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 (*e.g.,* regular types, repeating subsequences), data model (*e.g.,* XML *information set* constituents such as, elements, attributes, processing instructions), schema languages for document description (*e.g.,* XSD, DTD, RELAX NG), programming languages (*e.g.,* XPath [9], 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. It may also be argued that such an approach may lead to malicious *code injection* attacks analogous to the SQL injection technique. Second, the results of such a query often require type casting to appropriate types, which is often computationally expensive.

Listing 2 C++ classes generated by a typical XML data binding tool for the catalog object-model

```
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 [6, 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). Listing 2 shows such an object model that reflects the structure of the book catalog XML. Well-known OO programming idioms are used to encapsulate the data behind intuitive classes and natural ways are provided for inspection/manipulation of the data through member functions only.

The downside to the OO-based approach are manifold stemming primarily from the fact that it leaves no opportunity to express the solution as succinctly as 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 query decouples itself from the concrete XML structure by omitting the "book" and "catalog" 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 `author` objects.

To alleviate 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 [2], generative programming [3], and strategic programming [8, 5] in combination with operator overloading can coexist in a single framework to resolve the gap between XPath-like notation and OO type-safety. The following solution is presented using our previous work on the **L**anguage for **E**mbedded qu**E**ry and trave**rSA**l (LEESA) [7].

```
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()
```

In the above code snippet, a key observation is that the C++ compiler type-checks the expression because it is not encoded as a string. The return value of the expression is a standard container of `name`s, where `name` is a C++ type derived from the schema. `AllDescendantsOf` is LEESA's manifestation of the descendant axis, which serves a purpose simular to that of "//" in XPath. Finally, a user-defined predicate function `from_england` is used to filter authors.

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 [4] remain important but are not discussed further.

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 2. 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* way to access the children could be via a generic accessor that can be parameterized by the desired children types. A key benefit of the *generic type-driven* access to data is that we can abstract the vocabulary-specific interface behind a uniform interface without losing type-safety. Furthermore, this approach is significantly more amenable to composition than classic OO *dot* notation

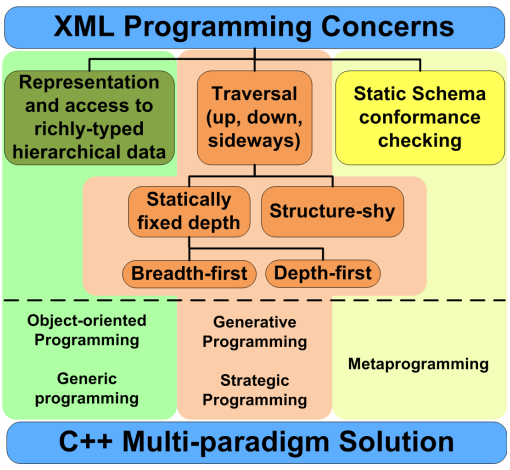

Figure 1: Major concerns of the XML programming paradigm and the proposed multi-paradigm solutions

as discussed in Section 2.2. 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 3 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 3. 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++ templates mechanism. In fact, this technique has been employed in our earlier work where the type parameter serves the same purpose as the second parameter in Listing 3. We choose overloaded functions here primarily for their simplicity.

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 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` would be indistinguishable if they are both represented using `std::string`. Nevertheless, such optimizations 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. 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. Alternatively, the script can be instructed to drop the ambiguous set of functions altogether.

2.2 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 significant 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*.

We achieve this desired goal using the generic program-

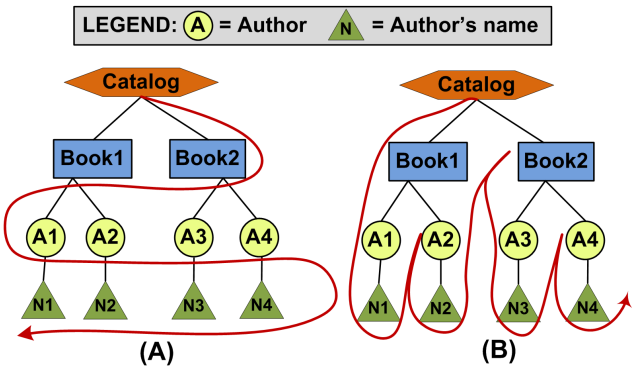

Figure 2: Variations in child axis traversal: (A) breadth-first (B) depth-first

ming paradigm and operator overloading in C++ to construct a highly intuitive, composable, and reusable notation for child axis that is well beyond the expressiveness of the OO paradigm alone.

Listing 4 A reusable, generic infrastructure in (simplified) 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 4 shows the implementation of a generic class, `Carrier<Kind>`, and a generic function, `operator ≫`, that allows us to express type-safe queries using a notation that aligns itself along the spirit 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 `Kinds`. The overloaded member function, `push_back`, treats a singular `Kind` object and a collection of `Kinds` uniformly. This uniformity is exploited on line #18 in the generic `operator ≫` function. 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, which subsequently selects an appropriate overloaded `children` function from Listing 3. Finally, the `operator ≫` function itself is reminiscent of the conventional overloaded extraction (≫) operators used for I/O, which can be chained to an arbitrary length.

Using the above data access technique realized in LEESA, we provide generic, reusable functionality for traversal along child, parent, and sibling axes. In particular, ≫ and ≫= operators are used for breadth-first and depth-first traversal of child axis whereas ≪ and ≪= operators are used for breadth-first and depth-first traversal of the parent axis. For instance, the following query implements the depth-first traversal shown in Figure 2(B).

```
catalog_root >>=  book() >>= author() >>= name()
```

It uses the depth-first (≫=) operator that executes the expression to its right in the context of a *single* object. In our example, `≫= book()` selects one book at a time and continues execution in the context of that book object. Subsequently, `≫= author()` and `≫= name()` do the same for authors and their names respectively. As a result, depth-first operators provide highly expressive notation for nested loops.

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.

```
std::vector<tuple<name, country> > tuples =
catalog_root  >>  book()  >>  author()  >>
  ChildrenAsTupleOf(author(), tuple<name, country>()))
```

These examples indicate that the ability to combine axes is a powerful technique for expressing complex queries succinctly and intuitively. To enable such expressiveness natively in C++, LEESA implements these operators using the sophisticated operator overloading idiom of *Expression Templates*. Details of LEESA's design appear in [7].

2.3 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 highly 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) [8, 5] and *Generative Programming* [3] paradigms supported by C++, we can achieve support for structure-shyness that rivals XPath.

Sidebar 1: Strategic Programming in a Nutshell

Strategic Programming (SP) began as a general-purpose program transformation [8] technique, which later evolved into a paradigm [5] 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. For instance, *FullTopDown* traversal scheme, composed using the *Sequence* and *All* combinators, descends into a tree and visits every node in depth-first manner.

Sidebar 1 presents an overview of SP. In our earlier work [7] 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 identifies the `Descendant` type objects and accumulates them as the recursive strategy descends into the XML tree. For instance, `AllDescendantsOf(catalog(), country())` collects all the "country" objects irrespective of their depth. Therefore, `AllDescendantsOf` presents an opportunity for XML programmers to express *typed structure-shy* queries.

In spite of the generality of strategic programming, constructing LEESA queries that emulate *wildcards* ('/*/') remains a challenge to the XML programmer. LEESA resolves this challenge using `LevelDescendantsOf`, which is an intuitive facade that mimics XPath's wildcards. For instance, the following expression is a typed equivalent of earlier presented query `"/*/*/country"`:

```
std::vector<country> countries =
  catalog_root >> LevelDescendantsOf(catalog(), _, _, country())
```

`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*.

2.4 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 5 shows the automatically generated meta-information for the catalog object model. For 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 5.

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 #13) inherits from `False`, indicating that the descendant re-

Sidebar 2: C++ Metaprogramming and Boost MPL

C++ templates, due to their support for *specialization*, give rise to a unique, purely functional (no side effects allowed) 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 [2] 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 = 1
mpl::contains<Integral, float>::value; // value = 0
```

Listing 5 Automatically generated meta-information (partial) for the catalog object model

```
 1: template <class Kind> struct SchemaTraits {
 2:  typedef mpl::vector<> Children;
 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: // Similarly SchemaTraits<author> contains name, country.
11: struct True  { enum { value = 1 }; };
12: struct False { enum { value = 0 }; };
13: template<class A, class D> struct IsDescendant : False {};
14: template<> struct IsDescendant<catalog,book>   : True {};
15: template<> struct IsDescendant<catalog,title>  : True {};
16: template<> struct IsDescendant<catalog,price>  : True {};
17: template<> struct IsDescendant<catalog,author> : True {};
18: template<> struct IsDescendant<catalog,name>   : True {};
19: template<> struct IsDescendant<catalog,country>: True {};
20: // Similar specializations for book (5) and author (2).
```

lationship 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 4).

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

Using the Boost static-assert library and the externalized meta-information, it 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 for industrial strength XML documents, we report on the performance comparisons between LEESA's multi-paradigm approach and the pure object-oriented approach used by contemporary XML data binding tools.

3.1 Compile-time Performance

LEESA's reliance on C++ template metaprogramming requires 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. We evaluated a single LEESA expression of each query type shown in the table against equivalent programs written using OO abstractions only.

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 highly 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.

Comparisons of the compilation times for the test programs written using the large schema are shown in Figure 3. 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 that the succinctness and intuitiveness of LEESA not only requires fewer key-strokes but also fewer compilations than the equivalent object-oriented programs.

Furthermore, *variadic templates*, an upcoming C++ language feature that allows arbitrary number of template parameters will help reduce the compilation overhead dramatically. Our initial results (not shown) revealed an order of magnitude improvement in the compilation times for LEESA's internal metaprograms written using variadic templates as opposed to their library-level emulation using MPL typelists.

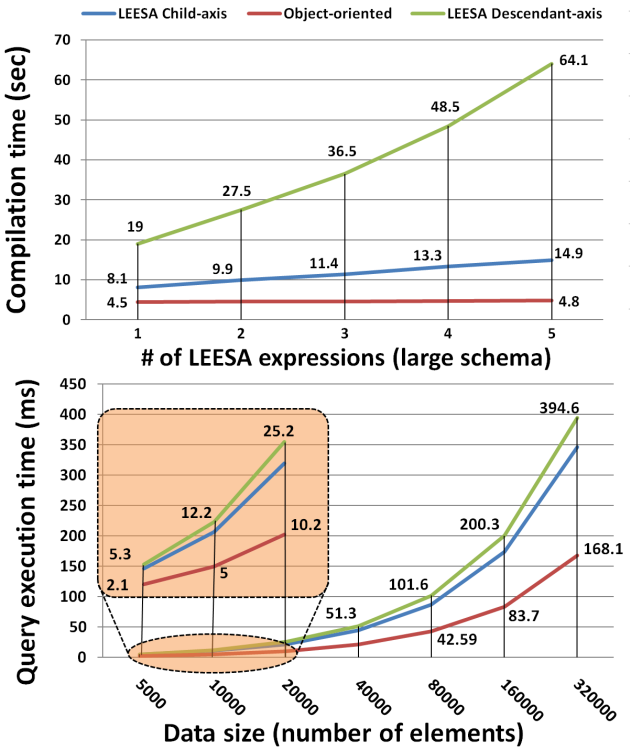

Figure 3: Run-time and compile-time comparisons of LEESA and the pure OO solution

3.2 Run-time Performance

Figure 3 compares the run-time performance of a LEESA query with an optimized hand-written object-oriented solution. We compared the time needed to construct a standard container of `name` objects from a set of large book catalogs. LEESA's descendant axis has consistently higher overhead by a factor of 2.5 compared to the hand-written solution. This *abstraction penalty* stems from the construction, copying, and destruction of the internal dynamic data structures LEESA maintains.

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 36 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 allows transparent performance improvements using the upcoming C++ language features such as *rvalue references* and threading facilities for multi-core processing, which may otherwise require costly OO refactoring.

4 Conclusion

Language integrated XML querying has been deployed successfully in other languages (*e.g.,* Linq in C#). LEESA is a step forward toward integrating typed XML querying using multi-paradigm capabilities 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 further 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://codesynthesis.com/products/xsd.

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

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

[4] 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.

[5] R. Lämmel, E. Visser, and J. Visser. The Essence of Strategic Programming. 18 p.; Draft; Available at `http://homepages.cwi.nl/~ralf/eosp`, Oct.15 2002.

[6] Ronald Bourret. XML Data Binding Resources. http://www.rpbourret.com/xml/XMLDataBinding.htm.

[7] 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.

[8] 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.

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