Comput Optim Appl (2010) 45: 181–203
DOI 10.1007/s10589-008-9169-6

OSiL: An instance language for optimization

Robert Fourer · Jun Ma · Kipp Martin

Received: 31 January 2006 / Revised: 30 August 2007 / Published online: 25 January 2008

Abstract 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 common to separate modeling languages from optimization solvers. In a completely distributed environment, 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

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

R. Fourer · J. Ma
Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL 60208, USA

R. Fourer
e-mail: 4er@iems.northwestern.edu

J. Ma
e-mail: maj@iems.northwestern.edu

K. Martin (✉)
Graduate School of Business, University of Chicago, Chicago, IL 60637, USA
e-mail: kipp.martin@chicagogsb.edu

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.

Keywords Linear programming · Nonlinear programming · Modeling languages · Information systems · Web services · XML

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 might 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 [14]. 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 instance input formats, as can be seen by even a cursory look at the list of solvers available on the NEOS Server ([7, 9] 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 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—Optimization Services instance Language—a text-based format powerful and flexible enough to represent continuous and mixed-integer instances of linear, quadratic, and very general nonlinear programs. Its underlying principles are sufficiently powerful to allow for future extensions to such problem areas as cone, constraint, disjunctive, and stochastic programming. Thus it has the potential to serve as a new standard that subsumes the many currently used formats for optimization instances.

OSiL's definition as a so-called 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 a corresponding data structure and application programming interface (or API) that provide solvers with a standard way of extracting instance data from files written in OSiL. 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.

The demands of web services, object-oriented programming, and nonlinear problem types make the design of such a standard problem-instance representation much more challenging than the design of formats for linear problems that were created 50 years ago with punch cards in mind. Thus the design of OSiL has properly been the focus of a research project that has extended over several years. This paper is not intended to be a user's guide to OSiL, but rather a description of the major features and key design decisions that were necessary to make OSiL workable. A user's manual is available at `projects.coin-or.org/OS`. Other documentation is available at `www.optimizationservices.org`.

1.1 Related work

A number of new standards for instance representation 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 [18]. We are also not the first to incorporate XML into this area. Fourer, Lopes and Martin [13] propose the LPFML XML schema for representing instances of mixed-integer linear programs; Chang [6] and Kristjánsson [19] have also proposed XML representations for linear programming instances. Ezechukwu and Maros [10] describe an Algebraic Markup Language that uses XML to describe the model rather than the instance, and Bradley [5] introduces an XML markup grammar for networks. See also [4] for a good overview of the uses of XML technologies in operations research.

One potential alternative to OSiL is Content MathML [25], which is also an XML vocabulary capable of representing general nonlinear optimization. It is not specifically designed for representing optimization problem instances, however, and after some study we concluded that it would not be useful. For example, Content MathML is considerably more verbose than OSiL when representing optimization instances. This is particularly true when a large part of the model is linear or quadratic. Furthermore, Content MathML does not contain key optimization constructs. For example, in Content MathML there is no built-in element to represent a decision variable with a coefficient and an index. Content MathML is not under control of the optimization community. This is perhaps the single most important reason not to use MathML. We can add optimization-related features to OSiL as needed. Using MathML to support optimization features is awkward at best, and it is unlikely we can get the W3C to adopt optimization-specific features in a timely fashion. Control of a standard for optimization is better left to an organization under the control of the Operations Research community. See [23] for considerably more detail on this decision. Another di-

alect, CapeML (A Common Model Exchange Language for Chemical Process Modeling [3]), has been used in a system for solving dynamic optimization problems via automatic differentiation, but is very domain specific and not designed for broad classes of optimization. In summary, MathML is too broad and CapeML is too specific.

New API proposals have also been a subject of recent activity. The extensive COIN-OR (COmputational INfrastructure for Operations Research) project (Lougee-Heimer [21] 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 [18]; 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 [20]. 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.

1.2 Outline

After providing the requisite background for XML in the next section, we present the OSiL language in detail in Sect. 3. We first describe aspects of the language common to all optimization 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 Sect. 3.4.

An OSiL representation of an optimization problem instance may persist in a repository of test problems, or may be encapsulated in a SOAP envelope for use in a distributed computing environment. (SOAP stands for Simple Object Access Protocol—a high-level networking protocol for encapsulating XML data.) To be useful to solvers, however, a problem instance represented in OSiL must at some point have an analogous in-memory representation. In Sect. 4 we describe the object class `OSInstance` that we have designed for this purpose. The `OSInstance` class has an API consisting of `get()` and `set()` methods that are used for extracting components of an optimization instance, or for creating and modifying instances in memory. The `OSInstance` class also has a set of `calculate()` methods for calculating function values and derivatives (both gradients and Hessians). 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 Sect. 4.2, `OSExpressionTree` is designed to allow for easy extraction of expressions in postfix, prefix, and infix formats amenable to solvers, and also to facilitate function and derivative evaluation as required by solvers during their execution.

In Sect. 5 we conclude by describing several directions in which this research is continuing. We are extending OSiL to other problem types, designing and building

open-source libraries for reading and writing OSiL, and pursuing a broader Optimization Services research agenda, described by Ma [23], which provides a host of similarly-named XML languages for use in optimization over the Internet. The resulting software is available as a project at the COIN-OR open-source repository for operations research software. See `projects.coin-or.org/OS`.

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 [13]. See also the excellent general overview of XML by Skonnard and Gudgin [26]. This section introduces aspects of XML relevant to this paper, including the basics of XML files, parsing, and schemas.

2.1 XML files

OSiL, our proposed standard, stores optimization problem instances as XML files. Consider the following problem instance that is a modification of an example of Rosenbrock [24]:

$$\begin{aligned} \text{Minimize} \quad & (1-x_0)^2 + 100(x_1 - x_0^2)^2 + 9x_1 && (1)\\ \text{Subject to} \quad & x_0 + 10.5x_0^2 + 11.7x_1^2 + 3x_0x_1 \leq 25 && (2)\\ & \ln(x_0x_1) + 7.5x_0 + 5.25x_1 \geq 10 && (3)\\ & x_0, x_1 \geq 0. && (4) \end{aligned}$$

We introduce XML by use of illustrations from the corresponding OSiL representation of this problem.

There are two continuous variables, x_0 and x_1, in this instance, each with a lower bound of 0. Figure 1 shows how we propose to represent this information in an XML-based OSiL file. Like all XML files, this is a text file that contains both *markup* and *data*. In this case 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 Fig. 1 and that consequently takes the default value `"INF"` (denoting ∞).

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

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

2.2 XML schemas

To be useful for communication between solvers and modeling languages, OSiL instance files must conform to a standard. An XML-based representation standard is imposed through the use of a *W3C XML Schema.* The W3C, or World Wide Web Consortium (`www.w3.org`), promotes standards for the evolution of the web and for interoperability between web products. XML Schema (`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 thus 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. An XML file that conforms to a schema is called *valid* for that schema.

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. Just as a class in C++ very explicitly describes member and method names and properties, a schema explicitly describes element and attribute names and properties.

Figure 2 is a piece of our schema for OSiL. In W3C XML Schema jargon, it defines a *complexType,* whose purpose is to specify elements and attributes that are allowed to appear in a valid XML instance file such as the one excerpted in Fig. 1. In particular, Fig. 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 Fig. 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 (such as `Variable`) 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. Thus we will often refer to complexTypes as classes and to elements as objects. This object-oriented analogy continues in Sect. 4 where we define an `OSInstance` class that is the in-memory analogue of an OSiL instance file.

Our definition of the `Variable` complexType allows only the attributes listed in Fig. 3 to be present in a `<var>` element. All of these attributes are specified as optional. Characteristics 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

```
<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>
```

Fig. 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>
```

Fig. 3 The `Variable` complexType in the OSiL schema

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 Sect. 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. Useful as validation is, however, it is concerned with *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 collections of quadratic terms and more general nonlinear expressions. 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.

We describe OSiL in general terms via its XML schema, and more specifically by use of the example given by (1)–(4). The actual schema file for OSiL resembles the excerpts in Figs. 2 and 3, except that it is much longer and harder to read; it may be found at `www.optimizationservices.org/schemas/OSiL.xsd`. In this paper we depict the schema and its parts by means of tree diagrams, with schema elements and attributes as the nodes. Specialized schema development software lets people work directly with these graphical representations.

3.1 OSiL basics

Figure 4 depicts the top two levels of the OSiL schema in tree-diagram form, as drawn by the `<oXygen/>` XML Editor (`http://www.oxygenxml.com/`). Solid rectangles

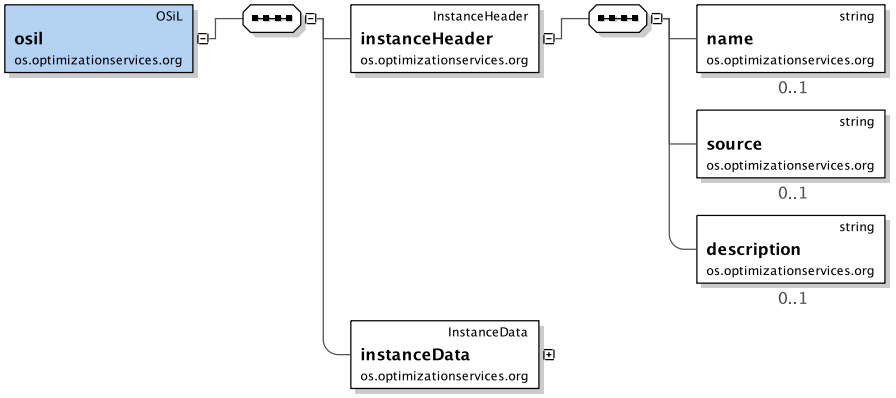

Fig. 4 The top two levels of the OSiL schema, in tree-diagram form

```
<?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>
        .......
    </instanceData>
</osil>
```

Fig. 5 The root and `<instanceHeader>` elements for instance (1)–(4)

depict elements. Thus the `<osil>` root element is seen to have two child elements. The first such element, `<instanceHeader>`, and its child elements are depicted in Fig. 4, and the corresponding parts of the OSiL file for the instance (1)–(4) are shown in Fig. 5.

In Fig. 4 the "ellipsoid" icon containing four small rectangles denotes a *sequence* of elements. Thus the `<instanceHeader>` element consists of a sequence of `<name>`, `<source>`, and `<description>` elements. A plain rectangle indicates an element that must appear exactly once, while a `0..1` icon beneath a rectangle indicates an optional element that may appear at most once. Thus Fig. 4 implies that a valid OSiL file must have exactly one `<instanceHeader>` element, which may optionally have one child `<name>` element.

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

The first child element of `<instanceData>` is a `<variables>` element that comprises one `numberOfVariables` attribute and a sequence of `<var>` elements, one for each variable in the problem instance. Attributes are distinguished from elements by dotted boxes and the symbol `@`. Attributes of the `<var>` elements provide standard information such as domain type, bounds, and optional names, as illustrated for our

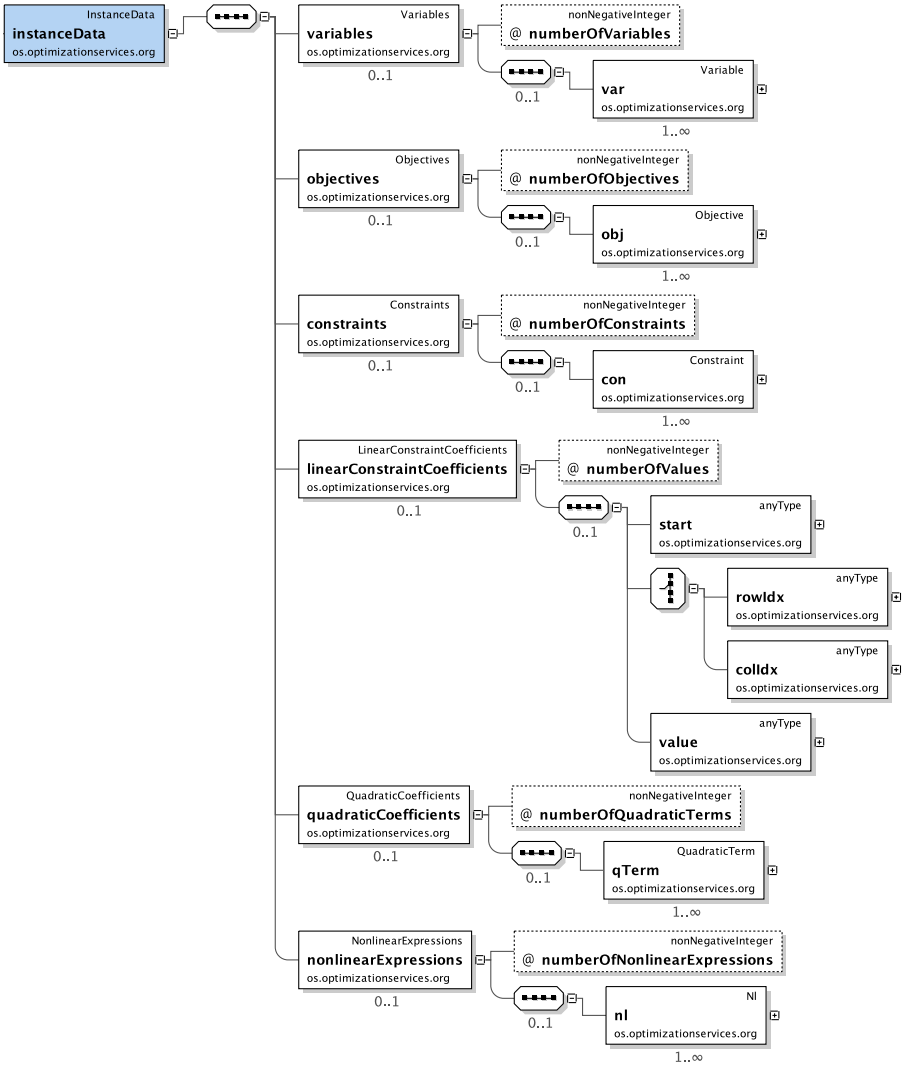

Fig. 6 Detail of the `instanceData` element

example in Fig. 7. (We omit the schema diagrams for `<var>` and other elements that have no significant child elements.)

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 an objective function has linear terms,

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

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

these are stored in the <coef> child elements of the <obj> element. For example, in Fig. 7 the linear term $9x_1$ from the objective function (1) is represented by <coef idx="1">9</coef>, with the attribute idx="1" indicating that x_1 is the variable whose coefficient is being specified as 9.

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 Fig. 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.

Subsequent child elements of <instanceData> refer to the variables, objectives, and constraints by index number. Variables and constraints are numbered increasing from 0 according to the order in which they are defined by the <variables> and <constraints> elements; objectives are similarly numbered but decreasing from −1. Each constraint and objective thus has a unique index number, which will be seen in Sects. 3.3 and 3.4 to be convenient for locating quadratic and nonlinear terms. We considered requiring identification of variables, objectives, and constraints by name rather than by index. However, we decided that the resulting representation would be too verbose and thus names are treated as optional attributes. The main advantage of requiring names rather than indices would be to help people read OSiL files, but OSiL is intended to be read by computers rather than by people.

The <variables> element is required and must contain at least one <var> element, as indicated by the 1..∞ icon beneath the rectangle for the latter in Fig. 6. 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 Sect. 3.4.

3.2 Representing linear constraints

Almost invariably, most of the linear constraint coefficients in a large optimization problem are zero. Thus OSiL must adopt the usual approach of recording only the nonzeros. This is done by use of Fig. 6's <linearConstraintCoefficients> element, which stores the coefficient matrix using three arrays as proposed in the earlier LPFML schema [13]. There is a child element of <linearConstraintCoefficients> to represent each array: <value> for an array of nonzero coefficients,

Fig. 8 The `<linearConstraintCoefficients>` element for constraints (2) and (3)

```
<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.</el><el>7.5</el><el>5.25</el>
    </value>
</linearConstraintCoefficients>
```

`<rowIdx>` or `<colIdx>` for a corresponding array of row indices or column indices, and `<start>` for an array that indicates where each row or column begins in the previous two arrays.

The *choice* element group icon immediately preceding the `<rowIdx>` and `<colIdx>` element rectangles indicates that only one of these may appear in any valid OSiL representation. Thus the `<linearConstraintCoefficients>` element may represent the constraint coefficients either by column or by row. With modern solvers offering to set up the dual problem or to accelerate calculations by using both row-wise and column-wise lists, it makes sense to allow for row-wise as well as column-wise specification of coefficients in a new standard.

Individual array entries are specified by `<el>` children (not shown in Fig. 6) of `<value>`, `<rowIdx>` or `<colIdx>`, and `<start>`. The `<linearConstraintCoefficients>` element for the constraints (2) and (3) of our example is shown (using the column-wise option) in Fig. 8. There are three linear coefficients, one from the linear part of the first constraint $x_0 + 10.5x_0^2 + 11.7x_1^2 + 3x_0x_1 \leq 25$, and two from the linear part of the second constraint $\ln(x_0x_1) + 7.5x_0 + 5.25x_1 \geq 10$.

3.3 Representing quadratic terms

Any quadratic expression is easily represented as a general nonlinear expression using the format to be described in Sect. 3.4. Nevertheless there are good reasons for including a special quadratic expression representation in OSiL. Each quadratic term admits a particularly compact representation as a list of index-index-value triples. This representation is also readily recognized by software that classifies or analyzes optimization problems. Moreover there are numerous specialized solvers for the case in which the objective and all constraints are 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 accept 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.

As seen in Fig. 6, OSiL represents quadratic terms by `<qTerm>` child elements of the `<quadraticCoefficients>` element. A `<qTerm>` element has four required attributes: an integer that indicates the constraint (if nonnegative) or objective to which the term belongs, two nonnegative integers that specify the indices of the variables in the quadratic term, and a floating-point number that is the coefficient of the term. Figure 9 illustrates a representation of the quadratic term for constraint (2) of our example.

```
<quadraticCoefficients numberOfQuadraticTerms="3">
     <qTerm idx="0" idxOne="0" idxTwo="0" coef="10.5"/>
     <qTerm idx="0" idxOne="1" idxTwo="1" coef="11.7"/>
     <qTerm idx="0" idxOne="0" idxTwo="1" coef="3."/>
</quadraticCoefficients>
```

Fig. 9 The <quadraticCoefficients> element for constraint (2)

We could instead have designed the quadratic representation to be more like the linear one, with two arrays of variable indices, an array of coefficients, and an array of pointers to indicate where each objective or constraint begins. Instead we decided to adopt this more flexible approach, which is analogous to the representation that we chose for nonlinear terms.

3.4 Representing nonlinear terms

General nonlinear terms are represented by the final child of <instanceData> in Fig. 6, the <nonlinearExpressions> element. Our goal for this part of OSiL is to represent a comprehensive collection of optimization problems while keeping parsing relatively simple.

One of our most challenging design issues was to provide general nonlinear expressions with an XML-based representation that could be easily validated against a schema. Clearly an expression tree or the equivalent is required for representing each nonlinear term in a problem instance. As an example, Fig. 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 Fig. 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. Nevertheless, 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, in the XML schema for OSiL is illustrated in Fig. 12.

The first line of Fig. 12 defines OSnLNode to be an abstract element type. Think of it as a "template" for a nonlinear element; in C++ an analogous statement would be

```
class OSnLNode{ ...
}
```

The following group of lines in Fig. 12 defines OSnLNodePlus to be an "extension" of the "base" OSnLNode abstract type; the analogous C++ statement would be

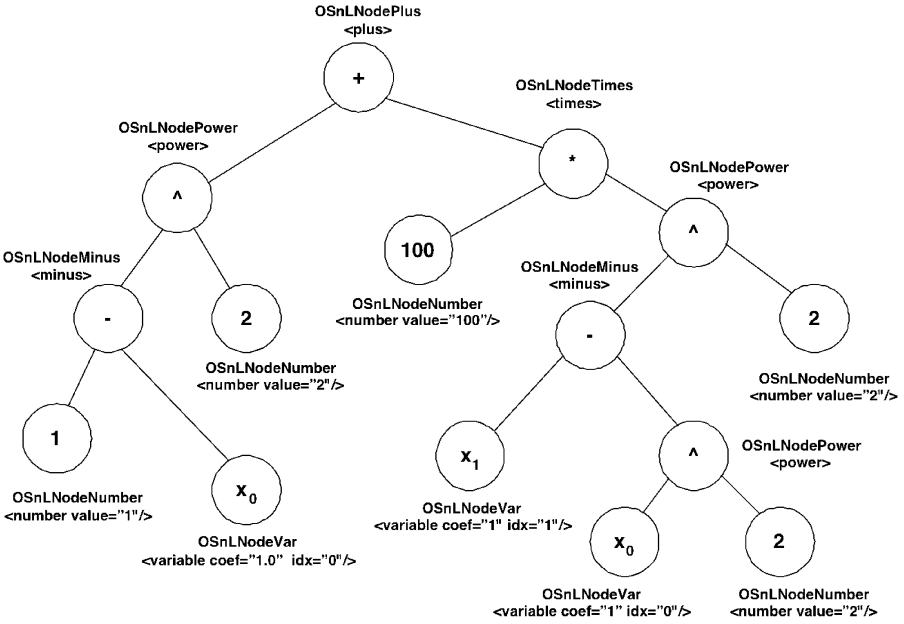

Fig. 10 Conceptual expression tree for the nonlinear part of the objective (1)

```
<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>
```

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

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

This definition of `OSnLNodePlus` also specifies that it must have exactly two children, each of which must be an `OSnLNode`. Thus we avoid the problem of explicitly

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

<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>

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

Fig. 12 Defining and using `OSnLNode` and its extension to `OSnLNodePlus`

Fig. 13 The `<nl>` element for the constraint (3)

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

listing every type of nonlinear operator and function that may be a child of `OSnLNodePlus`. Finally, in the last line of Fig. 12 the actual element `<plus>` is declared, to be of the type `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 C++ would be `OSnLNodePlus *plus = new OSnLNodePlus();`.

Armed with the `OSnLNode` concept, we can represent each parse tree for a nonlinear expression in an optimization instance as an OSiL element that is easily validated against a schema. As seen in Fig. 6, the `<nonlinearExpressions>` element contains a sequence of `<nl>` children, one for each constraint and objective that has nonlinear terms. The `idx` attribute of each `<nl>` child identifies its corresponding objective function (if negative) or constraint. Each `<nl>` element also has a single child element, of type `OSnLNode`, that specifies the required nonlinear expression.

Figures 11 and 13 illustrate the `<nl>` elements for the nonlinear parts of the objective function (1) and constraint (3), respectively. The OSiL XML representation in Fig. 11 corresponds exactly to the structure of the parse tree for the expression in Fig. 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 Fig. 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 Fig. 13, the `<nl>` element with index 1 (corresponding to the second constraint) has a single child element `<ln>` whose child represents the expression whose natural logarithm is to be taken.

Terminal tree elements have special attributes. The `<number>` element represents the literal numerical value given by its `value` attribute, which must be a floating-

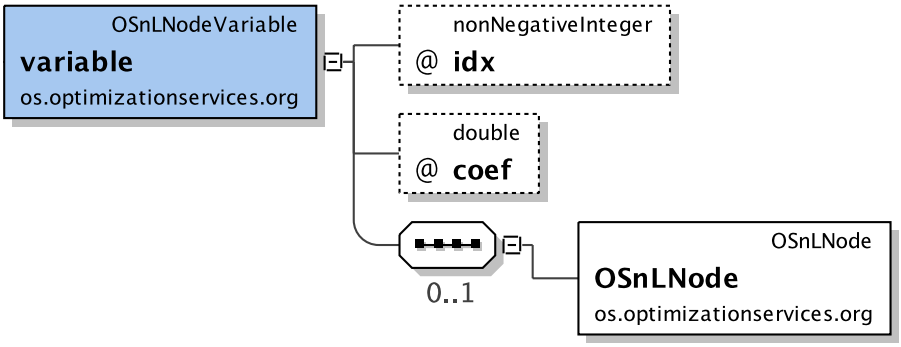

Fig. 14 Detail of the <variable> element, showing its optional child

point number. The <variable> element represents a variable whose index is given by the nonnegative integer idx attribute; an optional floating-point coef attribute is intended mainly for linear sub-expressions of nonlinear terms.

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. The OSiL schema defines over 200 OSnLNode elements, which fall into the following broad categories:

- ◇ *Arithmetic operator elements*: Standard binary arithmetic operators such as <plus>, <times>, <power>, and <divide>, which take exactly two OSnLnode children. We also define <sum> and <product> operators that allow an arbitrary number of children, to avoid unnecessary nesting when there is a summation or product with many terms.
- ◇ *Elementary function elements*: Standard functions such as <abs>, <ln>, <exp>, <sin>, and <arctan>, which have exactly one OSnLNode child.
- ◇ *Statistical and probability function elements*: A large variety of statistical and probability functions such as <mean>, <variance>, <uniform>, <normal>, and <lognormal>, especially those relevant to applications in finance and operations management.
- ◇ *Operand* (*terminal*) *elements*: The previously mentioned <number> element and the constant elements <PI>, <E>, <EULERGAMMA>, <TRUE>, <FALSE>, <EPS>, <INF>, and <NAN>. These have no child elements.
- ◇ *Variable element*: The <variable> element that represents decision variables in an optimization instance. It may be a terminal element as previously described, or may have a more general form (see Fig. 14) in which the index is specified through an OSnLNode child element that evaluates to a nonnegative integer. The latter is important to constraint programming formulations [22, 27] whose variables may be indexed ("subscripted") by expressions involving variables.
- ◇ *Logic and relational operator elements*: Operators to represent numerical relations such as <leq> and <gt>, and logical relations such as <if>, <and>, and <or>.
- ◇ *Special elements*: Representations for user-defined functions, real time data, and XPath data references, to be detailed in subsequent extensions.

We have drawn operator and function elements from the supported operators and functions in Microsoft Excel, in Content MathML (Sandhu [25]), and in standard solvers such as LINDO [20]. Our naming conventions are consistent with MathML and Excel.

In the OSiL representation of (1)–(4) we have taken the linear and quadratic terms in general nonlinear constraints and put them respectively into the `<linearConstraintCoefficients>` and `<quadraticCoefficients>` sections. This may be desirable for taking advantage of structure, however this is not necessary and the linear and quadratic terms can all be put into the `<nonlinearExpressions>` section.

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 an optimization instance. This class has an API defined by a collection of `get()` methods for extracting various components (such as bounds and coefficients) from a problem instance, a collection of `set()` methods for modifying or generating an optimization instance, and a collection of `calculate()` methods for function, gradient, and Hessian evaluations. We now describe the close relationship between the OSiL schema and the `OSInstance` class.

4.1 Mapping rules

As shown in Fig. 15, the `OSInstance` class has two member classes, `InstanceHeader` and `InstanceData`. These correspond to the OSiL schema's complexTypes `instanceHeader` and `instanceData` (Fig. 4), and to the XML elements `<instanceHeader>` and `<instanceData>` (Fig. 5).

Moving down one level, Fig. 16 shows that the `InstanceData` class has in turn the member classes `Variables`, `Objectives`, `Constraints`, `LinearConstraintCoefficients`, `QuadraticCoefficients`, and `NonlinearExpressions`, corresponding to the complexTypes in Fig. 6 and the elements in Figures 7, 8, 9, 11, 12 and 13.

Figure 17 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

Fig. 15 The `OSInstance` class

```
class OSInstance{
public:
     OSInstance();
     InstanceHeader *instanceHeader;
     InstanceData *instanceData;
}; //class OSInstance
```

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

Fig. 16 The `InstanceData` class

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 osinstance;
osinstance.instanceData.variables.number=2;
osinstance.instanceData.variables.var=new Variable[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;
```

Fig. 17 The `<variables>` element as an `OSInstance` object

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.

- Each complexType in an OSiL schema corresponds to a class in `OSInstance`. Thus the OSiL schema's complexType `Variable` corresponds to `OSInstance`'s class `Variable`. Elements in an actual XML file then correspond to objects in `OSInstance`; for example, the `<var>` element that is of type `Variable` in an OSiL file corresponds to a `var` object in class `Variable` of `OSInstance`.
- An attribute or element used in the definition of a `complexType` is a member of the corresponding `OSInstance` class, and the type of the attribute or element matches the type of the member. In Fig. 17, 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 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.
- A schema sequence corresponds to an array. For example, in Fig. 17 the complexType `Variables` has a sequence of `<var>` elements that are of type `Variable`, and the corresponding `Variables` class has a member that is an array of type `Variable`.

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

The `OSInstance` class has a set of `get()`, `set()`, and `calculate()` 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, the method `getOSInstance()` of the `OSiLReader` class parses an OSiL file and returns a corresponding object `osinstance` of type `OSInstance`. Then the nonzero coefficient values in the linear terms of the constraints 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`. The `calculate()` methods are discussed in Sect. 4.2.2.

4.2 Nonlinear expressions

Perhaps the greatest challenge posed by `OSInstance` lies in the `OSExpressionTree` class, which provides the in-memory representation of the nonlinear terms. Our goal in designing `OSExpressionTree` has been to allow for efficient parsing of OSiL instances, while providing an API that meets the needs of diverse solvers.

Conceptually, any nonlinear expression in the objective or constraints is represented by a tree. The expression tree for the nonlinear part of the objective function (1), for example, has the form illustrated in Fig. 10. The choice of a data structure to store such a tree—along with the associated methods of an API—is a key aspect in the design of the `OSInstance` class.

```
double calculateFunction (expr *e, double x[]){
    ...
    opnum = e->op
    switch(opnum){
        case PLUS_opno: ...
          return( calculateFunction(e->L.e, x) + calculateFunction(e->R.e, x) );
        case MINU_opno: ...
        ...
    }
}
```

Fig. 18 Excerpt from a function calculation method without polymorphism

One approach uses a C-style structure (a `struct` or `union`) for each node in the expression tree. The ASL data structure [16] of the AMPL modeling language [12, 15] has this form. The structure stores information as to operator or operand type, along with pointers to child nodes. Thus a tree-walking method may be used to perform operations on the expression such as function or derivative evaluations. Figure 18 illustrates the essential idea (from [11]) in the context of function evaluation; `expr` is a C structure that represents nodes of the expression tree, `*e` is a particular node passed to the function, and `opnum` is an integer value in the structure, 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 method, risking the introduction of errors.

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 are avoided as much as possible, this is implemented by having each node class extend a single fundamental node abstract class, `OSnLNode`, using the object-oriented concept of polymorphism. We have adopted this alternative in our implementation.

The design of the `OSInstance` API also provides the flexibility to work with different types of nonlinear solvers. For example, the API has a method that can return a list of `OSnLNodes` in postfix format, making it easy to communicate with a solver such as LINDO that expects a postfix instruction list as part of its initial input. A similar method returns a prefix instruction list. Another part of the API supplies `calculate()` methods to work with solvers such as IPOPT and KNITRO that use callbacks to evaluate function and derivative values at specific iterates, as described in Sect. 4.2.2.

4.2.1 The `OSnLNode` *class*

As Sect. 3.4 has explained, all of an OSiL file's operator and operand elements used in defining a nonlinear expression are extensions of the base element type `OSnLNode`. There is an element type `OSnLNodePlus`, for example, that extends `OSnLNode`; then in an OSiL instance file, there are `<plus>` elements that are of type `OSnLNodePlus`.

The `OSExpressionTree` class and its API follow an identical design, based on the abstract class `OSnLNode`. Specifically, each `OSExpressionTree` object contains

Fig. 19 The function calculation method for the "plus" node class with polymorphism

```
double OSnLNodePlus::calculateFunction(double *x){
   m_dFunctionValue =
      m_mChildren[0]->calculateFunction(x) +
      m_mChildren[1]->calculateFunction(x);
   return m_dFunctionValue;
} //calculateFunction
```

a pointer to an `OSnLNode` object that is the root of the corresponding expression tree. To every element that extends the `OSnLNode` type in an OSiL instance file, there corresponds a class that derives from the `OSnLNode` class in an `OSInstance` data structure. 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.

The `OSInstance` class has a variety of `calculate()` methods, based on two pure virtual functions in the `OSInstance` class. The first of these, `calculateFunction()`, 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. As an example, the `calculateFunction` method for the `OSnLNodePlus` class is shown in Fig. 19. 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 Fig. 18 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 Automatic differentiation

Gradient and Hessian calculations via callbacks at specified iterates are carried out by use of automatic differentiation (AD) [17]. This is the same approach used by proprietary interfaces between modeling languages and nonlinear solvers; it can be much faster than symbolic differentiation, but unlike finite differencing it gives up nothing in accuracy.

To provide AD services, a second pure virtual function is introduced into the `OSInstance` class. In our initial implementation this function, `constructCppADTape()`, builds an AD "tape" that is read by CppAD, a robust and efficient AD package developed by Bradley Bell [2]. Each differentiable operator class that extends `OSnLNode` must implement a method based on this package, which makes heavy use of C++ operator overloading; Fig. 20 illustrates taping the plus operator.

The `calculate()` methods in the `OSInstance` class for first and second derivative calculations are built on the CppAD facilities. These methods operate in two phases, first constructing the tape and then using it to compute the requested derivatives. For example, the API method `calculateAllConstraintFunctionGradients()` tapes an operation sequence by applying `constructCppADTape()` to the nonlinear part of each objective function and constraint, then uses the CppAD method `Jacobian` to calculate first partial derivatives. This method returns a `SparseJacobian` object that contains the values of the partial derivatives, their sparsity pattern,

```
AD<double> OSnLNodePlus::constructCppADTape(std::map<int, int> *cppADIdx,
CppAD::vector< AD<double> > *XAD){
   m_CppADTape =
      m_mChildren[0]->constructCppADTape( cppADIdx, XAD) +
      m_mChildren[1]->constructCppADTape( cppADIdx, XAD);
   return m_CppADTape;
} //calculateCppADTape
```

Fig. 20 The AD function taping method for the "plus" node class

and information (useful to some solvers) on which partial derivatives are constant. Likewise there is a method to return the sparsity pattern and second derivatives for the Hessian of the Lagrangian function.

5 Conclusions and future work

OSiL is an XML-based language for representing general nonlinear optimization instances. It represents a major advance over the old MPS format for linear and integer problems. There are open-source C++ libraries available at `projects.coin-or.org/OS` that implement all of the features described in this paper including special sparse treatments of linear and quadratic components of a model instance. These libraries implement classes described in this paper for writing, validating, and parsing optimization instances in the OSiL format. In addition, the OSInstance class with its `get()`, `set()`, and `calculate()` methods provides a powerful API to interface with modeling languages and solvers. It is the flexible design of the `OSInstance` class with its underlying `OSExpressionTree` class (see Sect. 4.2) that provides first and second derivative information for solver callback functions or the instance representation in postfix or prefix instruction list format. Examples are given in a User's Manual available at `https://projects.coin-or.org/OS`.

The design of OSiL and the corresponding in-memory `OSInstance` class has also facilitated the development of libraries for feeding any instance in OSiL format into the following commercial and open-source solvers: Cbc, Clp, Cplex, DyLP, Glpk, Knitro, Lindo, SYMPHONY, and Vol. On the modeling language side we have a component `amplClient` that provides access from the AMPL modeling language to any solver (including those mentioned above) that recognizes OSiL files, in much the same way that the Kestrel client [8] provides access from AMPL to the NEOS Server [7, 9].

It terms of future work, we are in the process of 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 of simulations. Each class has its own special features that pose new kinds of challenges.

Finally, OSiL is only one schema in a much larger OS—Optimization Services—framework [23]. Of most immediate concern, the OS framework is planned to incorporate OSoL (Optimization Services option Language) for passing algorithmic options to solvers, and OSrL (Optimization Services result Language) 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.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley, Reading (1986)
2. Bell, B.: CppAD: A package for C++ algorithmic differentiation. http://www.coin-or.org/CppAD/ (2006)
3. Bischof, C.H., Bücker, H.M., Marquardt, W., Petera, M., Wyes, J.: Transforming equation-based models in process engineering. In: Bücker, H.M., Corliss, G., Hovland, P., Naumann, U., Norris, B. (eds.) Automatic Differentiation: Applications, Theory, and Implementations. Lecture Notes in Computational Science and Engineering, pp. 189–198. Springer, Berlin (2005)
4. Bradley, G.: Introduction to extensible markup language (XML) with operations research examples. ICS Newsl. **24**, 1–20 (2003)
5. Bradley, G.: 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
6. Chang, T.-H.: Modelling and presenting mathematical programs with xml:lp. Masters thesis, Department of Management, University of Canterbury, Christchruch, NZ (2003)
7. Czyzyk, J., Mesnier, M.P., Moré, J.J.: The NEOS server. IEEE J. Comput. Sci. Eng. **5**, 68–75 (1998)
8. Dolan, E.D., Fourer, R., Goux, J.-P., Munson, T.S., Sarich, J.: Kestrel: An interface from optimization modeling systems to the NEOS server. Technical report, Optimization Technology Center, Northwestern University, Evanston, IL and Mathematics & Computer Science Division, Argonne National Laboratory, Argonne, IL (2006). http://www.optimization-online.org/DB_HTML/2007/01/1559.html
9. Dolan, E.D., Fourer, R., Moré, J.J., Munson, T.S.: Optimization on the NEOS server. SIAM News **35**(6), 4–9 (2002).
10. Ezechukwu, O.C., Maros, I.: OOF: open optimization framework. Technical Report ISSN 1469-4174, Department of Computing, Imperial College of London, London, UK (2003)
11. Fourer, R., Gay, D.M.: Extending an algebraic modeling language to support constraint logic programming. INFORMS J. Comput. **14**, 322–344 (2002)
12. Fourer, R., Gay, D.M., Kernighan, B.W.: A modeling language for mathematical programming. Manag. Sci. **36**, 519–554 (1990)
13. Fourer, R., Lopes, L., Martin, K.: LPFML: A W3C XML schema for linear and integer programming. INFORMS J. Comput. **17**, 139–158 (2005)
14. Fourer, R.: Modeling languages versus matrix generators for linear programming. ACM Trans. Math. Softw. **9**, 143–183 (1983)
15. Fourer, R., Gay, D.M., Kernighan, B.W.: AMPL: A Modeling Language for Mathematical Programming, 2nd edn. Brooks/Cole, Pacific Grove (2003)
16. Gay, D.M.: Hooking your solver to AMPL (revised 1994, 1997). Technical report, Bell Laboratories, Murray Hill, NJ (1993)
17. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. SIAM, Philadelphia (2000)
18. Halldórsson, B.V., Thorsteinsson, E.S., Kristjánsson, B.: A modeling interface to non-linear programming solvers an instance: xMPS, the extended MPS format. Technical report, Carnegie Mellon University and Maximal Software (2000)
19. Kristjánsson, B.: 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
20. Lindo Systems, Inc.: LINDO API user's manual. Technical report, Lindo Systems, Inc. (2002). http://www.lindo.com/lindoapi_pdf.zip
21. Lougee-Heimer, R.: The Common Optimization INterface for operations research. IBM J. Res. Dev. **47**(1), 57–66 (2003)
22. Lustig, I.J., Puget, J.-F.: Program $\neq$ program: Constraint programming and its relationship to mathematical programming. Interfaces **31**(6), 29–53 (2001)

23. Ma, J.: Optimization services (OS), a general framework for optimization modeling systems. Ph.D. Dissertation, Department of Industrial Engineering & Management Sciences, Northwestern University, Evanston, IL (2005)
24. Rosenbrock, H.H.: An automatic method for finding the greatest or least value of a function. Comput. J. **3**, 175–184 (1960)
25. Sandhu, P.: The MathML Handbook. Charles River Media, Hingham (2003)
26. Skonnard, A., Gudgin, M.: Essential XML Quick Reference. Pearson Education, Boston (2002)
27. Van Hentenryck, P.: Constraint and integer programming in OPL. INFORMS J. Comput. **14**, 345–372 (2002)