program should work when the above schema change occurs, since the fragment is well-typed both for `Old` and `New` types of `input`.

Conversely, a pattern may be more precise in the specification of the context. Consider this fragment:

```
match (input) {
  case [[ <cwn><cwn_title>pcdata x</>, <cwn_content>any</></cwn> ]]: ...
}
```

Changing the type of `input` from `Old` to `New` now triggers typing errors, indicating that some cases of the input type (namely the potential presence of a `cwn_url` element) are not covered by the matching clauses. The type checker now guides the programmer to the places in the program where changes must be made to reflect changes in the schema. We have found this technique of "change and type-check" (familiar from languages such as ML and Haskell) quite useful when programming in XTATIC.

6   Some Further Extensions

During the design and implementation of XTATIC, we have discussed a number of possible extensions that are not reflected in the current design. We briefly list these here.

One extension that is of immediate importance is integration with generics: Hosoya, Frisch, and Castagna's recent proposal of how to add polymorphism to XDUCE looks very attractive as the starting point.

Another promising extension involves allowing label patterns not just to match objects of some class but also to extract values from the object's instance variables and, if instance variables are of regular types, match their structure as well. This extension is desirable because of generality, but it might present difficulties both at the level of semantic definition and practical pattern matching implementation.

Several useful type system improvements are pending as well. As we have discussed in Section 5.3, introducing boolean label types can drastically improve the quality of type inference. The interleaving operator on types can be employed elegantly to describe data in which the order of elements is immaterial. It is quite difficult though to design efficient subtyping algorithms for regular types with interleaving. Finally, attribute-element constraints discussed in Section 3.4 can become the basis for future XTATIC's principled handling of attributes.

Another area of future work deals with high-level pattern matching primitives. There are several ongoing projects. XTATIC's ambiguous patterns and `iterate` discussed in Section 5.2 can be improved to support more sophisticated iteration strategies based on the hierarchical structure of the pattern. Non-linear patterns provide another way of collecting multiple results from a single pattern. Hosoya [10] describes filters—one mechanism for working with non-linear patterns. CDUCE also uses non-linear patterns but on a more basic level where a non-linear variable is bound to the sequence of all its possible bindings. Another potentially promising approach to iteration may involve a mechanism similar to XQUERY's FLOWR expressions [26].

7   Related work

XDUCE [13, 14, 15], a domain-specific language for statically typed XML processing, was the first language featuring XML trees as built-in values, a type system based on regular types for statically type-checking computations involving XML, and a powerful form of pattern matching based on regular patterns. Here, we have concentrated on integrating XDUCE's regular types and regular patterns with a modern object-oriented language to form the programming language XTATIC. This paper describes the design choices we have made to facilitate smooth integration while keeping XTATIC easy to understand for the programmer. Other aspects of the XTATIC design and implementation are described in three companion papers—one presenting the core language design, integrating the object and tree data models and establishing basic soundness results [7]; another proposing a technique for compiling regular patterns based on *matching automata* [18], and another describing the run-time system of XTATIC [6].

Another XDUCE descendant that is close to XTATIC in several respects is the CDUCE language of Benzaken, Castagna, and Frisch [5, 1]. Like XTATIC, CDUCE is based on XDUCE-style regular types and emphasizes a declarative style of recursive tree transformation based on algebraic pattern matching. In

other respects, the focus in CDUCE is quite different: its type system includes several features (such as intersection and function types) not present in XTATIC, it is not object-oriented, and it is not integrated with an existing language. XTATIC, by contrast, has taken a more conservative approach in its type system, instead emphasizing smooth compatibility with an existing mainstream object-oriented language. Two significant differences are the object-oriented flavor of our representations and our approach to various interoperability issues such as cross-language calls and compatibility with legacy XML representations.

Another close cousin of XTATIC is Meijer, Schulte, and Bierman's C_ω language (previously called XEN) [20, 21], an extension of $C^\sharp$ that smoothly integrates support for objects, relations, and XML. Some aspects of the C_ω language design are much more ambitious than XTATIC: in particular, the extensions to its type system (`sequence` and `choice` type constructors) are more tightly intertwined with the core object model—indeed, XML itself is simply a syntax for serialized object instances. In other respects, C_ω is more conservative than XTATIC: for example, its `choice` constructor is not a true least upper bound, and the subtype relation is defined by a conventional, semantically incomplete, collection of inference rules, while XTATIC's is given by a more straightforward (and, for the implementation, more demanding) "subtype = subset" construction.

XACT [17, 2] extends JAVA with XML processing, proposing another somewhat different programming idiom: the creation of XML values is done using XML *templates*, which are immutable first-class structures representing XML with named *gaps* that may be filled to obtain ordinary XML trees. XACT also features a static type system that guarantees that, at a given point in the program, a template statically satisfies a given DTD. XACT's implementation, developed independently and in parallel with XTATIC but driven by similar needs (supporting efficient sharing, etc.) and targeting a similar (object-oriented) runtime environment, has strong similarities to ours; in particular, lazy data structures are used to support efficient gap plugging.

XJ [9] is another extension of JAVA for native XML processing that emphasizes fidelity to the XML Schema and XPATH standards, for instance by only allowing subtyping by name (as opposed to the structural subtyping of the languages mentioned above). XJ is also one of the few XML processing languages that allow imperative modification of XML data. This feature, however, significantly weakens the safety guarantees offered by static typing: the updated tree must be re-validated dynamically, raising an exception if its new type fails to match static expectations. In keeping with its emphasis on standards and its imperative nature, XJ uses DOM for its run-time representation of XML data.

XOBE [16] is a source to source compiler for an extension of JAVA. From a language design point of view, it is very similar to XTATIC, allowing seamless integration of XML with JAVA, taking a declarative style of tree processing, and providing a rich type system and subtyping relation based on regular expression types. The run-time representation, like XJ, relies on DOM.

SCALA is a general-purpose experimental web services language that compiles into JAVA bytecode and therefore may be seen as an extension of JAVA since SCALA programs may still easily interact with JAVA code. SCALA is currently being extended with XML support [4].

Work also continues on XDUCE itself, including fully typed treatment of attributes *à la* RELAXNG [11] and regular expression filters [10]. These developments are highly relevant to future work on the XTATIC language design.

Finally, a recent survey paper by Møller and Schwartzbach [22] offers an excellent overview of recent work on static typechecking for XML transformation languages, with detailed comparisons between a number of representative languages, including XDUCE and XACT.

Acknowledgments

References

[1] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Uppsala, Sweden*, pages 51–63, 2003.

[2] A. S. Christensen, C. Kirkegaard, and A. Møller. A runtime system for XML transformations in Java. In Z. Bellahsène, T. Milo, and e. a. Michael Rys, editors, *Database and XML Technologies: International XML Database Symposium (XSym)*, volume 3186 of *Lecture Notes in Computer Science*, pages 143–157. Springer, Aug. 2004.

[3] J. Clark and M. Murata. RELAX NG. http://www.relaxng.org, 2001.

[4] B. Emir. Extending pattern matching with regular tree expressions for XML processing in Scala. Diploma thesis, EPFL, Lausanne; http://lamp.epfl.ch/ buraq, 2003.

[5] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. In *IEEE Symposium on Logic in Computer Science (LICS)*, 2002.

[6] V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. XML goes native: Run-time representations for Xtatic. Technical Report MS-CIS-04-23, University of Pennsylvania, Oct. 2004.

[7] V. Gapeyev and B. C. Pierce. Regular object types. In *European Conference on Object-Oriented Programming (ECOOP), Darmstadt, Germany*, 2003. A preliminary version was presented at FOOL '03.

[8] V. Gapeyev and B. C. Pierce. Paths into patterns. Technical Report MS-CIS-04-25, University of Pennsylvania, Oct. 2004.

[9] M. Harren, B. M. Raghavachari, O. Shmueli, M. Burke, V. Sarkar, and R. Bordawekar. XJ: Integration of XML processing into Java. Technical Report rc23007, IBM Research, 2003.

[10] H. Hosoya. Regular expression filters for XML. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2004.

[11] H. Hosoya and M. Murata. Boolean operations and inclusion test for attribute-element constraints. In *Eighth International Conference on Implementation and Application of Automata*, volume 2759 of *Lecture Notes in Computer Science*, pages 201–212. Springer-Verlag, 2003. Preliminary version in PLAN-X 2002.

[12] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language (preliminary report). In D. Suciu and G. Vossen, editors, *International Workshop on the Web and Databases (WebDB)*, May 2000. Reprinted in *The Web and Databases, Selected Papers*, Springer LNCS volume 1997, 2001.

[13] H. Hosoya and B. C. Pierce. Regular expression pattern matching. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), London, England*, 2001. Full version in *Journal of Functional Programming*, 13(6), Nov. 2003, pp. 961–1004.

[14] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, May 2003.

[15] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2000.

[16] M. Kempa and V. Linnemann. On XML objects. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2003.

[17] C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, Mar. 2004.

[18] M. Y. Levin. Compiling regular patterns. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Uppsala, Sweden*, 2003.

[19] M. Y. Levin and B. C. Pierce. Typed-based optimization for regular patterns. In *First International Workshop on High Performance XML Processing*, 2004.

[20] E. Meijer and W. Schulte. Unifying tables, objects and documents. In *Declarative Programming in the Context of OO Languages (DP-COOL)*, Sept. 2003.

[21] E. Meijer, W. Schulte, and G. Bierman. Programming with circles, triangles and rectangles. In *XML Conference and Exposition*, Dec. 2003.

[22] A. Møller and M. I. Schwartzbach. The design space of type checkers for XML transformation languages. In *Proc. Tenth International Conference on Database Theory, ICDT '05*, LNCS. Springer-Verlag, January 2005.

[23] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, Montreal, Canada, 2001.

[24] XML Schema Part 1: Structures, W3C Recommendation, May 2001. http://www.w3c.org/TR/xmlschema-1/.

[25] XML Path Language (XPath) Version 1.0, W3C Recommendation, Nov. 1999. http://www.w3c.org/TR/xpath.

[26] XQuery 1.0: An XML Query Language, W3C Working Draft, July 2004. http://www.w3.org/TR/xquery/.