LPFML: A W3C XML Schema for Linear Programming

Robert Fourer and Leo Lopes
Department of Industrial Engineering and Management Sciences
McCormick School of Engineering and Applied Science
Northwestern University

Kipp Martin
Graduate School of Business
University of Chicago

6th January 2004

Abstract

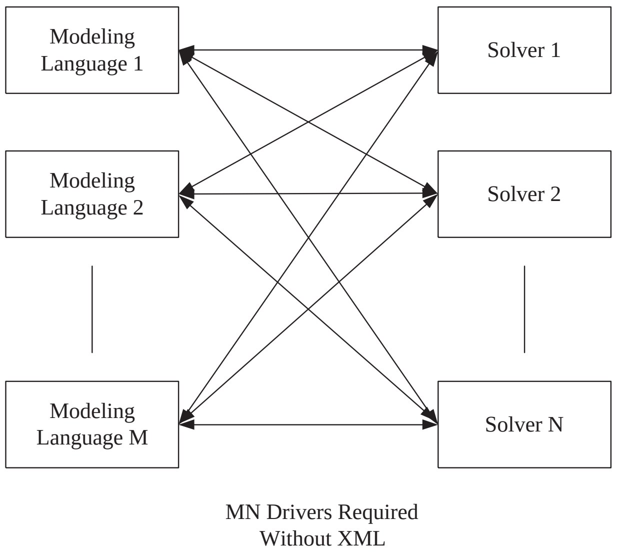
There are numerous algebraic modeling languages for generating linear programs and numerous solvers for computing solutions to linear programs. This proliferation of modeling languages and solvers is frustrating to modelers who find that only certain languages connect to certain solvers. One way to encourage modeler-solver compatibility is to use a standard representation of a problem instance, so that all modeling languages and all solvers deal with problem instances in the same form. Such a standard should be able to express instance-specific and vendor-specific information, should be simple to manipulate and validate, and should promote the integration of optimization software with other software.

Given the increasing importance of XML for data representation and exchange, and XML's ability to support the characteristics above, it is natural to base a proposal for a standard for representing problem instances on XML. In this paper, we present the LPFML Schema, a W3C Schema for representing linear programming problem instances in XML. We also describe a library of open-source C++ classes that we have written to facilitate the exchange of information between modeling languages and solvers. We show how these classes have been used to provide previously unavailable language-solver connections.

1 Introduction

There are many algebraic modeling languages for expressing linear programming models as input to computer systems. Examples include AIMMS [1], AMPL [10], GAMS [4], ILOG OPL [15], LINGO [26], MPL [28], and Xpress-Mosel [6]. There are also many efficient solvers for linear programs, such as CLP [11], CPLEX [14], GLPK [20], LINDO[25], MINOS [23], MOSEK [22], OSL [12], and Xpress-Optimizer [7]. This proliferation of languages and solvers is a difficulty for developers of linear programming software, as modelers may want to use any solver with any modeling language – see Figure 1. If there are M modeling languages and N solvers, then $M \times N$ "drivers" are required for complete interoperability.

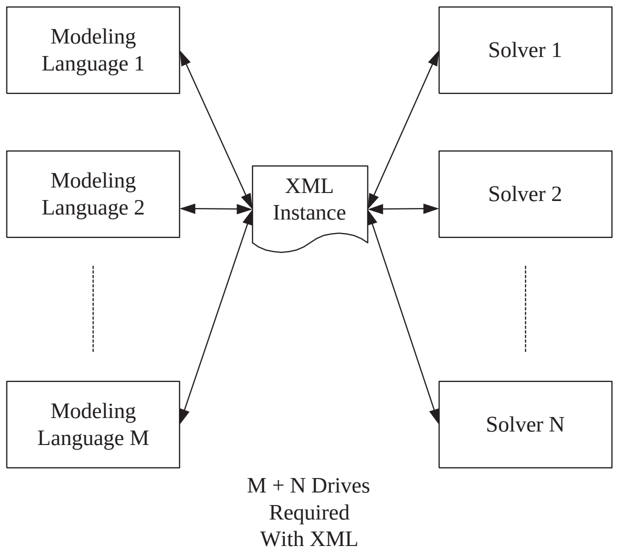
Figure 1: Software required without common interface



One way to increase modeler-solver compatibility is to adopt a standard representation of a problem instance. Here it is important to make a distinction between models and instances. A *model* is an abstract algebraic representation of a problem, an *instance* is an explicit description of a problem's objective and constraints. For linear programming, an instance can be represented as a list of nonzero coefficients of variables in the objective and constraint functions, along with bounds on the variables and constraint functions.

Using the standard representation of an instance, only $M + N$ software drivers

Figure 2: Software required with common interface



are needed for complete interoperability – see Figure 2. Each modeling language translator generates an instance in the standard format and each solver reads an instance in the standard format. Note that the arrows in Figures 1 and 2 are double-headed: a standard form should be able to receive results back from a solver, as well as send problems to a solver.

An algebraic representation of a simple product mix model, in the AMPL modeling language, is given in Example 1.1.

Example 1.1 *Product Mix Example*

```
set PROD;  # products
set DEP;   # processing departments
param hours {DEP};      # time available in dept i
param rate {DEP,PROD};  # hours used per time unit
param profit {PROD};    # profit per product made
var Make {PROD} >= 0;   # number of each product to be made

maximize TotalProfit:
   sum {j in PROD} profit[j] * Make[j];
```

```
subject to HoursAvailable {i in DEP}:
   sum {j in PROD} rate[i,j] * Make[j] <= hours[i];
```

This model describes the data needed by any product mix problem, but does not supply particular data. Instead the data for an instance of the product mix linear program is given separately. Here is an example of the data (see [2]) for an instance, in AMPL's format for data:

```
param: PROD: profit :=
   std    10
   del     9   ;
param: DEP:  hours :=
   cutanddye   630
   sewing      600
   finishing   708
   inspectandpack     135 ;
param: rate:   std   del :=
   cutanddye     0.7   1.0
   sewing        0.5   0.8333
   finishing     1.0   0.6667
   inspectandpack       0.1   0.25  ;
```

The AMPL model together with the above data produces a problem instance. People do not normally look at an instance representation, but AMPL can be directed to display it in a fairly readable form:

```
maximize TotalProfit:
   10*Make['std'] + 9*Make['del'];
subject to HoursAvailable['cutanddye']:
   0.7*Make['std'] + Make['del'] <= 630;
subject to HoursAvailable['sewing']:
   0.5*Make['std'] + 0.8333*Make['del'] <= 600;
subject to HoursAvailable['finishing']:
   Make['std'] + 0.6667*Make['del'] <= 708;
subject to HoursAvailable['inspectandpack']:
   0.1*Make['std'] + 0.25*Make['del'] <= 135;
```

The one widely used standard for representing problem instances in terms of coefficients and limits is the so-called MPS form [13]. The above instance has the following equivalent MPS-form representation:

```
NAME          PRODMIX
ROWS
 N  OBJ
 L  R0001
 L  R0002
 L  R0003
 L  R0004
COLUMNS
    C0001     OBJ       10
    C0001     R0001     0.7            R0002     0.5
    C0001     R0003     1              R0004     0.1
    C0002     OBJ       9
    C0002     R0001     1              R0002     0.8333
    C0002     R0003     0.6667         R0004     0.25
RHS
    RHS1      R0001     630
    RHS1      R0002     600
    RHS1      R0003     708
    RHS1      R0004     135
ENDATA
```

The MPS format illustrated in Example 1.1 has serious disadvantages, however. It is needlessly verbose, as can be seen in the repetitions of the column names and the right-hand side name (RHS1). It does not extend well, moreover. It can be generalized easily enough to specify some integer-valued variables, thus representing instances of *mixed-integer* programs (or MIPs). Occasional proposals for major extensions, such as for nonlinear expressions, have failed to catch on; nor is there any useful provision for conveying solver-specific algorithmic directives, such as branching preferences for MIP solvers.

Current modeling systems instead use their own proprietary model instance formats that various solvers have been adapted to recognize. The existence of all of these forms suggests that the MPS format, even in extended guises, has not proved to be all that modeling language implementers need. At the same time, separate implementations of the MPS form in different modeling systems and solvers have gradually diverged in their handling of spacing and defaults. As a result, the MPS form is today used almost exclusively in test problem libraries and in instances that accompany bug reports to solver vendors. A companion form for results from solvers has mostly fallen into disuse. In addition, there is no MPS format for reporting the results of a linear programming model solution.

Two factors have thus contributed to the proliferation of proprietary model instance formats: the lack of a formal mechanism for validating an instance that claims to adhere to a standard, and the presence of vendor-specific or problem-specific extensions. In addition, the inability of the standard to carry metadata – data about the data, such as branching rules in integer programming – has forced developers to adopt proprietary mechanisms even when the standard representation of data alone might be understood by all parties.

Our research specifically addresses these issues. In addition, we have given special consideration to designing a standard that can be more easily integrated into general information technology infrastructures. We accomplish this through the use of XML.

XML (Extensible Markup Language) is rapidly becoming an accepted format for storing data. In this research, we develop an XML-based dialect, LPFML, for representing instances of linear programs. An XML dialect is formally defined by an XML *schema* (explained in Section 2) against which every file written in the dialect can be automatically validated. This arrangement alone gives an XML dialect several important advantages over MPS and the various proprietary formats:

- Validation against a schema promotes stability of the standard.
- An XML schema can restrict data values to appropriate types – row names to `string`, row indices to `int`, and coefficient values to `double`, for instance.
- An XML schema can define *key* data to insure, for example, that no row or column name is used more than once.
- As the name suggests, XML schemas are extensible. Our LPFML Schema acts a base class that can be extended to include, for example, new constraint types or solver directives. Of course an instance that validates against the extended schema may not validate against the original schema.

Also, XML is increasingly being adopted as a standard for the interchange of information of many kinds. This broader relevance also has benefits for linear programming systems:

- XML is becoming a very popular format for storing data. By storing an instance in XML format we are bringing the model closer to the data source and facilitating the integration of optimization-based solutions into IT infrastructures.
- XML is the data interchange language of Web services. Future linear (and other) solvers are likely to be made available as Web services, in which case it will be important to have an XML representation of problem instances.

- XML lends itself very well to compression. In Section 5 we describe compressed representations of our XML representations of linear programs, which are still in XML format.

- XML-based Extensible Stylesheet Language Transformations (XSLT) offer a convenient way to specify translations of XML documents. If a linear program instance (with perhaps corresponding solution) is stored in XML, then XSLT can be applied to the instance to easily produce a Web browser (HTML) document that displays the linear program data or solution data in human-readable form.

- Encryption standards such as XML Encryption are emerging for XML data – see `http://www.w3.org/Encryption/2001/`. This option is important to commercial linear programming applications, where the problem instances may contain confidential data.

All of XML's advantages for validating files, defining keys, compression, and the like are provided by numerous XML tools designed for manipulating and parsing XML data. It suffices to define our XML dialect in the form of a schema that these tools can work with. This contrasts to ad hoc formats that require specially writing, debugging, and maintaining the routines equivalent to these tools.

A related problem to the one we address is that of developing an XML dialect for optimization *models*. It is certainly feasible to develop an XML dialect for expressing the concepts of set, index, variable, parameter, and so forth within an XML dialect. In fact, many of the necessary constructs are already present in the MathML dialect [29]. An XML modeling dialect could take its place alongside the other modeling languages in Figures 1 and 2. However, this is not the goal of the research we report here. Creating a standard for instances is fundamentally different from creating a standard for models. The mathematical components are different, the efficiency and representational considerations are different, and the context in which the solutions to each problem are to be applied are very different. Modeling systems are often audience-specific or application-specific or both. We want a methodology where a minimum amount of agreement about syntax is required among the user community. The model instance requires the least amount of agreement, and has the widest applicability.

This paper is not the first attempt to incorporate XML into mathematical modeling. See Chang [5] and Kristjánsson [17] for two proposals for representing linear program instances in XML. In contrast to others' proposals for XML dialects for representing linear programming instances, however, we are providing associated open-source libraries that developers of modeling languages or solvers can use to read and write files expressed in our proposed format.

Ezechukwu and Maros [8] describe their own AML (Algebraic Markup Language) which uses XML to describe the model rather than the instance. Finally, Martin [21] demonstrates how to bypass a traditional algebraic modeling language and use XSLT to transform raw data into an XML description of the problem instance that validates against the LPFML Schema of this paper. Finally, see the survey paper by Bradley [3] for a good presentation of the uses of XML technologies in Operations Research.

In the next section we provide the necessary background material on XML, schemas, and other technologies used in this paper. In Section 3 we describe LPFML, a W3C XML Schema used to define the format for representing instances of linear programs. Any format for linear programming can of course handle mixed-integer programming by allowing for some variables to be specified "integer." This is true of our LPFML format and in this section we describe how to specify integer valued variables. Furthermore, our schema allows for specification of branching strategies, cuts, on-the-fly column generation, etc.

In Section 4 we describe the libraries we have written based on the LPFML linear programming schema. These open-source libraries are designed to facilitate linking solvers with modeling languages, by taking care of low-level tasks such as parsing LPFML files. As an illustration, we have used these libraries to provide for conversion from the AMPL modeling language to LPFML, and for conversion from LPFML to the LINDO API and COIN OSI solver front-ends. The LINDO API is a product of LINDO Systems, Inc. and is an interface for the underlying LINDO solvers. The COIN (COmputational INfrastructure for Operations Research) OSI (Open Solver Interface) library is an open source optimization library, currently hosted by IBM, that provides a C++ API for numerous solvers such as CPLEX, GLPK, and CLP. Thus our libraries have helped make possible previously unavailable connections from AMPL models to the LINDO solver and any solver that supports an OSI interface.

We expect that under many scenarios, linear programming solvers will be used as Web services. In this case, instances of linear programs will be sent over the network, and size is an issue. In Section 5 we describe two methods for compressing the XML representation of a linear program instance. The compressed files are still in XML format.

The software libraries we developed are released as open source. In Section 6 we discuss the choice of our software license and provide details about the distribution. The paper concludes in Section 7 with several important extensions and implications for this work.

2 Basic XML Technologies

In this section we give a brief overview of the XML technologies used in this paper. See also the excellent overview of these technologies by Skonnard and Gudgin [27]. An XML file is a text file that contains both data and markup. Consider the text in Figure 3, describing the rows of a linear program.

Figure 3: XML Representation of Row Data

```
<rows>
   <row rowName="cutanddye" rowUB="630"/>
   <row rowName="sewing" rowUB="600"/>
   <row rowName="finishing" rowUB="708"/>
   <row rowName="inspectandpack" rowUB="135"/>
</rows>
```

This text contains both data, such as a row upper bound of 630; and a row name, `cutanddye`. The text also contains markup, or metadata, in the form of *elements* and *attributes* that describe or give meaning to the data. In this example there are two elements, `<rows>` and `<row>`. Elements are defined by an opening `<tag>` and closing `</tag>`. In this specific example, the `<row>` element has two attributes: `rowName` and `rowUB`. The attributes are used to define or characterize each `<row>` element. In this respect, the `<row>` elements correspond to records in a relational database and the attributes correspond to fields.

Unlike a relational database, the XML structure is tree-like or hierarchical and not restricted to a two dimensional table structure. For example, Figure 4 is an XML representation of the constraint matrix coefficients of Example 1.1. This matrix represented in this XML data is encoded in a sparse storage scheme typical of linear programs. The `<pntANonz>` element contains `<el>` elements whose values point to the start of each column. The `<rowIdx>` element contains the row indices of each nonzero, and the actual nonzero elements are contained in the `<nonz>` element. The complete tree-like structure corresponding to this XML data is illustrated in Figure 5.

In Figure 5 the icon next to the `<sparseMatrix>` tag of a line through the three dots implies a required sequence of child elements, whereas the “switch” icon represents a choice among child elements. For example, there is a choice between either a `<rowIdx>` child of `<sparseMatrix>` if we store in column major form, or a `<colIdx>` child of `<sparseMatrix>` if we store in row major form. Similarly, a `<nonz>` element may have `<el>` children or a

Figure 4: XML Representation of Constraint Matrix Data

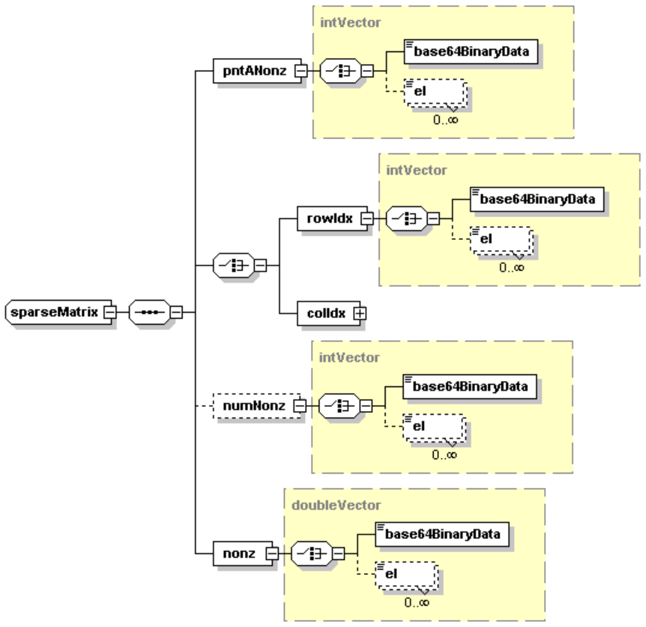
```
<sparseMatrix>
   <pntANonz>
      <el>4</el><el>8</el>
   </pntANonz>
   <rowIdx>
      <el>0</el><el>1</el><el>2</el><el>3</el>
      <el>0</el><el>1</el><el>2</el><el>3</el>
   </rowIdx>
   <nonz>
      <el>.7</el><el>.5</el><el>1.0</el><el>0.1</el>
      <el>1.0</el><el>0.8333</el><el>0.6667</el><el>0.25</el>
   </nonz>
</sparseMatrix>
```

`<base64BinaryData>` child if compression is used (see Section 5). The rectangles containing the element names are either solid or dashed. A solid rectangle indicates that the element is mandatory and must be a singleton. A dashed rectangle denotes that the element is optional. Note also the $0..\infty$ icon below the `<el>` element. This indicates that an unlimited number of elements of this type are allowed.

In the XML representation of the constraint matrix data illustrated in Figure 4, the text markers surrounding each tag (< and >), as well as other elements of the XML syntax, serve a very important purpose: they make XML instances very easy to parse and to validate. In order for a parser to parse and construct an appropriate tree such as the one illustrated in Figure 5 from an XML document, the document must be *well formed*. An XML document is well formed if 1) both opening and closing tags are present, 2) the opening and closing tag names exactly match both in name and case (XML is case sensitive), and 3) the tags are nested properly (the closing tag of a child element must precede the closing tag of its parent element). Numerous parsers, both open source and proprietary, are available for parsing an XML document and determining if the document is well formed. In our work we use the Xerces parser from The Apache Software Foundation (`www.apache.org`).

The concept of well formed relates only to the syntax of an XML file. An even more useful concept is that of a *valid* XML document. An XML document is valid if it is well formed and the use of elements and attributes in the document is consistent with an associated *schema*. Specifying the format for the instance of a

Figure 5: Sparse Matrix Element



linear program amounts to specifying a schema against which the XML document is validated. It is useful to think of the schema as a set of class descriptions and the actual XML document elements as objects in the classes.

A powerful feature of the XML Schema standard defined by the World Wide Web consortium is that it allows for both built-in and user-defined "types" (or classes). We illustrate this concept using the schema associated with Figures 3 and 4. In Figure 6 is a schema for the `<rows>` element illustrated in Figure 3. This schema, part of the LPFML Schema, is described further in Section 3. The `<complexType>` in Figure 6 is a user-defined type and can contain other elements, attributes, or text. In this case the `<rows>` element contains child elements of type `<row>`. Each `<row>` element has four optional attributes. These attributes are built-in types, not user-defined types. For example the attribute `rowName` is of type `string` and the row upper and lower bounds `rowUB` and `rowLB` are of type `double`.

A second example of a user-defined type is `intVector`. In order to support sparse storage of the constraint matrix, we require integer vectors. They are used

Figure 6: Rows element of LPFML Schema

```
 <xs:element name="rows">
   <xs:complexType>
      <xs:sequence>
      <xs:element name="row" minOccurs="0" maxOccurs="unbounded">
         <xs:complexType>
            <xs:attribute name="rowName" type="xs:string"
               use="optional"/>
            <xs:attribute name="rowUB" type="xs:double"
               use="optional"/>
            <xs:attribute name="rowLB" type="xs:double"
               use="optional"/>
            <xs:attribute name="mult" type="xs:int"
               use="optional"/>
         </xs:complexType>
      </xs:element>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

both as pointers and to store row or column indices. An integer vector is a good example of a user-defined type that is used in the definition of other types.

Consider the schema definition of the `intVector` type illustrated in Figure 7. The type `intVector` contains a `<choice>` between: exactly one element of type `<base64BinaryData>`; or (the mutually exclusive or) between 0 and an unbounded number of elements of type `<el>`. The element `<el>` is also a `<complexType>`. It contains text that is of built-in type `int`. This implies that the text contained in the `<el>` element of an `intVector` type must be parsed as integer data. A validating parser should give an error message if non-integer data are encountered. This is a desirable feature when validating the instance of a linear program. We can check for data of an incorrect type. The `<el>` element also has two attributes, `mult` and `incr`. These attributes are used for compression and their use is described in Section 5. The `intVector` type does not get instantiated as an element in an XML file. It is not valid to have an `<intVector>` element. Rather, the `intVector` type is used to define other elements that appear in the XML file. The elements `<pntANonz>` and `<rowIdx>` introduced in Figure 4, which are also a `<complexType>`, are of type `intVector`. The type `intVector` is like an abstract class in C++ with `<pntANonz>` and `<rowIdx>` classes that derive from it.

Figure 7: complexType intVector of LPFML Schema

```
<xs:complexType name="intVector">
   <xs:choice>
      <xs:element name="base64BinaryData"
         type="base64BinaryData"/>
      <xs:element name="el" minOccurs="0"
         maxOccurs="unbounded">
         <xs:complexType>
            <xs:simpleContent>
               <xs:extension base="xs:int">
                  <xs:attribute name="mult" type="xs:int"
                     use="optional"/>
                  <xs:attribute name="incr" type="xs:int"
                     use="optional"/>
               </xs:extension>
            </xs:simpleContent>
         </xs:complexType>
      </xs:element>
   </xs:choice>
</xs:complexType>
```

In the definition of the type `intVector` in Figure 7 the element `<el>` is defined and required to contain integer data. Thus `<el>` behaves much like a local type in a class definition. However, `<el>` is also a local type in the definition of the type `doubleVector` (part of LPFML Schema but not shown in Figure 7). The constraint matrix nonzero elements are stored in the `<el>` element. The `<el>` element is a child of the `<nonz>` element, as shown Figure 4, which is of type `<doubleVector>`. In this case the text of an `<el>` element must be a double precision real number.

We illustrated how the W3C XML Schema specification allows the attribute and element text to be of numerous built-in data types, e.g. string, int, double, base64, etc. These basic types are further enhanced by a `<simpleType>` type. The `<simpleType>` element specification allows the user to define the type of text that can make up an element or an attribute. Consider the example in Figure 8.

In this example we are defining an attribute type we call `colType`. This attribute must be a string consisting of a single character which is either `C` if the corresponding column element represents a continuous variable, `B` if the corre-

Figure 8: simpleType colType of LPFML Schema

```
<xs:simpleType name="colType">
   <xs:restriction base="xs:string">
      <xs:enumeration value="C"/>
      <xs:enumeration value="B"/>
      <xs:enumeration value="I"/>
   </xs:restriction>
</xs:simpleType>
```

sponding column element represents a binary variable, or `I` if the corresponding column element represents a general integer variable.

An XML schema is itself an XML document. Indeed, the W3C XML Schema standard is an XML vocabulary for defining schemas. Each `<complexType>` and `<simpleType>` in our examples is qualified with an `xs`. This qualification on a tag is saying that the tag is in a specified *namespace*. In this particular example, there is an attribute in the root element of the schema

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

This attribute in the root element tells the parser that any element qualified by `xs` belongs to the namespace uniquely identified by the URI (uniform resource identifier)

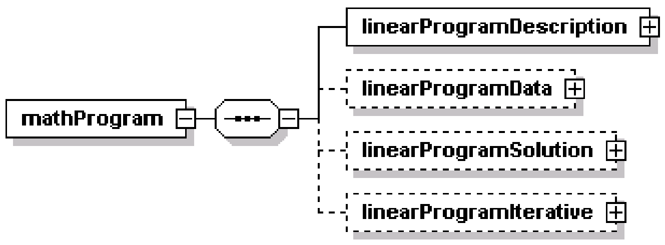
```
http://www.w3.org/2001/XMLSchema
```

Another important XML technology is Extensible Stylesheet Language Transformations (XSLT). This is an XML-based programming language for transforming XML files into other XML files. The transformation is based upon a *stylesheet*. 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. However, unlike C++ and Java, XSLT is a functional programming language, not a procedural programming language. With regard to our work, one important use of XSLT is to take the XML representation of a linear programming instance and solution and convert it into an HTML document that is easily readable by humans. For example, with XSLT it is easy to read the solution to a linear program in an XML file, select the variables of interest along with their solution values, and display the results in an HTML table. For a excellent treatment of XSLT programming see Kay [16].

3 LPFML Schema

In this section we describe our LPFML Schema. The root element is `<mathProgram>`. The root element has four children. See Figure 9.

Figure 9: Math Program Element and Children



The `<linearProgramDescription>` element is used to convey the basic properties of the linear program instance. See Figure 10. This element's children are self-explanatory except for the `<option>` element.

The `<option>` element is an extension mechanism. Its role is to support the transfer of information *related* to the model. For example, user preferences (such as precision digits in displaying results, or the frequency with which intermediary results are produced) or parameters useful to the solution method (such as useful cut generation strategies or stability and sparsity parameters) can be communicated using the `<option>` element.

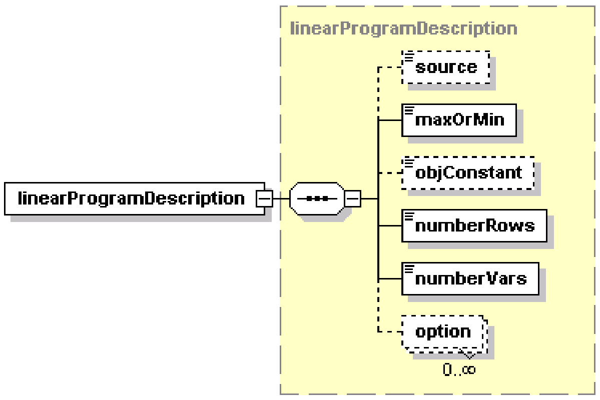
The `<option>` element has an optional `solver` attribute. This attribute is set when using solver-specific options (e.g. `<option solver="lindo">`). An application may issue a warning if it doesn't find an `<option>` for itself, especially if it finds `<option>` tags for other elements. If it does find an `<option>` tag for itself, it should attempt to parse the contents of the `<option>` tag. In this case, if it finds a discrepancy, it should raise an error message.

If the `solver` attribute is not used, then the content of the `<option>` element is free. It is up to the individual application to parse the content of the `<option>` tags. If an application does not understand the content of an `<option>` tag without a `solver` attribute, it should ignore the directive. The application may issue a warning, but should not raise an error condition.

The `<linearProgramDescription>` element and its children for the linear program instance in Example 1.1 are

```
<linearProgramDescription>
```

Figure 10: Linear Program Description Element



```
    <source>Par Inc. Problem from Anderson, Sweeny,
       and Williams </source>
    <maxOrMin>max</maxOrMin>
    <numberRows>4</numberRows>
    <numberVars>2</numberVars>
</linearProgramDescription>
```

The actual data that comprise the linear program instance are contained in the element `<linearProgramData>`. See Figure 11. This element has four children.

1. The `<rows>` element, which contains an unbounded number of `<row>` children. There is a one-to-one correspondence between rows in the instance and `<row>` elements. Each `<row>` has four optional attributes. The attributes are `rowName`, `rowUB`, `rowLB`, and `mult`. The `mult` attribute is used for compression and is described in Section 5. If row names are not provided, the rows are uniquely identified by an index assigned to them based on their order in the file. For the linear program instance in Example 1.1, the `<rows>` element and its children are

```
<rows>
   <row rowName="cutanddye" rowUB="630"/>
   <row rowName="sewing" rowUB="600"/>
   <row rowName="finishing" rowUB="708"/>
   <row rowName="inspectandpack" rowUB="135"/>
</rows>
```

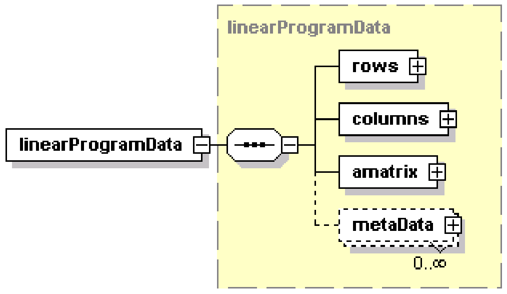
2. The `<columns>` element, which contains an unbounded number of `<col>` children. There is a one-to-one correspondence between columns in the instance and `<col>` elements. Each `<col>` has six optional attributes. The attributes are `colName`, `colUB`, `colLB`, `objVal`, `colType`, and `mult`. The `objVal` attribute is the objective function coefficient and is zero by default. The `colType` attribute has three possible values, i) `C` for continuous, ii) `B` for binary, and iii) `I` for general integer. If the colType attribute is not present, `C` is the default value. As with the rows, the `mult` attribute is used for compression and if column names are not provided, the columns are uniquely identified by an index assigned to them based on their order in the file. For the linear program instance Example 1.1, the `<columns>` element and its children are

   ```
   <columns>
      <col objVal="10" colName="x1" colType="C" colLB="0.0"/>
      <col objVal="9" colName="x2" colType="C" colLB="0.0"/>
   </columns>
   ```

3. The `<aMatrix>` element currently has only one child element which is `<sparseMatrix>` (described in detail below). In the future, users may wish to incorporate matrices stored differently than in `<sparseMatrix>`.

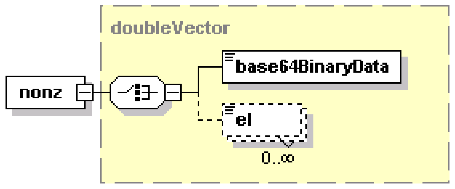
4. The `<metaData>` tag may appear in several sections of the file. It is another extension mechanism, similar to the `<option>` tag discussed earlier. Like `<option>`, it contains data about the data, its contents are application-specific, and it should be ignored if not understood by an application. Unlike `<option>`, it contains information about specific components of the instance, rather than about the instance as a whole. For example, to communicate branching priorities, a `<sparseVector>` might be enclosed in a `<metaData info=“priorities” object=“columns”>` tag at the end of `<linearProgramData>`. If the modeling environment and the language agree that “priorities” associated with “columns” stands for branching priorities, then the priorities can be correctly interpreted and used. Otherwise, the solver simply ignores the directive and possibly issues a warning. The labels “priorities” and “columns” used in `<metaData>` are not prescribed by the standard.

Figure 11: Linear Program Data Element



The `<sparseMatrix>` stores the nonzero elements of the constraint matrix (refer back to Figure 5 used earlier to illustrate the hierarchical nature of XML). It does so using a traditional sparse storage scheme. There is an element `<pntANonz>` which is an integer array of pointers. This element can have two types of children. If base 64 compression is not used, then `<pntANonz>` has an unbounded number of children elements `<el>`. In this case the *ith* `<el>` element points to the start of the nonzero elements for column (row) $i + 1$. These nonzero elements are stored in the element `nonz` which is a vector of elements where each element `<el>` must be a double precision value. The `<nonz>` element is illustrated in Figure 12. If the matrix is stored in column major form, the row indices are stored in the element `<rowIdx>` as an integer vector. Similarly column indices are stored in `<colIdx>` if the constraint matrix is stored in row major form.

Figure 12: nonz Element



In many modeling situations it is desirable to add rows or columns to the matrix *after* an initial solution has been obtained; for example, adding cuts when solving an integer program or adding columns when using column generation. Thus, in a sparse representation it is important to leave extra room for nonzero elements. This is done using the `<numNonz>` element. For example, if the matrix is stored in column major form, there is a child element of `<numNonz>` for each variable. The *ith* child element `<el>` of `<numNonz>` is the number of potential nonzero elements in column i.

In order to make for a smaller file, all of the vectors in `<sparseMatrix>` may be stored in base 64 format. More on this in Section 5. For the linear program instance in Example 1.1, the `<sparseMatrix>` element and its children are

```
<sparseMatrix>
   <pntANonz>
      <el>4</el><el>8</el>
   </pntANonz>
   <rowIdx>
      <el>0</el><el>1</el><el>2</el><el>3</el>
      <el>0</el><el>1</el><el>2</el><el>3</el>
   </rowIdx>
   <nonz>
      <el>.7</el><el>.5</el><el>1.0</el><el>0.1</el>
      <el>1.0</el><el>0.8333</el><el>0.6667</el><el>0.25</el>
   </nonz>
</sparseMatrix>
```

The `linearProgramSolution` element is used to store the solution to the linear program. See Figure 13. There are six child elements.

1. The `<primalSolution>` element contains an element `sol` for each element in the solution. The `<sol>` element has three attributes. They are `idx`, `val`, `name`. Both `idx` and `val` are required, while `name` is optional. The logic for this is that the user may wish to present the solution in sparse format, e.g. list only the nonzero primal or dual variables. In this case, the index on a variable cannot be based on the position of the element in the file. The solution value must be assigned a unique index and this is done with the attribute `idx`. The variable name associated with `idx` is optional.

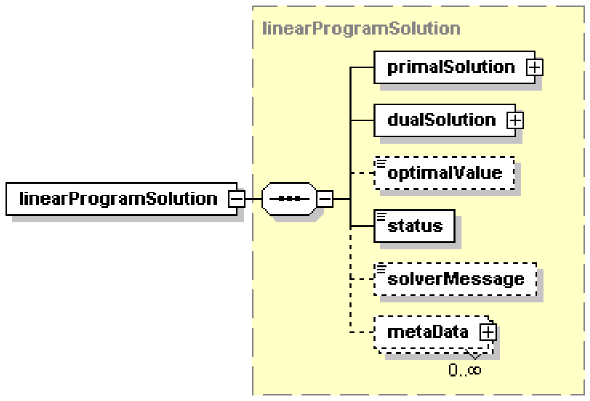
2. The `<dualSolution>` element is analogous to `<primalSolution>`.

3. `<optimalValue>` – self explanatory

4. The `<status>` element is used to indicate whether the given solution is optimal, or if the problem is unbounded, or infeasible.

5. `<solverMessage>` – any solution related message returned by the solver

Figure 13: Linear Program Solution Element



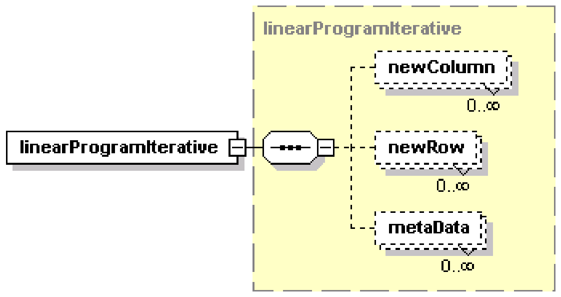
6. `<metaData>` (described earlier).

For the instance in Example 1.1, the `<linearProgramSolution>` element and its children are

```
<linearProgramSolution>
   <primalSolution>
      <sol idx="1" name="x1" val="540"/>
      <sol idx="2" name="x2" val="252"/>
   </primalSolution>
   <dualSolution>
      <sol idx="1" name="cutanddye" val="4.37457"/>
      <sol idx="3" name="finishing" val="6.9378"/>
   </dualSolution>
   <optimalValue>7667.94</optimalValue>
   <status statusId="optimalSolutionFound">Put in here
      any other status message desired</status>
   <solverMessage>This was solved using LINDO from LINDO
    Systems, Inc.</solverMessage>
</linearProgramSolution>
```

Users may wish to use a solver in an iterative fashion, for example, add columns or cuts (or both) on the fly. The `linearProgramIterative` element is de-

Figure 14: Linear Program Iterative Element



signed to handle this need without requiring parts of the problem which remain unchanged to be specified anew.

4 The Libraries

A major contribution of this work is a set of open source libraries for reading and writing LP instances in XML format. This library serves three purposes:

1. It allows the format to be used immediately.
2. It hides all the parsing code, allowing solver and modeling language developers to deal only with familiar mathematical objects, like objectives, constraints, etc.
3. It allows for future changes and extensions to be implemented without requiring any solver or modeling language code to be rewritten.

In Section 4.1, we explain how to use the library to parse an instance of a linear program. In Section 4.2, we explain how to use the library to write an instance of a linear program. Section 4.3 is a more detailed description of the classes in the library, how they relate to each other, and how they use the technology provided by XML parsers.

4.1 Parsing

The key service provided by the library is to parse the XML file, get the necessary data, put those data into a sparse matrix storage scheme convenient for linear programming, and present the data to the solver without burdening the solver with any of the parsing implementation. In order to parse an XML file with a linear programming instance, a `main` function creates a single class, derived from `FMLParser`. This class that inherits from the `FMLParser` class is a function of the solver used, for example an OSI solver or LINDO. This inherited class receives (via a `handler` class described in Section 4.3) the instance data in the XML file and transforms this instance data into the format that is compatible for the solver API. Which solver to use and corresponding `FMLParser` class to create is controlled by command line arguments to the `main` function. The `main` function is in the `FMLParse` utility that is described in further detail later in this section.

The `FMLParser` class contains several methods for accessing different components of a linear program. For example, `FMLParser` has a method called `onObjectiveSense`. This method is *called by the libary* when the library finds out whether the file being read contains a minimization or a maximization problem. When a solver-specific class is derived from `FMLParser`, it is the derived version of the method that is called. In that method, solver-specific initialization can be accomplished. For example, here is the OSI implementation of `FMLParser::onObjectiveSense`:

```
 void FMLOSIParser::onObjectiveSense(const bool isMin)
{
   isMin_ = isMin;
   solver_->setObjSense( isMin? 1. : -1.);
}
```

In the code above, solver_ is a pointer to an OSI-specific interface class, which has a method `::SetObjSense` and `isMin_` is class variable of type `bool` in the `FMLParser` class and defines the direction of the optimization. Notice that the person implementing this parser needed to know the solver's interface library, *but did not have to deal with any XML-specific concepts at any time*.

Analogous methods exist for variables, constraints, and coefficient matrices, as well as for initial points, solutions, and dual information. In each case, the method is invoked with regular C++ vectors or constants as parameters, *after* the XML parsing has been done by the library. By implementing the parser as an event-driven library, we achieve two very important benefits:

1. We avoid having to search the file at any point. The file is read sequentially, which is especially important when the solver and the modeling environment reside at different places on a network.

2. We reduce the number of simultaneous copies of the same data that have to exist at any given time. As soon as the method associated with the appropri-

ate event is called, the parsers' representation of the objects can be deleted, which is especially relevant for very large problems.

In addition to the event-driven methods, `FMLParser` also contains a `solve` method and a `write` method, that need to be specialized for each solver, and a few other utilility methods, that should not need to be overriden. The `solve` method is used to send the instance data to a solver using the solver specific API. The `write` method is described in more detail in the next subsection.

4.2 Writing

In addition to parsing services, the library also provides writing services. As with the parsing services, the writing services provided by the library also abstract completely the XML manipulation, providing instead an intuitive mathematical interface in terms of vectors and matrices.

Advanced features provided by the LPFML proposal, like structural compression and base64 encoding, can be enabled or disabled by a simple call. The library then takes care of writing an instance with the features selected by the user.

The library makes available, as members of the `FMLLPToXML` class, several `::setXYZ` types of methods. For example, there is a `::setRows` method, a `::setColumns` method, a `::setLPDescription` method, etc.

Each method has at least two signatures: one using C-style representations of arrays (using pointers); and one using C++ style representations of arrays (using the STL). In addition, more signatures might be available if they provide some convenient functionality. For example, here are three signatures for the `::setRows` method:

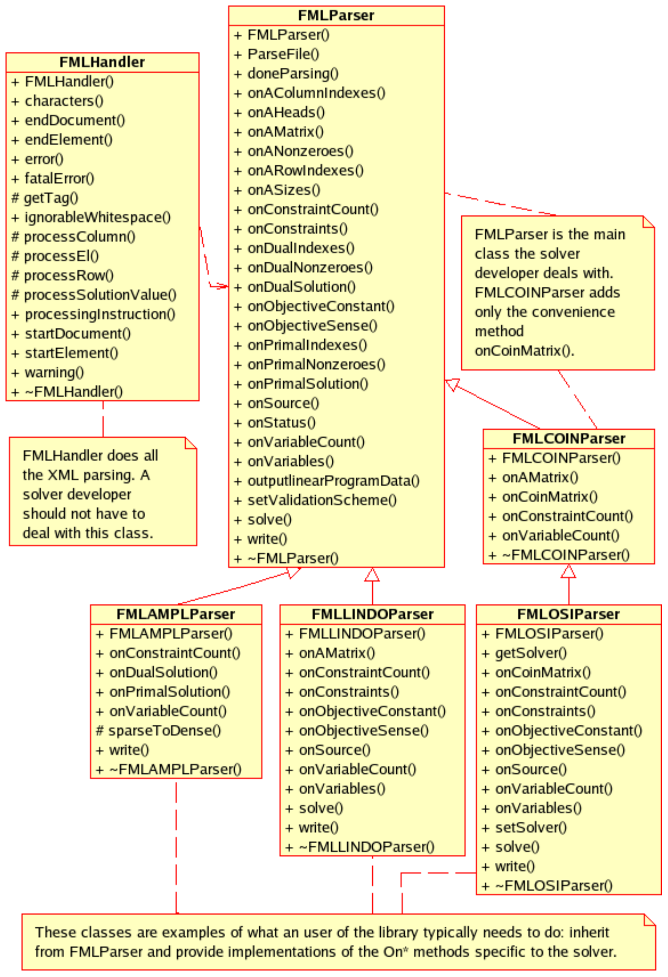
```
void setRows( char** const rowNames,
   const double* lhs, const double* rhs);
void setRows( const vector<string> &rowNames,
   const vector<double> &lhs,
   const vector<double> &rhs);
void setRows( const vector<double> &lhs,
   const vector<double> &rhs);
```

The first signature provides the C-style interface, while the second and third provide C++-style interfaces. In addition, the third signature dispenses with the use of row names.

4.3 Classes

The discussion in this subsection is especially relevant for those who wish to add functionality to the library, understand how it is implemented, or make significant changes to the library to try to improve its efficiency, like replacing the parser library we used (Xerces) with another parser library. Here we provide a more detailed description of the classes in the library, how they relate to each other, and how they use the technology provided by XML parsers.

Figure 15: The Parser Classes



There are several generic ways to read an XML file. Two widely accepted technologies are the Simple API for XML (SAX) 2.0 and Document Object Model (DOM). We chose to use SAX2 in this parser, because it is faster and requires less memory than DOM. We use DOM to write an XML file, as described later.

SAX parsers are event-based. They call functions in the user's code upon finding specific elements, attributes, characters, etc. Our library provides essentially the same functionality, but at an optimization level. Our library calls functions in *its* user's code when it finds objectives, matrices, constraints, etc. Below is a brief description of the main classes in our library. Complete documentation generated by Doxygen is available at our LPFML Web site. The classes used to read and parse the XML instance file are illustrated in Figure 15.

There are two key classes in the SAX 2.0 standard used to parse an XML file. The first class is `XMLReader`. An object of this class is used to actually parse the XML file. The second class is `DefaultHandler.` When the parser object detects various XML constructs such as elements (begin and end) and attributes, methods in the `DefaultHandler` class are called. The following classes in the LPFML Library use these two classes.

FMLHandler: This class inherits from the DefaultHandler class (which implements the default behavior for the SAX2 ContentHandler interface). When the SAX parser encounters the start and end of elements, the appropriate method (for example, `startElement` or `endElement`) in FMLHandler is called. These methods aggregate several pieces of data, and build the components of the linear program. These data are used in arguments for methods such as `onConstraints` in the class `FMLParser` described next.

FMLParser: This class takes care of initializing the Xerces library, including creation of an `XMLReader` parser object. It also provides numerous virtual methods that are called by an `FMLHandler` object. For example, the method `onConstraints` is used to get row information.

```
virtual int onConstraints(vector<std::string>
   const &label, vector<double> const &lhs,
   vector<double> const &rhs )
{return 0;};
```

When overriden by a solver specific implementation, these methods create or populate the necessary data structures in the solver. In `FMLParser` all these methods do nothing. The solver specific classes that inherit from `FMLParser` are responsible for the actual implementation of the methods.

FMLCOINParser: This class inherits from `FMLParser`, and adds a convenient method `onCoinPackedMatrix`, that provides the constraint matrix in a `CoinPackedMatrix` data structure. To accomplish this, `FMLCOINParser` implements the `onAMatrix` virtual method of `FMLParser` and creates an object in the class `CoinPackedMatrix` class.

FMLOSIParser: This class inherits from the `FMLCOINParser` class. This class implements all of the methods in `FMLCOINParser` (and consequently in `FMLParser`) needed to describe a linear program. It is used to connect an LPFML file (an XML file that validates against the LPFML Schema) to any solver that has an Open Solver Interface implementation. For example, as illustrated below, the method `onConstraints` is used to get the name of the rows and the upper and lower bounds on each row.

```
int FMLOSIParser::onConstraints(vector<std::string>
   const &label, vector<double> const &lhs,
   vector<double> const &rhs )
{
   int i;
   lhs_ = new double[nRows_];
   rhs_ = new double[nRows_];
   std::copy(&lhs[0], &*lhs.end(), lhs_);
   std::copy(&rhs[0], &*rhs.end(), rhs_);
   vector<string>::const_iterator iConNameLabel =
   label.begin();
   char *p;
   rowNames_ = new char*[ nRows_];
   cout << "nRows =  " << nRows_ << endl;
   for(i = 0; i < nRows_; i++)
   {
      p = new char[iConNameLabel->size() + 1];
      strcpy(p, iConNameLabel->c_str());
      rowNames_[ i] = p;
      iConNameLabel++;
   }
   return 0;
}
```

The `FMLOSIParser` implementation of `FMLParser::solve()` also uses only the OSI. Thus, this class can call any solver that has an OSI interface. This is done by including the solver specific OSI header file and creating the corresponding solver interface class. In our implementation we tested the CLP (Coin Linear Program) and GLPK (GNU Linear Programming Kit) solvers.

FMLLINDOParser: This is another implementation of a parser, specific for the LINDO solver. `FMLLINDOParser` inherits from `FMLParser` and implements the same methods that `FMLOSIParser` does. It plays the same role as

FMLOSIParser, but generates data structures for the LINDO API as opposed to those for an OSI solver. An interesting distinction between the FMLLINDOParser and FMLOSIParser classes is that the LINDO API makes copies of all the parameters passed to it. The OSI API allows the data to be assigned to the solver, which takes responsibility for the management of that memory from that point on. Our library supports either scheme, and in the OSI case this prevents another copy of the data from being made in memory.

FMLLPToXML: After the linear program instance is read into a solver and optimized, the class FMLLPToXML is used to output the primal and dual solution (with the original linear programming data if the boolean variable outputLPdata is true) to a document object model (DOM). In the present implementation the DOM is written to file. However, the DOM is a very flexible data structure and could be used several different ways. For example, the DOM output could be used in an iterative fashion where the dual variables are used to form a Langrangian relaxation of the problem. A second use of the DOM is in conjunction with an XSLT transformer to transform the linear programming solution into human readable HTML. This is illustrated in Section 7.

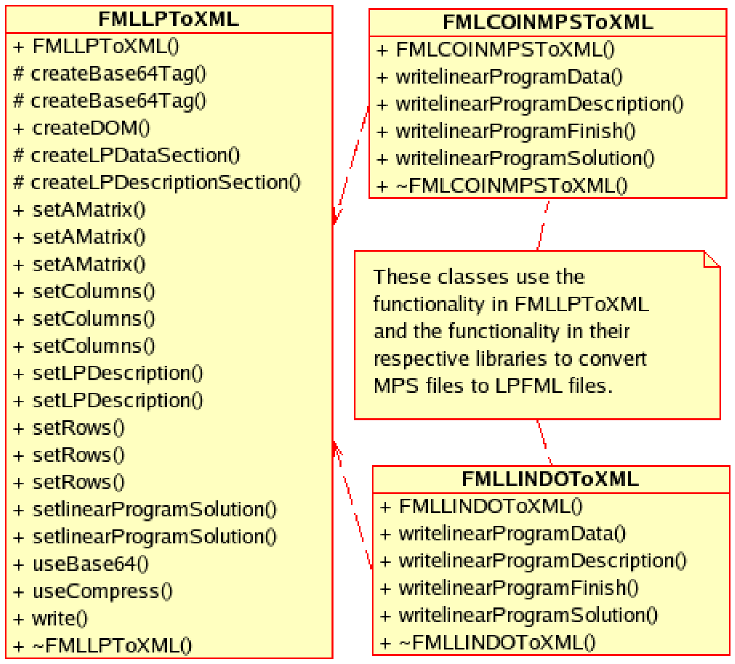
FMLAMPLParser: This class inherits from FMLParser. Unlike the other parser classes that implement methods for reading the *input* XML file, this class implements methods for reading the primal and dual solution in the XML file created by the FMLLPToXML class. There is an additional write method that is AMPL specific and returns the data to AMPL. If a different modeling language were used this method would need to be modified accordingly.

We also distribute some utilities with the library. These utilities serve as examples, provide some convenient functionality, and play a demonstration and debugging role. The classes used to write the solver solution in XML format and convert MPS format to XML format are illustrated in Figure 16.

FMLParse: This utility takes an LPFML file, and through our library, solves the linear program using any of the currently supported solvers. FMLParse creates and manipulates only an FMLParser object. Depending on which solver is selected by the user (currently GLPK, CLP, or LINDO), an appropriate child of FMLParse is instantiated and used to solve the problem. As new solvers become available, only the selection mechanism in FMLParse needs to be changed. The CLP solver does not work in the Windows environment with libraries compiled using Visual Studio .NET.

nl2fml: This utility is designed to work with the AMPL modeling language. In AMPL terms, it is a *driver*. A model instance is input into AMPL in AMPL format. Then the solver option in AMPL is set to nl2fml. Upon execution, nl2fml.exe converts an AMPL nl file into an XML problem instance. A parser object (for example FMLOSIParser or FMLLINDOParser) is created to parse

Figure 16: The Writer Classes

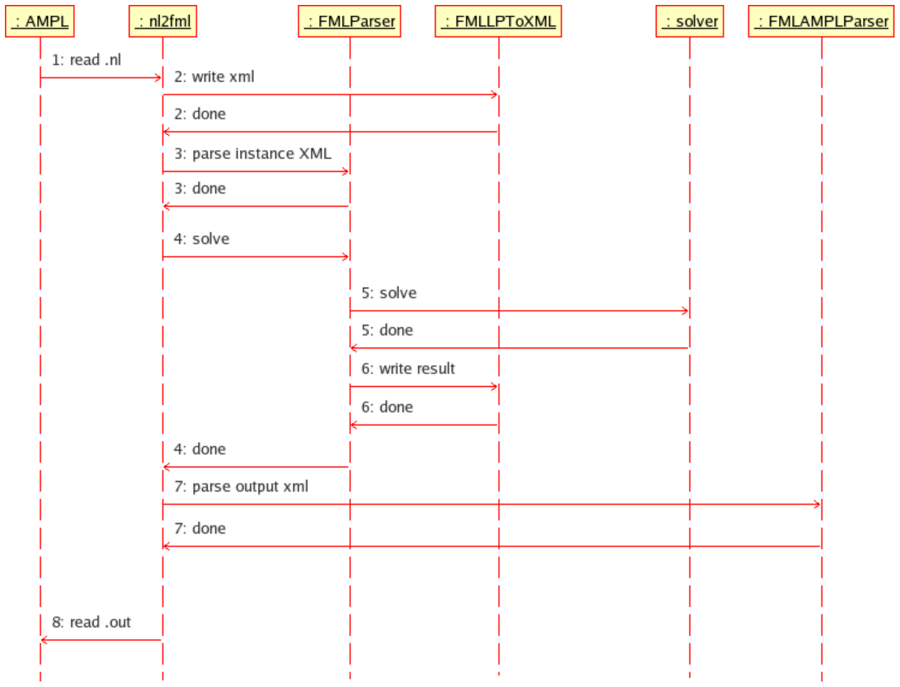

FMLLPToXML
+ FMLLPToXML()
# createBase64Tag()
# createBase64Tag()
+ createDOM()
# createLPDataSection()
# createLPDescriptionSection()
+ setAMatrix()
+ setAMatrix()
+ setAMatrix()
+ setColumns()
+ setColumns()
+ setColumns()
+ setLPDescription()
+ setLPDescription()
+ setRows()
+ setRows()
+ setRows()
+ setlinearProgramSolution()
+ setlinearProgramSolution()
+ useBase64()
+ useCompress()
+ write()
+ ~FMLLPToXML()
FMLCOINMPSToXML
+ FMLCOINMPSToXML()
+ writelinearProgramData()
+ writelinearProgramDescription()
+ writelinearProgramFinish()
+ writelinearProgramSolution()
+ ~FMLCOINMPSToXML()
These classes use the functionality in FMLLPToXML and the functionality in their respective libraries to convert MPS files to LPFML files.
FMLLINDOToXML
+ FMLLINDOToXML()
+ writelinearProgramData()
+ writelinearProgramDescription()
+ writelinearProgramFinish()
+ writelinearProgramSolution()
+ ~FMLLINDOToXML()

the XML file and call the appropriate solver. Then the solution XML file is parsed by `FMLAMPLParser` and the results read back into AMPL for further analysis. This sequence of operations is illustrated in Figure 17.

FMLCOINMPSToXML and FMLLINDOToXML: Finally, in order to provide a clean transition to XML, we have implemented two classes for converting MPS files, as well as files in other formats readable by LINDO, into LPFML files. These classes (and associated utilities) use the COIN class `CoinMpsIO` to parse MPS files, or the functionality of the LINDO API to parse other file types. They then write instances in XML format that validate against the LPFML Schema.

Using these utilities, we implemented a connection between AMPL and any solver that has an OSISolverInterface, as well as LINDO. We used the Xerces C++ XML open source software available from Apache.org. However, any SAX 2.0 compliant parser can be used. Unfortunately, C++ is not as XML friendly as Java and there is not a C++ equivalent of JAXP (Java API for XML Processing) that is parser independent. However, it would be relatively easy to modify our libraries to use another SAX parser.

Figure 17: Using AMPL with a solver



5 Compression

XML is verbose. For instance, it requires start and end tags for all elements. This is the price paid to allow easy conversion and validation. However, verbosity is potentially a problem in the context of data-intensive applications like optimization, especially if large files are sent over a network.

To support arrays, it may seem that every nonzero element would have to be stored using a tag. For example, in LPFML, elements of an array are described by the tag `<el>`. One alternative to this approach is to implement some additional notation to describe arrays. Unfortunately, arrays represented using additional notation would require specialized parsing code, virtually eliminating some of the main advantages of XML, like the ability to easily parse, convert, and validate the content of the files. Instead, our schema has provisions for two compression schemes. The first scheme is based upon the structure of the problem. The second scheme is based upon encoding the text description of the problem in a more efficient manner.

It is important to note that a class that inherits from `FMLParser` class in order to deliver the data to a solver does **not** need to implement the compression scheme used to write the file. The `FMLHandler` class uncompresses the XML file before calling the methods in `FMLParser`. Consider the following example taken from Winston [31].

Example 5.1

$$\begin{array}{lcccccccc}
\min & x_1 & +x_2 & +x_3 & +x_4 & +x_5 & +x_6 & & \\
\textit{s.t.} & x_1 & +x_2 & & & & & \geq & 1 \\
 & x_1 & +x_2 & & & & x_6 & \geq & 1 \\
 & & & x_3 & +x_4 & & & \geq & 1 \\
 & & & x_3 & +x_4 & +x_5 & & \geq & 1 \\
 & & & & +x_4 & +x_5 & +x_6 & \geq & 1 \\
 & & x_2 & & & +x_5 & +x_6 & \geq & 1
\end{array}$$

In Example 5.1 all of the constraint matrix nonzero elements are 1. Also, in every column, at least two nonzero elements appear in consecutive rows. We take advantage of these features by defining two attributes in LPFML Schema for the `<el>` element. The attributes are multiplicity `mult` and increment `incr`. These attributes are used to compress the nonzero elements and row (or column) index elements. First consider the nonzero elements. All of the nonzero elements are 1. We use the multiplicity attribute to record how many consecutive nonzero elements have the same value. In the case of Example 5.1 since all 16 nonzero elements have value 1, the multiplicity is 16 and the nonzero elements for this example are stored as follows.

```
<nonz>
   <el mult="16">1</el>
</nonz>
```

In the row index section we take advantage of the fact that nonzero elements in a column may appear in consecutive rows. Consider variable x_2. This variable has nonzero elements in rows indexed by 0,1, and 5. We store this information by setting `mult="2"` and `incr="1"` for the element with an index value of 0. This tells the parser that the first row index for variable x_2 is 0 and since the multiplicity is 2 with an increment of 1, that there is a second row index with value 1 (we increment 0 by `incr`). The third row index of 5 is stored separately. The row indicies for variable x_4 are stored using only one element with a multiplicity of 3 and increment of 1. The row index storage for Example 5.1 is illustrated below.

```
<rowIdx>
   <el incr="1" mult="2">0</el>
   <el incr="1" mult="2">0</el>
   <el>5</el>
   <el incr="1" mult="2">2</el>
   <el incr="1" mult="3">2</el>
   <el incr="1" mult="3">3</el>
   <el>1</el>
   <el incr="1" mult="2">4</el>
</rowIdx>
```

The compression scheme just described is very effective if there are numerous columns with nonzero elements of equal value in consecutive rows. If this is not the case an alternative compression scheme is available.

An XML file is a text file. For example, in Example 1.1 we store the nonzero element .8333 in the second constraint as:

```
<el>.8333</el>
```

Using UTF-8 encoding (typical of XML files) which requires one byte per ASCII character, this single nonzero element requires 14 bytes. The LPFML Schema also supports the base 64 data type through the `<base64BinaryData>` element. This element may appear as a child element of `<nonz>`, `<pntANonz>`, `<rowIdx>` and `<colIdx>`. The `<base64BinaryData>` has two required attributes, `numericType` and `sizeOf`. The `numericType` attribute value is the data type converted to base 64 (for example double or int). The `sizeOf` attribute value is the number of bytes used in representing each number of the indicated numeric type. The `FMLParser` and `FMLLPToXML` classes support the `<base64BinaryData>` element and can read and write the matrix nonzeros, indicies, and pointers in base 64.

Base 64 binary methods read a file in 6 bit chunks and convert the associated binary number between 0 and 63 into one of the ASCII characters a-z, A-Z, 0-9, and +,/. Each ASCII character then requires one byte of storage. To store a four byte integer using UTF-8 requires nine bytes for the `<el>` and `</el>` start and end tags, plus a byte for each character in the integer. This is a minimum of 10 bytes (for a single digit integer). Storing a four byte integer as base 64 requires at most 6 bytes for a 40% reduction. A similar analysis can be made for storing double precision nonzero elements.

To give the reader some feeling for how well these compression schemes work, we took a bank location set covering linear programming instance from Mairose, Sweeney, and Martin [19]. The problem had 89 rows and 176 variables. The MPS file is 39KB. The XML file with no compression is 50KB. With base 64 compression it is 28KB and using the increment and multiplicity compression it is 29KB. In Table 1 we give results for 10 linear programs from the Netlib library (see `www.netlib.org`). The linear programs selected were the 10 largest in terms of the size of the corresponding MPS file. In Table 1 the first column is the problem name and the second column is the size of the corresponding MPS file. The third column is the size of the file in compressed MPS format. The fourth column is the corresponding problem represented as XML that validates against LPFML Schema but without using any of the compression schemes above. The fifth column is the same problem instance with the data stored in base 64 format. The sixth column is the same problem instance using compression based upon the multiplicity of the nonzero elements. The last column is the size of the original XML instance compressed using an XML specific compression routine implemented in `xmill`. See [30]. (Unfortunately this compressed format is a binary format that is not a W3C standard. This is a potential problem if, for example, the solver is called as a Web service.) The results of this table clearly show that moving from an MPS format to an XML format is not a problem in terms of file size. The XML and MPS specific compression algorithms lead to far more compact files than general compression algorithms (e.g. LZW) hence we do not report sizes for .zip or .gz format.

Table 1: NETLIB Problem Sizes (in KB)

Problem	MPS Format	MPS Comp	XML Format	XML Base64	XML Mult	XML Comp
fit2d	4669	482	6515	2894	4866	121
fit2p	2255	439	3663	1976	2592	126
wood1p	2235	328	3518	1308	2892	74
dfl001	1402	353	2953	1715	2657	178
woodw	1231	240	2494	1230	2466	71
d2q06c	1208	258	2084	981	1877	112
80bau3b	1155	293	1997	1196	1896	111
greenbeb	1057	235	2047	990	1968	73
greenbea	1057	235	2047	990	1969	73
degen3	878	128	1315	589	542	33

6 The Software Distribution

The objective of this research is to propose a standard for representing the instance of a linear program using XML. However, it is clear from previous work, that without an open set of convenient tools for *using* the proposed standard, its adoption would be hindered. Thus, we are distributing a software library in conjunction with this paper. In order to facilitate the use of this library among the researcher and practitioner community, we decided to release the library as open source software.

People disagree about the meaning of "Open Source". For the purposes of this discussion, we define "Open Source" as software whose source code is available without additional charge. Releasing the source code provides the following advantages:

- Increased quality through peer review, frequent updates, and contributions from third parties.
- Transparency. Releasing the code as open source prevents users from having any concerns about functionality being deliberately changed or omitted for business motives, and makes it clear to all parties that their investment in the technology will not be wasted.
- Better documentation, especially from a technical perspective, of the goals and achievements of the project.

In releasing open source software, the license is a key consideration. There are numerous open source software licenses. These licenses differ in the restrictions they place on how the source code and binaries are used. Here are some common licenses, ordered from most restrictive to less restrictive in terms of redistribution requirements. For a thorough discussion of open source software licenses see Fink [9].

- The GPL (GNU General Public License) or *copyleft* license: The GPL is due to Richard Stallman and it is a *quid-pro-quo* license. The key feature of the GPL license is that if you *use* or modify GPL-licensed software, you must distribute the modifications, as well as any software you develop that incorporates GPL-licensed code, under the terms of the GPL. The Linux operating system is a well known example of open source software distributed under a GPL license.
- The LPGL (Lesser or Library GPL) license: As the name implies, this license often (but not always) applies to libraries. With this license you can

write software that *uses* LPGL code and then redistribute an executable that contains your proprietary software and the LPGL code *without having to distribute the source code for your own application*. This is under the assumption that the LPGL-licensed software is not altered, but merely used by your program.

- Non-copyleft licenses: These licenses do not insist that modified and redistributed software also be open source. They typically contain a copyright (in contrast to public domain software which does not carry a copyright) clause and require modified software to retain the clause. Examples of non-copyleft licenses include the Apache software license and the MIT license.

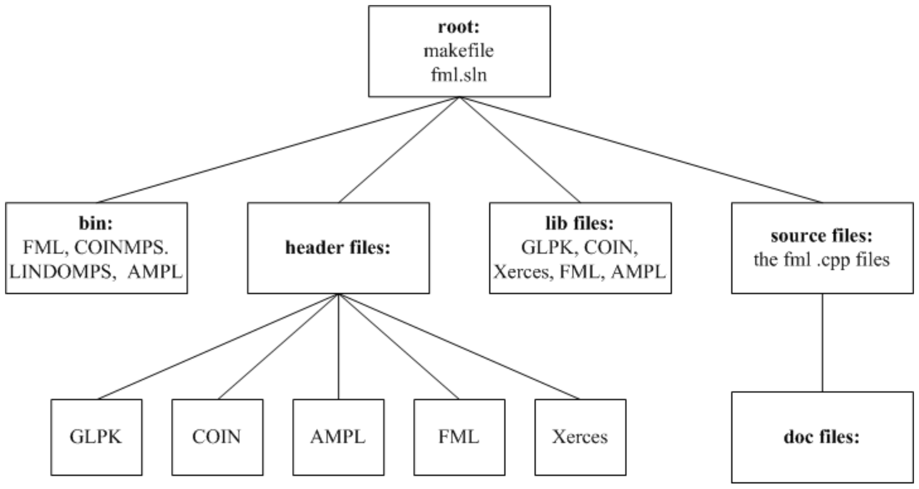
Our main goal with this research is to make available a thorough standard that can be used extensively in a wide variety of scenarios. We therefore chose to distribute our FML library under a non-copyleft license. This allows developers of proprietary software to modify our libraries and then include them in their proprietary software. However, two of our utilities that are not part of the FML library, `nl2fml` and `FMLParse`, link to the GLPK library which is licensed under the GPL. Because we are providing a complete distribution that contains all of the libraries that link with our code, we had to release `nl2fml` and `FMLParse`, under the GPL. (The `nl2fml` and `FMLParse` utilities can use LINDO or any OSI solver. We do this because the user might *wish* to use GLPK.)

The software described in this paper is available for download at `http://gsbkip.uchicago.edu/fml/fml.html`. We currently have distributions for the Windows and Linux operating systems. The file structure of our distribution is illustrated in Figure 18. This is a complete distribution and the user does not need to download any other software. As mentioned in Section 4, we use the Xerces parser in our libraries and include the necessary Xerces library and dll file. Similarly, COIN and Lindo libraries are provided. Makefiles are provided for both the Linux and Windows distribution. We also provide a Microsoft Visual Studio 2003 .NET solution file `fml.sln` for the Windows distribution.

7 Extensions and Future Work

There are a number of possible extensions of this work. First, it is important to develop schemas for more specialized linear programs and nonlinear programs. Currently, we only take advantage of special structure through the `incr` and `mult` attributes. For example, the nonzero element 1 in a set covering problem is stored only once using the `mult` attribute. However, the current schema does not take advantage of other structures such as network flows, variable upper and lower bounds,

Figure 18: Software distribution file structure



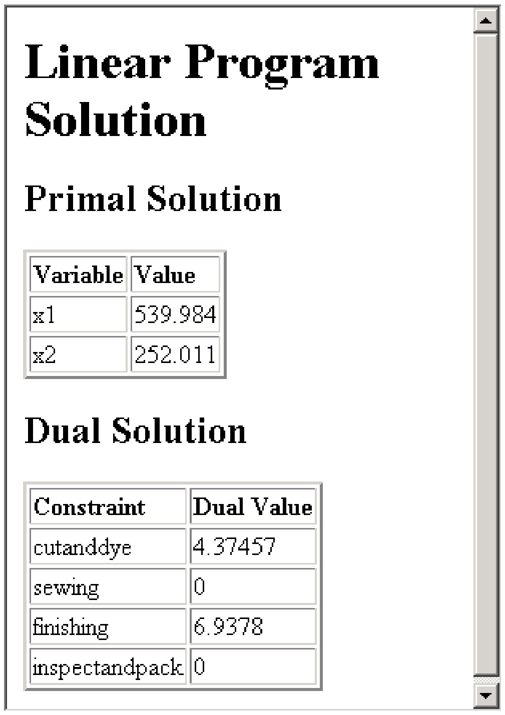
or stochastic programs (see Lopes and Fourer [18]). Representing instances of nonlinear programs is also important. An existing XML dialect that can be used to represent nonlinear terms is Content MathML. See, for example, Sandhu [24]. One direction of research is to develop a schema that uses the MathML namespace, but is specialized for optimization problems.

A natural extension of this work is to enhance the libraries with style sheet processing. As discussed in Section 2, XSLT is a language for transforming XML. We illustrate in Figure 19 a transformation of the solution to Example 1.1 into HTML. This was done using the library class `FMLLPToXML` with the Xalan XSLT processor from apache.org.

Clearly far more elegant transformations are possible. For example, problems might be solved in client server mode. The client machine would have the instance data and send it to the server machine where the FML libraries and the LP solver are located. After solving the problem instance the user at the client machine might query the server for information such as the value of all primal variables that start with an "x", or the dual values for all of the constraints where the dual value is above a threshold number. This could be done using Web service technologies and is facilitated by the use of XML.

Finally, an important feature of XML is that it supports encryption standards such as XML Encryption. With XML Encryption a user can specify which elements to encrypt. For example, if a user wished to encrypt the data in the constraint

Figure 19: Result of Style Sheet Transformation



Linear Program Solution

Primal Solution

Variable	Value
x1	539.984
x2	252.011

Dual Solution

Constraint	Dual Value
cutanddye	4.37457
sewing	0
finishing	6.9378
inspectandpack	0

matrix of a linear program this would be possible by choosing to encrypt all of the child elements of the `<sparseMatrix>` element.

References

[1] AIMMS. The AIMMS modeling language, 2003. `http://www.aimms.com/aimms/product/modeling_language.html`.

[2] D. R. Anderson, D. J. Sweeney, and T. A. Williams. *An Introduction to Management Science*. West Publishing, St. Paul, MN, sixth edition, 1991.

[3] G. Bradley. Introduction to extensible markup language (xml) with operations research examples. *Newletter of the INFORMS Computing Society*, 24:1–20, 2003.

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

[5] T-H. Chang. Modelling and presenting mathematical programs with xml:lp. Masters thesis, University of Canterbury, 2003.

[6] Dash Optimization. Xpress-mosel, 2003. `http://www.dashoptimization.com/pdf/mosel.pdf`.

[7] Dash Optimization. Xpress-optimizer reference manual, 2003. `http://computing.ee.ethz.ch/sepp/xpress-13b-et/optimizer/optimizer.pdf`.

[8] O.C. Ezechukwu and I. Maros. Oof: Open optimization framework. Technical Report ISSN 1469-4174, Imperial College of London, 2003.

[9] M. Fink. *The Business and Economics of Linux and Open Source*. Prentice Hall PTR, Upper Saddle River, PTR, first edition, 2003.

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

[11] IBM. Coin lp, 2003. `http://oss.software.ibm.com/developerworks/opensource/coin/index.html`.

[12] IBM. Optimization subroutine library, 2003. `http://www.research.ibm.com/osl/`.

[13] IBM. Passing your model using mathematical programming system (MPS) format, 2003. `http://www-306.ibm.com/software/data/bi/osl/pubs/Library/featur11.htm`.

[14] ILOG. Ilog cplex, 2003. `http://www.ilog.com/products/cplex/`.

[15] ILOG. Ilog tutorial, 2003. `http://www.ilog.com/products/oplstudio/tutorial/index.cfm`.

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

[17] B. Kristjánsson. Optimization modeling in distributed applications: How new technologies such as xml and soap allow or to provide web-based services, 2001. `http://www.maximal-usa.com/slides/Svna01Max/index.htm`.

[18] Leo Lopes and Bob Fourer. An xml-based format for communicating optimization problems. Presented at INFORMS Annual Meeting, Miami Beach, 2001.

[19] L. Mairose, D. Sweeney, and K. Martin. Strategic planning in bank location. *Proceeding of American Institute of Decision Sciences*, 1979.

[20] Free Software Foundation (Andrew Makhorin). GLPK (gnu linear programming kit), 2003. `http://www.gnu.org/software/glpk/glpk.html`.

[21] K. Martin. A modeling system for mixed integer linear programming using xml technologies. Technical report, University of Chicago, 2002.

[22] Mosek ApS. Mosek, 2003. `http://www.mosek.com/`.

[23] B.A. Murtagh and M.A. Saunders. MINOS 5.4 user's guide. Systems Optimization Laboratory SOL 83-20R, Stanford University, 1983.

[24] Pavi Sandhu. *The MathML handbook*. Charles River Media, 2003.

[25] L. Schrage. *Optimization Modeling with LINDO*. Brooks/Cole, Pacific Grove, CA, fifth edition, 1997.

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

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

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

[29] Neil Soiffer. Mathml: a proposal for representing mathematics in html. *ACM SIGSAM Bulletin*, 31(3):44–45, 1997.

[30] D. Suciu and H. Liefke. Xmill an efficient compressor for xml, 1999. `http://www.research.att.com/sw/tools/xmill/`.

[31] W. Winston. *Operations Research Applications and Algorithms*. Duxbury Press, Belmont, California, third edition, 1994.