Toward Native XML Processing Using Multi-paradigm Design in C++

Sumant Tambe and Aniruddha Gokhale
Department of EECS, Vanderbilt University, Nashville, TN, USA
{sutambe, gokhale}@dre.vanderbilt.edu

Abstract

XML programming has emerged as a powerful data processing paradigm with its own rules for abstracting, partitioning, programming styles, and idioms. Seasoned XML programmers expect, and their productivity depends on the availability of languages and tools that allow usage of the patterns and practices native to the domain of XML programming. The object-oriented community, however, prefers XML data binding tools over dedicated XML languages because these tools automatically generate a statically-typed, vocabulary-specific object model from a given XML schema. Unfortunately, these tools often sidestep the expectations of seasoned XML programmers because of the difficulties in synthesizing abstractions of XML programming using purely object-oriented principles. We demonstrate how this prevailing gap can be significantly narrowed by a novel application of multi-paradigm programming capabilities of C++. In particular, we demonstrate how generic programming, metaprogramming, generative programming, strategic programming, and operator overloading supported by C++ together enable native and typed XML programming.

Keywords: XML Processing, Object-oriented Programming, Generic Programing, Meta Programming, Generative Programming, C++.

1 Introduction

There is little doubt that XML has evolved from just a human readable serialization format to a sophisticated data description, storage, and processing technique used in a wide range of applications. XML programing – the paradigm that is native to the domain of XML processing – has its own type system [10], (*e.g.,* anonymous complex elements, repeating sub-sequences), data model [21] (*e.g.,* XML *information set* constituents, such as elements, attributes, and processing instructions), schema languages for document description (*e.g.,* XSD [22], DTD, RELAX NG), programming languages (*e.g.,* XPath [23], XSLT, XQuery), and styles and idioms (*e.g.,* child, descendant, sibling axes in XPath, pattern matching in XSLT). Naturally, the conceptual richness of XML processing has led many to identify it as a distinct paradigm in itself.

Listing 1 An XML document (catalog.xml) containing a book catalog.

```
<catalog>
  <book>
    <title>Hamlet</title>
    <price>9.99</price>
    <author>
      <name>William Shakespeare</name>
      <country>England</country>
    </author>
  </book>
  <book>...</book>
    ...
</catalog>
```

To reify this fact, consider the XML document shown in Listing 1 that we will use as a running example in the rest of this article. Suppose we need to extract the names of authors who lived in England. A solution using XPath would be

```
"//author[country/text() = 'England']/name/text()"
```

The succinctness and expressiveness of this solution is due to the idiomatic uses of XPath's child and descendant axes denoted by '/' and '//' respectively. The child axis selects immediate children elements whereas the descendant axis selects the specified element nodes ("author") anywhere in the XML tree, irrespective of their depth from the root ("catalog").

While a XML aficionado would appreciate the succinctness of the above solution, an object-oriented (OO)-biased developer would be reluctant to use this approach for several reasons. First, contemporary XPath libraries available to the OO programmers use strings to represent queries, which leaves no opportunity for static type checking. Incorrect XPath queries are identified only during testing when the result set is either a null set or an exception is raised. Second, the string encoded XPath queries may be vulnerable to XPath injection attacks [2]. Finally, the results of such

queries require type casting to appropriate types, which is often computationally expensive.

Listing 2 XML Schema Definition (XSD) of the catalog XML

```
<xs:element name="catalog">
 <xs:complexType> <xs:sequence>
  <xs:element name="book" maxOccurs="unbounded">
   <xs:complexType> <xs:sequence>
    <xs:element name="name" type="xs:string" />
    <xs:element name="title" type="xs:string" />
    <xs:element name="price" type="xs:double" />
    <xs:element name="author" maxOccurs="unbounded">
      <xs:complexType> <xs:sequence>
        <xs:element name="name" type="xs:string" />
        <xs:element name="country" type="xs:string" />
       </xs:sequence> </xs:complexType>
    </xs:element>
   </xs:sequence> </xs:complexType>
  </xs:element>
 </xs:sequence> </xs:complexType>
</xs:element>
```

Listing 3 C++ classes generated by a typical XML data binding tool from the XSD

```
class title   {...};
class price   {...};
class name    {...};
class country {...};
class author  { // Constructors are not shown.
  private: name name_;
           country country_;
  public:  name get_name() const;
           void set_name(name const &);
           country get_country() const;
           void set_country(country const &);
};
class book { // Constructors are not shown.
  private: title title_;
           price price_;
           std::vector<author> author_sequence_;
  public:  title get_title() const;
           void set_title(title const &);
           price get_price() const;
           void set_price(price const &);
           std::vector<author> get_author() const;
           void set_author(std::vector<author> const &);
};
class catalog {...};  // Contains a std::vector of books.
```

To overcome these limitations, OO-biased developers often use XML data binding tools [15, 1], which generate a statically-typed, vocabulary-specific object model from a description of the object structure in a schema language (*e.g.,* XSD, DTD, RELAX NG). For instance, Listing 2 shows the XML schema definition of the book catalog. Corresponding to that, Listing 3 shows the generated object model[1] that reflects the structure of the book catalog. Well-known OO programming idioms are used to encapsulate the data in intuitive classes and intuitive ways are provided for the inspection and manipulation of the data through member functions only.

Verbose queries are the primary downside of the object-oriented approach compared to XPath. First, obtaining children requires invocation of member functions, which cannot be composed as in the '/' operator of XPath. Although *Method chaining* can be exploited to a certain extent, its use is not anticipated in most XML data binding tools. Second, explicit loops are necessary to iterate over the containers of children, which is clearly low-level and tedious compared to XPath. Finally, the XPath features allow queries to be decoupled from the concrete XML structure by omitting intermediate tags. For instance, XPath query "//name" finds *name* tags anywhere in the XML tree, without having to mention the "book", "catalog", and "author" tags. Such decoupling reduces maintenance should the XML structure change in future. This commonly used XPath idiom is nowhere to be found in the generated classes. In fact, there is no way to bypass `catalog` and `book` objects before reaching the `name` objects.

To bridge this technical gap between the two approaches and thereby alleviate the "disappointment" of the XML programmer, we ask ourselves the question: "Is it possible to achieve the expressiveness of XPath and the type-safety of OO all at once?" As we will show in the rest of this article, C++ is a true multi-paradigm language that rises to the complexity of this problem, which is otherwise inaccessible using the OO paradigm alone. In particular, we demonstrate how generic programming, static metaprogramming [3], generative programming [7], and strategic programming [19, 14] in combination with operator overloading can co-exist in a single framework to resolve the gap between XPath-like notation and OO type-safety.

The following code snippet is a C++ program that is equivalent to the XPath query shown earlier.

```
bool from_england(author a){ return a.get_country()=="England"; }
std::vector<name> author_names =
catalog_root >>  AllDescendantsOf(catalog(), author())
             >>  Select(author(), from_england)
             >>  name()
```

This solution is presented using our previous work on the **L**anguage for **E**mbedded qu**E**ry and trave**r**s**A**l (LEESA) [17]. In the above code snippet, `catalog_root` is the root of the typed XML tree, which is instantiated after parsing (and optionally validating) an input XML file. `AllDescendantsOf` is LEESA's manifestation of the descendant axis, which serves a purpose similar to that of "//" in XPath. `Select` encapsulates the user-defined predicate function `from_england`.

The key distinction between the XPath query and the LEESA query is that the C++ compiler type-checks the LEESA expression because it is not encoded as a string. Moreover, LEESA validates the expressions against the

[1] By default, simple types such as *xs:double* are implemented using the language's native types. Classes can be generated for simple types using tool-specific command-line options or simple xsd transformations.

schema at compile-time. The return value of the expression is a standard container of `names`, which requires no type-casting.

2 XML Programming Concerns

In this section, we identify the key concerns of XML programmers that are left unresolved by contemporary OO-biased XML data binding tools. Figure 1 shows a decomposition of these concerns that we are interested in. Several other XML programming concerns, such as construction of XML literals, modularization of type-specific actions, and the dreaded problem of X/O impedance [13] remain important but are not discussed further. We describe how XML data access can be made more generic (yet type-safe) by using a *type-driven* approach.

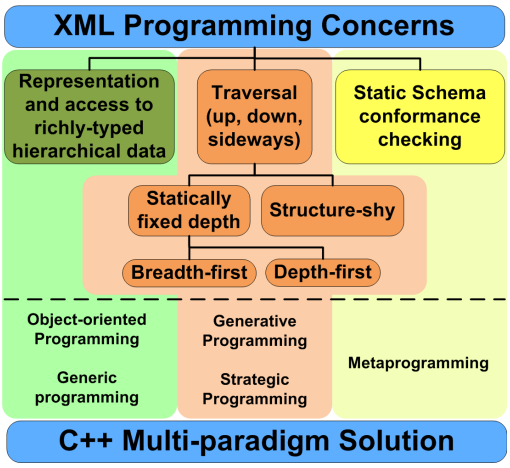

Figure 1: Major concerns of the XML programming paradigm and the proposed multi-paradigm solutions

2.1 Representation and Data Access

Contemporary XML data binding tools aptly represent XML's tree-shaped data using the Composite design pattern. Each element type is represented by a class that is specific to the XML vocabulary in question as shown in Listing 3. These classes, however, are hard to use in generic algorithms. Syntactically, vocabulary-specific *accessors/-mutators* of the generated classes have little, if any, commonality. The types of the children elements are encoded in the member function names (*e.g.,* `get_country`, `get_name` etc.), which force usage of OO's *dot* notation.

In contrast, a more *generic* approach to access the children is via a generic accessor function that can be parameterized by the desired children types. The key benefit of the *generic type-driven* access to data is that it allows us to abstract the vocabulary-specific interface behind a uniform interface without losing type-safety. Furthermore, this approach is more amenable to composition than classic OO *dot* notation as discussed in Section 2.3. Unfortunately, such a generic use is often not anticipated by the OO-centric XML data binding tools and hence the generic APIs are not synthesized.

Listing 4 Automatically generated overloaded functions for type-driven data access

```
name children (author a, name const *) {
  return a.get_name();
}
country children (author a, country const *) {
  return a.get_country();
}
title children (book b, title const *) {
  return b.get_title();
}
price children (book b, price const *) {
  return b.get_price();
}
std::vector<author> children (book b, author const *) {
  return b.get_author();
}
std::vector<book> children (catalog c, book const *) {
  return c.get_book();
}
```

To address this limitation we have developed a Python script that generates a set of overloaded functions that allow generic type-driven access to the composite data, as opposed to the common style of member function invocation. A sample of global (namespace-level) functions is shown in Listing 4. All the overloaded functions are named `children`, where the second formal parameter is a dummy *pointer* used to resolve ambiguities and also to provide type-driven access. Thus, for every parent-child pair one overloaded function is synthesized which maps the child type to the appropriate member function of the parent object.

Alternatively, a semantically equivalent interface can be synthesized using C++ member templates, where a generic member function, `children`, is made parameterizable using C++ template mechanism. In fact, this technique has been employed in our earlier work [17] where the type parameter serves the same purpose as the second parameter in Listing 4. We choose overloaded functions here primarily for their simplicity.

2.2 Resolving Ambiguities

Despite the simplicity of our approach, a new problem arises that must be handled. For example, XML data binding tools use mapping optimizations where simple content nodes such as attributes and simple elements are represented using standard library types or the language's built-in types instead of vocabulary-specific types. Such optimizations, however, are unable to distinguish objects that logically belong to different parts of the XML tree at the type level. For instance, `title` and `name` in Listing 2 would be indistinguishable at the type level if they are both represented using `std::string`. Nevertheless, such optimiza-

tions cause ambiguities in our type-driven approach when two or more types of children elements are represented using the same C++ class.

To address this limitation, our Python script provides two alternatives. First, the script can be used to transform the given XML Schema Definition (XSD) – without affecting its data semantics – such that XML data binding tools are forced to generate vocabulary-specific types for simple content nodes. This is achieved by inserting a combination of *xsd:simpleType* and *xsd:restriction* elements in the data definition of the simple content nodes. For instance,

```
<xs:element name="name" type="xs:string" />
<xs:element name="title" type="xs:string" />
```

from Listing 2 is automatically transformed into

```
<xsd:element name="name">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string" />
  </xsd:simpleType>
</xsd:element>
<xsd:element name="title">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string" />
  </xsd:simpleType>
</xsd:element>
```

Such transformation forces contemporary XML data binding tools to generate distinct `name` and `title` classes. Alternatively, the script can be instructed to drop the ambiguous set of functions, which leaves the programmer with the generated OO bindings (instead of LEESA) as the only way to access the leaf-level data.

2.3 Axis-oriented Fixed-depth Traversal

XPath programmers visualize every XML document as being conceptually partitioned along the so-called XML axes (*e.g.,* child, parent, sibling, ancestor, descendant) and the self-describing data organized around these axes. These axes represent a *commonality, i.e.*, an opportunity for reuse, that is central to the domain of XML programming which, however, is not recognized by the OO principles alone. These axes determine not only how data is structured but also how different variations in traversal allow access to the data. For instance, Figure 2 shows two variations: breadth-first and depth-first, along the child axis.

Complementing the axes, there exist numerous element tags, which capture the actual data in every XML document. Since the element tags are vocabulary-specific, the classes that correspond to element tags in the corresponding object model become the source of *variability*. Such an identification of the common and variable parts provides a opportunity to introduce a *reusable mechanism* that can capture the commonality of axes while leaving the richly-typed objects a mere matter of vocabulary-specific *policy*.

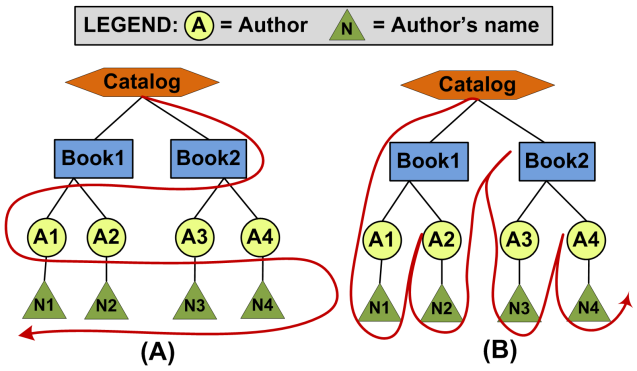

Figure 2: Variations in child axis traversal: (A) breadth-first (B) depth-first

We achieve this desired goal using the generic programming paradigm and operator overloading in C++ to construct an intuitive, composable, and reusable notation for axis-oriented expressions that is well beyond the *dot* notation prevalent in the OO paradigm.

Listing 5 A reusable, generic infrastructure in (pseudo) C++ simulating XPath-like child axis notation

```
 1: template <class Kind>
 2: class Carrier : public std::vector<Kind> {
 3:   // Default and copy-constructor are trivial.
 4:   Carrier (const Kind &k) {
 5:     this->push_back(k);
 6:   }
 7:   using std::vector<Kind>::push_back; // Accepts one Kind.
 8:   void push_back(std::vector<Kind> const &v) {
 9:     this->insert(this->end(), v.begin(), v.end());
10:   }
11: };
12: template <class Parent, class Child>
13: Carrier<Child>
14: operator >> (Carrier<Parent> carrier, Child const &c)
15: {
16:   Carrier<Child> all_children;
17:   foreach parent in carrier {
18:     all_children.push_back(children(parent, &c));
19:   }
20:   return all_children;
21: }
```

Listing 5 shows a mechanism for writing typed child axis expressions, which has been significantly simplified compared to that of LEESA in the interest of brevity. While LEESA uses the Expression Templates [18] idiom for supporting both breadth-first and depth-first traversal, the overloaded `operator` $\gg$ shown in Listing 5 supports breadth-first traversal only. It allows the expression of type-safe queries using a notation that is similar to the child axis of XPath. For instance, the following program implements the breadth-first traversal shown in Figure 2(A).

```
Carrier<catalog> catalog_root(getRoot("catalog.xml"));
std::vector<name> names =
  catalog_root >> book() >> author() >> name();
```

`Carrier<Kind>` is an abstraction that hides whether it is *carrying* a single `Kind` or a collection of `Kind`s. The overloaded member function, `push_back`, treats a singular `Kind` object and a collection of `Kind`s uniformly. This uniformity is exploited on line #18 in the generic `operator` $\gg$ function. The `children` function on line #18 is chosen from the overloaded functions for type-driven data access shown in Listing 4. Depending upon the return type of the `children` function, appropriate overloaded method of `push_back` is chosen. The `Child` type is automatically deduced by the compiler while evaluating the chained `operator` $\gg$ from left to right. Finally, the `operator` $\gg$ function itself is reminiscent of the conventional overloaded extraction ($\gg$) operators used for I/O, which can be chained to an arbitrary length.

LEESA provides generic, reusable functionality for traversal along child, parent, and sibling axes. In particular, $\gg$ and $\gg$= operators are used for breadth-first and depth-first traversal of child axis whereas $\ll$ and $\ll$= operators are used for breadth-first and depth-first traversal of the parent axis. The following query implements the depth-first child axis traversal shown in Figure 2(B).

```
Carrier<catalog> catalog_root(getRoot("catalog.xml"));
evaluate(catalog_root,
         catalog() >>=  book() >>= author() >>= name());
```

Note the use of function `evaluate`, which takes two parameters as input. The first parameter is a *context* object and the second parameter is the depth-first LEESA query implemented using the Expression Templates idiom. Expression Templates allow lazy evaluation of C++ expressions. Actual execution can be deferred much later in the program. At the point of definition, a temporary function object is created that embodies the computation. This function object must be explicitly invoked later in the program. LEESA provides `evaluate` function to execute previously constructed LEESA expression templates. In the above expression, LEESA query is executed in the context of the root of the XML tree: `catalog_root`.

Unlike the breadth-first expression, LEESA's depth-first expressions do not have return values. Instead, they are intended to be used in the context of the Visitor [8] design pattern. Our Python script generates a visitor class with empty `visit` functions for all the types in the schema. Users can extend specific `visit` functions to implement type-specific actions, such as printing. Assuming the user-defined visitor object is called `v`, the following expression invokes type-specfic `visit` functions in depth-first order as shown in Figure 2(B).

```
Visitor v;
Carrier<catalog> catalog_root(getRoot("catalog.xml"));
evaluate(catalog_root,
         catalog() >> v >>= book() >> v >>=
         author() >> v >>= name() >> v);
```

Further details on the use of the Visitor design pattern in LEESA can be found in [17].

LEESA also supports different variations of the sibling axis to query children at the same level. We present a variation of sibling axis that allows *on-demand* creation of tuples containing objects of different types. For example, the following LEESA query creates a collection of tuples containing every author's name and his/her country.

```
Carrier<catalog> catalog_root(getRoot("catalog.xml"));
std::vector<tuple<name, country> > tuples =
evaluate(catalog_root, catalog() >> book() >> author() >>
         ChildrenAsTupleOf(author(),tuple<name,country>()));
```

Note that XPath does not support depth-first traversal and *tuple-fication*[2] of XML data element as presented here. These capabilities demonstrate that due to the judicious use of operator overloading and generic programming in C++, succinct and expressive traversals can be written in a type-safe manner. More details of LEESA's design appear in [17].

2.4 Axis-oriented Structure-shy Traversal

XPath supports the *descendant* axis, which allows omission of the element tags between the document root and the elements of interest resulting in the so-called *structure-shy* queries. For instance, `"/*/*/country"` and `"//country"` are two structure-shy XPath queries that omit the "book" and "author" tags indicating interest in the "country" elements only. While the former query looks for the "country" elements at the third level from root, the latter looks for the same at any nested level in the XML tree. Such a decoupling from the concrete structure of the XML tree is desirable to write flexible queries that are resilient to changes in the schema.

Although the OO paradigm can exhibit structure-shyness in the form of *information hiding* and *encapsulation*, realizing XPath-style structure-shyness poses a significant challenge using only the OO features of C++. However, by leveraging the *Strategic Programming* (SP) [19, 14] and *Generative Programming* [7] paradigms supported by C++, we can achieve support for structure-shyness that rivals XPath.

Sidebar 1 presents an overview of SP. In our earlier work [17] we have demonstrated how LEESA implements basic strategic combinators using C++ templates. Commonly used traversal schemes such as FullTopDown are also provided out-of-the-box like most SP incarnations do. LEESA's manifestation of the descendant axis, `AllDescendantsOf(Ancestor, Descendant)`, uses FullTopDown traversal scheme to emulate XPath's '//' operator.

Specifically, the FullTopDown traversal scheme is parameterized with a `Collector<Descendant>` object that

[2] XPath's *or* operator computes a linear list of node elements; not a list of tuples as in LEESA.

identifies the `Descendant` type objects and accumulates them as the recursive strategy descends into the XML tree. For example, `AllDescendantsOf(catalog(), country())` uses `Collector<country>` that collects all the `country` objects irrespective of their depth. Therefore, `AllDescendantsOf` presents an opportunity for XML programmers to express *typed structure-shy* queries. The following expression is a typed equivalent of `"//country"` XPath query.

```
Carrier<catalog> catalog_root(getRoot("catalog.xml"));
std::vector<country> countries =
evaluate(catalog_root, catalog() >>
         AllDescendantsOf(catalog(), country()));
```

Sidebar 1: Strategic Programming in a Nutshell

Strategic Programming (SP) began as a general-purpose program transformation [19] technique, which later evolved into a paradigm [14] for generic tree traversal that supports reuse of the traversal logic while providing complete control over traversal. It warrants the status of a *paradigm* because it has been incarnated in disparate programming disciplines such as term rewriting, functional programming, logic programming, object-oriented programming (visitors). It is based on a small set of *combinators* (*Identity*, *Fail*, *Sequence*, *Choice*, *All*, and *One*) that can be composed to construct complex traversal schemes. Pseudo-code for the *Sequence* and *All* combinators is shown below.

```
template <class Strategy1, class Strategy2, class Datum>
Sequence (Datum d) {
  Strategy1(d);
  Strategy2(d);
}
template <class Strategy, class Datum>
All (Datum d) {
  foreach child c of d
    Strategy(c);
}
```

While the `Sequence` combinator invokes the parameter strategies in sequence on its input datum, the `All` combinator invokes the parameter strategy on all the children of the input datum, if any. The `strategy` template parameters could be simple unary functions, function objects, or other strategic combinators themselves. The real strength of SP lies in the various ways these combinators can be combined to give rise to complex traversals. For example, `Sequence` and `All` can be composed to obtain a recursive traversal scheme called `FullTopDown`, which descends into each sub-tree and applies the specified strategy on every element in the sub-tree.

```
template <class Strategy, class Datum>
FullTopDown (Datum d) {
  Sequence<Strategy, All<FullTopDown<Strategy> > > (d);
}
```

Furthermore, LEESA applies strategic programming to construct type-safe queries that emulate XPath wildcards ('/*/') using `LevelDescendantsOf`. For instance, the following expression is a typed equivalent of `"/*/*/country"` XPath query. The intermediate types between `catalog` and `country` are omitted *without* losing the type-safety of the query.

```
Carrier<catalog> catalog_root(getRoot("catalog.xml"));
std::vector<country> countries =
evaluate(catalog_root, catalog() >>
         LevelDescendantsOf(catalog(), _, _, country()));
```

LEESA infers the omitted types using sophisticated metaprograms implemented in `LevelDescendantsOf`. `LevelDescendantsOf` makes use of the *All* strategic combinator, which applies its nested strategy to the *immediate* children of its input datum. For instance, when `All<All<All<Collector<country> > > >` composite strategy is applied to the root of the catalog object model, it collects the `country` objects found exactly at the 3rd level. Note that in the case of `LevelDescendantsOf`, the number of times *All* is composed is not known a priori but instead determined at compile-time based on the number of wildcards specified by the programmer. Such automatic compile-time composition of strategies is the novelty of LEESA's incarnation of SP. This technique of synthesizing complex types from basic types at compile-time is well-known in the C++ community as *generative programming* [7].

2.5 Compile-time Schema Conformance Checking

Although all the LEESA queries are type-checked by the compiler, it could be argued that the axis-oriented traversal is an *over-generalization* of OO's member access idiom leading to a possibility of writing unsafe or illegal XML queries. For instance, `catalog()` ≫ `book()` ≫ `book()` is an illegal query because "book" elements do not contain other books. The possibility of such illegal queries question the usefulness of the *type-driven* data access approach we presented earlier. We address these limitations using the static metaprogramming paradigm supported by C++.

We exploit the C++ compiler to check LEESA expressions at compile-time based on the meta-information of the XML object structure that is automatically *externalized* in a form understood by the C++ compiler. The Boost Metaprogramming Library (MPL) is used as the representation format for the externalized meta-information for the child and descendant axes. Sidebar 2 presents an overview of Boost MPL, which provides sophisticated facilities for static metaprogramming in C++.

We have developed a Python script that automatically externalizes the meta-information in the schema in the form of MPL sequences. Listing 6 shows the automatically generated meta-information for the catalog object model. For

Sidebar 2: C++ Metaprogramming and Boost MPL

C++ templates, due to their support for *specialization*, give rise to a unique, purely functional computation system that can be used to perform compile-time computations. It has become well-known as C++ template metaprogramming and has been exploited in countless applications including scientific computing, parser generators, functional programming among others.

Boost MPL [3] is a general-purpose C++ metaprogramming library with a collection of extensible compile-time algorithms, *typelists*, and *metafunctions*. Typelists encapsulate zero or more C++ types in a way that can be manipulated at compile-time using MPL metafunctions. For example, consider a typelist called `Integral`, which is represented using a compile-time MPL sequence `mpl::vector` (not to be confused with `std::vector`).

```
typedef mpl::vector<int, long, short, unsigned> Integral;
```

MPL provides several off-the-shelf capabilities to manipulate such a list of types at compile-time. For instance, a MPL metafunction called `mpl::contains` can be used to check existence of a type in a MPL sequence.

```
mpl::contains<Integral, int>::value;   // value = true
mpl::contains<Integral, float>::value; // value = false
```

Listing 6 Automatically generated meta-information for the catalog object model

```
 1: template <class Kind> struct SchemaTraits {
 2:  typedef mpl::vector<> Children; // Empty sequence
 3: };
 4: template <> struct SchemaTraits <catalog> {
 5:   typedef mpl::vector<book> Children;
 6: };
 7: template <> struct SchemaTraits <book> {
 8:   typedef mpl::vector<title, price, author> Children;
 9: };
10: template <> struct SchemaTraits <author> {
11:   typedef mpl::vector<name, country> Children;
12: };

13: struct True  { enum { value = 1 }; };
14: struct False { enum { value = 0 }; };
15: template<class A, class D> struct IsDescendant : False {};

16: template<> struct IsDescendant<catalog, book>    : True {};
17: template<> struct IsDescendant<catalog, title>   : True {};
18: template<> struct IsDescendant<catalog, price>   : True {};
19: template<> struct IsDescendant<catalog, author>  : True {};
20: template<> struct IsDescendant<catalog, name>    : True {};
21: template<> struct IsDescendant<catalog, country> : True {};

22: template<> struct IsDescendant<book, title>      : True {};
23: template<> struct IsDescendant<book, price>      : True {};
24: template<> struct IsDescendant<book, author>     : True {};
25: template<> struct IsDescendant<book, name>       : True {};
26: template<> struct IsDescendant<book, country>    : True {};

27: template<> struct IsDescendant<author, name>     : True {};
28: template<> struct IsDescendant<author, country>  : True {};
```

every class that has at least one child, a specialization of the `SchemaTraits` is generated that contains a MPL sequence of the children types. For other simple classes, the list of children is empty, represented by the generic `SchemaTraits` template (lines 1-3) in Listing 6.

The descendant axis information is represented using the specializations of `IsDescendant<A,D>` template. For every type `D` (for descendant) that is contained directly or indirectly under type `A` (for ancestor), an `IsDescendant` specialization is generated that inherits from the `True` type. For all other pairs, the generic `IsDescendant` template (line #15) inherits from `False`, indicating that the descendant relationship does not hold. Essentially, `IsDescendant` is a *transitive closure* of the child relationship.

LEESA leverages this meta-information to catch any illegal query expression at compile-time. LEESA implements generic, reusable *compile-time assertions* in its overloaded operators to disallow illegal queries along all the supported axes. For instance, the following compile-time assertion is used in the implementation of `operator` $\gg$ function (Listing 5).

```
typedef SchemaTraits<Parent>::Children Children;
BOOST_STATIC_ASSERT(mpl::contains<Children, Child>);
```

Using the Boost static-assert library and the externalized meta-information, LEESA constrains the formal parameter types of the function such that only those types that satisfy the parent-child relationship yield a successful compilation of the program. Similar static assertions are used in the implementation of `AllDescendantsOf` and `LevelDescendantsOf` based on the `IsDescendant<A,D>` meta-information. Such compile-time assertions are generic but still *schema-aware*. They act like vocabulary-specific extensions to the language's type system ensuring that the existential constraints in the schema are satisfied at compile-time.

3 Performance Evaluation

To determine whether our approach can be used in practice, we compare the performance[3] of LEESA with several contemporary XML processing tools, such as the open-source *xsdcxx* [1] XML data-binding tool for C++ and an open-source XML processing library in C language called *libxml2*[4]. We chose these tools because they are open-source and widely deployed.

3.1 Run-time Performance

Figure 3 compares the run-time performance LEESA with equivalent programs written using pure object-oriented techniques (xsdcxx) and the libxml2 C library. We compared the time needed to construct a standard container

[3]The testbed can be downloaded from `http://www.dre.vanderbilt.edu/LEESA`

[4]http://xmlsoft.org

(`std::vector`) of `name` objects from a set of large book catalogs. LEESA's descendant axis has consistently higher overhead by a factor of 2.3 compared to the XML data-binding solution. The performance of libxml2 library lies between the solutions using OO and LEESA. This *abstraction penalty* stems from the construction, copying, and destruction of the internal dynamic data structures LEESA maintains. In particular, our analysis using the GNU profiler (gprof) revealed that the test program spent the highest percentage of time in the `std::vector`'s `insert` member function and the iterator functions during the query execution.

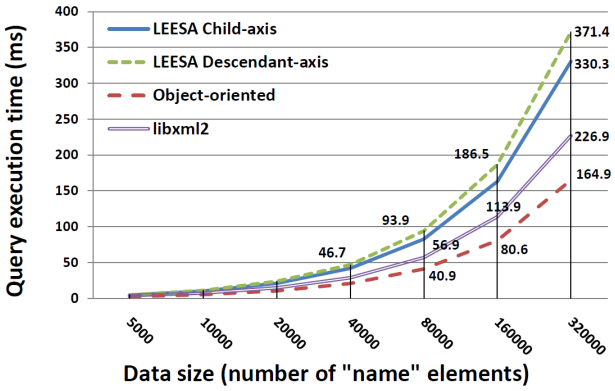

Figure 3: Comparison of run-time performance of LEESA, the OO solution, and libxml2

In practice, however, query execution amounts to a small fraction of the overall XML processing, which involves I/O, parsing, XML validation, construction of the in-memory object model, and the execution of business logic. For instance, our 320,000 elements test took over 33 seconds for XML parsing, validation, and object model construction, which is nearly two orders of magnitude higher than the query execution time.

Moreover, LEESA's higher-level of abstraction opens opportunities for transparent performance improvements using the upcoming C++ language features such as *rvalue reference* [11]. Rvalue references eliminate unnecessary expensive copies of objects, particularly when large objects are returned by value. LEESA, like most other generic libraries, will benefit from this language feature because it passes large container objects from the inner scope to the outer by value.

3.2 Compile-time Overhead

LEESA's heavy reliance on C++ template metaprogramming and generative programming motivates us to evaluate metrics such as compilation times, source code size, and the object code size. Table 1 shows the comparison of code sizes for one small (10 types) and one large (300 types) schema. The small schema has four nested levels where as the large schema has eight (not considering recursion). The large schema is the data representation format used by a component-based modeling language [5], which has 4 recursive and mutually recursive types. We evaluated a single LEESA expression of each query type shown in the table against equivalent programs written using OO abstractions only. The GNU C++ compiler collection (version 4.5) was used for the evaluations.

Schema size	Query type	Lines of code		Object code (Megabytes)	
		(A)	(B)	(A)	(B)
Small	Child-axis, AllDescendants, LevelDescendants	3	13	0.38	0.35
Large	Child-axis	3	39	7.42	7.15
	AllDescendants	3	136	7.46	7.19
	LevelDescendants	4	88	7.49	7.18

Table 1: Comparison of the static metrics. (A) = LEESA and (B) = Object-oriented solution

The difference in the lines of code (LOC) in Table 1 clearly shows that LEESA expressions are expressive and succinct compared to the OO-centric solution. Pure OO code is not only verbose but also unable to express XML idioms of structure-shyness. Data for the object code sizes reveals that LEESA's generative programming approach does not result in object-level code bloat.

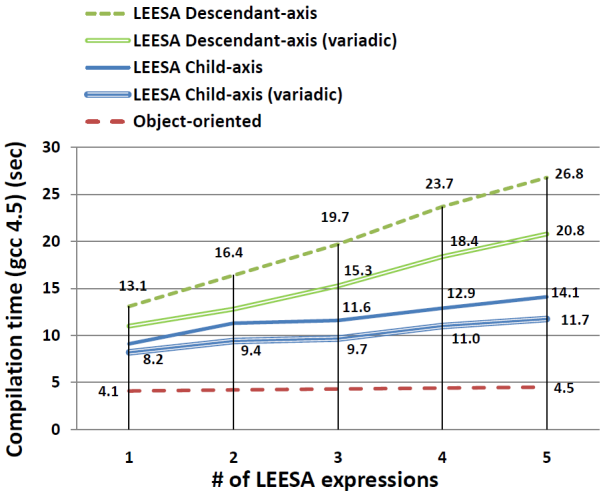

Figure 4: Comparison of compilation times of LEESA and the pure OO solution. Schema of 300 types.

Comparisons of the compilation times for the test programs written using the large schema are shown in Figure 4. LEESA-based programs consistently require more time to

compile than pure OO solutions because contemporary C++ compilers are not optimized for heavy metaprogramming. The increasing compilation-times may lengthen the *edit-compile-test* cycles. However, we believe hat the succinctness and intuitiveness of LEESA not only requires fewer key-strokes but also fewer compilations than the equivalent object-oriented programs.

Furthermore, the upcoming C++ language standard (informally know as C++0x) has improved generic programming features such as, *variadic templates* that allows arbitrary number of template parameters. With variadic templates, LEESA gains efficiency in compile-time computations due to their first-class status in the language. The results in Figure 4 show the improvement in compilation times for LEESA's internal metaprograms written using variadic templates as opposed to their library-level emulation using MPL typelists. We observed 10% to 23% reduction in compilation time depending upon the axis used. Savings were more pronounced for the descendant axis expressions because they use more complex meta-programs that that of the child axis.

4 Related Work

In this section we present a sampling of related research focused on integrating XML semantics in object-oriented languages.

Lämmel et al. have identified the fundamental differences in the type systems of the contemporary OO languages and that of XML, which is known as the X/O impedance [13] mismatch. XML data is naturally represented by *regular expression types* [10] or *regular types* in short. Popular schema definition languages (*e.g.,* XSD, DTD, RelaxNG) are all based on regular types. Common features of regular types, such as associative concatenation operator, untagged union, and Kneene-star give rise to a complex type system that is impossible to reflect in the type systems of contemporary OO languages.

Several research languages, such as XJ [9], XACT [12], Xtatic [12] embed XML type system into an OO language. XJ and XACT combine the Java and XSD type systems. XACT provides a programming model based on XML templates and XPath together with a type checker based on data-flow analysis. Xtatic is an extension of C# programming language with several features from the XML type system, such as tree and sequence constructors for data and regular types for data representation. Tight integration of the two type systems in the languages often gives static guarantees of validity of the generated XML data.

Language Integrated Query (LINQ) [4] is a Microsoft technology used to support SQL-like queries natively in a program to search, project and filter data in arrays, XML, relational databases, and other third-party data sources. LINQ expressions are designed to be embedded in .NET languages, particularly C#.

S-XML [6] and XML Translation Language [20] are functional languages embedded in Scheme and Haskell, respectively, for creating and processing XML-like trees. These languages are designed to exploit the functional characteristics of their host languages.

XTL [16] presents an extensible typing library in C++, which implements domain-specific enhancements to the C++ type system. XTL also defines a type system for the XML processing language, capable of statically guaranteeing that a program only produces valid XML documents according to a given XML Schema.

X/O impedance [13] identifies the lack of fidelity in contemporary X-to-O mapping (XML data binding) tools towards supporting XML semantics and programming patterns & practices native to the XML domain. LEESA, however, recognizes the practical usefulness of these mappings and attempts to improve the extraction of typed data by incorporating the idioms of XML querying (*e.g.,* XPath axes, wildcards) without losing type-safety. In that sense it supports typed XML processing and alleviates the disappointment of a XML programmer in the typed world.

5 Conclusion

XML data-binding tools are increasingly being used for XML document processing in C++ because the automatically generated language bindings from the XML schema improves the type-safety of the programs. The gain in type-safety, however, comes at the expense of native XML programming styles and idioms, such as succinct axis-oriented queries (XPath), structure-shyness, and the fidelity to the full XML information set semantics. As a consequence, typed XML processing programs in C++ are often verbose and tightly coupled to the underlying XML data structure.

This paper addresses the limitations of XML data-binding tools using the multi-paradigm capabilities of C++. It develops a programming model for XML processing that supports axis-oriented and structure-shy queries without losing type-safety. The viability of the approach is demonstrated in a C++ library named LEESA. Although LEESA can not support all the capabilities of XPath, it has valuable XML processing features, such as depth-first traversal and tuple-fication that XPath does not provide. Compile-time and run-time performance evaluations of LEESA show that it is a practical alternative to existing XML processing libraries.

LEESA is a step forward toward integrating typed XML processing in C++, but much remains to be accomplished. We believe that the upcoming C++ language standard, C++0x, which dramatically improves the generic programming support in C++, offers plethora of opportunities to advance the fidelity and performance of XML integration in C++. LEESA can be downloaded in open-source from `www.dre.vanderbilt.edu/LEESA`.

References

[1] XML Data Binding for C++. http://www.codesynthesis.com/products/xsd.

[2] A. Klain. Blind XPath Injection. Whitepaper from Watchfire, 2005.

[3] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[4] Anders Hejlsberg, Don Box et al. Language Integrated Query (LINQ).

[5] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt. A Platform-Independent Component Modeling Language for Distributed Real-Time and Embedded Systems. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 190–199, Washington, DC, USA, 2005. IEEE Computer Society.

[6] J. Clements, M. Felleisen, R. Findler, M. Flatt, and S. Krishnamurthi. Fostering Little Languages. *Dr. Dobb's J.*, Mar. 2004.

[7] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, Massachusetts, 2000.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[9] M. Harren, M. Raghavachari, O. Shmueli, M. G. Burke, V. Sarkar, and R. Bordawekar. Xj: integration of xml processing into java. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, WWW Alt. '04, pages 340–341, New York, NY, USA, 2004. ACM.

[10] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.

[11] Howard Hinnant, Bjarne Stroustrup, and Bronek Kozicki. A Brief Introduction to Rvalue References, C++ Source, 2008.

[12] C. Kirkegaard and A. Moller. Type checking with XML Schema in XACT, Technical Report RS-05-31, BRICS. *Programming Language Technologies for XML, PLAN-X*, 2005.

[13] R. Lämmel and E. Meijer. Revealing the X/O Impedance Mismatch (Changing lead into gold). In *In Datatype-Generic Programming, volume 4719 of LNCS*, 2007.

[14] R. Lämmel, E. Visser, and J. Visser. The Essence of Strategic Programming. Draft; Available at `http://homepages.cwi.nl/~ralf/eosp`, Oct.15 2002.

[15] Ronald Bourret. XML Data Binding Resources.

[16] Y. Solodkyy, J. JÃd'rvi, and E. Mlaih. Extending Type Systems in a Library — Type-safe XML processing in C++. In *Proceedings of the Second International Workshop on Library-Centric Software Design (LCSD'06)*, pages 55–64, 2006.

[17] S. Tambe and A. Gokhale. LEESA: Embedding Strategic and XPath-Like Object Structure Traversals in C++. In *DSL '09: Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages*, pages 100–124, 2009.

[18] T. Veldhuizen. Expression Templates. *C++ Report*, 7(5):26–31, June 1995.

[19] E. Visser, Z. Benaissa, and A. Tolmach. Building Program Optimizers with Rewriting Strategies. In *Proceedings of the International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, 1998.

[20] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proc. of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*.

[21] World Wide Web Consontium (W3C). XML Information Set, W3C Recommendation. http://www.w3.org/TR/xml-infoset, Feb. 2004.

[22] World Wide Web Consontium (W3C). XML Schema Part 0: Primer Second Edition, W3C Recommendation. http://www.w3.org/TR/xmlschema-0, Oct. 2004.

[23] World Wide Web Consontium (W3C). XML Path Language (XPath), Version 2.0, W3C Recommendation. http://www.w3.org/TR/xpath20, Jan. 2007.