Mapping Demeter to XML Schema and XPath

Will Guaraldi and Karl Lieberherr
{guaraldi,lieber}@ccs.neu.edu
College of Computer and Information Science
Northeastern University
360 Huntington Avenue 202 WVH
Boston, Massachusetts 02115 USA

ABSTRACT

The Demeter project is a collection of tools and technologies that allow for Adaptive Programming and structure-shy development. Software development using Demeter separates object definition from operations on objects giving both pieces room for change over the course of the software lifecycle [2].

This paper updates some of the material in the paper *Adaptive XML/Java Data-Binding* which discusses adaptive techniques for binding XML documents to Java classes [5] and the paper *Improving XPath Evaluation with Strategies* which discusses logical optimizations of evaluation of XPath expressions using traversal algorithms from Demeter [9]. Additionally, this paper provides a mapping of nomenclature, ideas and syntax of Demeter class dictionaries/graphs to XML Schema schema and also Demeter strategies to XPath expressions.

The end result of this paper is a mapping of Demeter things to their XML-world equivalents and also a strategy syntax that's a subset of XPath 2.0 for specifying strategies allowing Demeter implementations to alternatively use the more widely used XPath syntax.

Keywords

Demeter, XMLSchema, XPath, traversal strategies, class graphs

1. INTRODUCTION

Demeter is a set of technologies and tools that enable Adaptive Programming. At the time of this writing, the implementations of Demeter technologies include DJ, which is a library that allows runtime construction and interpretation of traversal strategies in Java [8]; DemeterJ, which is a code generation tool that takes a class dictionary file and behavior files and generates Java code combining the specifed structure and behavior together [3]; and DAJ [1], which is similar to DemeterJ but uses aspects to add behavior.

We'll start off by mapping Demeter class dictionaries and class graphs to XML Schema schema and XML documents. Then we'll work on mapping Demeter strategies to XPath 2.0 expressions.

FIXME - This section gives a rough explanation of the Demeter system including explanations of class graphs and strategies. This section also talks about XML Schema and XPath. This section should introduce all the things this paper talks about.

We'll be using variations on the BusRoute problem to illustrate Demeter class graphs/dictionaries, XML Schema schema, Demeter strategies, and XPath expressions. A BusRoute consists of two things: a BusList and a BusStopList. These hold the list of buses that belong to the route and the list of stops that are on the route. At any point in time, each Bus in the BusList has zero or more passengers on it and each BusStop in the BusStopList has zero or more passengers waiting for the next bus.

2. DEMETER CLASS DICTIONARIES AND XML SCHEMA

Demeter class graphs are finite directed graphs formed from a set of nodes which represent classes and a set of directed edges between nodes which represent relations between classes [7].

There are two kinds of classes that can be in a class graph: *abstract classes* and *concrete classes*. There are two kinds of edges in a class graph: *construction edges* (also known as *has-a edges*) and *subsclass edges* (also known as *is-a edges*). Edges can be labelled.

FIXME - add drawing of a class graph

In DemeterJ and DAJ, two tools that implement Demeter technologies, a class graph is specified using a class dictionary [1] [3] which is a textual representation of a class graph with additional syntactic sugar. The class dictionary takes the form of an LL(1) grammar with syntactic bits that guide a parser which takes in a stream of characters and generates an object graph.

In the class dictionary, there are several kinds of rules. Concrete rules have a `=` sign. A concrete rule defines a concrete class that has no subclasses. The classes to the right of the `=` sign are connected to the class with *has-a* edges. These edges may be labeled. Here is an example of a concrete rule that defines the `BusRoute` class. The `BusRoute` has two labeled edges: one named `buses` which leads to a `BusList` and one named `busstops` which leads to a `BusStopList`.

```
BusRoute = <buses> BusList <busstops> BusStopList .
```

Abstract rules have a `:` sign. Abstract rules define abstract classes and the classes to the right of the `:` sign are subclasses of the class and connected to the class with *is-a* edges. Here is an example of an abstract rule that defines a `StringList`. The `StringList` is either an `EmptyList` or a `NoneEmptyList`.

```
StringList     :   EmptyList | NonEmptyList .
EmptyList      =   .
NonEmptyList   =   <head> StringItem
                   <tail> StringList .
StringItem     =   String .
```

Here is an example of an abstract rule that defines the `BusList` class. The `BusList` class is either an `EmptyBusList` or it's a `NonEmptyBusList`.

Collection rules have a ˜ sign. Collection rules allow us to define parameterized classes such as lists. We can create a list more generally that can work for strings and any other classes.

```
List(S)      ~   { S } .
StringList   =   List(StringItem) .
StringItem   =   String .
```

IClass dictionaries also allow you to specify optional items. For example, if some of the people in our `BusRoute` had baggage that needs to be stored in the bins underneath the bus, we could have an optional `Bag` class. It's presence in the object graph signifies that the person has a bag; it's lack of presence indicates the person does not have a bag.

In this snippet, we denote the optionality of the bag with square brackets. A person has a name and may or may not have one bag.

```
Person   =   <name> String [Bag] .
Bag      =   <desc> String .
```

Class dictionaries allow us to label edges using the `<...>` syntax. If we didn't explicitly label the edge in the class dictionary, then the edge is assigned a label that is the lower-cased name of the target class.

To loosely summarize, we need to map the following things into XML Schema equivalents:

- concrete classes
- abstract classes
- subclasses that inherit from their parent class
- parameterized classes
- collection classes such as lists
- class recursion
- terminals
- labeled *has-a* edges
- root classes
- optional classes

The XML Schema definition language is used to define schemas that specify the structure and constraints for a class of XML documents [6]. Schemas themselves are XML documents composed of elements that are defined in the XML Schema Definition namespace which is often "xsd".

This namespace defines 13 schema-components which fall into three different groups: primary components, secondary components, and helper components [6]. For the purposes of this paper, I'm only going to talk about the primary components. The other components help to specify and constrain elements in a schema, but don't have a Demeter class graph equivalent. We'll talk more about this later.

The primary schema-components consist of:

- simple type definitions
- complex type definitions
- attribute declarations
- element declarations

Simple types are the set of built-in types and types that are derived from the built-in types by placing restrictions on them. Simple types cannot contain other elements and they cannot have attributes. For example, `xsd:string` is a built-in type and is a simple type. We could define another simple type `BusNumber` which is a four-character string where all characters are digits:

```
<xsd:simpleType name=”BusNumber”>
  <xsd:restriction base=”xsd:string”>
    <xsd:pattern value=”\d{4}” />
  </xsd:restriction>
</xsd:simpleType>
```

Complex types can contain elements, have attributes, and can extend other complex types.

Complex types are very much like classes in Demeter class dictionaries. Without loss of expressiveness, we can ignore specifying attributes in complex types and stick with elements when mapping complex types to Demeter classes.

Elements have a name attribute and are contained by complex types. Elements defined at the top level of a schema are root elements of the document.

When you specify that an element is part of a complexType, you can also specify the `minOccurs` and `maxOccurs` attributes which specify the number of times the element should occur. For example:

```
<xsd:complexType name=”PersonList”>
  <xsd:sequence>
    <xsd:element name=”person”
                 minOccurs=”0”
                 maxOccurs=”unbounded” />
  </xsd:sequence>
</xsd:complexType>
```

Complex types can inherit from other complex types using `extension`. For example:

```
<xsd:complexType name=”A”>
  ...
</xsd:complexType>

<xsd:complexType name=”B”>
  <xsd:complexContent>
    <xsd:extension base=”A”>
      ...
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Complex type definitions can recurse. We can also declare complex types as "abstract" which prevents XML document elements from having that type.

Here's the `StringList` example we mentioned before in XML Schema:

```
<xsd:element name="stringlist"
             type="StringList" />
<xsd:complexType name="StringList"
                 abstract="true" />
<xsd:complexType name="EmptyList">
  <xsd:complexContent>
    <xsd:extension base="StringList" />
  </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="ConsList">
  <xsd:complexContent>
    <xsd:extension base="StringList">
      <xsd:sequence>
        <xsd:element name="head"
                     type="xsd:string" />
        <xsd:element name="tail"
                     type="StringList" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

XML Schema doesn't allow for parameterization of types like Demeter class dictionaries do.

This gives us the mapping found in Figure 1 of Demeter class dictionary things to XML Schema equivalents.

3. DEMETER STRATEGIES AND XPATH

A Demeter strategy specifies a pathset between classes in a graph Γ. A pathset has a starting point A (also called a *source*) and an ending point B (also called a *target*). There are several ways to manipulate pathsets [7]:

$$D ::= [A,\ B] \mid D \cdot D \mid D + D$$

where:

$[A,\ B]$	the pathset from A to B in graph Γ
$D_1 \cdot D_2$	concatenation of pathsets D_1 and D_2 by connecting the targets of D_1 to the sources of D_2
$D_1 + D_2$	the union of pathsets D_1 and D_2

These rules allow us to think about strategies in terms of single edges since single edges form the building blocks of a bigger strategy.

There are two primary strategy syntaxes used by DemeterJ [3]: line graph notation and strategy graph notation.

The line graph notation notation looks like this:

```
from A via B via C to D
```

This strategy starts with node `A` passes through nodes `B` and `C` and ends at node `D`.

We can also bypass edges:

```
from A bypassing B to D
```

which will skip edges going through `B`.

Demeter Class Dictionary	**XML Schema**
class dictionary	schema
object graph	XML document
classes	types (complex and simple)
terminals examples: String, Integer, Float, ...	built-in atomic types examples: xsd:string, xsd:integer, xsd:float, ...
classes on the left-hand-side of rules classes on the right-hand-side of rules labeled edges example: BusRoute = <buses> BusList <busstops> BusStop .	complexType definition element element *name* attribute example: <xsd:complexType name=”BusRoute”> <xsd:complexContent> <xsd:sequence> <xsd:element name=”buses” type=”BusList”> <xsd:element name=”busstops” type=”BusStop”> </xsd:sequence> </xsd:complexContent> </xsd:complexType>
abstract rule example: StringList : EmptyList | NonEmptyList . EmptyList = . NonEmptyList = <head> StringItem <tail> StringList . StringItem = String .	extension example: <xsd:element name="stringlist" type="StringList" /> <xsd:complexType name="StringList" abstract="true" /> <xsd:complexType name="EmptyList"> <xsd:complexContent> <xsd:extension base="StringList" /> </xsd:complexContent> </xsd:complexType> <xsd:complexType name="ConsList"> <xsd:complexContent> <xsd:extension base="StringList"> <xsd:sequence> <xsd:element name="head" type="xsd:string" /> <xsd:element name="tail" type="StringList" /> </xsd:sequence> </xsd:extension> </xsd:complexContent> </xsd:complexType>
root classes	elements declared at top level
lists of zero or more items	minOccurs=”0” maxOccurs=”unbounded”
lists of one or more items	minOccurs=”1” maxOccurs=”unbounded”
optional classes	minOccurs=”0” maxOccurs=”1”

Figure 1: Mapping Demeter class dictionary to XML Schema

You can also specify sets of nodes:

```
from A via { B, C } to D
```

Instead of nodes, we can specify edges that should be traversed. There are two edges we can refer to: construction edges and subclass edges.

`-> A, b, B` - construction edge from `A` to `B` through labeled edge `b`.

`-> A, B` - subclass edge from `A` to `B`.

We can use `*` as a wildcard.

The other notation we can use is the strategy graph notation. It's more general and allows us to express strategies that are difficult to express with line graph notation. Strategy graph notation consists of a set of edges like this:

```
{ A -> B
  B -> C
  C -> D }
```

XPath 2.0 is a specification for expressions that denote node sequences. XPath 2.0 expressions read more like the Demeter line graph strategy notation, so we'll focus on mapping that notation. It's not clear whether we could map strategy graph notation to XPath 2.0 expressions.

Demeter strategies specify a pathset. While the pathset has a set of target nodes and you can retrieve those target nodes using the `gather` method, there's a large difference between what an XPath 2.0 expression results in and what a Demeter strategy results in.

For now we're going to ignore the fact that XPath 2.0 expressions only result in a node sequence and treat them as if expressions specified pathsets.

The XPath 2.0 specification [4] has an EBNF grammar at the end that walks through the expression grammar. We'll talk about the portions that are equivalents to Demeter strategy syntax.

XPath 2.0 calls edges steps. You can step from one node to another or you can step from one node past a bunch of nodes to a target node. The first kind of step is specified with `/` and the second kind with `//`.

Examples:

`A/B` - from `A` to `B` with no steps in between.

`A//B` - from `A` to `B` with one or more steps in between.

XPath 2.0 allows you to connect multiple steps together with the step syntax. The nodes specified in the step syntax are abbreviations.

There are element tests that can match nodes by name or type notation:

`element(*, A)` - is the element of type `A`.

`element(a, *)` - is the element of name `a`.

`element(a, A)` - is the element of type `A` and of name `a`.

FIXME - this section is incomplete.

4. XLINQ? AND OTHER WORK

FIXME

5. CONCLUSIONS

FIXME

6. REFERENCES

[1] DAJ. http://daj.sourceforge.net/ .

[2] Demeter project. http://www.ccs.neu.edu/research/demeter/ .

[3] DemeterJ. http://www.ccs.neu.edu/research/demeter/software/DemeterJ/ .

[4] http://www.w3.org/TR/2007/REC-xpath20-20070123/ , October 2007.

[5] P. Adak, H. He, and K. Lieberherr. Adaptive XML/Java Data-Binding. Technical Report NU-CCS-2001-04, College of Computer Science, Northeastern University, Boston, MA, May 2001.

[6] M. M. Henry S. Thompson, David Beech and N. Mendelsohn. XML schema part 1: Structures second edition. http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/, October 2004.

[7] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.

[8] D. Orleans and K. Lieberherr. DJ: Dynamic Adaptive Programming in Java. Technical Report NU-CCS-2001-02, College of Computer Science, Northeastern University, Boston, MA, March 2001.

[9] J. Palm and K. J. Lieberherr. Improving XPath evaluation with strategies. Technical report, Northeastern University, September 2005.

Demeter Strategy	XPath 2.0 Expression
from A to B	A//B
from A via (-> *, b, B) to B	A//element(b, B)
FIXME - complete this table	

Figure 2: Mapping Demeter strategy to XPath 2.0 expression

```
BusRoute             =   <buses> BusList <busstops> BusStopList .
BusList              :   EmptyBusList | NonEmptyBusList .
EmptyBusList         =   .
NonEmptyBusList      =   <first> Bus <rest> BusList .
Bus                  =   <passengers> PersonList .
BusStopList          :   EmptyBusStopList | NonEmptyBusStopList .
EmptyBusStopList     =   .
NonEmptyBusStopList  =   <first> BusStop <rest> BusStopList .
BusStop              =   <waiting> PersonList .
PersonList           :   EmptyPersonList | NonEmptyPersonList .
EmptyPersonList      =   .
NonEmptyPersonList   =   <first> Person <rest> PersonList .
Person               =   <name> String
```

Figure 3: The BusRoute class dictionary with recursions

```
BusRoute   =   <buses> List(Bus) <busstops> List(BusStop) .
Bus        =   <passengers> List(Person) .
BusStop    =   <waiting> List(Person) .
List(S)    ~   { S } .
Person     =   <name> String .
```

Figure 4: The BusRoute class dictionary with collections

[1]	`XPath`	::=	`Expr`
[2]	`Expr`	::=	`ExprSingle ( “,” ExprSingle)*`
[3]	`ExprSingle`	::=	`OrExp`
...			
[8]	`OrExpr`	::=	`AndExpr ( “or” AndExpr)*`
[9]	`AndExpr`	::=	`ComparisonExpr`
[10]	`ComparisonExpr`	::=	`RangeExpr`
[11]	`RangeExpr`	::=	`AdditiveExpr`
[12]	`AdditiveExpr`	::=	`MultiplicativeExpr`
[13]	`MultiplicativeExpr`	::=	`UnionExpr`
[14]	`UnionExpr`	::=	`IntersectExceptExpr ( (“union” | “|”)`
			`IntersectExceptExpr )*`
[15]	`IntersectExceptExpr`	::=	`InstanceofExpr`
[16]	`InstanceofExpr`	::=	`TreatExpr`
[17]	`TreatExpr`	::=	`CastableExpr`
[18]	`CastableExpr`	::=	`CastExpr`
[19]	`CastExpr`	::=	`UnaryExpr`
[20]	`UnaryExpr`	::=	`ValueExpr`
[21]	`ValueExpr`	::=	`PathExpr`
...			
[25]	`PathExpr`	::=	`(“/” RelativePathExpr?)`
			`| (“//” RelativePathExpr)`
			`| RelativePathExpr`
[26]	`RelativePathExpr`	::=	`StepExpr ((“/” | “//”) StepExpr)*`
[27]	`StepExpr`	::=	`AxisStep`
[28]	`AxisStep`	::=	`ForwardStep PredicateList`
[29]	`ForwardStep`	::=	`(ForwardAxis NodeTest)`
			`| AbbrevForwardStep`
[30]	`ForwardAxis`	::=	`(“child” “::”)`
			`| (“descendent” “::”)`
			`| (“attribute” “::”)`
			`| (“self” “::”)`
			`| (“descendent-or-self” “::”)`
			`| (“following-sibling” “::”)`
			`| (“following” “::”)`
			`| (“namespace” “::”)`
[31]	`AbbrevForwardStep`	::=	`“@”? NodeTest`
...			
[35]	`NodeTest`	::=	`KindTest`
...			
[46]	`ParenthesizedExpr`	::=	`“(” Expr? “)”`
...			
[54]	`KindTest`	::=	`ElementTest`
...			
[64]	`ElementTest`	::=	`“element” “(” (ElementNameOrWildcard`
			`(“,” TypeName “?”?)?)? “)”`
[65]	`ElementNameOrWildcard`	::=	`ElementName | “*”`
...			
[69]	`ElementName`	::=	`QName`
[70]	`TypeName`	::=	`QName`

Figure 5: Subset of XPath for specifying strategies