INFORMS Journal on Computing
Vol. 17, No. 2, Spring 2005, pp. 139–158
ISSN 1091-9856 | EISSN 1526-5528 | 05 | 1702 | 0139

DOI 10.1287/ijoc.1040.0120
© 2005 INFORMS

LPFML: A W3C XML Schema for Linear and Integer Programming

Robert Fourer
Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, Illinois 60208, USA, 4er@iems.northwestern.edu

Leo Lopes
Department of Systems and Industrial Engineering, University of Arizona, Tucson, Arizona 85721, USA, leo@sie.arizona.edu

Kipp Martin
Graduate School of Business, University of Chicago, Chicago, Illinois 60637, USA, kipp.martin@gsb.uchicago.edu

There are numerous modeling systems for generating linear programs and numerous solvers for optimizing them. However, it is often impossible for modelers to combine their preferred modeling system with their preferred solver. Current modeling systems use their own proprietary model-instance formats that various solvers have been adapted to recognize. The existence of all of these formats suggests that one way to encourage modeling-system and solver compatibility is to use a standard representation of a problem instance. Such a standard must be simple to manipulate and validate, be able to express instance-specific and vendor-specific information, and promote the integration of optimization software with other software.

In this paper we present LPFML, an XML Schema for representing linear-programming (LP) instances. In addition, we provide open-source C++ libraries that simplify the exchange of problem-instance and solution information between modeling systems and solvers. We show how our system is used to enable previously unavailable language-solver connections and how our design improves on the state of the art under three different scenarios relevant to communication between solvers and modeling systems.

Key words: linear programming; information systems; XML
History: Accepted by John W. Chinneck, Area Editor for Modeling: Methods and Analysis; received January 2004; revised September 2004; accepted October 2004.

1. Introduction

There are varied modeling languages for expressing linear-programming models as input to computer systems, and there are many efficient implementations of algorithms for solving linear programs—see Table 1. This proliferation of languages and solvers presents a difficulty for developers of linear-programming software, as modelers may want to use any solver with any modeling language. If there are M modeling languages and N solvers, then $M \times N$ "drivers" are required for complete interoperability, as depicted in Figure 1.

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, symbolic representation of an optimization problem, while an *instance* is an explicit description of a problem's objective and constraints. For linear programming, a model can be described by linear algebraic expressions; 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 the constraint functions.

Using a standard representation of an instance, only $M + N$ software drivers are needed for complete interoperability, as seen in 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 express results returned from a solver as well as problems sent to a solver.

Any representation mechanism for linear-programming instances is readily extended to specify that certain variables take only integer or zero-one values. Thus in the remainder of this paper, every claim we make regarding linear programming extends to mixed-integer linear programming. Our proposed representation also permits communication of solver output options (e.g., quantity of output), branching strategies, and other solver directives.

This paper motivates and describes a standards proposal based on the concept of a markup language

Table 1 **Examples of Modeling Languages and of Solvers**

Modeling languages	Solvers
AIMMS (AIMMS 2003)	CLP (COIN 2003)
AMPL (Fourer et al. 1993)	CPLEX (ILOG 2003a)
GAMS (Brooke et al. 1988)	GLPK (Makhorin 2003)
LINGO (Schrage 2000)	LINDO (Schrage 1997)
Mosel (Dash Optimization 2003a)	MINOS (Murtagh and Saunders 1983)
MPL (Maximal Software 2002)	MOSEK (Mosek ApS 2003)
OPL (ILOG 2003b)	Xpress-MP (Dash Optimization 2003b)

Note. Many other examples are listed in the Linear Programming Frequently Asked Questions, `http://www-unix.mcs.anl.gov/otc/Guide/faq/linear-programming-faq.html`.

and using the technology known as XML. This approach offers substantial advantages over any of the likely alternatives, with few drawbacks. An XML-based approach also offers significant strategic benefits for the mathematical-programming community, as we will explain.

We are not the first to incorporate XML into mathematical modeling. Chang (2003) and Kristjánsson (2001) have also proposed XML representations for linear-program instances. In contrast to these, our proposal provides a broader range of representational options; equally important, we provide open-source libraries to read and write instances expressed in our proposed format, whether from files or directly in memory. Martin (2002) demonstrates how to bypass a traditional algebraic modeling language and use XSLT (a language for transforming XML files) to convert raw data into an XML description of a problem instance such as we propose in this paper. Bradley (2003) provides a good overview of the uses of XML technologies in operations research.

Section 2 describes the problem of representing instances of linear programs and explains why XML is an appropriate technology on which to base a linear-programming standard. Section 3 provides the necessary background material on XML, schemas, and

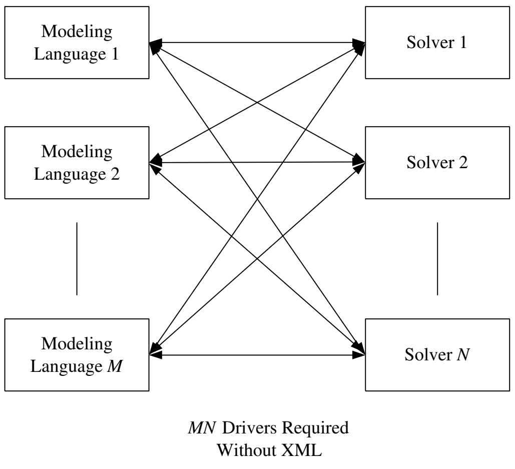


Figure 1 $M \times N$ **Drivers Required Without Common Interface**

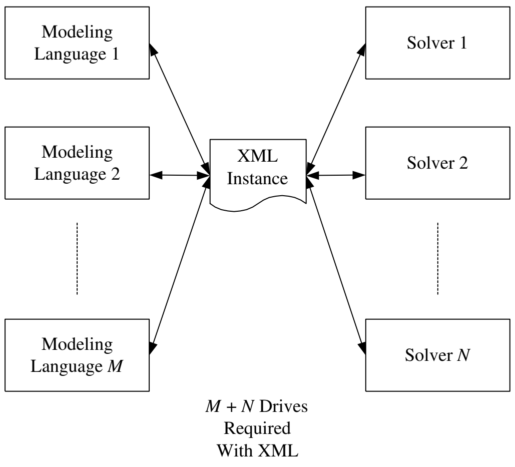


Figure 2 $M + N$ **Drivers Required With Common Interface**

other concepts used in this paper, using some snippets of our representation as examples but assuming no specific knowledge of it. These set the stage for §4, in which we introduce LPFML, the W3C XML Schema that defines our proposed format for representing instances of linear programs.

Section 5 describes open-source libraries that support the LPFML schema. These libraries hide the XML details completely and reduce interacting with LPFML to manipulating vectors and sparse matrices. As an illustration, we use these libraries to make previously unavailable connections between the AMPL modeling language and the LINDO solver and the OSI solver interface. Section 6 deals with issues raised by the large data requirements of LP instances by presenting the results of computational tests. We describe several methods for compressing LP instances, and demonstrate that LPFML is competitive in file size and processing speed.

Section 7 discusses our strategies to prevent fragmentation of the standard, including our choice of software licensing, distribution structure, and official standardization steps. Finally, §8 proposes several extensions and discusses important implications of this work.

2. Instance Representation

An algebraic representation of a simple product-mix model in the AMPL modeling language is given in Figure 3. 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. Figure 4 shows an example of the data (see Anderson et al. 1991) for an instance, in AMPL's format for data.

The AMPL model together with the data in Figure 4 produces a problem instance. AMPL can be directed to display an instance in a somewhat readable text

```
set PROD;  # products
set DEP;   # processing departments

param hours {DEP};     # time available in each department
param rate {DEP,PROD}; # hours used in each dept per product unit made

param profit {PROD};   # profit per unit of each product made
var Make {PROD} >= 0;  # number of units 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];
```

Figure 3 Product-Mix Example in the AMPL Modeling Language

```
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;
```

Figure 5 An Instance of the Product-Mix Example Corresponding to Figure 3's Model and Figure 4's Data

format as shown in Figure 5, and formats similar to this are accepted by a number of solvers. For efficient communication of instances to solvers, AMPL instead uses a more expressive yet more concise proprietary representation, shown in its text form (with optional comments) in Figure 6. Normally people do not need to look at this representation, however, and so by default AMPL uses an even more concise but equivalent binary format. Other modeling systems use their own proprietary representations for the same purpose.

The work presented in this paper is concerned with describing instances, not models. Instance descriptions are very different from model descriptions, as can be seen by comparing Figure 3 to Figure 6.

Figure 3's representation of the product-mix model is *symbolic, general, concise*, and *understandable* (Fourer 1983). An LP model contains only a few (commonly less than 50) declarations of high-level algebraic components such as sets, parameters, variables, objectives, and constraints. Thus the central problem in representing models is handling the diverse algebraic modeling concepts available to the user. Ezechukwu and Maros (2003) describe an Algebraic Markup Language that uses XML to describe the model rather than the instance. In fact, many of the necessary modeling constructs are already present in the MathML (Soiffer 1997) vocabulary.

```
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  ;
```

Figure 4 Data for the Product-Mix Example in AMPL's Data Format

Figure 6's representation of the product-mix instance is, by contrast, *explicit, specific, verbose*, and *convenient* for the solver rather than for the human modeler. A typical LP instance might include millions of entries, each corresponding to, for instance, a particular linear coefficient. Thus, the central problem in representing instances is managing these entries, ensuring that they are handled correctly and efficiently, and taking advantage of low-level regularities such as block structures and repetitions of coefficient values.

An XML modeling vocabulary could take its place alongside the other modeling languages in Figures 1 and 2, but 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 contexts in which the designs are to be applied are very different. Modeling languages and systems are intended to be used by people and so have a variety of designs reflecting different personal preferences and applications. Forms for communication of instances between modeling systems, which are largely hidden from people, are much more readily standardized. Thus a standard for instances has much better prospects for being widely adopted.

2.1. Existing Instance-Representation Standards

The one widely used standard for representing problem instances in terms of coefficients and bounds is the so-called MPS form (IBM 2003). An MPS-form representation for our product-mix problem instance is shown in Figure 7.

The MPS format is needlessly verbose, as can be seen in the repetitions of the column names and the right-hand side name (RHS1). It is moreover not quite standard, as different implementations have adopted

```
g3 2 1 0 # problem prodmix
 2 4 1 0 0  # vars, constraints, objectives, ranges, eqns
 0 0  # nonlinear constraints, objectives
 0 0  # network constraints: nonlinear, linear
 0 0 0   # nonlinear vars in constraints, objectives, both
 0 0 0 1 # linear network variables; functions; arith, flags
 0 0 0 0 0  # discrete variables: binary, integer, nonlinear (b,c,o)
 8 2  # nonzeros in Jacobian, gradients
 0 0  # max name lengths: constraints, variables
 0 0 0 0 0  # common exprs: b,c,o,c1,o1
C0 #HoursAvailable['cutanddye']
n0
C1 #HoursAvailable['sewing']
n0
C2 #HoursAvailable['finishing']
n0
C3 #HoursAvailable['inspectandpack']
n0
O0 1  #TotalProfit
n0
r  #4 ranges (rhs's)
1 630
1 600
1 708
1 135
b  #2 bounds (on variables)
2 0
2 0
k1 #intermediate Jacobian column lengths
4
J0 2
0 0.7
1 1
J1 2
0 0.5
1 0.8333
J2 2
0 1
1 0.6667
J3 2
0 0.1
1 0.25
G0 2
0 10
1 9
```

Figure 6 Another Representation of the Problem Instance Shown in Figure 5 but in the Proprietary Format that AMPL Uses to Communicate Instances to Solvers

different defaults for values such as the bounds on integer variables and the lower bound on a variable whose upper bound is specified as zero. Various modeling systems and solvers have also introduced competing alternatives for a free-format version that does not force information to appear in certain fields within each line. Occasional proposals for major extensions, such as for nonlinear expressions, have failed to catch on. MPS forms also do not make useful general provision for conveying solver-specific directives, such as branching preferences for MIP solvers; an MPS file does not even specify whether the objective (`TPROFIT` in Figure 7) is to be minimized or maximized.

```
NAME          PRODMIX
ROWS
 N  TPROFIT
 L  HRSCUT
 L  HRSSEW
 L  HRSFIN
 L  HRSINS
COLUMNS
    MAKESTD   TPROFIT   10
    MAKESTD   HRSCUT    0.7          HRSSEW   0.5
    MAKESTD   HRSFIN    1            HRSINS   0.1
    MAKEDEL   TPROFIT   9
    MAKEDEL   HRSCUT    1            HRSSEW   0.8333
    MAKEDEL   HRSFIN    0.6667       HRSINS   0.25
RHS
    RHS1      HRSCUT    630
    RHS1      HRSSEW    600
    RHS1      HRSFIN    708
    RHS1      HRSINS    135
ENDATA
```

Figure 7 Another Instance of the Product-Mix Example in the MPS Standard Format

Current modeling systems instead use their own proprietary instance formats, such as the AMPL format in Figure 6. Each solver has to be interfaced to recognize each instance format. The existence of all of these proprietary forms confirms that MPS form, even in extended guises, has not proved to be all that modeling-language implementers need. Current uses of MPS form in large-scale optimization are confined almost exclusively to maintaining test-problem libraries and to representing instances that accompany bug reports to solver vendors. A companion form for returning results from solvers has mostly fallen into disuse.

2.2. The Case for XML

An XML vocabulary is formally defined by an XML *schema* against which every file written in the vocabulary can be automatically validated. This arrangement gives an XML vocabulary several important advantages over MPS and other proprietary formats:

- Validation against a schema promotes stability of the standard.
- The schema can restrict data values to appropriate types—row names to `string`, row indices to `int`, and coefficient values to `double`, for instance.
- The schema can define *key* data to insure, for example, that no row or column name is used more than once.
- The schema can be extended to include, for example, new constraint types or solver directives. Files that validated under the original schema continue to validate under the extended one (though, of course, the reverse is not guaranteed).

XML schemas and validation are explained further in §3.

XML is increasingly being adopted as a standard for the interchange of information in diverse fields of science (O'Reilly 1999) and operations research (Bradley 2003). This broader relevance has benefits for optimization systems:

• When instances are stored in XML format, optimization technology solutions are more readily integrated into broader information technology infrastructures.

• XML is the data interchange language of Web services. As solvers become available in the form of Web services, they will require XML representations of problem instances. Fourer et al. (2004) propose a comprehensive framework for XML-based optimization services extending the functions of the popular NEOS Server for Optimization (Moré et al. 2004).

• XML lends itself very well to compression. In §6 we describe compressed XML representations of linear programs.

• 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 is easily applied to the instance to produce a Web-browser document that displays the linear-program data or solution data in reports that are suitable for people to read.

• Encryption standards such as XML Encryption are emerging for XML data—see `http://lists.w3.org/Archives/Public/xml-encryption/`. This option is important to commercial linear-programming applications where the problem instances contain confidential data.

Libraries for these purposes—validating files, defining keys, compressing files, and the like—are provided by numerous XML tools designed for manipulating and parsing XML data. It suffices to define our XML vocabulary in the form of a schema with which these tools can work. This contrasts to ad hoc formats that require writing, debugging, and maintenance of routines equivalent to these tools.

3. Basic XML Technologies

In this section we give a brief overview of the XML technologies used in this paper. See also the excellent overview by Skonnard and Gudgin (2002).

An XML file is a text file that contains both data and markup. Consider the text in Figure 8, describing the rows of a linear program. This text contains both data, such as a row upper bound of 630 and a row name `cutanddye`, and markup, or metadata, in the form of *elements* and *attributes* that describe or give meaning to the data. Elements that contain other ("child") elements are defined by an opening `<tag>` and closing `</tag>`, like `<rows>` in the example. Other elements are defined by a single construct

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

Figure 8 An XML Representation of Row (Constraint) Data

of the form `<tag.../>`, like the example's element `<row>`. Attributes, such as `rowName` and `rowUB` in `<row>`, are used to define or characterize elements. In this respect, the `<row>` elements correspond to records in a relational database and the attributes correspond to fields. However, unlike a relational database, the XML structure is tree-like or hierarchical and not restricted to a two dimensional table structure.

In the XML representation of the row data illustrated in Figure 8, 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 construct an appropriate tree from an XML document, the document must be *well formed*. An XML document is well formed if

• both opening and closing tags are present, or a single `<tag.../>` is used with no child elements,

• the opening and closing tag names exactly match both in name and case, and

• the tags are nested properly, with the closing tag of a child element preceding 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 have used the Xerces parser from The Apache Software Foundation (`www.apache.org`).

Well-formedness 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 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 instances of the classes.

A powerful feature of the XML Schema is that it allows for both built-in and user-defined "types." We illustrate this concept using the schema associated with Figure 8. The schema in Figure 9 describes the `<rows>` element used in Figure 8. (This is part of our LPFML Schema proposal, described further in §4.) The `<complexType>` in Figure 9 is a user-defined *anonymous complex type* and can contain other elements,

```
<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>
```

Figure 9 The `rows` Element of a Schema for Figure 8's Data

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.

To support sparse storage of the constraint matrix we require integer vectors. They are used both to hold pointers and to store row or column indices. The schema definition of the intVector type from our LPFML Schema proposal is shown in Figure 10. It is an example of a *named complex type.* The type intVector contains a <choice> between two possibilities: exactly one element of type <base64BinaryData>, and anywhere from 0 to an unbounded number of elements of type <el>. The intVector is an example of a user-defined type that is used in the definition of other types but that does not get instantiated as an element in an XML file. It is not valid to have an <intVector> tag in an LPFML file. However, we can define a tag such as <colIdx> that is of type <intVector>.

The element <el> is also a user-defined type. It contains text that is of built-in type int. This implies that the text contained in the <el> element of an intVector

```
<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>
```

Figure 10 The `intVector` Element of the LPFML Schema Proposal

```
<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>
```

Figure 11 The `colType` Element of the LPFML Schema Proposal

type must be parsed as integer data. A validating parser should give an error message if noninteger data are encountered. Checking for data of an incorrect type is a desirable feature when validating the instance of a linear program. The <el> element also has two attributes, mult and incr. These attributes are used to take advantage of structure in linear constraint coefficients; their use is described in §4.

The built-in data types like int and string are supplemented by a <simpleType> element specification that allows a schema to define the type of text that can make up an element or an attribute. Consider the example in Figure 11, where we are defining an attribute type we call colType. This attribute must be a string consisting of a single character that is either C if the corresponding column element represents a continuous variable, B if the corresponding 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 tag and data type in our schema examples is qualified with an xs, as in xs:complexType and xs:int. This qualification is saying that the tag or data type 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"
```

that 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* that 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 in C++ or Java. 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 human analysts through a web browser. For example, with XSLT it is easy to

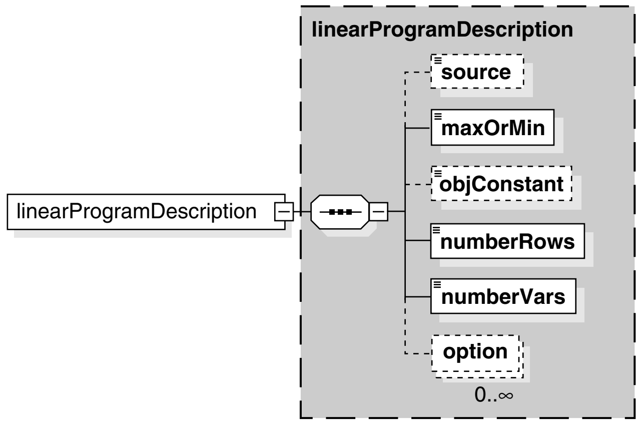


Figure 12 The Form of the Linear-Program Description Element

read the solution to a linear program from an XML file, select the variables of interest along with their solution values, and display the results in an HTML table. For an excellent treatment of XSLT programming see Kay (2004).

4. LPFML Schema

The LPFML Schema describes an LP instance in up to four parts. The first contains information about the instance, such as dimensions and objective sense. This is the only mandatory section. The second part contains the actual vectors and matrices that define the instance. The third contains the solution information. Finally, the fourth section provides constructs for dynamically adding new columns and rows to a problem.

4.1. Linear-Program Description

The `<linearProgramDescription>` element is used to convey the basic properties of the linear-program instance. Its general form is depicted in Figure 12 and shown for the product-mix example in Figure 13. 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, it can be used to communicate output preferences (such as the frequency with which intermediary results are produced) or parameters useful to the solution method (such as choice of pricing or branching strategy).

There may be more than one `<option>` element, and each may specify a `solver` attribute. This attribute indicates solver-specific options. For example, the element

```
<option solver="lindo" outlev="3">
```

specifies that the option `outlev=3` is to be used when the instance is sent to the solver `lindo`. A solver may issue a warning if it doesn't find an `<option>` for itself, especially if it finds `<option>` tags for other solvers. If it does find an `<option>` tag for itself, it should attempt to parse the contents. If it finds a discrepancy, it should raise an error condition.

If an `<option>` element does not contain a `solver` attribute, then any solver is free to try to parse the element's contents. If a solver cannot interpret an `<option>` element lacking a `solver` attribute, it should ignore the element's contents. The solver may issue a warning, but should not raise an error condition.

```
<linearProgramDescription>
   <source>Par Inc. Problem from Anderson, Sweeney,
      and Williams </source>
   <maxOrMin>max</maxOrMin>
   <numberRows>4</numberRows>
   <numberVars>2</numberVars>
   <option solver="lindo" outlev=3>
</linearProgramDescription>
```

Figure 13 The Linear-Program Description Element for the Product-Mix Example

4.2. Linear-Program Data

The data that comprise the linear-program instance are contained in the element `<linearProgramData>`, diagrammed in Figure 14. This element has four children: `<rows>`, `<columns>`, `<aMatrix>`, and `<metaData>`.

The `<rows>` element contains an unbounded number of `<row>` children, one for each row (constraint) in the problem instance. Thus the `<rows>` element for the product-mix example is

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

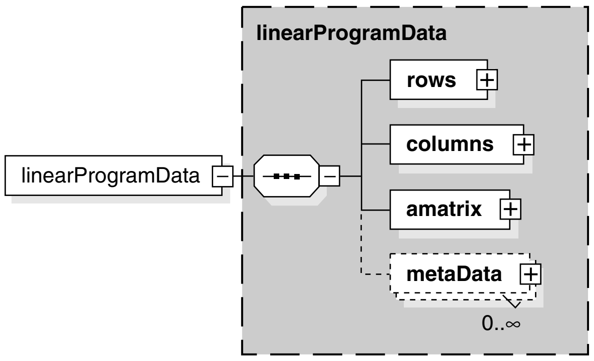


Figure 14 The Form of the Linear-Program Data Element

Each `<row>` has four optional attributes: `rowName`, `rowUB`, `rowLB`, and `mult`. The `mult` attribute is used for compression and is described shortly. If row names are not provided, the rows are uniquely identified by numerical indices assigned to them based on their order in the file.

The `<columns>` element contains an unbounded number of `<col>` children, one for each column (variable) in the product instance. The product-mix example's `<columns>` element is:

```
<columns>
   <col objVal="10" colName="Make['std']"
     colType="C" colLB="0.0"/>
   <col objVal="9" colName="Make['del']"
     colType="C" colLB="0.0"/>
</columns>
```

Each `<col>` has six optional attributes: `colName`, `colUB`, `colLB`, `objVal`, `colType`, and `mult`. The `objVal` attribute is the objective-function coefficient, zero by default. The `colType` attribute has three possible values, `C` for continuous (the default), `B` for binary, and `I` for general integer. As with the rows, the `mult` attribute is used for compression, and if column names are not provided then the columns are uniquely identified by indices assigned to them based on their order in the file.

The `<aMatrix>` element has a child element for each matrix-storage scheme. We implement only the `<sparseMatrix>` scheme, described in detail below. In the future it will be possible to incorporate other schemes, such as the Matrix Market or Harwell-Boeing Exchange Formats (`http://math.nist.gov/MatrixMarket/formats.html`), by including them in the LPFML schema as additional children of `<aMatrix>` as illustrated in Figure 14. For a detailed treatment of sparse matrix storage schemes, see Duff et al. (1986).

The `<metaData>` tag may appear in several sections of the file. It is an 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.

4.3. Sparse-Matrix Storage

The use of XML allows us considerable flexibility in representing sparse matrices. This flexibility is welcome because, depending on how instances are generated and transported, different representations are most appropriate. The resulting files differ in terms of parsing speed, uncompressed size, compression yields, manipulation flexibility, and portability. Different usage scenarios are discussed in §6. Our library (discussed in §5) makes the distinctions between different representations transparent to modelers.

We have initially implemented LPFML to use compressed row or column storage to represent sparse matrices. Compressed column storage uses three vectors: the nonzero entries of the matrix (`nonz`), the row indices of the nonzero entries (`rowIdx`), and the starting points of the columns within the other two vectors (`pntANonz`). We use zero-based counting, so that entry i of `pntANonz` indicates the start of the column $i+1$ entries in `nonz` and `rowIdx`. As an example, in a representation of the matrix

$$\begin{bmatrix} 0 & -3 \\ 1 & 4 \\ -2 & 0 \end{bmatrix},$$

`nonz` is $(1,-2,-3,4)$, `rowIdx` is $(1,2,0,1)$, and `pntANonz` is $(2,4)$. Compressed row storage is analogous but with the nonzero elements arranged rowwise in `nonz`, the column indices of the elements in `colIdx`, and the starting points of rows in `pntANonz`.

The sparse-matrix element of the LPFML schema is diagrammed in Figure 15. The compressed storage vectors are specified in the elements `<nonz>`, `<colIdx>` or `<rowIdx>`, and `<pntANonz>`. An optional element `<numNonz>` is provided for situations in which 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). Inserting a new row into a compressed column-storage matrix or a new column into a compressed row-storage matrix can be expensive. Thus, in a practical sparse representation it is often important to leave extra room for nonzero elements. To support this, LPFML sparse matrices provide for an extra `numNonz` vector that gives the number of places that should be reserved for nonzero elements in each row or column, regardless of how many nonzeros are present initially.

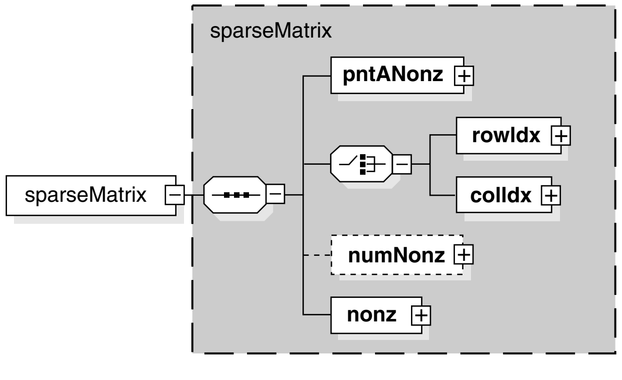


Figure 15 **Sparse Matrix Storage**

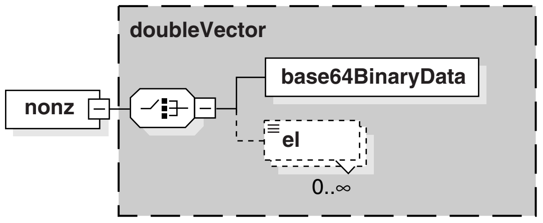


Figure 16 **Representation of a Vector**

LPFML offers two options for representing vector elements, as depicted for the case of `<nonz>` in Figure 16. In one case, each vector is encoded as a single string and enclosed in a `<base64BinaryData>` tag. In the other case, each *element* of each vector is stored individually in an `<el>` tag.

For the binary case, we use *base64 encoding* to convert the vector from its binary representation in the computer to a text representation (a requirement of XML files). This encoding converts each six bit chunk of the binary representation to one of the 64 characters `a–z`, `A–Z`, `0–9`, `+`, `/` (as explained in §6.8 of `http://www.ietf.org/rfc/rfc2045.txt`). We refer to LPFML instances using the base64 data type as *b64* LPFML. This is the default storage scheme in our libraries. The base64 representation for the constraint matrix of our product mix example is shown in Figure 17.

Compared to representations that store each nonzero of a vector separately, the base64 representation can be parsed faster and typically requires less storage. These performance benefits are attained by using fewer tags and by storing floating-point numbers exactly without incurring the extra overhead of techniques such as those described in Gay (1990) for converting binary numbers to and from their human-readable string representations.

```
<sparseMatrix>
   <pntANonz>
      <base64BinaryData numericType="int" sizeOf="4">
        BAAAAAgAAAA=
      </base64BinaryData>
   </pntANonz>
   <rowIdx>
      <base64BinaryData numericType="int" sizeOf="4">
        AAAAAAEAAAACAAAAAwAAAAAAAAABAAAAAgAAAAMAAAA=
      </base64BinaryData>
   </rowIdx>
   <nonz>
      <base64BinaryData numericType="double" sizeOf="8">
        ZmZmZmZm5j8AAAAAAADgPwAAAAAAAPA/mpmZmZmZuT8AAAAAA
        ADwP7U3+MJkquo/S8gHPZtV5T8AAAAAAADQPw==
      </base64BinaryData>
   </nonz>
</sparseMatrix>
```

Figure 17 **Base64 Representation of the Constraint Matrix for the Product-Mix Example**

```
<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>
```

Figure 18 **Element-by-Element Representation of the Constraint Matrix for the Product-Mix Example**

When it is desirable to access individual elements of each vector of the compressed column or row storage matrix, LPFML provides the ability to do so using a sequence of `<el>` elements. This representation for the product-mix example is illustrated in Figure 18.

Most linear programs exhibit characteristic structures in their nonzero values, especially in the constraint matrix. LPFML takes extensive advantage of special structure in several ways. Consider the following set-covering example (Winston 1994):

$$\begin{array}{ll}
\text{Minimize} & x_1+x_2+x_3+x_4+x_5+x_6 \\
\text{Subject to} & x_1+x_2 \qquad\qquad\qquad\quad \geq 1 \\
 & x_1+x_2 \qquad\qquad\quad +x_6 \geq 1 \\
 & \qquad\quad x_3+x_4 \qquad\qquad \geq 1 \\
 & \qquad\quad x_3+x_4+x_5 \qquad \geq 1 \\
 & \qquad\qquad\quad +x_4+x_5+x_6 \geq 1 \\
 & \quad x_2 \qquad\qquad +x_5+x_6 \geq 1.
\end{array}$$

Here 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 use of two attributes in LPFML Schema for the `<el>` element: `mult` (multiplicity) and `incr` (increment).

First consider the `<nonz>` element for the constraint matrix in this example. We use the multiplicity attribute to record how many consecutive nonzero elements have the same value. In this case all 16 nonzero constraint elements have value 1, the multiplicity is 16, and the constraint matrix is simply as follows:

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

In the `<rowIdx>` element we take advantage of the fact that nonzero elements in a column often appear in consecutive rows. Consider variables x_4 and x_5, that have nonzeros in rows 2–4 and 3–5, respectively. We store their row indices by setting `mult="3"` and `incr="1"` for the element containing their first row index:

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

This tells the parser that, for example, if the first row index for variable x_4 is 2, then the next two indices are $2+1$ and $2+2$ respectively. For efficiency, we do not use `mult` and `incr` together in this way unless `mult` is at least 3.

We refer to LPFML instances using the `mult` and `incr` attributes as *structural* LPFML. This representation achieves very high compression ratios, is easy to generate and manipulate using a wide variety of tools, and contains no computer-architecture dependencies.

4.4. Solution-Instance Representation

The `<linearProgramSolution>` element is used to store the solution to the linear program. It has six child elements, as seen in Figure 19. An instance associated with the product-mix example is shown in Figure 20.

The `<primalSolution>` and `<dualSolution>` elements contain a child element `<sol>` for each nonzero value in the primal and dual solution, respectively. Each `<sol>` element has attributes `idx` and `val` giving the index of a primal or dual variable and its corresponding value; an associated `name` attribute is optional.

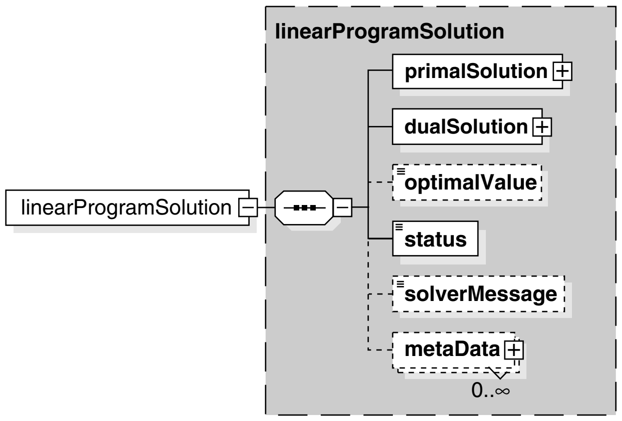


Figure 19 Form of the Linear-Program Solution Element

```
<linearProgramSolution>
  <primalSolution>
    <sol idx="1" name="Make['std']" val="540"/>
    <sol idx="2" name="Make['del']" val="252"/>
  </primalSolution>
  <dualSolution>
    <sol idx="1" name="HoursAvailable['cutanddye']" val="4.37457"/>
    <sol idx="3" name="HoursAvailable['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>
```

Figure 20 The Linear-Program Solution Element for the Product-Mix Example

The `<status>` element indicates whether the specified `<optimalValue>` is a successful optimum or whether the algorithm encountered some other stopping condition such as unbounded or infeasible. The solver can return any additional solution-related messages in `<solverMessage>`.

The `<metaData>` for a solution works in the same way as described earlier for a problem instance. It is used to specify information that a solver associates with the solution values, such as the components of a direction of unboundedness or membership in an irreducible infeasible subset when no feasible solution exists.

5. The LPFML Library

A major contribution of this work is a set of open-source libraries for reading and writing LP instances in XML format. We discuss in this section the innovations introduced by the library and the benefits provided. Appendix 8 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.

The LPFML library is designed to serve three main purposes: allowing LPFML to be used immediately, hiding all of the parsing and writing code, and promoting stability of the standard by providing a benchmark interface implementation. By hiding the parsing and writing code, the library offers three significant benefits:

- Solver and modeling-language authors deal only with familiar mathematical concepts, such as objectives, constraints, and vectors.
- Efficiencies can be designed into the format without increasing the complexity to users.
- Changes and extensions can be implemented without requiring any solver or modeling-language code to be rewritten.

Authors thus need only understand LP-specific concepts when writing or parsing an instance using the library. Vectors are passed to and from the library and library routines take care of encoding or decoding using any of the representations discussed earlier. The LPFML library's parsing services are described in §5.1, and its writing services in §5.2.

5.1. Parsing

The key service provided by the library is to parse an XML file, convert the data into a format convenient for LP, and present the data to the solver without burdening it with any parsing-related tasks. All that solver authors need to do is move the data from the library's memory space into the solver's memory space. This is done in the FMLParser class, the only class with which authors need to interact for parsing.

FMLParser contains several methods, typified by onObjectiveSense. This method is *called by the library* 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, and in which solver-specific initialization can be accomplished. For example, here is the implementation of FMLParser::onObjectiveSense for the OSI solver interface:

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

Here solver_ is a pointer to an OSI-specific interface class, that has a method ::setObjSense. The person implementing this parser needs to know the solver's interface library (OSI in this case) *but does 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 library invokes the method with regular C++ vectors or constants as parameters, *after* it has done the relevant XML parsing. By implementing the parser as an event-driven library of this kind, we achieve some very important benefits:

- We reduce to a minimum the number of library-specific concepts an author needs to learn.
- We completely separate the mathematical aspects of loading an LP into a solver from the file or network-management aspects.
- We avoid having to search the file at any point. The file is read sequentially.
- 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 a certain event is called, the parser's representation of the corresponding objects can be deleted (or ownership of the data can be assumed by the solver).

In addition to the event-driven methods, FMLParser contains a solve method that must be specialized to invoke the appropriate solver and a few other utility methods that should not need to be overridden.

5.2. Writing

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

Advanced features provided by the LPFML proposal, such as structural compression and base64 encoding, are 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, ::setRows, ::setColumns, and ::setLPDescription. Each such method has at least two signatures, one with C-style representations of arrays (using pointers), and one with C++-style representations of arrays (using the C++ Standard Template Library). More signatures are available if they provide some convenient functionality. As an 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. The third also dispenses with the use of row names.

6. Communicating Instances

LPFML is intended for use under three distinct scenarios: tightly-coupled environments, loosely-coupled environments, and pre- and post-processing environments. Requirements of one environment sometimes conflict with those of another. As a result, we have specifically designed the LPFML instance representation, and the associated libraries, to be as flexible as possible. This results in significant savings in modelers' time and facilitates incorporation of innovative features not previously available in mathematical-programming systems.

In this section we demonstrate that under each scenario the LPFML XML representation is competitive with the MPS standard form and AMPL's proprietary `nl` form, the two representations for problem instances described in §2. It is seen to be clearly superior to MPS form in several respects.

All the LPFML files used to produce the results in this section are stored using *canonical* XML. This is the way XML is typically handled in machine-to-machine communication. All tags and their content remain unchanged, but all superfluous whitespace, typically used (as in this paper) to improve human readability, is removed.

Our test problems are basically the 15 largest (in terms of nonzeros) from the netlib Kennington subset (`http://www.netlib.org/lp/data/kennington`) and the miplib collection (`http://miplib.zib.de`). We have dropped a few problems that are similar to larger problems, however, so that our test set includes only the largest problem from each of the four problem types in the netlib Kennington subset, and 11 from miplib. The netlib and miplib collections use MPS form; we have made equivalent LPFML and `nl` form representations for comparison.

Characteristics of these problems are shown in Table 2. This and subsequent tables are sorted by increasing numbers of nonzero constraint coefficients.

6.1. Tightly Coupled Environments

In the context of our research, a *tightly-coupled* system has only two components: a modeling system and a solver that are implemented specifically for use with optimization. They communicate directly with each other, run on the same machine under the same operating system, and have access to shared memory or disk space. They may be implemented as a single process (for example, LINGO) or as two different processes (for example, AMPL with CPLEX).

Table 2 Test Problems Used for Results Reported in this Section

Problem	Rows	Columns	Nonzeros
mzzv42z	10,460	11,717	151,261
nsrand-ipx	735	6,621	223,261
atlanta-ip	21,732	48,738	257,532
sp97ar	1,761	14,101	290,968
pds-20	33,875	105,728	304,153
t1717	551	73,885	325,689
cre-b	9,649	72,447	328,542
fast0507	507	63,009	409,349
ken-18	105,128	154,699	512,719
nw04	36	87,482	636,666
stp3d	159,488	204,880	662,128
rd-rplusc-21	125,899	622	852,384
momentum3	56,822	13,532	949,495
ds	656	67,732	1,024,059
osa-60	10,281	232,966	1,630,758

Table 3 MPS vs. LPFML Formats: Tightly-Coupled Case

	Parse times (ratios of MPS sec)			
Problem	CLP MPS	Specialized LPFML	Xerces from file LPFML	Xerces in memory LPFML
mzzv42z	0.187	2.4	0.6	0.7
nsrand-ipx	0.187	3.0	0.6	0.7
atlanta-ip	0.375	1.6	0.5	0.5
sp97ar	0.656	6.0	1.5	1.7
pds-20	0.656	2.3	0.6	0.6
t1717	1.062	4.2	1.2	1.2
cre-b	0.375	2.0	0.5	0.5
fast0507	0.500	2.1	0.6	0.6
ken-18	1.062	1.7	0.5	0.5
nw04	0.734	2.0	0.6	0.6
stp3d	1.437	2.0	0.5	0.5
rd-rplusc-21	1.312	2.9	0.8	0.9
momentum3	0.984	3.3	1.0	0.8
ds	0.968	2.1	0.6	0.7
osa-60	1.703	1.8	0.6	0.6

Note. All timings are from a Dell Precision 650 workstation with a 3.06 GHz Xeon processor and two Gbytes of RAM running under Windows XP Professional.

In this scenario, the modeling system and the solver together offer all the services of interest, such as preprocessing, visualization, data acquisition, and display of solution information.

In a tightly-coupled environment, performance tends to be more important than flexibility. Performance of an instance representation is measured primarily by the time needed to transfer an instance between a modeling language and a solver. Each may produce a file that is read by the other, or the transfer may be made in memory. The in-memory alternative may be faster and, more importantly, may yield more robust designs that need not address a variety of errors associated with file handling.

Four representative parse times for each test problem are compared in Table 3. The second column shows times for the open-source `readMps()` parser (`http://www.coin-or.org/Doxygen/Clp/functions.html`) applied to standard MPS form. The remaining columns show the *ratios* of the MPS times to parse times for various LPFML options.

For tightly-coupled situations in which the reader can trust the writer to generate LPFML files correctly, we have created a very efficient specialized parser that does a minimum of checking. An examination of the third column of Table 3 shows this specialized parser to be uniformly more efficient than the MPS parser. The LPFML parser's advantage is over 1.5 in all cases, and usually 2 to 3, though with no clear trend as problem size increases.

For other situations—such as the development of a new solver or modeling language—it may be better to use a parser based on a solid XML library that provides some automatic testing without any effort on the part of the developer. The fourth and fifth columns of Table 3 reflect timings of the Xerces XML parser, with testing for well-formed input turned on but validation against a schema off; they differ only in that one is based on reading from a file and the other on reading from an in-memory buffer. There is no significant difference between the file-based and memory-based results. Either runs slower than the MPS parser in most cases, and 3 to 4 times slower than our specialized parser (again with no trend as problem size increases). Considering that all of the Figure 3 times are only a small fraction of total solving times, particularly for the integer programs, the extra cost of using the Xerces parser is unlikely to make any practical difference.

6.2. Loosely Coupled Environments

In the context of our research, *loosely coupled* environments are those where solver and modeling system reside on different machines. Examples arise when a user reports a bug to a solver developer or employs a Web service such as NEOS (Dolan et al. 2002). The primary performance measure for exchanging information between the modeling system and the solver in this environment is file size, especially when compressed.

Table 4 compares the file sizes of seven different representations of our test problems. The second column shows the size required by uncompressed MPS form, and subsequent columns show *ratios* of the MPS size to other representations' file sizes.

The third and fourth columns show results for uncompressed structural LPFML and base64 LPFML format, respectively. Both are seen to be more concise than MPS typically by a factor between 1 and 2. The base64 alternative is more uniformly advantageous, however; the structural alternative's advantage is more varied because it depends on the degree to which LPFML's `mult` and `incr` attributes are applicable. Overall, neither LPFML alternative (base64 or structural) dominates the other, and in every case except two (`pds-20, stp3d`) both alternatives dominate MPS storage.

The fifth column gives results for AMPL's proprietary `nl` format, again uncompressed. This form is seen to be more concise than MPS format typically by a factor of 2 to 4, making it about twice as concise as LPFML on the whole. The smaller size of the `nl` files is not surprising since `nl` form was designed primarily for conciseness. What's more important is that we can achieve the advantages of XML described in §2.2 at the expense of only about a doubling in file size.

It is shown by Suciu and Liefke (1999) that when file formats are encoded as XML their compression yields improve. Confirming this observation, the sixth and seventh columns of Table 4 represent the sizes of the MPS and LPFML XML files after compression by gzip (`http://www.gzip.org`). Compression works well for both formats, but overall it makes LPFML's advantage even greater. (We did not test compressed base64 LPFML, as the base64 encoding lacks the patterns sought by most compression schemes and requires that the encoding and decoding be done on computers that represent numbers in the same way—not always a valid assumption in a loosely-coupled environment.)

Other compression schemes can be even more effective. The last column of Table 4 shows further reductions in files size after compression using bzip2 (`http://sources.redhat.com/bzip2/`), which reorders the file before searching for patterns and tends to do well on XML files. Using an XML-specific compression tool such as xmill (`http://www.research.att.com/sw/tools/xmill/`) can further improve the results, but that technology is not as widely available.

Table 4 **MPS vs. LPFML Formats: Loosely-Coupled Case**

	File sizes (ratios of MPS Mbytes)						
	Uncompressed				gzip		bzip2
Problem	MPS	struct LPFML	base64 LPFML	AMPL `nl`	MPS	struct LPFML	struct LPFML
mzzv42z	5,352	2.1	1.5	3.7	10.5	20.6	30.4
nsrand-ipx	7,060	3.2	1.8	4.1	10.7	103.8	220.6
atlanta-ip	11,288	1.2	1.4	2.6	7.9	14.7	19.7
sp97ar	9,372	1.4	1.7	3.7	10.7	31.2	49.9
pds-20	11,548	0.9	1.0	2.5	8.7	12.2	17.5
t1717	15,016	1.7	1.5	3.5	8.9	20.3	27.8
cre-b	10,436	1.4	1.2	2.8	13.8	19.8	26.4
fast0507	16,828	1.9	1.6	3.7	13.4	31.2	45.2
ken-18	29,912	1.2	1.4	2.8	7.9	13.4	16.9
nw04	25,376	2.1	1.6	3.6	13.8	32.9	44.1
stp3d	32,708	1.0	1.1	2.4	8.3	14.0	20.9
rd-rplusc-21	31,272	3.2	1.6	2.2	8.1	52.1	75.9
momentum3	30,988	1.7	2.4	1.8	5.1	15.4	25.9
ds	33,324	2.2	1.6	3.5	15.2	46.5	64.1
osa-60	52,156	1.3	1.4	2.4	8.1	19.2	29.2

6.3. Pre- and Post-Processing of LP Instances

Most current optimization modeling systems are monolithic. Visualization, data manipulation, preprocessing and post-processing services are built into either the modeling language or the solver. An XML-based format facilitates the development of independent component plug-ins that can be attached to a modeling system to provide such services as visualization, structure detection, and cut generation.

For example, in Figure 21 we see a representation of the sparsity matrix of a bank-location set-covering linear-programming instance from Mairose et al. (1979), created by a plug-in based on SVG (an open XML-based standard for scalable vector graphics).

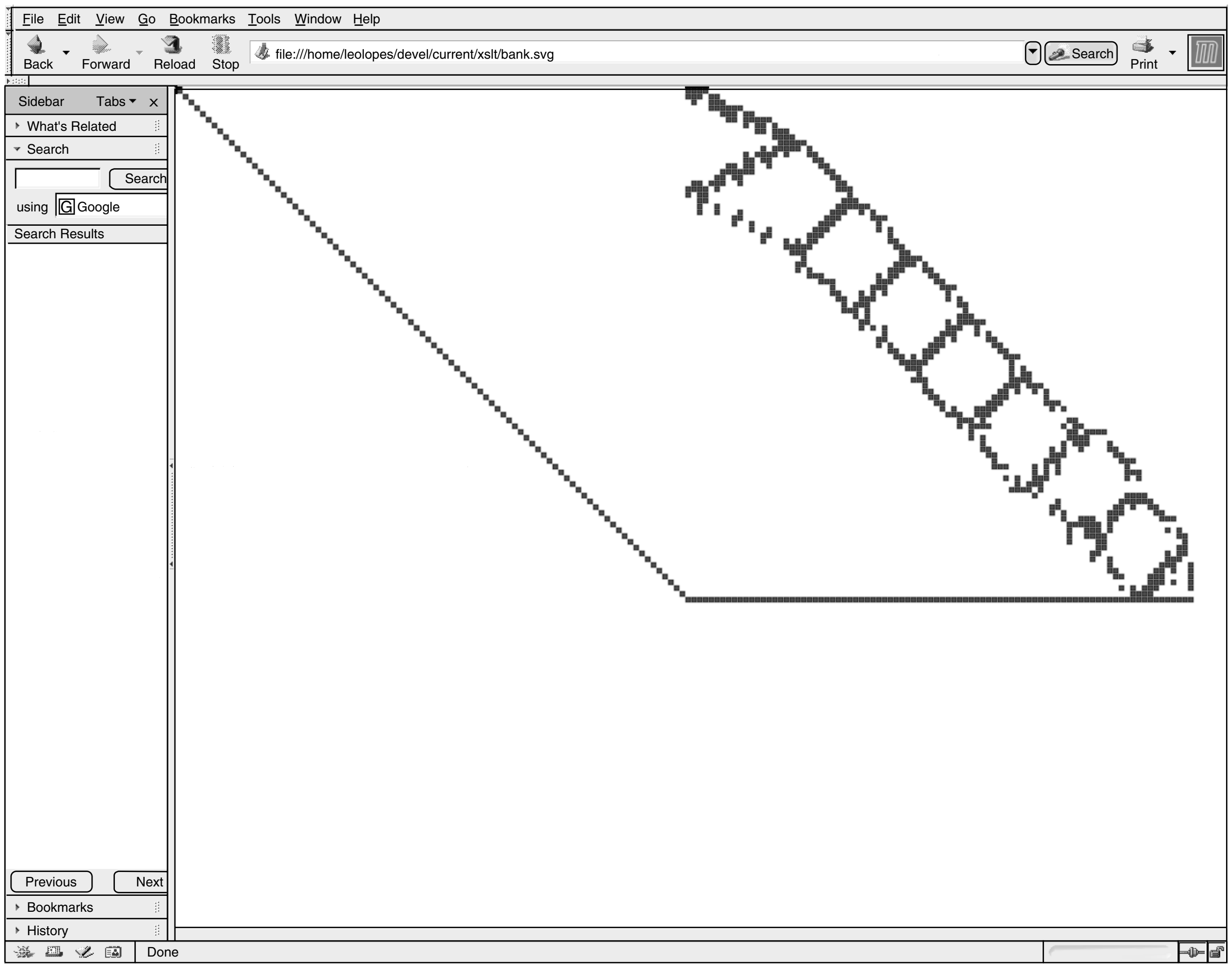


Figure 21 An SVG-Based Plug-In's Representation of a Coefficient Matrix Sparsity Pattern

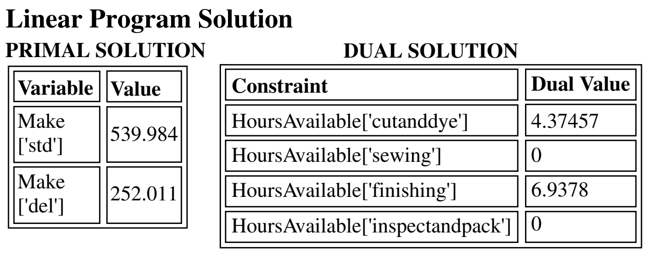
Notice that the image is displayed inside a regular web browser (Mozilla). Converting LPFML files to SVG files is accomplished using less than 100 lines of XSLT (Kay 2004), a language designed specifically for transforming XML files.

A second possible use of XSLT pre-processing is to convert from one XML instance dialect to another. For example, Bradley (2004) has developed an XML vocabulary, NaGML, for networks and graphs. By using XSLT one could readily convert a representation from NaGML into LPFML and then use all the associated libraries and connections to solvers described in this paper.

Post-processing of solution information is another useful feature enabled by the use of XML in LPFML. We illustrate in Figure 22 a transformation of the solution shown in Figure 20 into web-page (HTML) format. This was done using the library class `FMLLPToXML` with the Xalan XSLT processor from `http://www.apache.org/`.

The primary concern for pre- and post-processing environments is that plug-in components be easy to create. Thus it is desirable for the representation to be as simple as possible. In the context of LPFML, it is best to create an XML tag for every nonzero element of the sparse matrix (as illustrated in Figure 18), not using the complicating special-structure attributes `mult` and `incr` introduced in §4.3. Since the functionality provided under this mode cannot be achieved using MPS, there is no meaningful comparison to be drawn between LPFML and MPS here.

The representation for this case results in LPFML files that are roughly the same size as their MPS equivalents, so we do not recommend using it for communication with solvers as described earlier in



Linear Program Solution

PRIMAL SOLUTION

Variable	Value
Make ['std']	539.984
Make ['del']	252.011

DUAL SOLUTION

Constraint	Dual Value
HoursAvailable['cutanddye']	4.37457
HoursAvailable['sewing']	0
HoursAvailable['finishing']	6.9378
HoursAvailable['inspectandpack']	0

Figure 22 Results of an XSLT Style Sheet Transformation

this section. But the value of the service provided—flexible visualization or document preparation in the examples above—often outweighs any performance considerations.

7. Promoting the Standard

The objective of this research is to design and propose a standard for representing instances of linear programs, taking advantage of new XML technologies. Previous experience suggests, however, that an open set of convenient tools for *using* the proposed standard is essential to encouraging its adoption. Thus we are distributing a software library in conjunction with the proposed standard. To encourage the use of this library, we are releasing it as open-source software.

People disagree as to the meaning of open source, but for purposes of discussion we interpret it to include any software whose source code is available for modification and redistribution. Opening the source code offers significant advantages:

- increased quality through peer review, frequent updates, and contributions from third parties;
- greater transparency, by preventing deliberate changes or omissions of functionality for commercial purposes, and by making it clear to all parties that their investment in the technology will not be wasted;
- better documentation of the goals and achievements of the project, especially from a technical perspective.

In releasing open-source software, the license is a key consideration. There are numerous open-source software licenses, which differ in the restrictions they place on how the source code and binaries are used. Here are some common examples, ordered from greater to lesser restrictiveness in terms of redistribution requirements.

- The GPL (GNU General Public License) or *copyleft* license. This is a *quid-pro-quo* license. Its key feature 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 applies to libraries. It allows you to write software that *uses but does not modify* LPGL code, and then to redistribute an executable that contains your proprietary software and the LPGL code, *without having to distribute the source code for your own application*.
- Non-copyleft licenses. These licenses do not insist that modified and redistributed software also be open source. In contrast to GPL or LGPL software, they typically contain 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.

The Open Source Initiative has identified numerous open-source software licenses that meet its requirements; see `http://www.opensource.org/licenses`. A thorough discussion of open source licenses is provided by Fink (2003).

To encourage extensive use of the LPFML standard in a wide variety of scenarios, we have chosen to distribute the associated library under a non-copyleft license. This allows developers 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 `FMLSolve`, can link to the GLPK solver library (`http://www.gnu.org/software/glpk/glpk.html`, Makhorin 2003), which is licensed under the GPL. Because we are providing a complete distribution, we have to release `nl2fml` and `FMLSolve` under the GPL to give users the option of employing the GLPK solver.

A link for the software described in this paper is available from the online supplement to this paper on the journal's website. We currently have complete distributions, including makefiles, for the Windows and Linux operating systems; the user does not need to download any other software. We also provide a Microsoft Visual Studio 2003 .NET solution file `FML.sln` for the Windows distribution.

Maintaining an open-source standard and preventing its fragmentation can be a daunting task. Currently we are accepting suggestions for change and bug reports through LPFML's Bugzilla Web site (`http://senna.sie.arizona.edu/fmlzilla/`). The authors have formed an OASIS discussion group, `math-optimize-discuss`, with the hope of this discussion group leading to the formation of a Technical Committee devoted to establishing and promoting XML instance representations of optimization models. The archive for the discussion group is `http://lists.oasis-open.org/archives/math-optimize-discuss/`. The OASIS organization (`http://www.oasis-open.org/home/index.php`) is a nonprofit global consortium devoted to providing an open forum for developing, promoting, and maintaining XML standards.

One measure of an open source project's success is the willingness of people to contribute software that adheres to the standard. We have already received a significant contribution from OptiRisk Systems (2004), who have contributed the necessary software to parse LPFML XML files into the input format for their FortMP solver. Also, we learned very recently that one of the developers of `lp_solve` (see `http://groups.yahoo.com/group/lp_solve/`) is writing an LPFML based parser for this open-source LP code.

8. Extensions and Future Work

In its current state, LPFML only takes advantage of special structure through the `mult` and `incr` attributes introduced in §4.3. It would be desirable to take advantage of other structures such as network flows, variable upper and lower bounds, and stochastic-programming forms (Lopes and Fourer 2001).

Representing instances of nonlinear programs is also important. An existing XML vocabulary, Content MathML (Sandhu 2003), can be used to represent nonlinear terms such as those that appear in constraints and objectives. One direction of research is to develop a schema that uses the MathML namespace but that is specialized for optimization problems. See Fourer et al. (2004).

Finally, an important feature of XML is that it supports encryption standards such as XML Encryption (`http://www.w3.org/Encryption/2001/`). This standard has the flexibility to specify encryption of specific elements. Thus, for example, a user could encrypt the data in the constraint matrix of a linear program by choosing to encrypt all child elements of the `<sparseMatrix>` element.

Appendix: The LPFML Library Classes

This appendix provides a detailed description of the classes in the LPFML library, how they relate to each other, and how they use the technology provided by XML parsers. This material is especially relevant for those who wish to add functionality to the library, understand how it is implemented, or make significant changes (such as by replacing the Xerces parser that we used by another parser that might be faster). Complete documentation generated by Doxygen is available in the online supplement to this paper on the journal's website.

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

SAX2 parsers are event based. They call functions in the user's code upon finding specific elements, attributes, characters, and other components. 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, and similar LP components. The classes used by our library to read and parse the XML instance file are illustrated in Figure 23.

There are two key SAX2 classes used to parse an XML file. Objects of the `XMLReader` class are used to actually parse the XML file. When the parser 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 Apache Xerces `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. The aggregated 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;;.
```

None of these methods do anything in `FMLParser`. But when overridden by solver-specific implementations, they create or populate the necessary data structures in the solver.

FMLCOINParser. This class, inheriting from `FMLParser`, adds a convenient method `onCoinPackedMatrix` to provide 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 `CoinPackedMatrix` class. `CoinPackedMatrix` is an open-source sparse matrix class distributed as part of the Open Solver Interface of the COIN project (`http://www.coin-or.org/`).

FMLOSIParser. This class, implements all of the methods of its Superclass, `FMLCOINParser`, (and consequently `FMLParser`) that are 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 (OSI) implementation. For example, the method `onConstraints` is used as follows to get the name and 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;
}
```

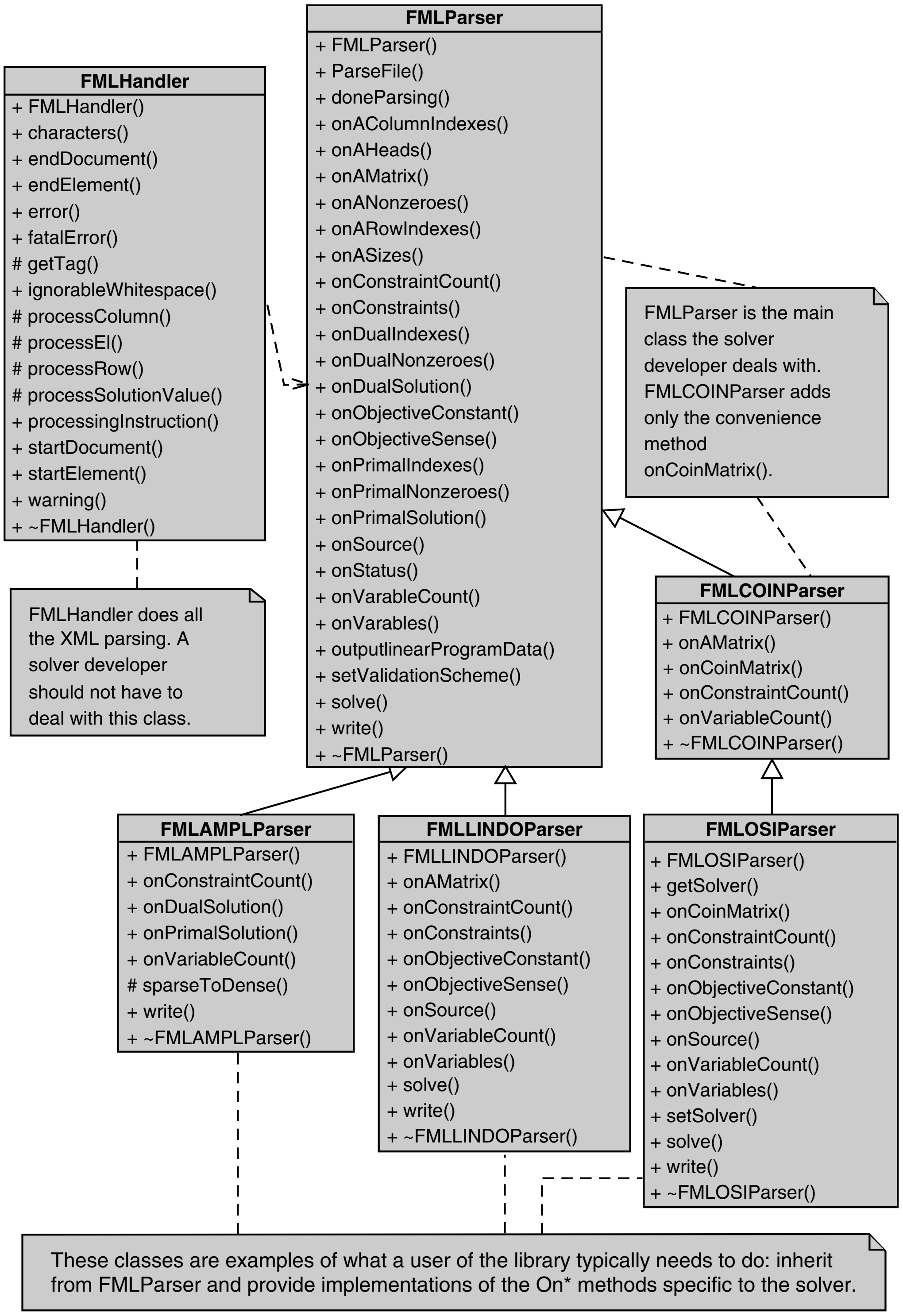


Figure 23 **The Parser Classes**

The `FMLOSIParser` implementation of `FMLParser::solve()` also uses only the OSI. Thus this class can call any solver that has an OSI interface, 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 (Schrage 1997, `http://www.lindo.com/`). `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 two is

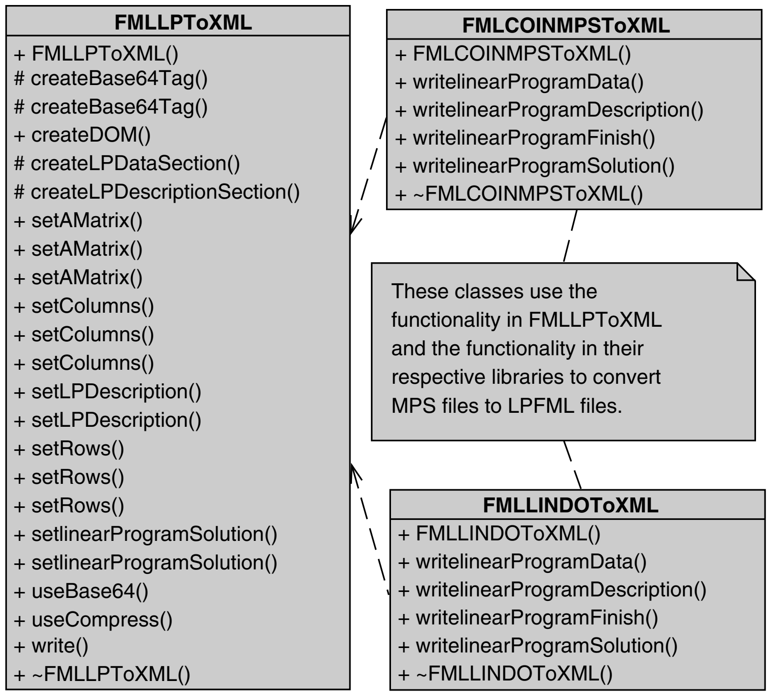

FMLLPToXML
+ FMLLPToXML()
# createBase64Tag()
# createBase64Tag()
+ createDOM()
# createLPDataSection()
# createLPDescriptionSection()
+ setAMatrix()
+ setAMatrix()
+ setAMatrix()
+ setColumns()
+ setColumns()
+ setColumns()
+ setLPDescription()
+ setLPDescription()
+ setRows()
+ setRows()
+ setRows()
+ 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()

Figure 24 The Writer Classes

that the LINDO API makes copies of all the parameters passed to it, while 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.

FMLFortMPParser. This is another implementation of a parser, specific for the FortMP solver (OptiRisk Systems 2004). `FMLFortMPParser` inherits from `FMLParser` and implements the same methods that `FMLOSIParser` does. (The authors thank Patrick Valente and OptiRisk Systems for this contribution to the library.)

FMLLPToXML. After the LP instance is read into a solver and optimized, the class `FMLLPToXML` is used to output the primal and dual solution (with the original LP data if the boolean variable `outputLPdata` is `true`) to a document object model (DOM). In the default implementation the DOM is written to a file. However, the DOM is a very flexible data structure and could be used in several different ways. For example, the DOM output could provide input for an in-memory data transfer (see the discussion of `nl2fml` below). A second use of the DOM is in conjunction with an XSLT transformer to convert the LP solution into HTML files that can be read by people through a Web browser, as illustrated in §6.

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 24.

FMLSolve. This utility takes an LPFML file, and through our library, solves the LP using any of the currently supported solvers. `FMLSolve` creates and manipulates only an `FMLParser` object. Depending on which solver is selected by the user (currently GLPK, CLP, FortMP, or LINDO), an appropriate child of `FMLParser` is instantiated and used to solve the problem. As new solvers become available, only the selection mechanism in `FMLSolve` needs to be changed.

nl2fml. This utility is a *driver* designed to work with the AMPL modeling language. A model instance is input into AMPL, and the solver option in AMPL is set to `nl2fml`. Upon execution, AMPL translates the instance to a file in its proprietary nl format, and then `nl2fml.exe` converts the nl file into an XML problem instance in an in-memory DOM tree. A parser object (for example `FMLOSIParser` or `FMLLINDOParser`) is created to parse the XML data and call the appropriate solver, by copying in memory the DOM tree into a SAX data structure for the appropriate parser object. Then the solution is parsed by `FMLAMPLParser` and the results read back into AMPL for further analysis. This sequence of operations is illustrated in Figure 25.

FMLCOINMPSToXML, FMLLINDOToXML. 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

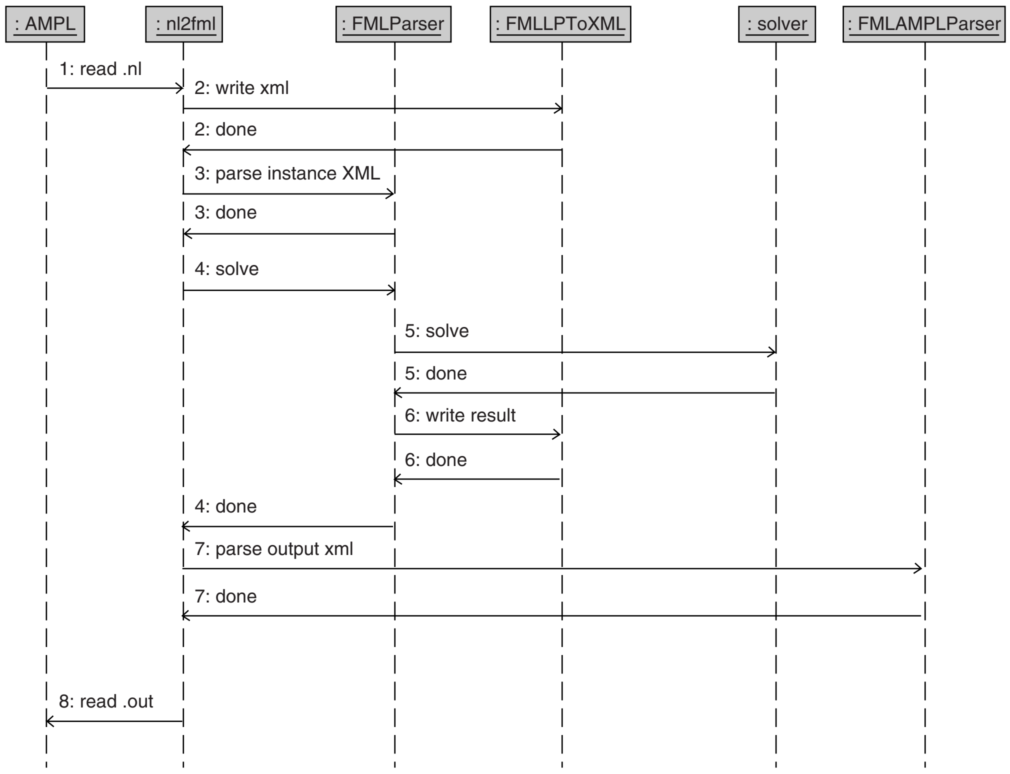


Figure 25 Using AMPL with a Solver

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 have implemented a connection between AMPL and any solver that has an OSI solver interface, as well as LINDO. We used the Xerces C++ XML open-source software available from `http://www.apache.org/`, but any SAX 2.0 compliant parser would have done as well. 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.

References

AIMMS. 2003. The AIMMS modeling language. Paragon Decision Technology, B.V., Haarlem, The Netherlands, `http://www.aimms.com/aimms/product/modeling_language.html`.

Anderson, D. R., D. J. Sweeney, T. A. Williams. 1991. *An Introduction to Management Science*, 6th ed. West Publishing, St. Paul, MN.

Bradley, G. 2003. Introduction to extensible markup language (XML) with operations research examples. *Newsletter INFORMS Comput. Soc.* **24** 1–20.

Bradley, G. 2004. Network and graph markup language (NaGML)—data file formats. Technical report NPS-OR-04-007, Department of Operations Research, Naval Postgraduate School, Monterey, CA.

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

Chang, T.-H. 2003. Modelling and presenting mathematical programs with xml:lp. Masters thesis, Department of Management, University of Canterbury, Christchruch, New Zealand.

COIN. 2003. COIN LP, `http://www.coin-or.org/`.

Dash Optimization. 2003a. Xpress-Mosel, `http://www.dashoptimization.com/pdf/mosel.pdf`.

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

Dolan, E. D., R. Fourer, J. J. Moré, T. S. Munson. 2002. Optimization on the NEOS server. *SIAM News* **35**(6) 8–9.

Duff, I. S., A. M. Erisman, J. K. Reid. 1986. *Direct Methods for Sparse Matrices*. Oxford University Press, New York.

Ezechukwu, O., I. Maros. 2003. OOF: Open optimization framework. Technical report ISSN 1469-4174, Department of Computing, Imperial College of London, London, UK.

Fink, M. 2003. *The Business and Economics of Linux and Open Source*, 1st ed. Prentice Hall PTR, Upper Saddle River, NJ.

Fourer, R. 1983. Modeling languages versus matrix generators for linear programming. *ACM Trans. Math. Software* **9** 143–183.

Fourer, R., D. Gay, B. Kernighan. 1993. *AMPL: A Modeling Language for Mathematical Programming*. Scientific Press, San Francisco, CA.

Fourer, R., J. Ma, K. Martin. 2004. OSiL: An instance language and API for optimization. Technical report, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL.

Gay, D. M. 1990. Correctly rounded binary-decimal and decimal-binary conversions. Numerical Analysis Manuscript No. 90-10, AT&T Bell Laboratories, Murray Hill, NJ.

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

ILOG. 2003a. ILOG CPLEX, `http://www.ilog.com/products/cplex/`.

ILOG. 2003b. ILOG tutorial, `http://www.ilog.com/products/oplstudio/tutorial/index.cfm`.

Kay, M. 2004. *XSLT Programmer's Reference 3rd Edition*. Wrox Press, Birmingham, UK.

Kristjánsson, B. 2001. Optimization modeling in distributed applications: How new technologies such as XML and SOAP allow OR to provide web-based services, `http://www.maximal-usa.com/slides/Svna01Max/index.htm`.

Lopes, L., B. Fourer. 2001. An XML-based format for communicating optimization problems. Presented at INFORMS Annual Meeting, Miami Beach, FL.

Ma, J. 2004. Optimization services (OS), a general framework for optimization modeling systems. Ph.D. dissertation, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL.

Mairose, L., D. Sweeney, K. Martin. 1979. Strategic planning in bank location. *Proc. Amer. Inst. Decision Sci.*

Makhorin, A. 2003. GLPK (GNU linear programming kit), `http://www.gnu.org/software/glpk/glpk.html`.

Martin, K. 2002. A modeling system for mixed integer linear programming using XML technologies. Technical report, Graduate School of Business, University of Chicago, IL.

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

Moré, J., T. Munson, J. Sarich. 2004. NEOS optimization server, `http://www-neos.mcs.anl.gov/neos/`.

Mosek ApS. 2003. MOSEK, `http://www.mosek.com/`.

Murtagh, B., M. Saunders. 1983. MINOS 5.4 user's guide. Systems Optimization Laboratory SOL 83-20R, Stanford University, Stanford, CA.

OptiRisk Systems. 2004. Fortmp, `http://www.optirisk-systems.com/`.

O'Reilly. 1999. Science XML vocabularies, `http://www.xml.com/pub/rg/Science`.

Sandhu, P. 2003. *The MathML Handbook*. Charles River Media, Hingham, MA.

Schrage, L. 1997. *Optimization Modeling with LINDO*, 5th ed. Brooks/Cole, Pacific Grove, CA.

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

Skonnard, A., M. Gudgin. 2002. *Essential XML Quick Reference*. Pearson Education, Inc., Boston, MA.

Soiffer, N. 1997. MathML: A proposal for representing mathematics in HTML. *ACM SIGSAM Bull.* **31**(3) 44–45.

Suciu, D., H. Liefke. 1999. XMILL an efficient compressor for XML, `http://www.research.att.com/sw/tools/xmill/`.

Winston, W. 1994. *Operations Research Applications and Algorithms*, 3rd ed. Duxbury Press, Belmont, CA.