OSiL: An Instance Language for Optimization

Robert Fourer
Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, Illinois 60208, USA 4er@iems.northwestern.edu
Jun Ma
Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, Illinois 60208, USA maj@iems.northwestern.edu
Kipp Martin
Graduate School of Business, University of Chicago, Chicago, Illinois 60637, USA kipp.martin@uchicagogsb.edu

Distributed computing technologies such as Web Services are growing rapidly in importance in today's computing environment. In the area of mathematical optimization, it is becoming increasingly common to separate modeling languages from optimization solvers. In fact, the modeling language software, solver software, and data used to generate a model instance might reside on different machines using different operating systems. Such a distributed environment makes it critical to have an open standard for exchanging model instances.

In this paper we present OSiL (Optimization Services instance Language), an XML-based computer language for representing instances of large-scale optimization problems including linear programs, mixed-integer programs, quadratic programs, and very general nonlinear programs. OSiL has two key features that make it much superior to current standard forms for optimization problem instances. First, it uses the object-oriented features of XML schemas to efficiently represent nonlinear expressions. Second, its XML schema maps directly into a corresponding in-memory representation of a problem instance. The in-memory representation provides a robust application program interface for general nonlinear programming, facilitates reading and writing postfix, prefix, and infix formats to and from the nonlinear expression tree, and makes the expression tree readily available for function and derivative evaluations.

Key words: linear programming; nonlinear programming; modeling languages; information systems; web services; XML

Acknowledgment: This work was supported in part by National Science Foundation grants CCR-0082807 and DMI-0322580 to Northwestern University.

---

1. Introduction

Web Services and other distributed computing technology standards are becoming increasingly important to Internet applications. This trend has growing implications in large-scale optimization, where modeling language software, solver software, and data used to generate a model instance may reside on different machines using different operating systems.

Such a distributed environment makes it critical to have an open standard for exchanging problem instances. By *instance* we mean a particular problem for which answers can be sought in the form of specific values for decision variables, in contrast to a *model* that is a description of a broad class of optimization problems. Typically a model is a *symbolic*, *general*, *concise,* and *understandable* representation of an optimization problem, whereas an instance is an *explicit*, *specific*, *verbose,* and *convenient* description of a problem's objective and constraints [13]. Thus a model plus data is required to generate an instance. A linear programming model is typically described by linear algebraic expressions, for example, while the corresponding instance is represented as a list of nonzero coefficients of variables in the objective and constraint expressions, along with bounds on the variables and the constraint expressions.

Current optimization software is hobbled by its reliance on a plethora of input formats, as can be seen by even a cursory look at the list of solvers available on the NEOS Server ([5, 6] or `www-neos.mcs.anl.gov/neos`). The nearly 50 solvers in the NEOS lineup require instance inputs of about a dozen different kinds, including MPS [16] and LP formats for linear and integer programming, SMPS extensions to the MPS format for stochastic programming, formats such as SDPA specific to semidefinite programming, DIMACS min-cost flow and other formats for network linear programming, and proprietary formats used by two modeling language processors. Other solvers recognize input programmed as functions in various languages including Fortran, C, C++, and Matlab.

This paper presents OSiL, an XML-based language designed as a new standard for representing optimization problem instances. OSiL serves as an instance-level format flexible enough to handle linear and mixed-integer programs, quadratic programs, and very general nonlinear programs. Its underlying principles are sufficiently powerful to allow for future extensions for such problem areas as cone, constraint, disjunctive, stochastic, and semidefinite programming. Thus it has the potential to serve as a new standard that subsumes the many currently used input formats.

OSiL's definition as an XML vocabulary makes it superior to current formats for use in the increasingly common distributed optimization contexts that we have cited. The OSiL standard also incorporates an application programming interface (an API) that provides solvers with a standard way of accessing instance data. The combination of a common XML format for instance representation and a common API for data access offers clear benefits to the optimization community. Modeling language developers are not taxed with creating a separate "driver" for each supported solver, and solver developers can use essentially the same interface to connect to all modeling systems and applications.

A number of new standards have been proposed (if not widely adopted) in recent years. Various extensions of the MPS format to nonlinear programming have been put forward, notably the xMPS format described by Halldórsson, Thorsteinsson and Kristjánsson [15]. We are also not the first to incorporate XML into this area. Fourer, Lopes and Martin [12] propose the LPFML XML schema for representing instances of mixed-integer linear programs; Chang [4] and Kristjánsson [17] have also proposed XML representations for linear programming instances. Ezechukwu and Maros [8] describe an Algebraic Markup Language that uses XML to describe the model rather than the instance, and Bradley [3] introduces an XML markup grammar for networks. See also [2] for a good overview of the uses of XML technologies in operations research.

New API proposals have also been a subject of recent activity. The extensive COIN-OR (COmputational INfrastructure for Operations Research) project (Lougee-Heimer [19] or `www.coin-or.org`) includes the OSI (Open Solver Interface) library, an API for linear programming solvers, and NLPAPI, a subroutine library with routines for building nonlinear programming problems. Another nonlinear interface, MOI (Modeler-Optimizer Interface), is proposed along with xMPS in [15]; it specifies the format for a callable library based on representing the nonlinear part of each constraint and the objective function in postfix (reverse Polish) notation [1] and then assigning integers to operators, characters to operands, and integer indices to variables so that the data structure corresponds to the implementation of a stack machine. A similar interface is used in the LINDO API [18].

In comparison to other proposals, our OSiL instance representation is notable for its broader range of representational options, very flexible API based on an expression-tree representation, and open source libraries founded on modern XML technology. After providing the requisite background for XML in the next section, we present the OSiL language in detail in Section 3. We first explain the aspects of the language that are common to all optimiza-

tion instances, and then describe the OSiL representations of linear programs, quadratic programs, and general nonlinear programs. A key aspect of our approach to representing nonlinear terms in an optimization instance is to use XML elements that all derive from one base type of element. This idea is developed in Section 3.4.

The OSiL schema defines an XML vocabulary for storing an optimization instance. This instance may persist in a repository of test problems, or it may be encapsulated in a SOAP (Simple Object Access Protocol — a high-level networking protocol for encapsulating XML data) envelope for use in a distributed computing environment. However, an optimization instance represented in OSiL must, at some point, have an in-memory representation in order to be useful to solvers. In Section 4 we describe the `OSInstance` class for this purpose. The `OSInstance` class has an API consisting of `get()` and `set()` methods that are used for extracting various components of the optimization instance, or for creating and modifying instances in memory. A key aspect of the in-memory instance representation is the `OSExpressionTree` class, which is used for the internal representation of the nonlinear part of an optimization instance. As explained in Section 4.2, `OSExpressionTree` is designed for ease of parsing in order to generate instances in varied postfix, prefix, and infix formats amenable to solvers, and also for ease of function and derivative evaluation when required by solvers.

In the concluding remarks of Section 5, we discuss extensions of OSiL to other problem types, which are to be described in subsequent papers. OSiL is moreover a part of a much broader Optimization Services research agenda, described by Ma [20], which provides a host of similarly-named XML languages for use in optimization over the Internet. In addition to OSiL (for Optimization Services instance Language), these include OSrL (Optimization Services result Language) for representing problem solutions and results, and OSoL (Optimization Services option Language) for communicating options to solvers.

2. XML Background

The logic and advantages of using XML as a markup language to represent optimization instances are set forth by Fourer, Lopes and Martin [12]. See also the excellent general overview of XML by Skonnard and Gudgin [23]. This section introduces aspects of XML relevant to this paper, including the basics of XML files, parsing, and schemas.

2.1. XML Files

In this paper we propose storing optimization problem instances as XML files. Consider the following problem instance that is a modification of an example of Rosenbrock [21]. The corresponding OSiL representation of this problem is given in the Appendix.

$$\text{Minimize} \quad (1 - x_0)^2 + 100(x_1 - x_0^2)^2 + 9x_1 \tag{1}$$

$$\text{Subject to} \quad x_0 + 10x_0^2 + 11x_1^2 + 3x_0x_1 \leq 25 \tag{2}$$

$$\ln(x_0x_1) + 7x_0 + 5x_1 \geq 10 \tag{3}$$

$$x_0, x_1 \geq 0 \tag{4}$$

There are two continuous variables, x_0 and x_1, in this instance, each with a lower bound of 0. For our XML-based standard format, OSiL, we propose to represent this information in an XML-formatted file as shown in Figure 1.

```
<variables numberOfVariables="2">
    <var lb="0" name="x0" type="C"/>
    <var lb="0" name="x1" type="C"/>
</variables>
```

Figure 1: The `<variables>` element for the example (1)–(4).

An XML file is a text file that contains both *markup* and *data*. In the Figure 1 XML file there are two types of markup, *elements* (or *tags*) and *attributes* that describe the elements. Specifically, there are a `<variables>` element and two `<var>` elements. Each `<var>` element has attributes `lb`, `name`, and `type` that describe properties of a decision variable: its lower bound, "name", and domain type. The actual values of the attributes, such as `"0"` (zero) for `lb` and `"C"` (denoting a continuous domain) for `type`, are the data in the file. An attribute may also assume a default value when it does not appear; for example, the `<var>` element has a `ub` attribute that is absent in Figure 1 and that consequently takes the default value `INF` (denoting ∞).

2.2. XML Schemas

To be useful for communication between solvers and modeling languages, the XML instance files must conform to a standard. Parsing optimization instance files in a meaningful way

is impossible otherwise. A representation standard is imposed through the use of a *W3C XML Schema.* The W3C, or World Wide Web Consortium (`http://www.w3.org`), promotes standards for the evolution of the web and for interoperability between web products. XML Schema (`http://www.w3.org/XML/Schema`) is one such standard. A schema specifies the elements and attributes that define a specific XML vocabulary. The W3C XML Schema is a schema for schemas; it specifies the elements and attributes for a schema that in turn specifies elements and attributes for an XML vocabulary such as OSiL. By analogy to object oriented programming, a schema is akin to a header file in C++ that defines the members and methods in a class. An XML file that conforms to the schema is called *valid.* Just as a class in an object oriented programming language very explicitly describes member and method names and properties, a schema explicitly describes element and attribute names and properties.

```
<xs:complexType name="Variables">
    <xs:sequence>
        <xs:element name="var" type="Variable" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="numberOfVariables" type="xs:positiveInteger" use="required"/>
</xs:complexType>
```

Figure 2: The `Variables` complexType in the OSiL schema.

```
<xs:complexType name="Variable">
    <xs:attribute name="name" type="xs:string" use="optional"/>
    <xs:attribute name="init" type="xs:string" use="optional"/>
    <xs:attribute name="type" use="optional" default="C">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="C"/>
                <xs:enumeration value="B"/>
                <xs:enumeration value="I"/>
                <xs:enumeration value="S"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="lb" type="xs:double" use="optional" default="0"/>
    <xs:attribute name="ub" type="xs:double" use="optional" default="INF"/>
</xs:complexType>
```

Figure 3: The `Variable` complexType in the OSiL schema.

Figure 2 is a piece of our schema for OSiL. In W3C XML Schema jargon, Figure 2 defines what is known as a *complexType.* A complexType specifies elements and attributes that are allowed to appear in a valid XML instance file such as the one illustrated in Figure 1. In particular, Figure 2 defines the complexType named `Variables`, which comprises an element named `<var>` and an attribute named `numberOfVariables`. The `numberOfVariables` attribute is of a standard type `positiveInteger`, whereas the `<var>` element is a user-defined complexType named `Variable`. Thus the complexType `Variables` contains a sequence of `<var>` elements that are of complexType `Variable`. OSiL's schema must also provide a specification for the `Variable` complexType, which is shown in Figure 3.

We follow the convention that elements and attributes in the XML instance file begin with lowercase letters, whereas the user-defined complexTypes begin with uppercase letters. A complexType is the XML schema analogue of a class in an object-oriented programming language, while an element (such as `<var>`) in the XML instance file corresponds to an instantiated object of a class (such as `Variable`). Thus we will often refer to complexTypes as classes and to elements as objects. This object-oriented analogy is continued in Section 4 where we define an `OSInstance` class that is the in-memory representation of an OSiL instance file.

Our definition of the `Variable` complexType allows only the attributes listed in Figure 3 to be present in a `<var>` element. All of these attributes are specified as optional. Properties of the attributes are explicitly defined: the `lb` attribute must be a double precision number, for example, and the `type` must be a string value that is either `C`, `B`, `I`, or `S`. (These designate continuous, binary, integer, and string-valued variables, respectively.) We discuss the OSiL schema in further detail in Section 3.

The key benefit of defining the OSiL schema is to impose a problem instance standard that can be applied by parsers to validate instance files. If a problem instance is valid then the parser knows, for example, exactly where in the instance file to locate information on the constraint upper bounds or to determine if there are integer variables present in the model. However, as useful as validation is, the concept of validation is about *syntax* rather than *semantics.* For example, a problem instance file may be valid for the OSiL schema even though it contains a value for the attribute `numberOfVariables` in the `<variables>` element that is not consistent with the number of `<var>` elements in the `<variables>` section; to catch this error requires an additional check in the parser.

3. The OSiL Schema

Our approach is to write a nonlinear optimization problem as a linear program plus a collection of quadratic and more general nonlinear terms. We begin by describing the aspects of OSiL that are the same for all problem instances. Then we describe how linear, quadratic, and general nonlinear problems are represented.

3.1. OSiL Basics

We illustrate the OSiL schema using the example given by equations (1)–(4). The actual schema file resembles the XML excerpts in Figures 2 and 3, except that it is much longer and harder to read; it may be found at `www.optimizationservices.org/schemas/OSiL.xsd`. Instead we depict the schema and its parts by means of tree diagrams, with schema elements as the nodes. Specialized schema development software lets people work directly with these graphical representations.

Figure 4 depicts the top two levels of the OSiL schema in tree-diagram form. It shows that the `<osil>` root element has two child elements. The first element, `<instanceHeader>`, and its child elements are depicted in Figure 4, and the corresponding part of the OSiL file for the instance (1)–(4) is shown in Figure 5. The child elements `<name>`, `<source>` and `<description>`, of the parent `<instanceHeader>`, describe general properties of the problem and require no further explanation.

In Figure 4 the solid rectangles denote elements. The "ellipsoid" icon containing four

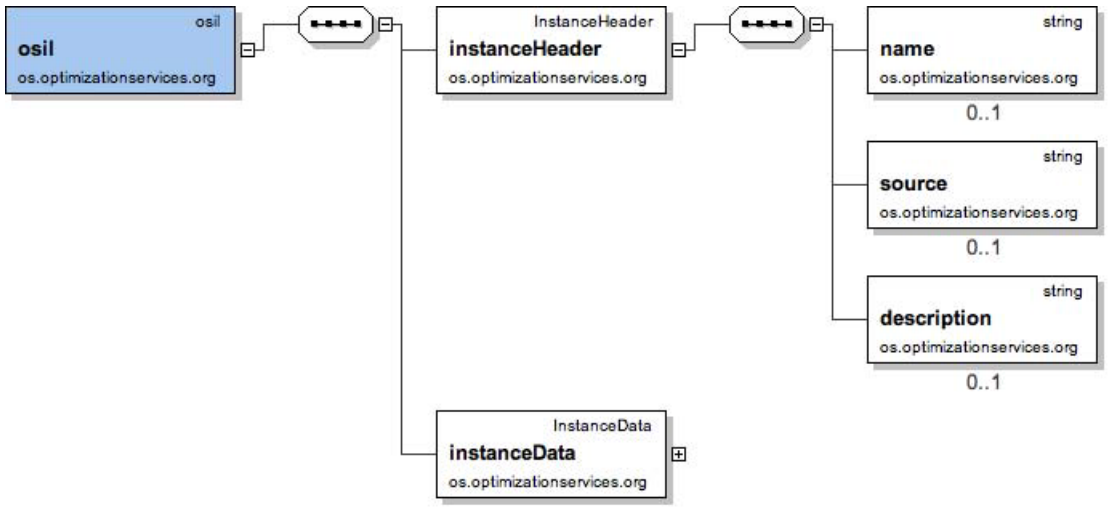

Figure 4: The OSiL schema.

```
<instanceHeader>
    <name>Modified Rosenbrock</name>
    <source>Computing Journal3:175-184, 1960</source>
    <description>Rosenbrock problem with constraints</description>
</instanceHeader>
```

Figure 5: The `instanceHeader` element for instance (1)–(4).

small rectangles denotes a *sequence* of elements. Thus the `<instanceHeader>` element consists of a sequence of at most three elements, `<name>`, `<source>`, and `<description>`. The 0..1 icon below a rectangle indicates that the corresponding element is optional, but that if present it may appear only once. For example, the `<name>` element is optional and may appear at most once. If no icon appears below the rectangle, then that element must appear exactly once. For example, there must be exactly one `<instanceHeader>` element in a valid OSiL file.

The second child element of the `OSiL` root element is the `<instanceData>` element, illustrated in Figure 6. This element holds all problem data, and so has a more complicated structure.

The first child element of `<instanceData>` is a `<variables>` element that consists of a sequence of `<var>` elements, one for each variable in the problem instance. Attributes of these elements provide standard information such as domain type, bounds, and optional names, as illustrated for our example in Figure 7.

The second child element of `<instanceData>` is an `<objectives>` element that consists of a sequence of `<obj>` elements, one for each potential objective function in the problem instance. Each `<obj>` has a set of attributes that include `name`, `maxOrMin`, `constant`, and `weight.` If the objective function has linear terms, these are stored in the `<coef>` child elements of each `<obj>` element. See for example, how in Figure 7 the linear term $9x_1$ in the objective function in Equation 1 is represented.

The third child element of `<instanceData>` is a `<constraints>` element that consists of a sequence of `<con>` elements, one for each constraint in the problem instance. The most important attributes of these elements are the constraint lower and upper bounds, as also illustrated in Figure 7. A constraint is an equality when its bounds are equal, and is a one-sided inequality when one or the other bound is absent. Hence the bound attributes provide the "right-hand side" values for the constraints.

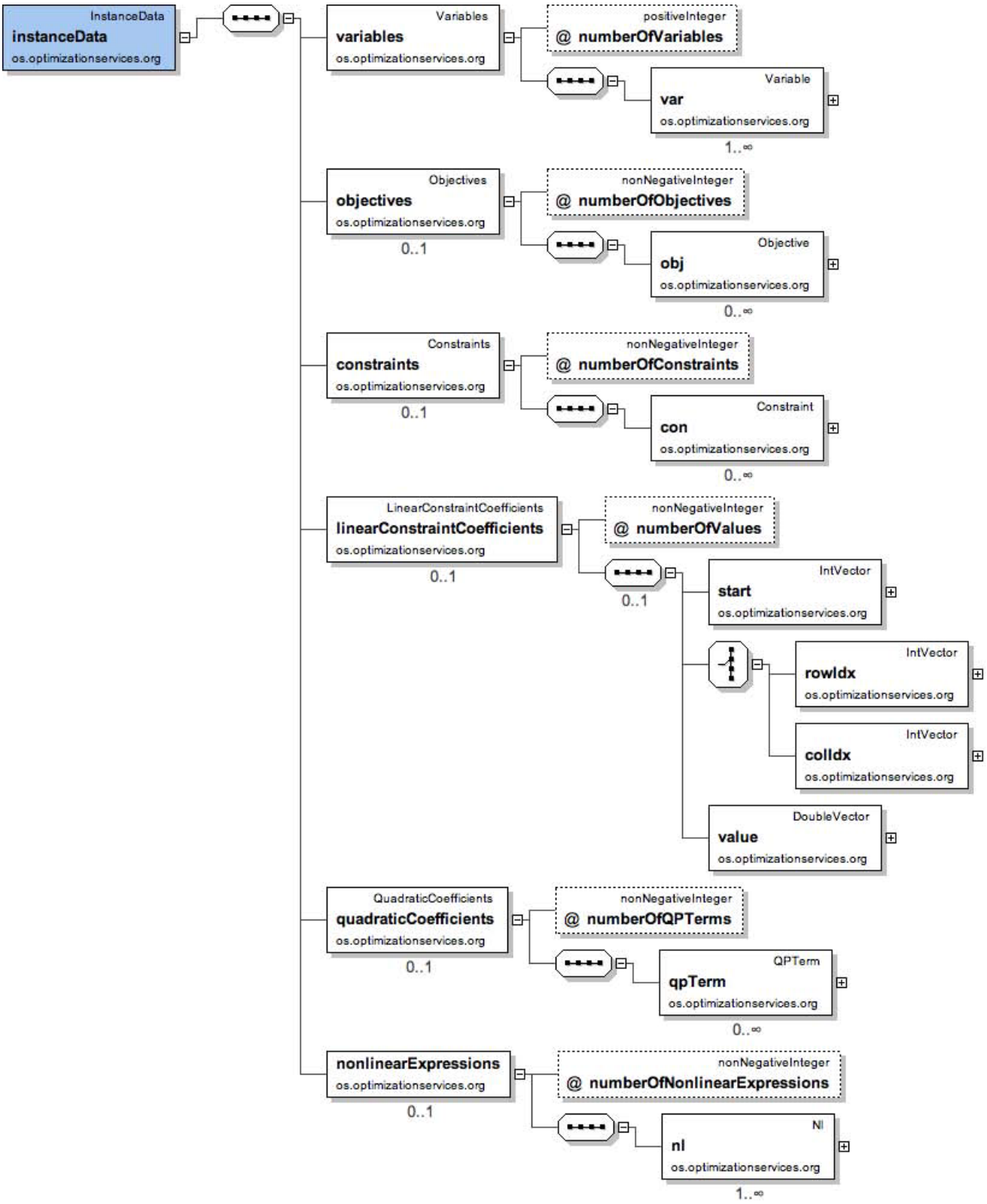

Figure 6: Detail of the `instanceData` element.

```
<variables number="2">
    <var lb="0" name="x0" type="C"/>
    <var lb="0" name="x1" type="C"/>
</variables>
<objectives number="1">
    <obj maxOrMin="min" name="minCost" numberOfObjCoef="1">
        <coef idx="1">9</coef>
    </obj>
</objectives>
<constraints number="2">
    <con ub="25.0"/>
    <con lb="10.0"/>
</constraints>
```

Figure 7: The <variables>, <objectives>, and <constraints> elements for (1)–(4).

The <variables> element is required and must contain at least one <var> element. The <constraints> element is optional because OSiL allows for unconstrained problems, which are of interest when nonlinear terms are specified in the objective as explained in Section 3.4.

3.2. Representing Linear Programs

Almost invariably, most of the linear constraint coefficients in a large optimization problem are zero. Thus OSiL adopts the usual approach of recording only the nonzeros. This is done by use of Figure 6's <linearConstraintCoefficients> element, which follows the sparse coefficient matrix storage scheme of the earlier LPFML schema [12]. The coefficient matrix is stored using three arrays. Each one of these arrays is represented by a child element of <linearConstraintCoefficients>: an array of nonzero coefficients <value>, a corresponding array of row indices <rowIdx>, and a pointer array <start> that indicates where each row begins in the previous two arrays. Individual array entries are specified by <el> children (not shown in Figure 6).

Figure 8 depicts the <linearConstraintCoefficients> element for the two constraints of the instance (1)–(4). There are three linear coefficients, one from the linear first constraint $x_0 + 3x_0x_1 \leq 25$, and two from the linear part of the second constraint $\ln(x_0x_1) + 7x_0 + 5x_1 \geq 10$.

The <linearConstraintCoefficients> element uses <rowIdx> when storing the matrix column-wise (and <colIdx> when storing the matrix row-wise). (Note in Figure 6 the

```
<linearConstraintCoefficients  numberOfValues="3">
    <start>
        <el>0</el><el>2</el><el>3</el>
    </start>
    <rowIdx>
        <el>0</el><el>1</el><el>1</el>
    </rowIdx>
    <value>
        <el>1.0</el><el>7.0</el><el>5.0</el>
    </value>
</linearConstraintCoefficients>
```

Figure 8: The `<linearConstraintCoefficients>` element for constraints (2)–(4).

"switch" icon immediately preceding the `<rowIdx>` and `<colIdx>` element icons. This icon denotes a *choice* element group.) With modern implementations offering options to set up the dual problem or to speed up calculations by using both row-wise and column-wise lists, it makes sense to provide for row-wise specification of coefficients in any new standard form.

3.3. Quadratic Programming

Any quadratic expression is easily represented as a general nonlinear expression using the format described in Section 3.4. Nevertheless there are good reasons for including a special quadratic expression representation in OSiL. First, each quadratic term admits a particularly compact representation as a list of index-index-value triples. Moreover there are numerous specialized solvers for the case in which the objective and all constraints are restricted to be linear or quadratic. These solvers look for the quadratic as well as the linear terms to be passed to them in full at invocation, rather than being evaluated at particular iterates as would be required by solvers that take more general smooth nonlinear functions. A special representation of quadratic terms in a problem instance facilitates passing the full list of quadratic terms to the solver.

In OSiL, a `<qTerm>` element is used to represent each quadratic term. It has two required integer attributes that specify the variable indices in the quadratic term. The coefficient of the quadratic term is specified using either a third optional real attribute or by a single child element. An added advantage of using the `<qTerm>` elements is that if an analyzer applied to the problem instance discovers that the only nonlinear terms are `<qTerm>` terms, it can

```
<quadraticCoefficients numberOfQuadraticTerms="3">
     <qTerm idx="0" idxOne="0" idxTwo="0" coef="10"/>
     <qTerm idx="0" idxOne="1" idxTwo="1" coef="11"/>
     <qTerm idx="0" idxOne="0" idxTwo="1" coef="3"/>
</quadraticCoefficients>
```

Figure 9: The `<quadraticCoefficients>` element for constraint (2).

classify the problem as a quadratic program. Figure 9 illustrates a representation of the quadratic term for constraint (2) of our example.

3.4. Nonlinear Terms and the OSnL Schema

The parts of OSiL shown so far are sufficient to describe a linear objective and constraints, with any combination of continuous and integer variables, and quadratic terms. General nonlinear terms are handled separately, by defining an OSnL schema that is imported by the OSiL schema. It is not necessary to have a separate schema for representing nonlinear terms; we do so for convenience. Our design goal for OSnL is to represent a comprehensive collection of optimization problems while keeping parsing relatively simple.

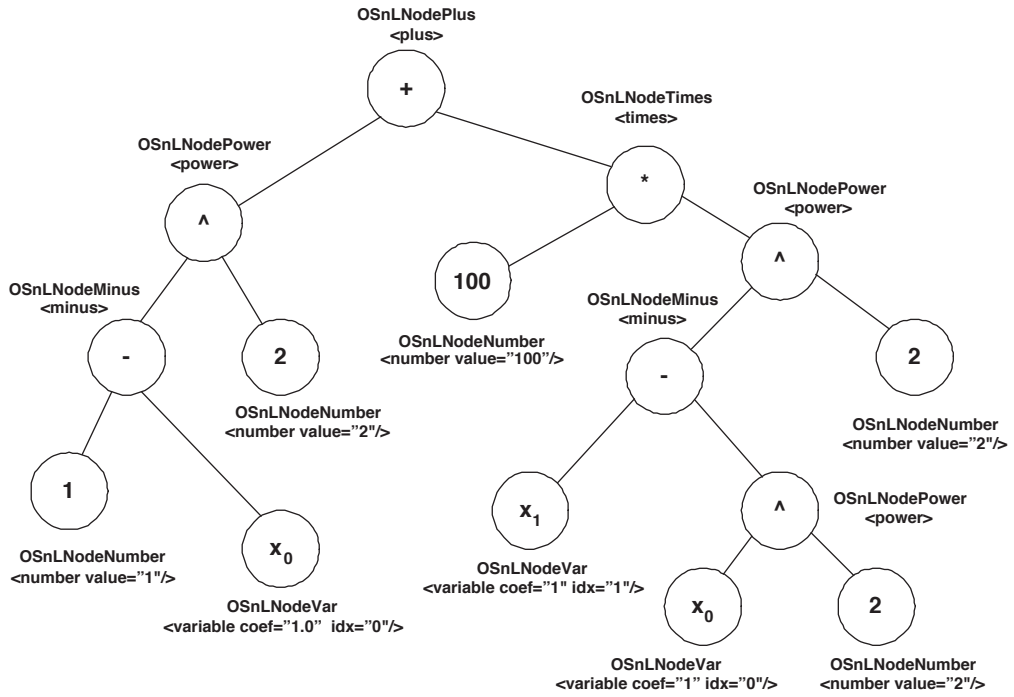

Figure 10: Conceptual form of an expression tree for the nonlinear part of (1).

```
<nl idx="-1">
     <plus>
          <power>
               <minus>
                    <number  value="1.0"/>
                    <variable coef="1.0" idx="0"/>
               </minus>
               <number  value="2.0"/>
          </power>
          <times>
               <power>
                    <minus>
                         <variable coef="1.0" idx="0"/>
                         <power>
                              <variable coef="1.0" idx="1"/>
                              <number  value="2.0"/>
                         </power>
                    </minus>
                    <number  value="2.0"/>
               </power>
               <number value="100"/>
          </times>
     </plus>
</nl>
```

Figure 11: The `<nl>` element for the nonlinear part of the objective function (1).

Developing a schema for representing a general nonlinear program in an XML format that is easily validated against the schema is a difficult problem. Clearly some type of expression tree is required for representing a nonlinear problem instance. As an example, Figure 10 shows how the nonlinear part of the objective function (1) is represented as an expression tree. The corresponding representation in OSiL is given in Figure 11. The root of the expression tree is a `<plus>` element. For purposes of validation, any schema needs an explicit description of the children allowed in a `<plus>` element, but it is clearly inefficient to list every possible nonlinear operator or nonlinear function allowed as a child element. Indeed, in our OSiL Schema we define over two hundred nonlinear elements. In general, if there are n allowable nonlinear elements (functions and operators), listing every potential child element, of every potential nonlinear element, leads to $O(n^2)$ possible combinations. This approach is obviously inefficient as the number of operators grows. However, in order to validate the problem instance a schema must know, for example, that the `<plus>` element

requires exactly two children and that each child element is an allowed operator or function.

Fortunately, the W3C XML Schema standard addresses this situation, by providing a construct very similar to that of an abstract class in an object oriented programming language such as C++ or Java. The use of this construct, called a `substitutionGroup`, is illustrated in Figures 12–14.

In Figure 12 we define an `OSnLNode` as an abstract class. Think of the `OSnLNode` as a "template" for a generic nonlinear element. In Java an analogous statement is

```
public abstract class OSnLNode{ ...
}
```

Next, in Figure 13 the class `OSnLNodePlus` is defined. The `OSnLNodePlus` class "extends" the `OSnLNode` abstract class. The `OSnLNodePlus` requires exactly two children, and each child must be an `OSnLNode`. Thus, we avoid the problem of explicitly listing every type of nonlinear operator and function. In Java an analogous statement is

```
public class OSnLNodePlus extends OSnLNode{ ...
}
```

```
<xs:element name="OSnLNode" type="OSnLNode" abstract="true">
```

Figure 12: The `OSnLNode` Abstract Class

```
<xs:complexType name="OSnLNodePlus">
    <xs:complexContent>
        <xs:extension base="OSnLNode">
            <xs:sequence minOccurs="2" maxOccurs="2">
                <xs:element ref="OSnLNode"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

Figure 13: Extending to the `OSnLNode` class to the `OSnLNodePlus` class

```
<xs:element name="plus" type="OSnLNodePlus" substitutionGroup="OSnLNode"/>
```

Figure 14: Declaring a `plus` Object

Finally, in Figure 14 the actual object (element) `plus` is declared. It is in class `OSnLNodePlus` that extends `OSnLNode.` Like an object that is instantiated in an object oriented programming language, the `plus` element is what actually appears in the XML file. An analogous statement in Java is `OSnLNodePlus plus = new OSnLNodePlus()`.

Armed with the `OSnLNode` concept, we can represent the parse tree for nonlinear expressions in an optimization instance as an XML file that is easily validated against a schema. As shown in Figure 6, the `<instanceData>` element in the OSiL schema has a child element `<nonlinearExpressions>` consisting of a sequence of `<nl>` children, one for each constraint and objective that has nonlinear terms. The `idx` attribute of each `<nl>` child refers to its corresponding objective function or constraint; the constraints are indexed by nonnegative integers starting from 0 and the objectives are indexed by the negative integers starting from -1. Each `<nl>` element has a single child element of type OSnLNode, that specifies the required nonlinear expression. For example, Figures 11 and 15 illustrate the `<nl>` elements for the objective function (1) (not including the linear term $9x_1$) and constraint (3), respectively.

The OSiL XML representation in Figure 11 corresponds exactly to the structure of the parse tree for the expression in Figure 10. There is an `<nl>` node with an index of -1, which indicates the first objective function. The single child of the `<nl>` element is a `<plus>` element which is the root of the expression tree illustrated in Figure 10. The root element has two children, representing the two sub-expressions, $100(x_1 - x_0^2)^2$ and $(1 - x_0)^2$, that are added to form the nonlinear part of the objective function; the `<power>` child has also two children, representing the sub-sub-expressions, $(1 - x_0)$ and 2 that are the base and the exponent of an exponentiation operator; and so forth. Similarly, in Figure 15, the `<nl>` element with index 1 has a single child element `<ln>` representing the expression whose natural logarithm is to be taken. In both cases the `<nl>` node does not represent the entire

```
<nl idx="1">
    <ln>
        <times>
            <variable idx="0"/>
            <variable idx="1"/>
        </times>
    </ln>
</nl>
```

Figure 15: The `<nl>` element for the constraint (3).

constraint expression, but only the part of it that is nonlinear.

Terminal tree elements have special attributes. The `<number>` element represents the literal numerical value given by its `value` attribute, which must be a real number. The `<variable>` element represents a variable whose index is given by the `idx` attribute, which must be a nonnegative integer.

OSiL is intended to accommodate not only the mathematical operators and functions of classical continuous optimization, but other "not linear" functions that can be meaningful in formulations of optimization problems. Over 200 elements are supported in the OSnL schema. The operator and function elements that we selected were drawn from the supported operators and functions in Microsoft Excel, in Content MathML (see, for example, Sandhu [22]), and in standard solvers such as LINDO [18]. Our naming conventions are consistent with MathML and Excel. The OSnL elements fall into the following broad categories:

- ◇ *Arithmetic operator elements:* The standard arithmetic operators such as `plus`, `minus`, `times`, `power`, and `divide`. These operator elements are `OSnLNode` elements that take exactly two children. We also define `sum` and `product` operators that allow an arbitrary number of children. This avoids unnecessary nesting if there is a summation or product with many terms.
- ◇ *Elementary function elements:* Standard elementary functions such as `abs`, `ln`, and `exp`. These are examples of `OSnLNode` elements that have exactly one child that is another `OSnLNode`.
- ◇ *Statistical and probability function elements:* A large variety of statistical and probability functions, especially those relevant to financial, stochastic programming, or operations management applications such as mean, variance, uniform, normal, lognormal, etc.
- ◇ *Operand (terminal) elements:* these include a `number` element and the constant elements `PI`, `E`, `INF`, `TRUE`, `FALSE`, `EULERGAMMA`, `NAN`, and `EPS`. Terminal elements have no child elements.
- ◇ *Variable element:* an element to represent a variable in an optimization instance. A variable element is usually terminal node but not always. See Figure 16 and the ensuing discussion.
- ◇ *Logic and relational operator elements:* Operators to represent numerical relations such as `leq`, `geq`, and logical relations such as `if`, `and`, `or`.

◇ *Trigonometric elements:* All the standard trigonometric functions.

◇ *Special elements:* Representations for user-defined functions, real time data, and XPath data references. These are to be explained in a later paper.

```
<xs:complexType name="OSnLNodeVar">
    <xs:complexContent>
        <xs:extension base="OSnLNode">
            <xs:sequence minOccurs="0" maxOccurs="1">
                <xs:element ref="OSnLNode"/>
            </xs:sequence>
            <xs:attribute name="idx" type="xs:int" use="optional"/>
            <xs:attribute name="coef" type="xs:double" use="optional" default="1"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name="variable" type="OSnLNodeVar" substitutionGroup="OSnLNode"/>
```

Figure 16: Schema for the `<variable>` element.

The OSnL `<variable>` element, defined by the schema in Figure 16, has an attribute `idx` that defines the variable's index and an attribute `coef` that specifies the variable's coefficient. If the `<variable>` element is specified using these attributes then it has no children, and describes a terminal node that behaves like an operand. However, we allow the alternative of specifying the variable's index through an `OSnLNode` child element that evaluates to a positive integer value. This feature is important, for example, in constraint programming where variables may be indexed ("subscripted") by expressions involving variables.

4. The `OSInstance` Class

The OSiL schema defines an XML vocabulary for storing an optimization instance. This instance may persist in a repository of test problems, or it may temporarily be encapsulated in a SOAP envelope for use in a distributed computing environment. However, at some point before the instance is optimized by a solver, it must be stored in memory. Our `OSInstance` class is the in-memory representation of the optimization instance. This class has an API defined by a collection of `get()` methods for extracting various components from the problem instance — such as variables' upper bounds or the coefficients of the linear terms — and a

collection of `set()` methods for modifying or generating an optimization instance. We now describe the close relationship between the OSiL schema and the `OSInstance` class.

4.1. Mapping Rules

As shown in Figure 17, `OSInstance` has member classes `InstanceHeader` and `InstanceData`. These correspond to the complexTypes `instanceHeader` (Figure 4) and `instanceData` (Figure 6), and to the XML elements `<instanceHeader>` (Figure 5) and `<instanceData>` (listed in full in the Appendix).

Moving down one level, Figure 17 also shows that the `InstanceData` class has in turn the member classes `Variables`, `Objectives`, `Constraints`, `LinearConstraintCoefficients`, `QuadraticCoefficients`, and `NonlinearExpressions`, corresponding to the complexTypes in Figure 6 and the elements in Figures 7–9.

```
class InstanceData{
public:
     InstanceData();
     Variables variables;
     Objectives objectives;
     Constraints constraints;
     LinearConstraintCoefficients linearConstraintCoefficients;
     QuadraticCoefficients quadraticCoefficients;
     NonlinearExpressions nonlinearExpressions;
}; // class InstanceData
class OSInstance{
public:
     OSInstance();
     InstanceHeader instanceHeader;
     InstanceData instanceData;
}; //class OSInstance
```

Figure 17: The `OSInstance` class

Figure 18 uses the `Variables` class to provide a closer look at the correspondence between schema and class. On the right, the `Variables` class contains a `number` data member and a sequence of `var` objects of class `Variable`. The `Variable` class has `lb` (double), `ub` (double), `name` (string), `init` (double), and `type` (char) data members. On the left the corresponding XML complexTypes are shown, with arrows indicating the correspondences. The following

rules describe the mapping between the OSiL schema and the `OSInstance` class.

1. Each XML schema complexType corresponds to a class in `OSInstance`. Thus the OSiL schema's complexType `Variable` corresponds exactly to the `Variable` class in `OSInstance`. Elements in the actual XML file then correspond to objects in the `OSInstance` class. For example, the `<var>` element that is of type `Variable` in a file that validates against the OSiL schema corresponds to a `var` object in class `Variable`.

2. An attribute or element used in the definition of a `complexType` is a member of the corresponding in-memory class; moreover the type of the attribute or element matches

Schema complexType / **In-memory class**

```
<xs:complexType name="Variables">  <-----------------------------------------------> class Variables{
                                                                                     public;
  <xs:sequence>                                                                        Variables();
    <xs:element name="var" type="Variable" maxOccurs="unbounded"/>  <-------------->   Variable *var;
  </xs:sequence>
  <xs:attribute name="number" type="xs:positiveInteger"  use="required"/>  <------>   int number;
</xs:complexType>                                                                    }; // class Variables


<xs:complexType name="Variable">  <-----------------------------------------------> class Variable{
                                                                                     public;
                                                                                       Variable();
  <xs:attribute name="name" type="xs:string" use="optional"/>  <------------------>   string name;
  <xs:attribute name="init" type="xs:double" use="optional"/>  <------------------>   double init;
  <xs:attribute name="initString" type="xs:string" use="optional"/>  <------------>   string initString;
  <xs:attribute name="type" use="optional" default="C">  <------------------------>   char type;
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="C"/>
        <xs:enumeration value="B"/>
        <xs:enumeration value="I"/>
        <xs:enumeration value="S"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="lb" type="xs:double" use="optional" default="0"/>  <-------->   double lb;
  <xs:attribute name="ub" type="xs:double" use="optional" default="INF"/>  <------>   double ub;
</xs:complexType>                                                                    }; // class Variable
```

OSiL elements

```
<variables number="2">
   <var lb="0" name="x0" type="C"/>
   <var lb="0" name="x1" type="C"/>
</variables>
```

In-memory objects

```
OSInstance osintance;
osinstance.instanceData.variables.number=2;
osinstance.instanceData.variables.var=new Var[2];
osinstance.instanceData.variables.var[0].lb=0;
osinstance.instanceData.variables.var[0].name=x0;
osinstance.instanceData.variables.var[0].type=C;
osinstance.instanceData.variables.var[1].lb=0;
osinstance.instanceData.variables.var[1].name=x1;
osinstance.instanceData.variables.var[1].type=C;
```

Figure 18: The `<variables>` element as an `OSInstance` object

the type of the member. In Figure 18, for example, `lb` is an attribute of the OSiL `complexType` named `Variable`, and `lb` is a member of the `OSInstance` class `Variable`; both have type `double`. Similarly, `var` is an element used in the definition of the OSiL `complexType` named `Variables`, and `var` is a member of the `OSInstance` class `Variables`; the `var` element has type `Variable` and the `var` member is a `Variable` object.

3. A schema sequence corresponds to an array. For example, in the excerpt from the OSiL schema shown on the left side of Figure 18, the complexType `Variables` has a sequence of `<var>` elements that are of type `Variable.` In the `OSInstance` class shown on the right side of the figure, the corresponding `Variables` class has a member that is an array of type `Variable`.

General nonlinear terms are stored in-memory as `OSExpressionTree` objects, which are the subject of the next section.

The `OSInstance` class has a set of `get()` and `set()` methods that act as an API for the optimization instance. The `get()` methods are used by other classes to access data in an existing object. For example, an OSiL instance is parsed by the `OSiLReaderClass` and a corresponding `osinstance` is returned by the method `getOSInstance()`. Then the nonzero coefficient values in the linear terms of each constraint are retrieved in column major form by

```
osinstance->getLinearConstraintCoefficientsInColumnMajor()->values
```

and in other forms by analogous calls. Similarly, the `set()` method provides an API for creating or modifying an `OSInstance`.

4.2. Nonlinear Expressions

A key part of the `OSInstance` class is the `OSExpressionTree` class which is the in-memory representation of the nonlinear terms. Our `OSExpressionTree` is designed to allow for the efficient parsing of an OSiL instance and to provide an API for a solver interface through `get()` methods. These `get()` methods allow the solver to extract the nonlinear expressions in postfix, infix, or prefix formats, or to request function and gradient evaluations at specific iterates via callbacks.

Consider again our example. Conceptually, an expression tree for the nonlinear part of the objective function (1) has the form illustrated in Figure 10. The choice of a data

structure to store the expression tree — for use by the API methods — is a key aspect in the design of the `OSInstance` class.

One approach uses a C-style structure for each node in the expression tree. The ASL data structure [14] of the AMPL modeling language [10, 11] has this form. The structure stores information as to operator or operand type, along with pointers to child nodes, and a tree-walking method is used to perform operations on the expression tree such as function or derivative evaluations. Figure 19 illustrates the essential idea (from [9]); `expr` is a C-structure, `*e` a pointer to the root node of the expression tree, and `opnum` is an integer value denoting the node type. A fundamental problem with this approach is that every method that operates on the expression tree requires a whole series of `switch` statement `case` clauses, making for a very large function. Updating the code to reflect new operators requires modifying every such function, and hence risking the introduction of errors.

```
double evaluate_function (expr *e, double x[]){
    ...
    opnum = e->op
    switch(opnum){
        case PLUS_opno: ...
        case MINU_opno: ...
        ...
    }
}
```

Figure 19: Sample Code for Parsing an OSiL Instance Without Polymorphism

A second approach is to use an object-oriented language such as C++ or Java and to define a class for each type of node in the expression tree: a "plus" node class, a "minus" node class, an "exponential function" class, and so forth. So that switches and complicated logic can be avoided as much as possible, this is implemented by having each node class extend a single fundamental node class, `OSnlNode`, using the object-oriented concept of polymorphism.

4.2.1. The `OSnlNode` Class

All of an instance file's operator and operand elements used in defining a nonlinear term are extensions of the base element type `OSnLNode`. There is an element class `OSnLNodePlus`, for

example, that extends `OSnLNode`; then in an OSiL instance file, there are `<plus>` elements that are of type `OSnLNodePlus`.

The Java libraries that support the `OSInstance` API follow an identical design and are based on the Java abstract class `OSnlNode`. Every element in an OSiL instance corresponds to a Java class that derives from the `OSnlNode` class. Thus we can construct an expression tree of homogenous nodes, and methods that operate on the expression tree to calculate function values, derivatives, postfix notation, and the like do not require switches or complicated logic.

```
protected double calculateFunction(double[] x){
   m_dFunctionValue =
      m_mChildren[0].calculateFunction(x) +
      m_mChildren[1].calculateFunction(x);
   return m_dFunctionValue;
} //calculateFunction
```

Figure 20: The function calculation method for the "plus" node class

As an example, the abstract class `OSnlNode` has an abstract method `calculateFunction` that takes an array of `double` values corresponding to decision variables, and evaluates the expression tree for those values. Every class that extends `OSnlNode` must implement this method. For instance the `calculateFunction` method for the `OSnlNodePlus` class is shown in Figure 20. Because the OSiL instance file must be validated against its schema, and in the schema each `<OSnlNodePlus>` element is specified to have exactly two child elements, this `calculateFunction` method can assume that there are exactly two children of the node that it is operating on. Thus through the use of polymorphism and recursion the need for switches like those in Figure 19 is eliminated. This design makes adding new operator elements easy; it is simply a matter of adding a new class and implementing the `calculateFunction()` method for it.

4.2.2. The `OSExpressionTree` Class

It is necessary to parse an OSiL instance file or string in order to create the corresponding in-memory `OSExpressionTree` object that consists of `OSnLNode` objects. In the Java library implementation we use the Apache Xerces XML parser, which creates an instance of the W3C Document Object Model (DOM) data structure. The DOM data structure is tree-like and consists of `Element` nodes, `Attribute` nodes, and `Text` nodes. Unfortunately, this data

structure is not conducive for use by optimization solver programming interfaces. Reading a DOM tree to make function evaluations, calculate derivatives, convert an instance into its postfix representation, and perform similar tasks requires complicated logic and is not computationally efficient. Therefore, after reading the OSiL instance into the DOM tree, the `OSInstance` API converts the DOM tree into an object in the `OSExpressionTree` class that is designed specifically for optimization applications. (For the C++ libraries we are writing our own parser, which will write `OSExpressionTree` objects directly.)

Each `Element` node in a DOM tree is an object in the DOM `Element` class. Each DOM `Element` is converted into an `OSnlNode` of an `OSExpressionTree`. By design, each `OSnLNode` type defined in the OSnL schema corresponds to an `OSnLNode` class in the API library. That is, there is a one-to-one mapping between elements in the DOM that define nonlinear terms in the problem instance, and the `OSnlNode` objects in the `OSExpressionTree`. For example, a `<plus>` element tag in the OSiL instance is instantiated as an `OSnlNodePlus` object in an `OSExpressionTree`.

There are several approaches to convert the DOM nodes into `OSExpressionTree` nodes. One approach is to loop over the appropriate DOM nodes, get the name of the element, and through a series of switches, create the appropriate `OSnLNode` class. A second, easier approach, is to use RTTI (Run-Time Type-Identification) [7]. Using RTTI is very easy in Java. There is a static method `forName` in the Java `Class` object that returns a `Class` reference at run time. For example, executing the statement

```
Class nlNodeClass = Class.forName("OSnLNodeTimes");
```

returns a `Class` reference. An actual object in the `OSnLNodeTimes` class is created by executing the statement `OSnLNode nlNode = (OSnLNode)nlNodeClass.newInstance()`.

The conversion of every DOM element node into an OS Expression Tree objects is accomplished using the method `convertElementToNlNode`, whose code is shown in Figure 21.

In the first line of the code in Figure 21, a generic object `nlNode` in the abstract class `OSnLNode` is created. The string `sNodeName` is given the name of the concrete class corresponding to the appropriate element in the line `sNodeName = ele.getLocalName()`. The string `sNlNodeClass` contains the fully qualified class name (that is, the package name is included). Then the line `Class nlNodeClass = Class.forName(sNlNodeClass)` gets a reference for the class `sNlNodeClass`. Next the line

```
OSnLNode nlNode = null;
String sNodeName = "";
try{     sNodeName = ele.getLocalName();
    String sNlNodeClass = m_sPackageName + "." + m_sNlNodeStartString +
        sNodeName.substring(0, 1).toUpperCase() + sNodeName.substring(1);
    Class nlNodeClass  = Class.forName(sNlNodeClass);
    nlNode = (OSnLNode)nlNodeClass.newInstance();
} // now process attributes
```

Figure 21. Code from the `convertElementToNlNode()` method for converting a DOM element into an `OSnLNode` object.

```
if(sNodeName.equalsIgnoreCase("number")){
    String sType = ele.getAttribute("type");
    if(sType != null && sType.length() > 0){
        ((OSnLNodeNumber)nlNode).setNumberType(sType);
    }
    else{
        ((OSnLNodeNumber)nlNode).setNumberType("real");
    }
    String sValue = ele.getAttribute("value");
    if(sValue != null || sType.length() > 0){
        ((OSnLNodeNumber)nlNode).setNumberValue(Double.parseDouble(sValue));
    }
    else{
        ((OSnLNodeNumber)nlNode).setNumberValue(0);
    }
}
```

Figure 22: Logic for Processing Attributes of a `Number` element

```
nlNode = (OSnLNode)nlNodeClass.newInstance();
```

creates at runtime an actual object in one of the concrete classes that extend `OSnLNode`. It is important that the `sNlNodeClass` string be the name of a concrete class as this is the class that gets created. Note the lack of switches or any complicated logic in the code.

Unfortunately there are more than just `Element` nodes in the DOM tree. There are also `Attribute` nodes with associated `Text` nodes. These `Attribute` nodes do require additional logic switches. For example, the processing of the attributes `type` and `value` for the `number` element is illustrated in Figure 22. The logic in this figure is part of the

code for the `convertElementToNlNode()` method in Figure 21. The key aspect of this logic is to extract the actual double value for this number from the attribute `real`, and to use the `setNumberValue` method of the `OSnLNodeNumber` class to set the value in the `OSExpressionTree` object so that it can be used for further evaluation.

Thus there is an `OSnLNode` object created for each `OSnLNode` element in the DOM tree. For each `<nl>` element in an OSiL instance, an `OSExpressionTree` object is created. This object's root element is an `OSnLNode` with an array `OSnLNode[]` of child nodes.

5. Future Work

The research described in this paper is continuing in three directions.

First, we are extending the OSiL schema to include other classes of optimization problems, such as optimization under uncertainty, semidefinite and cone programming, constraint programming, disjunctive programming, and optimization using simulation. Each class has its own special features that pose new kinds of challenges.

A second direction is the ongoing work to design and build open-source libraries in popular programming languages that implement classes for parsing and writing optimization instances in the OSiL format. These libraries derive from the `OSInstance` object and methods described in the preceding section. Examples in this paper are based on a full implementation in Java and an implementation in C++ lacking only the nonlinear features.

Finally, OSiL is only one schema in a much larger OS — Optimization Services — framework. Of most immediate concern, the OS framework is planned to incorporate a language OSoL for passing algorithmic options to solvers, and a language OSrL for passing results back to modelers and modeling systems. These languages are an initial part of a complete system that allows for platform-independent synchronous and asynchronous communication between client modeling systems and optimization solvers in a distributed environment.

6. Appendix

This is the complete OSiL representation for the optimization problem given in (1)–(4).

```
<?xml version="1.0" encoding="UTF-8"?>
<osil xmlns="os.optimizationservices.org">
     <instanceHeader>
          <name>Modified Rosenbrock</name>
          <source>Computing Journal 3:175-184, 1960</source>
          <description>Rosenbrock problem with constraints</description>
     </instanceHeader>
     <instanceData>
          <variables number="2">
               <var lb="0" name="x0" type="C"/>
               <var lb="0" name="x1" type="C"/>
          </variables>
          <objectives number=" 1">
               <obj maxOrMin="min" name="minCost" numberOfObjCoef="1">
                    <coef idx="1">9.0</coef>
               </obj>
          </objectives>
          <constraints number="2">
               <con ub="25.0"/>
               <con lb="10.0"/>
          </constraints>
          <linearConstraintCoefficients numberOfValues=" 3">
               <start>
                    <el>0</el><el>2</el><el>3</el>
               </start>
               <rowIdx>
                    <el>0</el><el>1</el><el>1</el>
               </rowIdx>
               <value>
                    <el>1.0</el><el>7.0</el><el>5.0</el>
               </value>
          </linearConstraintCoefficients>
          <quadraticCoefficients numberOfQuadraticTerms="3">
               <qTerm idx="0" idxOne="0" idxTwo="0" coef="10"/>
               <qTerm idx="0" idxOne="1" idxTwo="1" coef="11"/>
               <qTerm idx="0" idxOne="0" idxTwo="1" coef="3"/>
          </quadraticCoefficients>
```

```
        <nonlinearExpressions number="2">
             <nl idx="-1">
                  <plus>
                       <power>
                            <minus>
                                 <number type="real" value="1.0"/>
                                 <variable coef="1.0" idx="0"/>
                            </minus>
                            <number type="real" value="2.0"/>
                       </power>
                       <times>
                            <power>
                                 <minus>
                                      <variable coef="1.0" idx="0"/>
                                      <power>
                                           <variable coef="1.0" idx="1"/>
                                           <number type="real" value="2.0"/>
                                      </power>
                                 </minus>
                                 <number type="real" value="2.0"/>
                            </power>
                            <number type="real" value="100"/>
                       </times>
                  </plus>
             </nl>
             <nl idx="1">
                  <ln>
                       <times>
                            <variable coef="1.0" idx="0"/>
                            <variable coef="1.0" idx="1"/>
                       </times>
                  </ln>
             </nl>
        </nonlinearExpressions>
   </instanceData>
</osil>
```

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1986.

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

[3] Gordon Bradley. Network and graph markup language (NaGML) — data file formats. Technical Report NPS-OR-04-007, Department of Operations Research, Naval Postgraduate School, Monterey, CA, USA, 2004. Available from the author, `bradley@nps.navy.mil`.

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

[5] J. Czyzyk, M.P. Mesnier, and J.J. Moré. The NEOS server. *IEEE Journal on Computational Science and Engineering*, 5:68–75, 1998.

[6] Elizabeth D. Dolan, Robert Fourer, Jorge J. Moré, and Todd S. Munson. Optimization on the NEOS server. *SIAM News*, 35(6):4, 8–9, July/August 2002.

[7] B. Eckel. *Thinking in JAVA*. Prentice-Hall, Upper Saddle River, NJ, second edition, 2000.

[8] O.C. Ezechukwu and I. Maros. OOF: open optimization framework. Technical Report ISSN 1469-4174, Department of Computing, Imperial College of London, London, UK, 2003.

[9] R. Fourer and D.M. Gay. Extending an algebraic modeling language to support constraint logic programming. *INFORMS Journal on Computing*, 14:322–344, 2002.

[10] R. Fourer, D.M. Gay, and B.W. Kernighan. A modeling language for mathematical programming. *Management Science*, 36:519–554, 1990.

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

[12] R. Fourer, L. Lopes, and K. Martin. LPFML: A W3C XML schema for linear and integer programming. *INFORMS Journal on Computing*, 17:139–158, 2005.

[13] Robert Fourer. Modeling languages versus matrix generators for linear programming. *ACM Transactions on Mathematical Software*, 9:143–183, 1983.

[14] D.M. Gay. Hooking Your Solver to AMPL (revised 1994, 1997). Technical report, Bell Laboratories (Murray Hill, NJ), 1993.

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

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

[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] Lindo Systems, Inc. LINDO API user's manual. Technical report, Lindo Systems, Inc., 2002. `http://www.lindo.com/lindoapi_pdf.zip`.

[19] Robin Lougee-Heimer. The Common Optimization INterface for operations research. *IBM Journal of Research and Development*, 47(1):57–66, 2003.

[20] J. Ma. Optimization services (OS), a general framework for optimization modeling systems, 2005. Ph.D. Dissertation, Department of Industrial Engineering & Management Sciences, Northwestern University, Evanston, IL.

[21] H.H. Rosenbrock. An automatic method for finding the greatest or least value of a function. *Comp. J.*, 3:175–184, 1960.

[22] Pavi Sandhu. *The MathML handbook*. Charles River Media, Hingham, MA, 2003.

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