A Modeling System for Mixed Integer Linear Programming Using XML Technologies

Kipp Martin
Graduate School of Business
University of Chicago

December 11, 2002
Revised: December 29, 2003

Abstract

We show that XSLT and XPath can be used in lieu of a traditional algebraic modeling language such as AMPL, GAMS, LINGO, MPL, etc. for generating the instance of a mixed integer linear program. Our approach is to take raw data files in XML format and transform them using XPath and XSLT into a single XML file that represents the instance of a linear program. An advantage of this approach is that the operations research community can easily share and reuse XSLT model templates regardless of computing platform. All of the required software is open source and available on all major platforms. Also, this is the natural way to work with XML data, which is becoming an increasingly popular standard for storing and transmitting data. We *are not* proposing a new modeling language syntax. Rather, we demonstrate that the existing W3C Recommendations for XPath 1.0 and XSLT 1.0 contain the necessary functionality for defining sets and indices, and operating on them by looping and summing. Supporting documents for this paper are available at `http://gsbkip.uchicago.edu/xslt/xslt.html`.

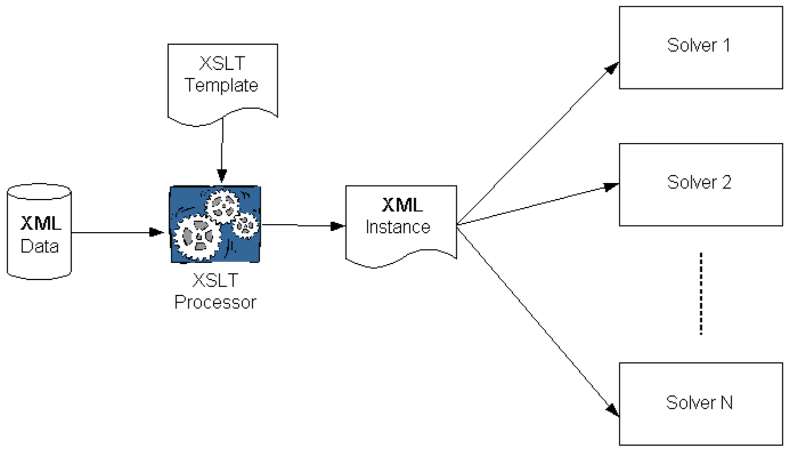
1 Introduction

XML (Extensible Markup Language) is hot. Whether in the the technical press or mainstream business press, each day is filled with new articles about XML-related technologies. XML is about data. One might argue that mathematical modeling is also about data. Indeed, a mathematical programming modeling language, and associated solver tools, will not be used unless they are closely integrated with corporate data. The importance of integrating mathematical programming tools with corporate data has been stressed before. See, for example, Atamtürk et al. [3], Choobineh [6], and Mitra et al. [22]. Perhaps one reason for the success of Excel based solvers such as Excel Solver from Frontline Systems [27] and What's *Best!*, from Lindo Systems, Inc. [20] is their close integration with a spreadsheet.

This paper is based on two premises. First, that XML is rapidly becoming an accepted format for storing data and that if the data of interest are not in an XML format, tools exist to easily transform the non-XML data into an XML format (this is the theme of Section 7 "Getting the Data"). Second, that there are powerful open source platform independent tools for taking XML data stored in one format and transforming it into another XML format. Thus, if there is an agreed upon XML standard for what a linear program instance should look like, we can use the transformation tools to transform the raw XML data into the linear programming XML instance format.

The objective of this paper is to demonstrate that the XPath and XSLT technologies are sufficient for transforming the raw problem data in an XML format into a linear programming instance in an XML format. This process is illustrated in Figure 1. An advantage of this approach is that an algebraic modeling language is not required and the operations research community can easily share and reuse XSLT model templates regardless of computing platform. All of the required software is open source and available on all major platforms. Also, this is the natural way to work with XML data, which is becoming an increasingly popular standard for storing and transmitting data.

Figure 1: The XSLT template approach to mathematical modeling



This is a proof of concept paper. In this respect, it is similar to the work of Atamtürk et al. where these authors demonstrate that SQL is sufficient for generating linear programming problem instances. Earlier, Choobineh [6] developed SQLMP, an SQL based modeling system for linear programming.

Throughout, we make a key distinction between the *algebraic representation* of a model and an *instance* of a model. For example, an algebraic representation of a knapsack linear program in a pseudo modeling language is

```
MODEL:
SET a, c, x;
MAX = SUM(SET: c*x);
SUM(SET: a*x) <= b;
```

Algebraic modeling languages with features such as sets, indices, summation, and looping include AMPL [11], GAMS [5], LINGO [23], and MPL [25]. An example instance of this linear program is

```
MAX = 2*x1 + 3*x2 + x3;
5*x1 + 3*x2 + 4*x3 <= 10;
```

Our goal is generating mixed integer linear programming model instances by transforming raw XML data into an XML model instance using XPath and XSLT. Our goal *is not* the development of an XML dialect for the algebraic representation of a linear programming problem. We want a methodology where the minimum amount of agreement about model syntax is required among the user community. One need only witness the proliferation of algebraic modeling languages mentioned earlier to be pessimistic about the development of an industry standard XML based modeling language. This is why Fourer, Lopes, and Martin are currently working on an XML dialect for representing an instance of a linear program. See [12] (work in progress). This is the lowest common denominator and probably requires the least amount of agreement.

XPath and XSLT are existing W3C standards. The transformation illustrated in Figure 1 is accomplished via XSLT *templates.* An XSLT template is used as a modern version of a matrix generator. However, XSLT is based upon functional programming and is not a procedural language. XSLT makes it very easy to transform raw data into a linear programming instance. With the XSLT template approach, users can share templates. This differentiates our approach from using an algebraic modeling language. For example, a person with a LINGO model cannot share it with an AMPL user. Templates, on the other hand, are platform neutral. They can be used with any of the open source software discussed in Section 6.

This paper covers the following. In Section 2, "XML Technologies," we provide a brief description of the technologies used in this paper. They are XML, W3C XML Schemas, Namespaces, XPath, and XSLT. In Section 3, "The Basic Approaches," we describe three possibilities for incorporating XML into a mathematical programming modeling system. Algebraic modeling languages are based on sets, indices, and the manipulation of sets. In Section 4, "Sets, Indices, and Data," we show how to use XPath to create desired node-sets of data. These node-sets can be manipulated (e.g. set difference, intersection, union) and used to generate indices. Perhaps the key section is Section 5, “Model Generation with XSLT and XPath,” where we show how to actually create an XML instance of a linear program using XSLT and XPath. All mathematical modeling systems depend upon data. Indeed, one reason for writing this paper is to provide a modeling system based on the increasingly ubiquitous XML format. In Section 6, “Software,” we provide the reader with a litany of available software for implementing the modeling methodology developed in this paper. In Section 7, “Getting the Data,” transforming the most common non-XML formats such as enterprise relational databases, spreadsheets, text flat files, or desktop databases into XML format is described. One motivation for using the XML technologies is the availability of open source software for multiple platforms. In Section 8, “Computational Results,” we give some preliminary computational results demonstrating that using XSLT and XPath is a viable modeling approach. This paper is based upon XPath and XSLT Recommendations 1.0. Versions 2.0 are working drafts of the W3C. The 2.0 versions offer important enhancements for mathematical modeling. The importance of these enhancements is described in Section 9, “Conclusion and Future Trends.” We also extend our ideas to nonlinear programming.

The complete source listing (XML files, XSLT templates, etc.) for the examples used in this paper are contained in a separate appendix. They are also available for download at `http://gsbkip.uchicago.edu/xslt/xslt.html`.

2 XML Technologies

In this section we give a brief overview of the XML technologies used in this paper. For a good overview of all of these technologies see Skonnard and Gudgin [24].

2.1 XML

It is common practice to store data in a relational database system. Two aspects of commercial relational database systems are 1) the data are stored in multiple tables or relations, and 2) the files containing the data are typically binary files. XML data is 1) stored using a tree structure, and 2) stored as a text file containing both tags and the data.

We illustrate with a multiproduct dynamic lot sizing model. See Wagner and Whitin [29]. This model is used throughout the paper. Assume there are two products with a four period planning horizon and that inventory holding cost, marginal production cost, and fixed production cost depend on product but not time period. These data are represented in Tables 1 through 3. The keys are in boldface type. The cost data are in Table 1. The costs are functionally dependent on only the **productID.** The demand data are in Table 2. The demand is functionally dependent on the **productID** and **periodID.** The capacity data are in Table 3. The capacities are functionally dependent on only the **periodID.**

We could store all of these data in a single table, however, such a table would not be in second normal form. These tables represent a normalized presentation of the data. See Ullman [28].

Table 1: Product costs data table

productID	holdCost	prodCost	fixedCost
1	1	7	150
2	2	4	100

Table 2: Demand data table

productID	**periodID**	demand
1	1	60
1	2	100
1	3	140
1	4	200
2	1	40
2	2	60
2	3	100
2	4	40

The demand data in Table 2 for the first product, i.e., the first four records of the relation, expresssed as XML appear in Figure 2. In this XML representation there are two *element* tags. The `root` element is `demandRoot`. There is also a `demand` element tag corresponding to each row of the relation in Table 2. Each `demand` element has three *attributes*. They are `productID`, `periodID`, and `demand`. These attributes describing the `demand` element correspond exactly to the attributes in the table of a relational database. We could use a similar XML "rectangular" storage format for the relations in Tables 1 and 3.

Figure 2: An XML representation of demand data

```
<?xml version="1.0" encoding="UTF-8"?>
<demandRoot>
   <demand productID="1" periodID="1" demand="60"/>
   <demand productID="1" periodID="2" demand="100"/>
   <demand productID="1" periodID="3" demand="140"/>
   <demand productID="1" periodID="4" demand="200"/>
</demandRoot>
```

However, the tree structure of XML is more flexible than the table structure of a relational database. An alternative XML representation is given in Figure 3 where we show the fragment of an XML file that contains both the cost and demand information for each product. This file contains the data in both Table 1 and Table 2. In this fragment of the file we show only the cost and demand data for the first product. By nesting the `period` elements as children of the `product` elements we store both the

Table 3: Time period production capacity data table

periodID	capacity
1	200
2	200
3	200
4	200

Figure 3: A second XML representation of demand data

```
<?xml version="1.0" encoding="UTF-8"?>
<linearProgram>
    <product productID="1" holdCost="1"  prodCost="7"
        fixedCost="150">
        <period periodID="1">
            <demand>60</demand>
        </period>
        <period periodID="2">
            <demand>100</demand>
        </period>
        <period periodID="3">
            <demand>140</demand>
        </period>
        <period periodID="4">
            <demand>200</demand>
        </period>
```

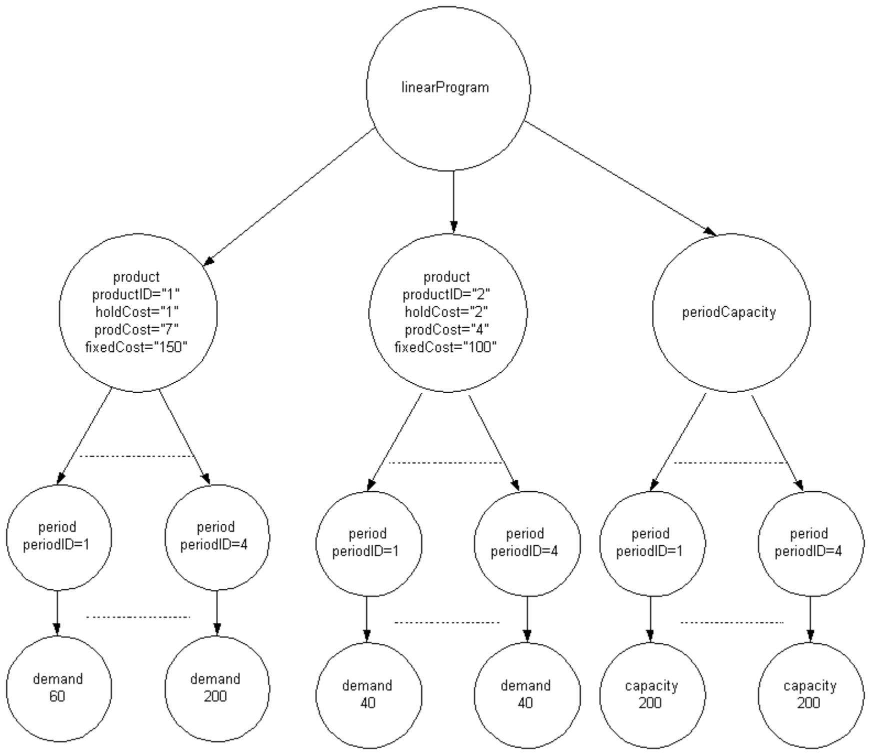
demand and product cost data together without creating any redundancy. It is also possible to store the `product` elements as children of the `period` elements. However, this would cause redundancy in the data since the holding cost, production cost, and fixed cost are independent of time and it is not necessary to repeat their values for each time period. A graphical representation of the XML file in Figure 3 appears in Figure 4. Notice that the nodes corresponding to the time periods are *children* of the nodes corresponding to products.

2.2 Schemas

An XML document must be *well formed* in order to be parsed and the appropriate tree constructed. Unlike an HTML document, a well formed XML document requires that:

- opening and closing tags be present,
- tags are case sensitive and opening and closing tags agree on case,
- tags be nested properly

Figure 4: Tree structure for dynamic lot size model



Even a well formed XML document may be ambiguous. For example, there are numerous hedge funds actively trading financial options. Different funds might wish to exchange financial information concerning executed trades. One hedge fund might use the tag pair `<strike>` and `</strike>` to denote the strike price of an option. However, if another fund chooses to use the tag pair `<strikeprice>` and `</strikeprice>` there is a problem if the two funds wish to exchange XML data files. It is crucial that there exist an industry standard vocabulary, and that XML files be *validated* against the industry standard.

One way to accomplish this is through the use of the W3C XML Schema (or the older document type definition (DTD)). Think of an XML document instance as an object (instance of a class) and the XML Schema as a class or set of classes. The Schema describes contents of the XML document. For example, in our dynamic lot size model there is a `product` element. In the XML Schema the product is described as

```
<xs:complexType name="product">
   <xs:sequence>
      <xs:element name="period" type="period"
        maxOccurs="unbounded"/>
   </xs:sequence>
   <xs:attribute name="productID" type="xs:string"
        use="required"/>
   <xs:attribute name="holdCost" type="xs:double"
        use="required"/>
   <xs:attribute name="prodCost" type="xs:double"
        use="required"/>
   <xs:attribute name="fixedCost" type="xs:double"
        use="required"/>
</xs:complexType>
```

A nice feature of the W3C XML Schema is the `<xs:key>` tag. This tag is used just as a key is used in a relational database. For example, in the lot sizing schema we could declare `productID` and `periodID` to be keys. Then when the XML instance file is validated against a schema we enforce the condition that there is a unique `productID` and that for each product the `periodID` is unique. This is obviously a nice feature for maintaining data integrity.

2.3 Namespace

XML tags are used to give the meaning to data. However, there is a potential conflict with the use of tags to markup data. Different XML dialects could select the same tag name. For example, the tag `<title>` in one dialect might refer to the title of a book, in another dialect it might refer the title of a person, e.g Doctor or Ms. The potential conflict over tag names is resolved through the use of *namespaces*. A namespace is much like an area code for a phone number. A good example of the use of a namespace is in an XML Schema. For example, in our dynamic lot size schema the root tag is

```
<xs:schema
targetNamespace="http://gsbkip.uchicago.edu/lotsize"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="http://gsbkip.uchicago.edu/lotsize"
elementFormDefault="qualified"
attributeFormDefault="unqualified">
```

There are two namespace declarations in this tag. The first,

```
xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

declares that any element with the prefix `xs` belongs in the namespace `http://www.w3.org/2001/XMLSchema`. For example, the tag

```
<xs:complexType name="product">
```

qualifies the element `complexType` as an element in the namespace defined by the `xs:` prefix. When an XML validating parser validates an XML document against a W3C XML Schema the parser verifies that each `element` tag in the instance file appears in the Schema file with an appropriate definition using an `xs:` prefix. In sub-Section 2.5 we introduce the XSLT XML dialect. An XSLT processor knows how to process the XSLT document by reading `xsl` elements with names qualified by an `xsl` prefix. In the root element there is a second namespace declaration,

```
xmlns="http://gsbkip.uchicago.edu/lotsize"
```

This namespace is *unqualified.* Since it appears in the root element all of the unqualified elements used in the document are assumed to be in this namespace.

2.4 XPath

XPath is used to locate data in an XML database. The function of XPath is similar to the `SELECT` command in SQL. However, the syntax of XPath is similar to the syntax used to locate files in a directory with a tree structure. We illustrate XPath on the following file.

```
<?xml version="1.0" encoding="UTF-8"?>
<data>
   <plants>
      <plant name="Uvalde"/>
      <plant name="Sanderson"/>
      <plant name="Marathon"/>
      <plant name="Alpine"/>
      <plant name="Terlingua"/>
      <plant name="Lajitas"/>
      <plant name="Lajitas"/>
   </plants>
   <openPlants>
      <plant name="Uvalde"/>
      <plant name="Sanderson"/>
      <plant name="Marathon"/>
   </openPlants>
</data>
```

The typical use of an XPath command is a *location path* to locate a set of nodes in a tree. This is called the *node-set.* For example, the location path

```
/data/openPlants/plant/@name
```

locates the node-set $\{\texttt{Uvalde}, \texttt{Sanderson}, \texttt{Marathon}\}$.

XPath is quite robust and can be used to perform set operations such as union, intersection, and set difference. For example, the node-set of closed plants is the set difference `plants` - `openPlants`. The XPath location path for this is

```
/data/plants/plant[not(@name=
    /data/openPlants/plant/@name)]/@name
```

This yields the node-set {`Alpine`, `Terlingua`, `Lajitas`, `Lajitas`}. Notice that the `Lajitas` node appears twice. If we want a node-set with unique city names then we use the following location path

```
/data/plants/plant[not(@name=/data/openPlants/plant/@name)
 and not(@name=preceding-sibling::plant/@name)]/@name
```

2.5 XSLT

Extensible Stylesheet Language Transformations (XSLT) is an XML based programming language for transforming XML files into other XML files, or HTML files, or plain text files. We use XSLT to convert XML files with raw instance data (e.g. costs, demands, etc.) into an XML file representing the instance of a linear program. This is described in greater detail in Section 5. In this section we provide a very brief introduction on how stylesheet transformations work. For a thorough treatment of XSLT programming see Kay [17].

A stylesheet consists of a set of *templates*. A template specifies what action to take when the XSLT processor encounters a given pattern in the input document. A template is somewhat similar to a function or method in a procedural language such as C++ or Java. Template order does not matter. There is a root template where processing starts much like the `main` method in C++ or Java. All of our XSLT style sheets have the form:

```
<xsl:template match="/">
    <xsl:apply-templates select="pattern 1 node-set"/>
    <xsl:apply-templates select="pattern 2 node-set"/>
</xsl:template>
<xsl:template match="pattern 1 node set">
    Do transformations for pattern 1 node-set
</xsl:template>
<xsl:template match="pattern 2 node set">
    Do transformations for pattern 2 node-set
</xsl:template>
```

Processing begins with the root template indicated by `match="/"`. Then the XSLT processor recursively processes each node in the node-set that matches pattern 1 by calling the template that matches the pattern 1 node-set. When this node-set is exhausted the processor repeats the process for pattern 2 node-set.

Consider a more specific example. In the previous section we used XPath to select the node-set consisting of the unique names of the closed plants. Below is the XSLT template that 1) locates the unique names of closed plants and then 2) puts the result in an XML file.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:output method="xml" version="1.0"/>
   <xsl:template match="/">
      <closedPlants>
         <xsl:apply-templates select="data/plants"/>
      </closedPlants>
   </xsl:template>
   <xsl:template match="data/plants">
      <xsl:for-each select="plant
         [not(@name=/data/openPlants/plant/@name)
         and not(@name=preceding-sibling::
         plant/@name)]/@name">
         <plant>
            <xsl:value-of select="."/>
         </plant>
      </xsl:for-each>
   </xsl:template>
</xsl:stylesheet>
```

When this document is processed by an XSLT processor, the processor first locates the tag

```
<xsl:template match="/">
```

and sets the *context node* of the processor to the root node. Next, the XSLT processor recursively processes each node in the node-set `data/plants` and calls the template beginning with the tag

```
<xsl:template match="data/plants">
```

Once inside this this template the processor encounters the loop

```
<xsl:for-each select="XPath statement">
```

and processes each node in the node-set located by the XPath statement, which in our case is

```
/data/plants/plant
   [not(@name=/data/openPlants/pl:plant/@name) and
   not(@name=preceding-sibling::plant/@name)]/@name
```

The selected node-set city names are then placed inside the `plant` element tag using the XSLT tag

```
<xsl:value-of select="." />
```

The resulting XML file is given below.

```
<?xml version="1.0" encoding="UTF-8"?>
<closedPlants>
   <plant>Alpine</plant>
   <plant>Terlingua</plant>
   <plant>Lajitas</plant>
</closedPlants>
```

3 The Basic Approaches

How can the XML technologies described in the previous section best be used to enhance the generation of mathematical programming models? The amount of XML involvement in the modeling process is categorized as follows.

Level 1: Use XML to represent instances of mathematical programs.

Level 2: Enhance current modeling languages with features such as XPath commands in order to better facilitate accessing data stored in XML format.

Level 3: Use XML technologies, in particular XPath and XSLT, to generate an XML instance of a mathematical program.

We consider each of these categories in turn in the next three subsections.

3.1 XML Instances

Level 1 requires representing the *instance* of a problem in XML, not the algebraic representation of the model. This approach is currently being pursued by Fourer, Lopes, and Martin [12] who are developing a W3C XML Schema to represent instances of mixed integer linear programs. We refer to this as the FML Schema. This approach requires no changes to current mathematical programming modeling languages. It does require drivers to interface with various solvers and modeling languages. The native format for representing a problem instance in each modeling language must be converted to the XML instance. Then the XML instance must be converted to the native format required by a solver. For example, an AMPL instance file *.ampl must be converted into a XML instance and the XML instance converted into a LINDO compatible format.

Having an industry standard for representing an instance of a mathematical program in XML is an important development. If there are M modeling languages and N solvers, then without a standard way to represent a problem instance, $M \cdot N$ drivers are required for every modeling language to be compatible with every solver. Different platforms, e.g. Unix, Linux, and Windows only exacerbates the problem. With an industry standard XML Schema only $M + N$ drivers are required. Fourer, Lopes, and Martin discuss these issues in greater detail.

3.2 Incorporate XML Technologies Into Algebraic Modeling Languages

XML is becoming a popular format for data storage. Database products such as Total XML [7] from Cincom and Tamino XML Server [26] from Software AG support data storage in native XML format. The amount of corporate data stored in native XML format will probably increase over time. Current algebraic modeling languages such a AMPL, LINGO, and MPL provide capabilities for interfacing with relational databases. This is usually done through ODBC drivers that are database specific. The Level 2 use of XML technologies is incorporating into these modeling languages the ability to access data stored in XML format in a manner analogous to the access of data stored in a relational database. With Level 2 we are not suggesting changing the basic syntax of the algebraic modeling language used to represent sets, loop, perform sums, etc.

For example, in LINGO, one might declare a set of time periods and a capacity for every time period. A more complete discussion on sets and indices is is given in Section 4. Denote the set of capacities by `CAP.` In LINGO, this set of capacities is populated from an ODBC database in the `DATA` section. This is illustrated below.

```
DATA:
CAP = @ODBC( ’capacitydata’, ’capacity’);
ENDDATA
```

However, if these data were in the XML file illustrated in Figure 4 we might instead incorporate an XPath command in LINGO like

```
DATA:
CAP = /linearProgram/periodCapacity/capacity
ENDDATA
```

A command is also needed to locate the XML file with the data. Software developers of algebraic modeling languages could also add features allowing the software to read and write an XML instance based on an accepted W3C XML Schema.

3.3 Using XPath and XSLT for Model Generation

The focus of this paper is Level 3. We show how to use XML technologies to generate mixed integer linear programming models. There are several approaches one could take to achieve this. One approach is to define the model within an XML input file. With this approach, an XML input file would contain both raw data and information about the algebraic structure of the model. For example, sets and indices would be defined within the XML input file. We choose not to take this approach. It would require the development of a new XML based modeling language syntax. Although feasible, a problem with this approach is achieving consensus on the syntax of such an XML dialect. Indeed, one reason Fourer, Lopes, and Martin are developing an XML Schema for instances of linear programs, rather than an XML based modeling language, is that the instance is the lowest common denominator and requires the least amount of agreement.

Our approach is to take as input the XML files that contain the problem instance data and then transform the input files into an output file that is an instance of a linear program. The most natural way to do this is to use something expressly designed to transform one XML file into another. We show how to use XSLT 1.0 (in conjunction with XPath 1.0) to generate instances of a mixed integer linear program. This approach was illustrated in Section 1 in Figure 1. In Figure 1 the XSLT template replaces the algebraic modeling language. The function of the template is to generate the industry standard XML instance. The template serves as a matrix generator. The details of XSLT transformation process and examples are given in the next two sections. With this approach a total of only N software drivers is necessary. For each of the N solvers a driver is required to translate the XML data instance into a format acceptable to the solver API.

4 Sets, Indices, and Data

In this section we assume that the input data for the model is in XML format in a single file. The single file assumption is not necessary. It is made only for ease of exposition. In sub-Section 5.2 we illustrate accessing input data simultaneously from multiple sources.

The first step in building an algebraic model using a modeling language is to identify the primitive sets. See Geoffrion [13] for a discussion of sets and indices in mathematical programming modeling. The sets often correspond to the indices on the decision variables. In the relational database world these are often attributes that correspond to keys in a relation. Algebraic modeling languages have commands to create sets. Sets may be either primitive or derived sets through such operations as cartesian product or set union. In the dynamic lot sizing example introduced in Section 2, primitive sets correspond to products and time periods. A derived set is the cartesian product of the product and time period sets. Here is an example of set declarations in LINGO.

```
SETS:
   product /1, 2/;
   period /1..4/;
   prodperiod(product, period);
ENDSETS
```

The analogous concept in the XML world is the XPath node-set. Node-sets corresponding to `product`, `period`, and `demand` listed above are:

```
/linearProgram/product
/linearProgram/periodCapacity/period
/linearProgram/product/period
```

We show in Section 5 we show how to generate these node-sets, on-the-fly, as needed. However, in XSLT, node-sets can also be stored as variables for later reference. For example, the node-set corresponding to the products is stored using XSLT in the variable `product` as follows:

```
<xsl:variable
   name="product"
   select="/linearProgram/product"
/>
```

In an algebraic modeling language, once the sets are identified, parameters and variables are associated with the sets and referenced by indices. For example, considering only parameters, in the lot sizing example we have

```
SETS:
   product /1, 2/: holdCost, prodCost, fixedCost;
   period /1..4/: capacity;
   prodperiod(product, period): demand;
ENDSETS
```

For example, `holdCost(1)` is the holding cost of the first product. The holding cost node-set is referenced in XPath by

```
/linearProgram/product/@holdCost
```

The `position()` function in XPath is then used as an index. For example, the holding cost of the first product is

```
/linearProgram/product[position()=1]/@holdCost
```

or, in terms of the variable `product`

```
$product[position()=1]/@holdCost
```

Similarly, the demand for product 2 in periods 3 and 4 is given by

```
/linearProgram/product[position()=2]/
   period[position()>2]/demand
```

or, in terms of the variable `product`

```
$product[position()=2]/period[position()>2]/demand
```

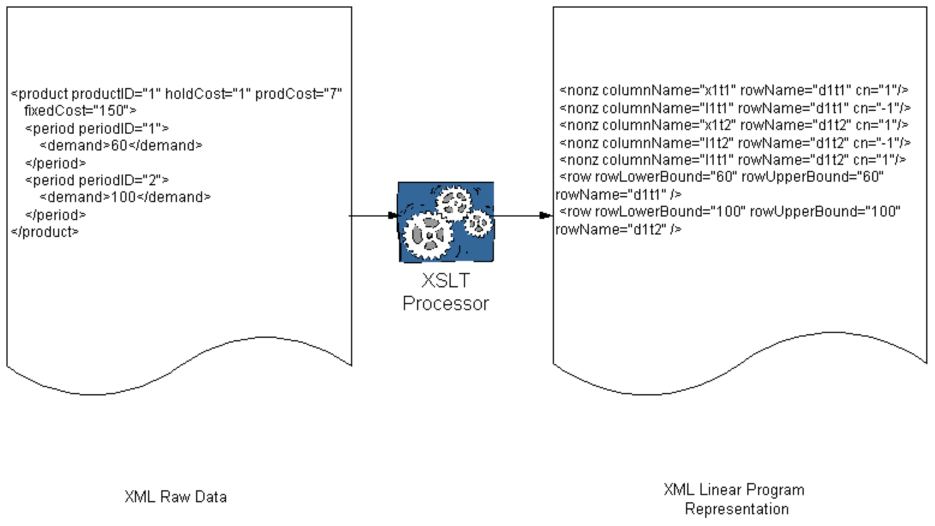
One advantage of nesting time period nodes within product nodes (illustrated in Figure 3) over the more traditional tabular approach (illustrated in Figure 2) is that we can use the position function to easily index both the product and time period. Applying the position function to the data structure in Figure 2 only allows us to index each row of the table.

An important aspect of this approach is that we are using the input XML for data only, the input files do not contain any information about constraints or variables. The input XML files need only contain all of the model parameters (or sufficient information to generate them). In Section 2.4 we showed how to locate the input data using XPath. In the next section we show how to use XSLT to generate the actual linear programming instance.

5 Model Generation with XSLT and XPath

XSLT is designed to transform an XML file into another XML file (or HTML or text). We use XSLT much like a matrix generator. If we know the format of the XML for storing the instance of a linear program, we write a template to transform the raw data file into the linear programming instance data file. For example, we could use XSLT to transform the input file into one which validates against the FML Schema of Fourer, Lopes, and Martin. However, the FML Schema is designed to minimize file size and for easy integration with solver API's. Rather than write an instance file in this format, we write an *intermediate* instance file that has a syntax that makes the XSLT template very easy to construct. Then the intermediate XML instance is transformed into a final XML instance file. More details on the transformation process are given in Section 8.

Figure 5: Transforming raw data into a linear program instance



```
<product productID="1" holdCost="1" prodCost="7"
  fixedCost="150">
  <period periodID="1">
    <demand>60</demand>
  </period>
  <period periodID="2">
    <demand>100</demand>
  </period>
</product>
```

```
<nonz columnName="x1t1" rowName="d1t1" cn="1"/>
<nonz columnName="I1t1" rowName="d1t1" cn="-1"/>
<nonz columnName="x1t2" rowName="d1t2" cn="1"/>
<nonz columnName="I1t2" rowName="d1t2" cn="-1"/>
<nonz columnName="I1t1" rowName="d1t2" cn="1"/>
<row rowLowerBound="60" rowUpperBound="60"
rowName="d1t1" />
<row rowLowerBound="100" rowUpperBound="100"
rowName="d1t2" />
```

The intermediate XML file used to store the linear program has two types of elements. For each row in the linear programming formulation there is a single element of the form

```
<row rowName="*" rowUpperBound="*" rowLowerBound="*" />
```

where `rowUpperBound` is a double precision attribute that holds the value of the right hand side in the case of a less-than-or-equal-to constraint, `rowLowerBound` is a double precision attribute that takes on the value of the right hand side in the case of a greater-than-or-equal-to constraint (both are present for an equality or range constraint), and `rowName` is a string attribute that must uniquely identify the row. Next, for each nonzero in the constraint matrix and objective function there is an element of the form

```
<nonz columnName="*" rowName="*"
   columnType="continuous/binary/integer"
   colLowerBound="*" colUpperBound="*" cn="*"/>
```

where `columnName` is a string attribute that must uniquely identify the variable, `cn` is a double precision attribute that is the coefficient of the variable `columnName` that appears in the row identified by the string attribute `rowName`, `columnType` is an optional string attribute used to declare either integer or continuous variables (we assume continuous variable when attribute is absent), `colLowerBound` is a double precision attribute that is the variable lower bound (we assume 0 when attribute is absent), and `colUpperBound` is a double precision attribute that is the variable upper bound (we assume $+\infty$ when attribute is absent). We assume the objective function has a `rowName` of `obj`. The details of generating this file are explained shortly.

The `row` and `nonz` elements may appear in an order in the intermediate XML document. Indeed, this allows for great flexibility in the model generation. There are different philosophies on how to generate a linear program. With our approach one can follow the algebraic modeling language approach which usually is to specify the constraints in a row wise fashion. Or, one can generate the model via column and row strips. See, for example, Welch [30].

The transformation of an XML file with raw data into the intermediate XML file with the necessary `row` and `nonz` elements representing the linear program instance is illustrated in Figure 5. In this figure we illustrate the nonzero elements in the demand constraints for the first product in the first two time periods. The row elements for first product in the first two time periods are also shown. The transformation is done using a style sheet that consists of a set of templates. In the next two subsections we illustrate the process with a lot sizing and a logistics example. Complete templates are provided in the Appendix document.

5.1 Lot Size Example

The first example is a *multiproduct lot size problem*. The raw data for this problem were used in the XML example in Section 2.1. The algebraic statement of the problem is given below.

Parameters:

$d_{it}-$ demand for product i in period t

$f_{it}-$ fixed cost associated with production of product i in period t

$h_{it}-$ marginal cost of holding one unit of product i in inventory at the end of period t

$c_{it}-$ marginal production cost of one unit of product i in period t

g_t- production capacity available in period t

$M_{it}-$ an upper bound on the production of product i in time period $t,$ often called “big M” in the integer programming literature

Variables:

$x_{it}-$ units of product i produced in period t

$I_{it}-$ units of product i held in inventory at the end of period t

$y_{it}-$ a binary variable which is fixed to 1 if there is nonzero production of product i in period t, otherwise it is fixed to 0

The mixed integer linear program for the multiproduct dynamic lot size problem is

$$\min \sum_i \sum_t (c_{it}x_{it} + h_{it}I_{it} + f_{it}y_{it}) \tag{1}$$

$$\begin{aligned} \texttt{s.t.} \qquad \sum_i x_{it} &\leq g_t, \quad \forall\, t && (2) \\ I_{i,t-1} + x_{it} - I_{it} &= d_{it}, \quad \forall\, i,t && (3) \\ x_{it} - M_{it}y_{it} &\leq 0, \quad \forall\, i,t && (4) \\ x_{it}, I_{it} &\geq 0, \quad \forall\, i,t && (5) \\ y_{it} &\in \{0,1\}, \quad \forall\, i,t && (6) \end{aligned}$$

The objective function (1) is the minimization of the sum of the production, inventory holding and setup costs. Constraint (2) is a capacity constraint. Constraint (3) is a *conservation of flow* or *sources and uses* requirement. Constraint (4) is a *fixed charge* or *setup forcing constraint.* The formulation (1)-(6) is a very simplified version of more realistic problems. We show how to take raw cost and demand data in the XML file illustrated in Figure 3 transform these data into the instance of a linear program given by (1)-(6). The root template for this model is

```
<xsl:template match="/">
   <xsl:apply-templates select=
      "ls:linearProgram/ls:periodCapacity/ls:capacity"/>
   <xsl:apply-templates select=
      "ls:linearProgram/ls:product"/>
   <!-- Code to generate LP statistics  -->
</xsl:template>
```

(The `ls:` prefix that appears in all of the XPath commands refers to the namespace of `ls` used to qualify the elements that define the lot size parameters.) In this example we generate the model by rows. The first template is used to generate the capacity constraints (2). The second template is used to generate the demand constraints (3) and the fixed charge constraints (4). The entire style sheet appears in the Appendix document. We only illustrate aspects of it. Consider the template used to generate the demand and fixed charge constraints. A match is made on the products, `match="ls:product"`, so the demand and fixed charge constraints are generated by product. Next, for each product, we generate the demand and fixed charge constraints for each time period. This is accomplished using the `<xsl:for-each select="ls:period">` element which is analogous to the `for` element in C++ or Java.

```
<xsl:template match="ls:product">
   <xsl:variable name="productIndex"
      select="position()"/>
   <xsl:variable name="prodCost"
      select="@prodCost"/>
   <xsl:variable name="holdCost"
      select="@holdCost"/>
   <xsl:variable name="fixedCost"
      select="@fixedCost"/>
   <xsl:for-each select="ls:period">
      <xsl:variable name="timeIndex"
         select="position()"/>
      <!--  more code here   -->
   </xsl:for-each>
```

We create an index for the products, `productIndex`, and an index for the time periods `timeIndex` using `<xsl:variable>`. These indices are used as subscripts on the variables. The index corresponds to a position in the node-set. This is given by the XPath function `position().` The output file is an XML file. This means that the transformation process must create the elements that are required in the output file. This is accomplished using the XSLT element `<xsl:element>`. The use of the element tag is illustrated below. In this segment of code illustrated below, we generated the x_{it} and I_{it} variables in the $I_{i,t-1} + x_{it} - I_{it} = d_{it}$ constraint. The `concat` function is used to concatenate the *x* or *I* with the product and time index. The $ in front of the `productIndex` and `timeIndex` variables is used when making a reference to a variable.

```
<xsl:element name="nonz" >
   <xsl:attribute name="columnName">
      <xsl:value-of select="concat('x',$productIndex,
         't',$timeIndex)"/>
   </xsl:attribute>
   <xsl:attribute name="rowName">
      <xsl:value-of select="concat('d',$productIndex,
         't',$timeIndex)"/>
   </xsl:attribute>
   <xsl:attribute name="cn"> 1</xsl:attribute>
</xsl:element>
```

```
<xsl:element name="nonz" >
   <xsl:attribute name="columnName">
      <xsl:value-of select="concat('I',
         $productIndex,'t',$timeIndex)"/>
   </xsl:attribute>
   <xsl:attribute name="rowName">
      <xsl:value-of select="concat('d',$productIndex,
         't',$timeIndex)"/>
   </xsl:attribute>
   <xsl:attribute name="cn"> -1</xsl:attribute>
</xsl:element>
```

The segment of code illustrated above for the first product in the second time period generates elements that look like

```
<nonz columnName="x1t2" rowName="d1t2" cn="1">
</nonz>
<nonz columnName="I1t2" rowName="d1t2" cn="-1">
</nonz>
```

This balance constraint still needs the inventory variable corresponding to the previous period. This is easy to do. The node-sets are ordered which allowed creation of indices. We can refer back to `$timeIndex - 1` to generate the variable $I_{i,t-1}$ that appears in every constraint for a time period greater than one.

```
<xsl:if test="$timeIndex > 1">
   <xsl:element name="nonz" >
      <xsl:attribute name="columnName">
         <xsl:value-of select="concat('I',$productIndex,
            't',string($timeIndex-1))"/>
      </xsl:attribute>
      <xsl:attribute name="rowName">
         <xsl:value-of select=
            "concat('d',$productIndex,'t',$timeIndex)"/>
      </xsl:attribute>
        <xsl:attribute name="cn"> 1</xsl:attribute>
   </xsl:element>
</xsl:if>
```

The `if` statement used here behaves as in any procedural langauge. The objective function is treated like any other row and objective function coefficients are generated exactly like the coefficients in the demand balancing row illustrated above. In the case of the objective function, the `rowName` is always `obj.` The right hand side information of each constraint in the linear program is contained in an `row` element. For example, the right hand side element of the demand balancing constraints is generated by

```
<xsl:element name="row">
   <xsl:attribute name="rowName">
      <xsl:value-of select=
        "concat('d',$productIndex,'t',$timeIndex)"/>
    </xsl:attribute>
    <xsl:attribute name="rowLowerBound">
        <xsl:value-of select="ls:demand"/>
    </xsl:attribute>
    <xsl:attribute name="rowUpperBound">
        <xsl:value-of select="ls:demand"/>
    </xsl:attribute>
</xsl:element>
```

and appears in the output file as

```
<row rowName="d1t2" rowLowerBound="100"
   rowUpperBound="100"/>
```

XSLT can also be used to generate constraints where there are conditions on the coefficients in the constraints. For example, in the fixed charge constraints (4) the *big M* parameter is often set to $M_{it} = \min\{g_t, \sum_{k=t}^{T} d_{ik}\}$. Using the XPath `sum()` function and XSLT elements `<xsl:choose>`, `<xsl:when>`, and `<xsl:otherwise>` gives

```
<xsl:variable name="remDemand"
   select="sum(/ls:linearProgram/ls:product[position()=
   $productIndex]/ls:period[position()>=
   $timeIndex]/ls:demand)"/>
<xsl:variable name="bigM">
   <xsl:choose>
      <xsl:when test="$capacity[position()=$timeIndex] >
         $remDemand">
         <xsl:value-of select="$remDemand"/>
      </xsl:when>
      <xsl:otherwise>
         <xsl:value-of select="$capacity[position()=
            $timeIndex]"/>
      </xsl:otherwise>
   </xsl:choose>
</xsl:variable>
```

This is not the most efficient way to calculate the `bigM` coefficient. This coefficient must be computed NT time where N is the number of products and T the number of time periods. The total work is then $O(NT^2.)$ The calculation of `bigM` is easily reduced to $O(NT)$ by looping backwards over time and keeping a running total of the remaining demand. Unfortunately, this requires updating the value of a variable within a loop which cannot be done in XSLT. This problem is easily overcome by using recursion. The recursive version of this template appears in the Appendix and can be downloaded from the Web site as `lotsizerecurse.xslt.`

Although we generate the lot sizing model by row, we generate the objective function coefficients for the inventory and production variables when we generate the demand rows and we generate the objective function coefficients for the fixed charge variables when we generate the fixed charge constraints. Thus, in a sense, we are mixing the by row and by column model generation methods.

5.2 Logistics

We illustrate our approach with a second example. It a logistics example. The instance data we use is taken directly from Atamtürk et al. [3]. The model is based on the one in Mairs et al. [21]. The mixed integer model formulation for this single source multiechelon distribution system is

Parameters:

d_{il}– demand for product i at demand center l

c_{ij}– marginal production cost of one unit of product i at plant j

f_{jk}– marginal cost of shipping one unit of a product from plant j to warehouse k

h_{kl}– marginal cost of shipping one unit of a product from warehouse k to demand center l

g_{ij}– production capacity available for product i at plant j

Variables:

x_{ijk}– units of product i produced at plant j and shipped to warehouse k

z_{ikl}– units of product i shipped from warehouse k to demand center l

y_{kl}– a binary variable which is fixed to 1 if center l is supplied from warehouse k otherwise it is fixed to 0

$$\min \sum_i \sum_j \sum_k (c_{ij} + f_{jk}) x_{ijk} + \sum_i \sum_k \sum_l h_{kl} z_{ikl} \tag{7}$$

$$\texttt{s.t.} \quad \sum_k y_{kl} = 1, \quad \forall\, l \tag{8}$$

$$d_{il} y_{kl} = z_{ikl}, \quad \forall\, i, k, l \tag{9}$$

$$\sum_l z_{ikl} = \sum_j x_{ijk}, \quad \forall\, i, k \tag{10}$$

$$\sum_k x_{ijk} \leq g_{ij}, \quad \forall\, i, j \tag{11}$$

$$x_{ijk},\, z_{ikl} \geq 0, \quad \forall\, i, j, k, l \tag{12}$$

$$y_{kl} \in \{0, 1\}, \quad \forall\, k, l \tag{13}$$

In this formulation the objective is to minimize the cost of production plus shipping from plants to warehouses and warehouses to demand center. Constraint (8) is a single sourcing constraint. Each demand center is supplied by only one warehouse. Constraint (9) is a demand constraint and requires that the demand for each product at each demand center be satisfied by supply at a warehouse. Constraint (10) is a balance constraint. The quantity of each product sent from a warehouse to all of the demand centers must equal the quantity of the product sent to the warehouse from the plants. Equations (12) and (13) are nonnegativity and integrality constraints, respectively.

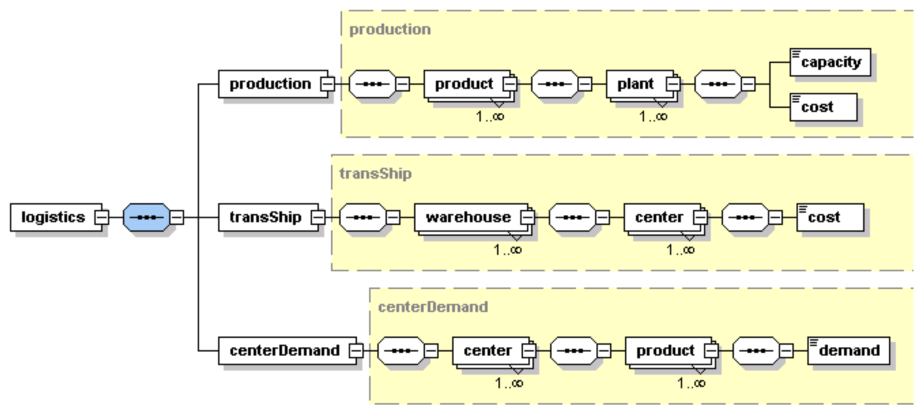
In this logistics example we illustrate two things. First, that the XML data may reside in physically distinct files. Second, that template approach is very flexible and allows for generating a problem either by row or by column. This example we generate the model using column and row strips as described in Welch [30].

We use two XML input files in this example. The first file contains the information on products, plants, warehouses, and demand centers. The second file contains the information on shipping costs from plants to warehouses. We again use a hierarchial tree storage as opposed to a tabular storage. The tree structure of the logistics schema is illustrated in Figure 6. For example, the plant production data is stored as

```
<production>
   <product prodID="chips">
      <plant plantID="topeka">
         <capacity>200</capacity>
         <cost>230</cost>
      </plant>
      <plant plantID="newyork">
         <capacity>600</capacity>
         <cost>255</cost>
      </plant>
   </product>
   <product prodID="nachos">
      <plant plantID="topeka">
         <capacity>800</capacity>
         <cost>280</cost>
      </plant>
   </product>
</production>
```

The template for the logistics example also appears in the Appendix document. We highlight a few important aspects. The root template is

Figure 6: Tree structure for logistics schema



```
<xsl:template match="/">
   <xsl:apply-templates select=
      "log:logistics/log:production/log:product"
      mode="xijk"/>
   <xsl:apply-templates select=
     "log:logistics/log:centerDemand/log:center"
      mode="ykl"/>
   <xsl:apply-templates select=
      "log:logistics/log:production/log:product"
      mode="zikl"/>
</xsl:template>
```

Three distinct templates are used to generate the model: a template to generate the production variables x_{ijk}, a template to generate the transshipment variables z_{ikl}, and a template to generate the binary variables y_{kl}. In this example note the use of the `mode` attribute which was not used in the lot sizing example. The mode attribute allows for matching more than one template to the same pattern. In this example, we generate both the production and transshipment variables based on the pattern

```
log:logistics/log:production/log:product
```

but use different templates depending upon the variable set. This is accomplished by using the `mode` attribute. For example, the template for the production variables has the form

```
<xsl:template match="log:production/log:product"
   mode="xijk">
   <!-- more code here -->
</xsl:template>
```

Recall that in the dynamic lot sizing example we had variables I_{it}, x_{it}, and y_{it}. The i, t subscripts were generated from a node-set with the property that the subscript t was generated using nodes that were direct descendants of the nodes used to generate the i subscripts. However, this is not necessary. With XPath and XSLT we can generate indices from node-sets that are not even in the same XML file. Consider the production variables x_{ijk}. Generate the indices as follows:

```
<xsl:template match="log:production/log:product"
   mode="xijk">
   <xsl:variable name="productIndex" select=
      "position()"/>
   <xsl:for-each select="log:plant">
      <xsl:variable name="plantIndex" select=
         "position()"/>
      <xsl:variable name="shipCostData" select=
         "document('shipcost.xml')"/>
      <xsl:for-each select=
         "$shipCostData/logistics/shipCost/
         plant[@plantID=$plantID]/warehouse">
         <xsl:variable name="whseIndex" select=
            "position()"/>
      </xsl:for-each>
   <xsl:for-each>
</xsl:template>
```

The i subscript (`productIndex` in the XSLT code) is generated by the `position()` of the context node in the product node-set. Next, the j subscript (`plantIndex` in the XSLT code) is generated by the `position()` of the `plant` children nodes of the `product` parent nodes. However, the node-set to generate the k warehouse subscript is in a separate XML file `shipcost.xml` and is selected using the XSLT `document()` function. The root node of this document is stored in the variable `shipCostData`. This variable is then used in XPath statement

```
<xsl:for-each select="$shipCostData/logistics/shipCost/
    plant[@plantID=$plantID]/warehouse">
```

to select the `warehouse` node-set. In this case the file `shipcost.xml` resides in the same directory as the other XML file, `logistics.xml`. However, these files could reside on different servers.

In this example, the XPath statement is analogous to the SQL commands `SELECT`, `FROM`, and `WHERE` that are used to join different relations based on a key-foreign key relationship and then project out the attributes of interest. In our case, the attributes `prodID` and `plantID` constitute a key for the node-set with root node `production` in the XML file `logistics.xml` In the XML file `shipcost.xml` the attributes `plantID` and `whseID` constitute a key for the node-set with root node `shipCost`. The two node-sets are effectively "joined" on the foreign key `plantID` and the desired node-set `warehouse` is projected out.

6 Software

Numerous software packages are available that implement XPath and XSLT. There are two major camps: Microsoft .NET and Java. The most recent release of the Microsoft development tool, Visual Studio .NET [8], contains numerous classes such as `XPathDocument` and `XslTransform` for manipulating and transforming XML data. These classes are available to all of the .NET languages. All of the transformation examples in this paper could easily be performed with an application developed using Visual Studio .NET. Indeed, a major advantage of using .NET software is that Microsoft has done such an excellent job of integrating XML into Visual Studio .NET. The downside of .NET is that .NET software written to process XSLT templates will run only on the Windows platform (although Ximian has announced the launch of the Mono project [31] to create an open source implementation of the .Net development framework). However, the actual XSLT templates are platform independent. Therefore, a template that meets the XSLT 1.0 standard will run on both Windows and non-Windows platforms. There is no problem with sharing model templates among users of different platforms.

A number of Java open source XSLT tools are also available. There is Saxon written by Michael Kay [18] and Xalan (a C++ version is also available) by the Apache organization [2]. Both Saxon and Xalan can be used from the command line or called from an Servlet, or stand alone Java program. Both Xalan and Saxon implement the Java API for XML Processing (JAXP). This makes it very convenient to write portable software that can call either Saxon or Xalan for the transforming XML. The most recent version of Saxon offers an experimental implementation of XPath 2.0 and XSLT 2.0. There is also XML Spy from Altova [1] which is a proprietary XML development environment. Spy includes an XML parser and XSLT tool along with some very nice graphical tools for constructing XML Schemas. The examples used in this paper have been successfully tested on Saxon, Xalan, XML Spy, and the Microsoft processor (MSXML 4.0).

7 Getting the Data

XSLT 1.0 is designed to work with input data in an XML format (XSLT 2.0 has features for accessing text files). In this section we show that there are numerous tools for transforming non-XML data into XML data. Most of the data used in a linear program will reside in a

- spreadsheet
- desktop database (e.g. Microsoft Access)
- ASCII flat file
- enterprise database (e.g. DB2, Oracle, SQL Server)
- XML file

We discuss converting each source into XML. There are several options with a spreadsheet or desktop database. If the spreadsheet or database are part of Microsoft Office 2002 (or later) it is possible to directly export each table in the database, or range in the spreadsheet, as an XML file. If the desktop spreadsheet or database are ODBC or OLE-DB compliant, then one can write a program in a procedural language such as C++ or Java to access this data using ODBC or OLE-DB, read it into memory, and then using DOM (document object module) create an XML representation of data. There is some overhead in creating the DOM and storing it in main memory. An alternative approach is to write a custom SAX parser and feed the information directly into a JAXP compliant XSLT processor. DOM and SAX are alternative APIs for processing XML.

If the flat file is an ASCII flat file several options exist. First, one could import the flat file into a desktop database such as Microsoft Access and then save it as an XML file. A second option is to write a C++ or Java program to parse the file and then use DOM or SAX create an XML representation of the data.

Much of the data for large models is stored in enterprise corporate databases. Fortunately, the major database vendors are adding features to their products that allow the user to submit an SQL query to the database and get the result back in XML format. There are JDBC drivers for the most widely used databases. Thus, one could write a Java program and use JDBC and SQL to query the database, get the result as XML, and then transform the XML using a JAXP transformation engine such XALAN or Saxon. This process is also easily carried out using Visual Studio .NET. There are many classes available to any of the .NET languages for reading data in XML format from a relational database and then transforming it with the `XslTransform` class.

Ideally, the input data is initially in XML format. However, some XML structures are more amenable to transformation into a mathematical model than others. For example, the hierarchal structure illustrated in Figure 3 is more amenable to transformation into a linear program with doubly subscripted variables than the table structure illustrated in Figure 2. Of course XSLT is designed to transform one XML file into another and it is not difficult to transform the XML file illustrated in Figure 2 into the one illustrated in Figure 3. Use XPath to select each unique `productID` and then select each `periodID` for the given `productID`. This is accomplished with the following XPath statement to get the set of unique productIDs.

```
/linearProgram/record[not(productID=
preceding-sibling::record/productID)]/productID
```

Then for each `productID` one can select the `periodID`. This type of grouping is now even easier in XSLT 2.0 that has an `xsl:for-each-group` command.

There are new products being released expressly for the purpose of accessing data stored in different formats and viewing the data as XML. Two such products include BEA's *Liquid Data* [4] and IBM's *XPeranto* [16]. The trend is obvious: make it easy to gather data in various sources and convert it into XML. This makes the methodology we are proposing even more viable over time.

8 Computational Results

XSLT is a functional programming language and is not as efficient as a procedural language such as C++ for generating mathematical models. Nevertheless, we tested the XSLT template approach on some small to medium size capacitated lot sizing problems with structure (1)-(6). The lot sizing problems are based on the Dixon and Silver [9] data reported in Eppen and Martin [10]. The problem data and CPU times in seconds appear in Table 4. The CPU times include file input and output time. This is nontrivial since the output XML files are over five megabytes in size for the largest problems. We report times for three XSLT transformers: Saxon 6.5.2 and Xalan 2.4.1 which are written in Java and a Microsoft .NET C# implementation that uses the .NET `XslTransform` class for transforming the stylesheet. The problems were solved on a Dell Latitude C600 notebook with a 1.1 GHz Pentium III processor and 512 MB of main memory.

Table 4: Intermediate Instance Generation Time

				Saxon 6.5.2	Xalan 2.4.1	C# .NET
Problem	Rows	Columns	Nonz	CPU Sec	CPU Sec	CPU Sec
ds100m	2010	3000	9000	1.67	2.22	.992
ds200m	4010	6000	18000	2.59	3.26	1.98
ds400m	8010	12000	36000	4.05	5.07	4.06
ds800m	16010	24000	72000	7.04	8.91	8.63

The CPU times in Table 4 use the lot sizing template described in Section 5.1 (where recursion is used to calculate the coefficients in the fixed charge constraints). The model generation template is designed to transform the raw XML data into an intermediate XML output file which represents the instance of the linear program. The structure of this intermediate XML output file is described in Section 5. The intermediate output XML file exists only to make the model generation template easy to construct. Recall that we can generate the model by either row or column and mix row and column element tags in the output file. However, storing the instance of a linear program with no structure on the rows and columns is not ideal for archival purpose, or for use with linear programming solver APIs. Therefore, we take the intermediate XML file representing the linear programming instance and transform it into another XML file that validates against the FML Schema.

There are two alternatives to transform the intermediate output file into the final instance format. The first alternative is to again use XSLT to transform the intermediate instance file into the final instance that validates against the FML Schema. A second alternative is to write a program in Java or C++ that takes the intermediate instance and transforms it into the final instance. Both of these alternatives are model independent, that is, the same XSLT/Java/C++ program could be used for lot sizing, logistics, scheduling etc.

We chose the second alternative and wrote a Java program to read the intermediate linear programming instance and transform it into an instance that validates against the FML Schema. Our implementation uses the Saxon transformer and the SAX (Simple

API for XML) and TrAX (Transformation API for XML) APIs. Rather than transform the raw data into an XML file, the raw data is transformed into a `SAXResult` which is processed. The `SAXResult` is processed by reading the elements and attributes of the intermediate XML instance. The elements and attributes of the `SAXResult` are then reorganized by the Java program into an XML file that validates against the FML Schema. For more details on the SAX and TrAX APIs we refer the reader to Harold [15] or Kay [17]. The total processing time to get the FML instance is given Table 5. A dedicated modeling language is more efficient. For example, LINGO takes approximately 1.7 CPU seconds to generate the linear programming instance for a `ds800m` problem. (LINGO cannot read files in XML format so we converted our XML data files into a flat file for LINGO to read. Also, LINGO cannot write XML files and we are reporting the time to generate an instance in MPS format.) Given the early stage of our technology we feel that the times reported in Table 5 show that the XSLT approach is worth pursuing.

Table 5: Final Instance Generation Time

	Saxon 6.5.2 Total CPU Sec
ds100m.xml	2.13
ds200m.xml	3.50
ds400m.xml	5.33
ds800m.xml	9.65

9 Conclusion and Future Trends

All of the methodology discussed so far is based upon XPath 1.0 and XSLT 1.0. These two standards are implemented in Microsoft .NET, Saxon, Xalan, XML Spy and others. However, Working Drafts of XPath 2.0 and XSLT 2.0 are available from the W3C as of November 15, 2002. The 2.0 versions of these standards have numerous features that greatly increase their utility for mathematical modeling. Saxon 7.0 and above has a partial implementation of the 2.0 versions of XPath and XSLT. A key addition to XSLT 2.0 is user defined functions. For example, in the lotsizing model the nonzero term for the production variable x_{it} in a demand constraint is generated by

```
<xsl:element name="nonz" >
   <xsl:attribute name="columnName">
      <xsl:value-of select=
         "concat('x',$productIndex,'t',$timeIndex)"/>
   </xsl:attribute>
   <xsl:attribute name="rowName">
      <xsl:value-of select="concat('d',$productIndex,
         't',$timeIndex)"/>
   </xsl:attribute>
   <xsl:attribute name="cn">1</xsl:attribute>
</xsl:element>
```

Using XSLT 2.0, we have implemented the following function `xlp:nonz`

```
<xsl:function name="xlp:nonz">
   <xsl:param name="columnName"/>
   <xsl:param name="rowName"/>
   <xsl:param name="cn"/>
   <xsl:variable name="output">
      <xsl:element name="nonz" >
         <xsl:attribute name="columnName">
            <xsl:value-of select="$columnName"/>
         </xsl:attribute>
         <xsl:attribute name="rowName">
            <xsl:value-of select="$rowName"/>
         </xsl:attribute>
         <xsl:attribute name="cn">
            <xsl:value-of select="$cn"/>
         </xsl:attribute>
      </xsl:element>
   </xsl:variable>
   <xsl:result select="$output"/>
</xsl:function>
```

This function has three inputs (parameters): a column name, a row name, and the value of the nonzero. This is function is generic and can be applied to *any* template. This function is called by the statement

```
<xsl:copy-of select="xlp:nonz(concat('x',$productIndex,
   't',$timeIndex), concat('d',$productIndex,'t',
   $timeIndex), 1)"/>
```

An even more useful function is the `xlp:consum` function. The code for this is in the Appendix document. It behaves much like the `sum` function in a modeling language such as AMPL, LINGO, or MPL. For example, the XSLT code to express the capacity constraints $\sum_i x_{it} \leq g_t$ for all t, is

```
<xsl:template match="ls:capacity">
   <xsl:copy-of select="xlp:consum(’x’,
      /ls:linearProgram/ls:product, position(),
      true(), concat(’cap’,’t’,position()), 1)"/>
   <xsl:copy-of select="xlp:row(concat(’cap’,
      ’t’,position()), ’&lt;’, .)"/>
</xsl:template>
```

Note also the use of the `xlp:row` function. It behaves much like the `xlp:nonz` function and generates the necessary tags for the capacity rows.

Developing a set of template independent functions greatly enhances the utility of XSLT 2.0 for generating linear programming models. The functions are easily shared by users over the Web. They are included in a template much like the `#include` statement in C++, or `import` statement in Java and can be linked to a server by a URL. For example,

```
<xsl:include href="http://gsbkip.uchicago.edu/template
   /functions/function.xslt" />
```

This paper has focused on linear programming. However, the XSLT template approach applies to nonlinear programming. Unfortunately, there is not an agreed upon standard for representing a nonlinear optimization problem. Halldórsson, Thorsteinsson, and Kristjánsson [14] propose a nonlinear extension to the MPS format for representing a linear program. In the API manual from Lindo Systems, Inc. [19] a similar representation scheme is proposed. In both cases a postfix representation of the operators and operands is used. The postfix instructions are put into a stack for execution. In both proposals a set of tags with corresponding integer codes are used to represent the operators such as addition and exponentiation. It is certainly possible with XSLT to transform XML based text into plain text which is not marked up. Therefore, one could use XSLT to transform the raw data XML file into the text format proposed by Halldórsson, Thorsteinsson, Kristjánsson” or Lindo Systems, Inc.

Another approach is to use Content MathML as proposed by the W3C. This is a proposed XML dialect for representing mathematical expressions. It is quite verbose. For example the Content Markup representation of x^2 is

```
<math>
   <apply>
      <power/>
         <ci>x</ci>
         <cn>2</cn>
   </apply>
</math>
```

The MathML representation is similar in approach to what Halldórsson, Thorsteinsson, Kristjánsson” or Lindo Systems, Inc. propose except prefix notation is used instead of postfix and XML tags are used instead of integer codes for operands.

It is certainly possible to take an XML data file and use it to create a nonlinear mathematical program with the objective function and constraints written in Content

MathML. One possibility is simply to allow the `nonz` tag to hold nonlinear expressions. For example,

```
<nonz rowname="*" />
MathML goes here to describe a nonlinear term
that appears in this row.
</nonz>
```

This is a topic for future research.

Finally, we mention an important, related technology called XQuery. As of this writing, it is a Working Draft of the W3C, but not a Recommendation. XQuery is to XML what SQL is to relational databases. Like XSLT, it uses XPath to locate data in the XML tree. The goal of XQuery is to provide a concise query langauge. There is no need to declare templates in order to query data. However, XQuery is not designed to *transform* the entire structure of an XML document. It is designed to quickly and efficiently extract chunks of data – much like SQL. Unlike XSLT, XQuery is not an XML dialect. One reason we chose XSLT for our research is that it is farther along than XQuery and XSLT software is more widely available than XQuery software. Investigating the use of XQuery to build mathematical programming models is also a topic of future research.

References

[1] Altova. XML SPY 5, 2002. `http://www.altova.com/products_ide.html`.

[2] Apache Software Foundation. The Apache XML project, 2002. `http://xml.apache.org/`.

[3] A. Atamtürk, E.L. Johnson, J.T. Linderoth, and M.W.P. Savelsbergh. A relational modeling system for linear and integer programming. *Operations Research*, 4:263–283, 2000.

[4] BEA. Introducing BEA Liquid Data for WebLogic, 2002. `http://www.bea.com/products/weblogic/liquiddata/index.shtml`.

[5] A. Brooke, D. Kendrick, and A Meeraus. *GAMS, A User's Guide*. Scientific Press, Redwood City, CA, 1988.

[6] J. Choobineh. SQLMP: a data sublanguage for representation and formulation of linear mathematical models. *ORSA Journal of Computing*, 3:358–375, 1991.

[7] Cincom. Total XML, 2002. `http://www.cincom.com/en/products/about.asp?ProductID=12`.

[8] Microsoft Corporation. Visual Studio .NET, 2002. `http://msdn.microsoft.com/library/default.asp?url=/vs/techinfo/Default.%asp`.

[9] P.S. Dixon and E.A. Silver. A heuristic solution procedure for the multi-item single-level, limited capacity, lot sizing problem. *Journal of Operations Management*, 2:23–39, 1981.

[10] G. D. Eppen and R. K. Martin. Solving multi-item capacitated lot-sizing problems using variable redefinition. *Operations Research*, 35:832–848, 1992.

[11] R. Fourer, D.M. Gay, and B.W. Kernighan. *AMPL A Modeling Language for Mathematical Programming*. Scientific Press, San Francisco, CA, 1993.

[12] R. Fourer, L. Lopes, and K. Martin. A W3C XML schema for linear programming (work in progress). Technical report, Northwestern University and The University of Chicago, 2002.

[13] A.M. Geoffrion. Indexing in modeling languages for mathematical programming. *Management Science*, 38:325–344, 1992.

[14] B. V. Halldórsson, E. S. Thorsteinsson, and B. Kristjánsson. A modeling interface to non-linear programming solvers an instance: xMPS, the extended MPS format. Technical report, Carnegie Mellon University and Maximal Software, 2000.

[15] E.R. Harold. *Processing XML with Java: A Guide to SAX, DOM, JDOM, JAXP, and TrAX*. Addison-Wesley, Boston, 2003.

[16] IBM. Introducing the new information integration technology preview, 2002. `http://www7b.software.ibm.com/dmdd/library/demos/0203xperanto/0203xpera%nto.html`.

[17] M. Kay. *XSLT Programmer's Reference 2nd Edition*. Wrox Press, Birmingham, UK, 2001.

[18] M. Kay. Saxon the XSLT processor, 2002. `http://saxon.sourceforge.net/`.

[19] Lindo Systems, Inc. LINDO API user's manual. Technical report, Lindo Systems, Inc., 2002.

[20] Lindo Systems, Inc. What's *best!* 6.0, 2002. `http://www.lindo.com/cgi/frameset.cgi?leftwb.html;wbf.html`.

[21] T.G. Mairs, G.W. Wakefield, E.L. Johnson, and K. Speilbergh. On a production and distribution problem. *Management Science*, 24:1622–1630, 1978.

[22] G. Mitra, C. Lucas, S. Moody, and B. Kristjánsson. Sets and indices in linear programming modelling and their integration with relational data models. *Computational Optimization and Applications*, 4:263–283, 1995.

[23] L. Schrage. *Optimization Modeling with LINGO*. Lindo Systems, Inc, Chicago, IL, 2000.

[24] Aaron Skonnard and Martin Gudgin. *Essential XML Quick Reference*. Pearson Education, Inc, 2002.

[25] Maximal Software. MPL manual, 2002. `http://www.maximal-usa.com/mplman/mplwtoc.html`.

[26] Software AG. Tamino XML Server, 2002. `http://www.softwareag.com/tamino/`.

[27] Frontline Systems. Excel Solver: Premium Solver Platform Version 5.0, 2002. `http://www.solver.com/`.

[28] J. D. Ullman. *Principles of Database and Knowledge - Base Systems Volume*. Computer Science Press, 1988.

[29] H. M. Wagner and T. M. Whitin. Dynamic version of the economic lot size model. *Management Science*, 5:89–96, 1958.

[30] J.S. Welch. PAM–A Practioners' Approach to Modeling. *Management Science*, 33:610–625, 1987.

[31] Ximian, Inc. Ximian and the Mono Project, 2002. `http://developer.ximian.com/projects/mono/`.