Optimization Services: A Framework for Distributed Optimization

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

Kipp Martin
Graduate School of Business,University of Chicago, 5807 South Woodlawn Avenue, Chicago, Illinois 60637, USA, kipp.martin@chicagogsb.edu

We have undertaken a research project to design an innovative distributed optimization environment in which modeling languages, servers, registries, communication agents, interfaces, analyzers, solvers, and simulation engines can be implemented as services and utilities under a unified framework. Our work, which we call Optimization Services or OS, defines standards for all activities necessary to support decentralized optimization on the Internet: representation of optimization instances, results, and solver options; communication between clients and solvers; and discovery and registration of optimization-related software using the concept of Web Services. In this paper we place emphasis on issues in distributed computing that are posed by the special character of optimization. We also describe a reference implementation that is freely available as an open-source project of COIN-OR.

Key words: optimization, modeling languages, distributed computing, XML, Web Services.
History: Version 1, August 2007; version 2, August 2008.

---

1. Introduction

Optimization is a key paradigm for modeling in operations research and in related aspects of engineering, science, economics, and business. But to be a practical tool, optimization increasingly needs to become integrated into modern corporate information technology (IT) infrastructures. The OR community has focused on standalone tools like modeling languages and solvers designed to work on a single machine, while the IT community is has been moving to tools like Extensible Markup Language (XML), Service Oriented Architecture (SOA), and Web Services that facilitate distributed computing. The OR community could much more readily achieve its objectives if optimization tools were built into technologies that the IT community is already using.

XML, SOA, and Web Services have facilitated the growing prevalence of *software as a service:* that is, software residing on a server that is accessed by numerous client machines over a network, as opposed to software residing in multiple copies on its users' machines. Current examples of software as a service include customer relationship management (see `salesforce.com`), tax preparation, Gmail, and Google Calendar. Indeed, all of the major players in software have been promising software as a service; the trend is clearly away from the fat client loaded with heavyweight applications, and towards distributed computing.

The goal of the research described herein is to determine *how optimization can be conceived as a software service.* This is easier said than done, because optimization software embodies a number of difficulties that are inherent to its nature as a tool for numerical computing as well as symbolic modeling. There is a lack of standardized communication in almost every respect:

- ◇ There are numerous optimization modeling languages, each with its own representations for models and data. In consequence there are only the most primitive standards for representing model instances, and even those are mainly confined to the familiar case of linear programming.
- ◇ There are numerous optimization algorithms, or solvers, each with its own application program interface (API). There is no standard solver API.
- ◇ Optimization projects use such a variety of operating systems, processor architectures, and compilers that developers of optimization applications have great difficulty supporting all platforms that are in demand.
- ◇ There are virtually no standards for representing solvers' algorithmic options and results.
- ◇ There is no standard protocol for registration and discovery of solver services over a network.

Overall, optimization services exhibit a greater variety and complexity of information to be moved around and a much greater range of behavior to be dealt with than do typical business applications. To further complicate matters, solvers are categorized by mathematical problem types that do not readily correspond to the model types familiar to customers.

The results of our study are presented in this paper as a *framework* for distributed optimization, which we refer to as Optimization Services, or OS. By framework, we mean a set of standards (or protocols) for

- ▷ representation of optimization instances, results, and solver options;
- ▷ communication between clients and solvers; and

- ▷ registration and discovery of optimization-related services in a distributed environment.

To this end, OS provides a general and robust format for representing optimization model instances, a common solver interface with `get()`, `set()`, and `calculate()` methods, and standard registry and discovery protocols. It also provides communication protocols that allow a client machine of any type to communicate with a solver server on any kind of platform.

The ultimate goal of this project is *to make optimization as easy as hooking up to the network.* Our vision is for all optimization system components, including modeling language environments, servers, registries, communication agents, interfaces, analyzers, solvers, and simulation engines, to be implemented as services under the OS framework, and for customers to use these computational services much like utilities, with specialized knowledge of optimization algorithms, problem types, and solver options being potentially valuable but not required. We foresee OS being built upon standards that are independent of programming language, operating system, and hardware, and that are open and readily available for use by the optimization community.

1.1 Protocols

Figure 1 presents a summary of the OS protocols to be discussed in this paper. There are additional OS protocols, but those shown in the figure are sufficient to convey the major aspects of the OS framework. A description of all the OS protocols is given in Ma (2005).

The OS protocols are classified as either *communication protocols* or *representation protocols*. The former are at a "higher level" than the latter; loosely speaking, communication protocols specify what data are exchanged between client and server, and representation protocols specify detailed information about the data format. For example, a communication protocol might specify that in order to solve an optimization problem a client must send to the solver server a model instance and solver options. The representation protocol would specify detailed information about the formats of the instance and options.

Both kinds of OS protocols may apply to a client communicating with a server that performs an optimization service or that provides a registry service. By an optimization service we mean loosely a solver that performs optimization on a model instance, analyzes a problem, performs only a preprocessing service, or perhaps does simulation but not true

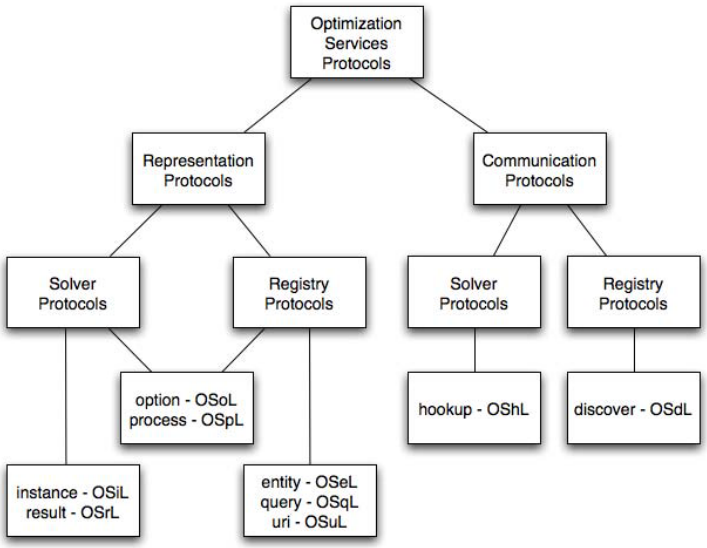

Figure 1: A summary of OS protocols.

optimization. A registry service allows servers to register their optimization services or allows clients to discover servers that perform optimization services.

1.2 Outline

In the next section we provide sufficient background information on XML, Service Oriented Architecture, and Web Services in order to make the protocols outlined in Figure 1 understandable. We also describe some previous work relevant to SOA and Web Services specifically for optimization.

In Section 3 we present the design of the key OS representation protocols. We describe OSiL for representing problem instances, OSrL for representing optimization results, and OSoL for specifying options to solvers or registry services. We also describe OSeL for registering an optimization service, and OSqL for querying a registry about which optimization services are available.

In Section 4 we describe key OS communication protocols, particularly OShL for communication between a client and a server hosting an optimization service, and OSdL for communication between a client and a server hosting a registry service.

The OS framework is just that — a framework. It does not specify implementation details, programming languages, and the like. However, in order to provide a reference implementation, many of the protocols described in this paper are implemented in a set of

open-source libraries. These libraries, along with the associated source code, have been donated to COIN-OR (COmputational INfrastructure for Operations Research) and constitute the COIN-OR OS project, which we describe briefly in Section 5.

Section 6 concludes with a summary of accomplishments and of current and future work. A more extensive example than this paper can provide is available in an online supplement (Fourer, Ma, and Martin (2008)).

2. Background

This section begins with a brief introduction to the concepts of XML, Service Oriented Architecture, and Web Services. Then we describe previous work in optimization that has direct relevance to these concepts.

2.1 XML

All of the Optimization Services protocols to be described in this paper are expressed in the Extensible Markup Language, XML. We chose XML because

- ◇ all of the Web Services protocols are expressed in XML;
- ◇ the language of the Web, HTML (Hypertext Markup Language), is being replaced by the XML version, XHTML;
- ◇ XML is becoming the *lingua franca* of data.

In sum, XML has become the best way to store text data. The XML standard is controlled by the W3C (World Wide Web Consortium, `www.w3c.org/XML`). A useful overview of XML technologies is given by Skonnard and Gudgin (2002).

XML is a *markup language.* An XML string or file is composed of *data* and of *markup* that describes the data. XML markup consists of a few kinds of components that must be organized according to certain general principles but that are quite flexible in their meaning, as we will show.

The following optimization problem instance (which is a modification of an example of Rosenbrock (1960)) is used to illustrate XML and other concepts throughout this paper:

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

$$\text{Subject to} \quad x_0 + 10.5x_0^2 + 11.7x_1^2 + 3x_0x_1 \leq 25 \tag{2}$$

$$\ln(x_0x_1) + 7.5x_0 + 5.25x_1 \geq 10 \tag{3}$$

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

In this problem there are two continuous variables, x_0 and x_1, each with a lower bound of 0. Figure 2 shows how this information about the variables could be stored as XML. There are two types of markup, *elements* (or *tags*) and *attributes* that describe the elements. Specifically, there are is a `<variables>` element, which serves to mark the start and end of a list of `<var>` elements. The two `<var>` elements correspond to x_0 and x_1, and 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

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

Figure 2: The `<variables>` element for (1)–(4).

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, specifying the upper bound, that is absent in Figure 2 and that consequently takes the default value `"INF"` (denoting ∞).

In the XML representation of the variables' properties illustrated in Figure 2, the text markers surrounding each tag (`<` and `>`), as well as other elements of the XML syntax, serve to make XML instances very easy to parse and to validate. Numerous parsers, both open-source and proprietary, are available for processing an XML document.

2.2 Service Oriented Architectures

A common distributed computing architecture is shown in Figure 3. In this design, there is a central server that intermediates between all of the clients and all of the other servers. All client requests must go through this central server. The NEOS architecture (to be detailed in Section 2.4) is a good illustration of the central server paradigm. All optimization instances (which could be very large) and solutions must pass through the central server, which then schedules a solver server to optimize the model and pass the result back to the client through the central server.

The central server paradigm does not scale up well. Indeed, the Internet works because it has a decentralized architecture; there is no such thing as a "central web server" through

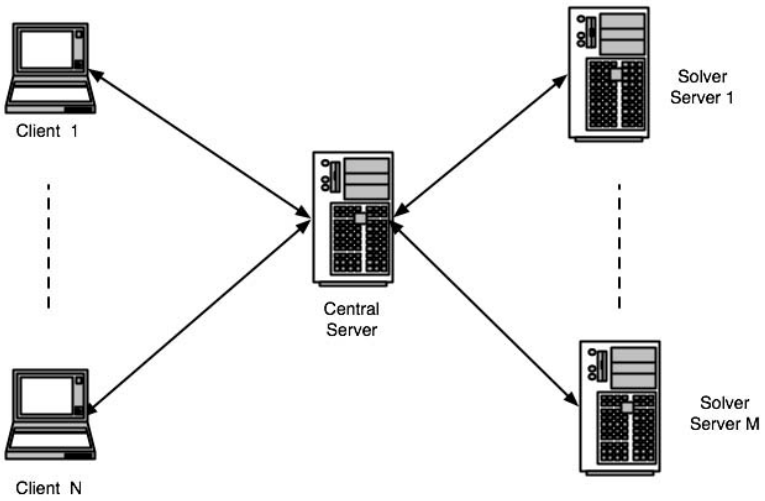

Figure 3: A centralized distributed computing architecture.

which all requests pass. Rather, there are directory services such as Google and Yahoo where a client can look up the resource of interest and then contact that resource directly rather than going through a central server.

Following the Internet model, IT departments are increasingly turning to service oriented architectures (SOA) when building their infrastructures. In the SOA paradigm (Figure 4), a service provider (perhaps a solver service) registers with a registry/discovery service. In a sense, this registry/discovery service does act as central server, but it functions as a lightweight service that only maintains information about available service providers. The client or service consumer "discovers" the service that is described in the registry. Then,

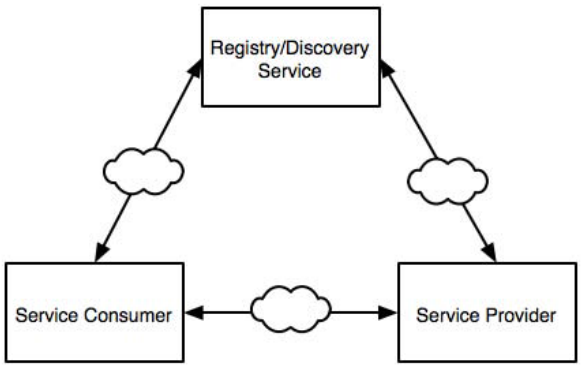

Figure 4: The Service Oriented Architecture (SOA) paradigm.

rather than interact directly with the registry/discovery server to consume the service from the service provider, the service consumer contacts the service provider directly and they work in a "peer-to-peer" fashion.

In Figure 5 we show the SOA version of the distributed optimization system first illustrated in Figure 3. The key contrast between the two architectures is that in the SOA version the clients and solvers are exchanging optimization instances and results directly in a peer-to-peer mode, thus enabling scaling of the system.

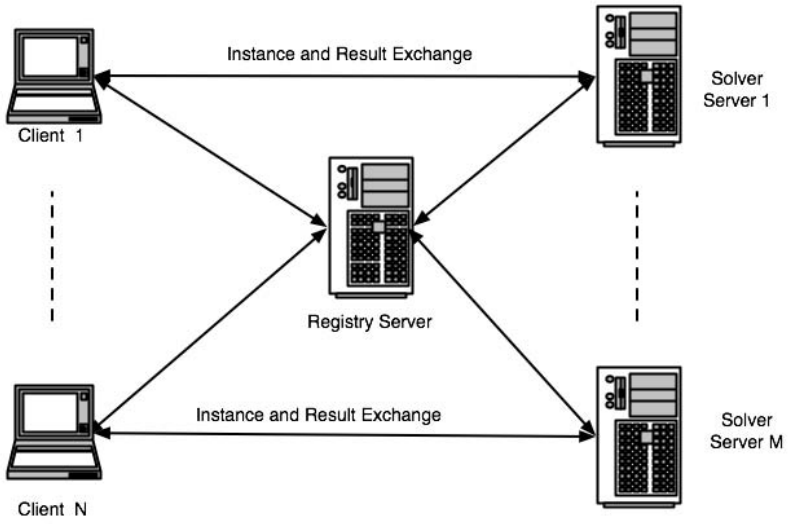

Figure 5: The Service Oriented Architecture version of Figure 3.

SOA is a philosophy of how to build a decentralized architecture, rather than a set of actual protocols or standards. We next describe the Web Services protocols for actually implementing an SOA.

2.3 Web Services

The Web Services concept consists of three XML-based protocols: SOAP (Simple Object Access Protocol), WSDL (Web Services Discovery Language), and UDDI (Universal Description, Discovery, and Integration).

In an SOA, service consumers make *requests* to service providers and get *responses* from the providers. SOAP is an XML-based protocol that specifies how information should be encoded in the request and response messages. These messages can then be sent over the net using an application layer protocol such as HTTP (Hypertext Transfer Protocol), FTP

(File Transfer Protocol), or SMTP (Simple Mail Transfer Protocol).

Figure 6 illustrates SOAP over HTTP. OS representation protocols (Section 3) are packaged inside OS communication protocols (Section 4) that are in turn packaged inside a SOAP envelope that constitutes the body of the HTTP message. For example, a client may send an optimization instance in the OSiL protocol, and solver options in the OSoL protocol using an OShL communication protocol that instructs a solver to optimize the problem. All of these protocols would be in a SOAP envelope in an HTTP body sent over the Internet to the solver service using the HTTP protocol.

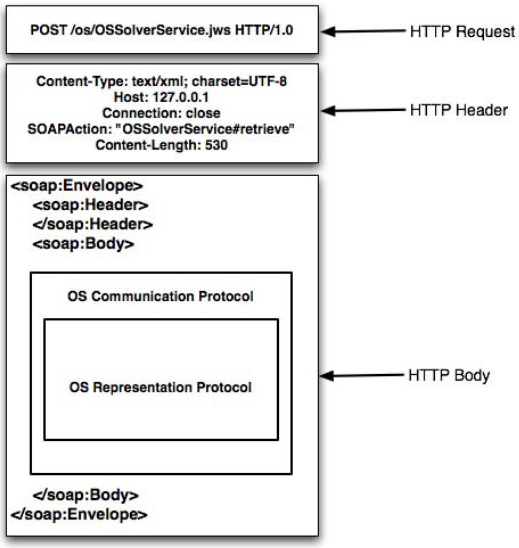

Figure 6: OS protocols inside a SOAP envelope inside an HTTP body.

WSDL is a protocol for expressing, in XML format, the methods (functions) and arguments provided by a Web service. WSDL is used by the provider of a service to tell the consumer how to use the service. OS communication protocols are expressed using WSDL, as will be seen in Section 4.

Finally, UDDI is an XML-based protocol describing how providers can join the registry and how consumers can query the registry. We discuss the OS registry protocols in Section 4.2.

2.4 Optimization Systems and Services

The idea of integrating optimization within a broader information system has a long and successful history. Optimization has long been a part of more general scientific software

such as MATLAB (www.mathworks.com/products/optimization) and statistical software such as SAS (www.sas.com/technologies/analytics/optimization). But undoubtedly the best known examples are the Excel Solver (www.solver.com) and What's Best (www.lindo.com), which make optimization conveniently available within the Microsoft Excel spreadsheet program, a ubiquitous business tool. As the IT community moves in the direction of distributed computing, the OR will do well to follow this lead by integrating optimization tools into a distributed computing environment.

Internet optimization servers began to appear almost immediately after the advent of the World Wide Web, mostly using browsers as interfaces for input and output. Summaries can be found in Czyzyk et al. (2000) and Fourer and Goux (2001).

The most ambitious and influential Internet optimization service has been NEOS (Czyzyk et al. (1998), neos.mcs.anl.gov) which has been widely used by the optimization community for over a decade. A central server maintains and queues submissions for solvers that run on a variety of workstations scattered around the Internet. At first, submissions were MPS-format files for linear problems and C or Fortran programs for nonlinear ones, but now the great majority of submissions are in high-level modeling languages, predominantly AMPL (www.ampl.com) and GAMS (www.gams.com). Submissions through the NEOS web portal (neos.mcs.anl.gov/neos/solvers) remain popular, and they can also be made by sending XML text files through email.

The most recent NEOS release, described in detail by Dolan et al. (2008), features a NEOS application programming interface (API) that permits all server functions to be accessed through remote function calls using the XML-RPC protocols (www.xmlrpc.com). This has brought NEOS more in line with the precepts of SOA and has made it much easier to integrate into optimization modeling environments. Nevertheless, its design still adheres in many respect to the central server paradigm of Figure 3. Also NEOS employs whatever file formats are supported by the various solvers; the over 40 solvers in the NEOS lineup require instance inputs of about a dozen different kinds. Similarly there is no NEOS standard format for communicating options to solvers or communicating results from solvers. As a result, while the NEOS API can effectively control and query the NEOS server, it cannot extend to reading and writing the files that are used for communication with the server.

The OS project has thus faced the challenge of remedying a variety of NEOS weaknesses. How it has faced this challenge, and the issues it has had to resolve in doing so, are detailed in the remainder of this paper.

Because OS is a framework, much of its effect occurs behind the scenes and is not perceived by the typical user. Indeed most users will never see the OS files and protocols described in Sections 3 and 4. For example, a user of the OSSolverService executable could simply enter the command

```
OSSolverService -osil testProblem.osil -osrl TestProblem.osrl
```

Then, using tools provided in the OS COIN-OR project (Section 5), the optimization solution could be displayed for the user in a browser window as shown in Figure 7.

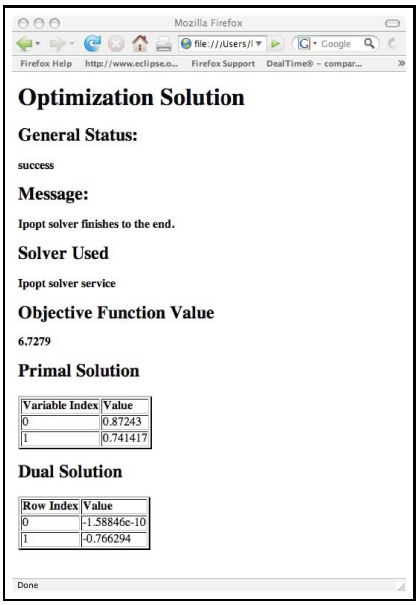

Figure 7: The optimization result displayed in a Web browser.

3. Optimization Services Representation Protocols

In the OS framework there are representation protocols for communication with solver servers and with registry servers. Section 3.1 introduces the protocols for solver servers, Sections 3.2–3.4 describe design principles and issues for the specific protocols that represent model instances, optimization results, and solver options, and Section 3.5 explains how these protocols are extended to encompass in-memory representations. Section 3.6 then describes the protocols for registry servers.

3.1 Representation Protocols for Solver Servers

Currently there are no comprehensive standards for communicating optimization problem instances, solver algorithmic options, or results of optimization. A few standard file formats such as MPS for linear and mixed-integer programs are well-known, but are inefficient, limited in scope, and not quite "standard" in the cases they cover. All of the widely-used optimization modeling systems have their own, incompatible representations for problem instances and related information.

In a tightly coupled environment where the optimization model instance generator and optimization solver are a single piece of software, standards are not an issue. A good example is LINGO (`www.lindo.com`), which is both a modeling language and a solver. Similarly, ILOG's OPL Development Studio (`www.ilog.com/products/oplstudio`) is a modeling system that is intended to be used only with its developer's solvers. However, beginning with the development of popular standalone modeling languages such as GAMS and AMPL in the 1970s and 1980s, it became increasingly common to have separate pieces of software for generating a model instance and for optimizing that instance. NEOS's appearance in the mid-1990s then completely broke the link between model generator and solver, allowing a model to be developed on one machine and then sent over the network to be solved on another machine.

The downside of these developments is that we now have a huge proliferation of modeling languages and solvers. NEOS alone has solvers that recognize three modeling languages, functions programmed in Fortran and C, and numerous file formats: MPS and LP for linear and integer programming, SMPS extensions to MPS for stochastic programming, SPARSE SDPA specific to semidefinite programming, and DIMACS, NETFLO, and RELAX4 for network linear programming. This is now a significant problem for software developers in the optimization community. If there are M modeling languages and N solvers, then $M \times N$ "drivers" are required for complete interoperability. However, if there are *standards* for representing model instances, optimization results, and solver options, then only $M + N$ drivers are required for interoperability. In the following sections we describe the design of the OS standards for these purposes.

3.2 OSiL: Optimization Services instance Language

OSiL is an XML-based language for representing instances of large-scale optimization problems including linear programs, mixed-integer programs, quadratic programs, and very general nonlinear programs. We illustrate the major features of OSiL using the optimization problem given by equations (1)–(4) in Section 2.1. A thorough account of all features can be found in Fourer, Ma, and Martin (2007).

There are two continuous variables, x_0 and x_1, in this instance, each with a lower bound of 0. Back in Figure 2 we showed how we represent this information in XML using both markup and data. We chose the name `<var>` for each markup element that represents a variable, but we could have chosen `<variable>`; clearly, there are countless ways to represent an optimization instance in XML. However, when parsing an XML file there must not be any ambiguity. So in order to be useful for communication between solvers and modeling languages, the markup in the instance files must conform to a naming convention.

A common way to impose a standard on the naming and structure of an XML file is to use a W3C (World Wide Web Consortium) XML *schema,* which specifies the elements and attributes that define a specific XML vocabulary such as OSiL. An XML file that conforms to a schema is called *valid* for that schema. Indeed, when we talk about an "Optimization Services instance Language," we are really talking about the OSiL 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.

Recall that Figure 2 is the XML representation of the variables for the model shown in equations (1)–(4). Figures 8 and 9 are the part of the OSiL W3C XML schema that defines the standard for what every `<variables>` section of an instance in OSiL should look like. In particular, these two figures are the schema specifications for the elements `<variables>` and `<var>` of Figure 2, respectively.

In Figure 8, a "complexType" named `Variables` is defined. Just as object-oriented programming languages such as C++ and Java allow the user to define data types that extend standard data types such as integer, double, and string, the W3C XML Schema standard permits user-defined data types called complexTypes. The complexType is used to specify the elements and attributes that are allowed to appear in a valid XML instance

```
<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 8: 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 9: The `Variable` complexType in the OSiL schema.

file such as the one shown in Figure 2. The `Variables` complexType of Figure 8 is made up of a sequence of elements named `<var>`. Each `<var>` element is in turn an instance of a user-defined complexType named `Variable`, which is defined in Figure 9. Every `<var>` element appearing in Figure 2 thus conforms to the Figure 9 schema.

The `Variables` complexType in Figure 8 also has an attribute `numberOfVariables` that specifies the number of `<var>` elements in the XML instance file. The `numberOfVariables` attribute has the standard type positiveInteger. Note that in Figure 2, `numberOfVariables` is defined to be `"2"`, which is indeed a positiveInteger. The `Variable` complexType in Figure 9 can be seen to have attributes of a variety of kinds. The `type` attribute is specified as a simpleType whose value must be one of four character strings — `"C"`, `"B"`, `"I"`, or `"S"` — indicating variables whose domain is continuous, binary (zero-one), integer, or symbolic, respectively.

Also in Figure 2, note the <var> "child" elements nested inside of the <variables> element; in our convention, the elements that appear in an actual XML instance file have element names that are all lower case, and the corresponding complexType in the schema begins with an upper case letter. The complexType appears only in the schema and not in the XML instance file, which is made up of elements that are instances of the complexTypes defined in the schema. Thus, the <variables> element of Figure 2 conforms to the definition of complexType Variables in Figure 8 by containing a sequence of <var> elements that are instances of complexType Variable defined in Figure 9.

In addition to Variables, there are complexTypes Objectives and Constraints that similarly define <objectives> and <constraints> sections. To complete the specification of a linear problem (possibly with integer variables), the OSiL schema incorporates the complexType LinearConstraintCoefficients, which defines a <linearConstraintCoefficients> section that contains the nonzero coefficients in the constraints. The coefficients are described by use of a standard three-array sparse matrix storage scheme: an array of nonzero coefficients in a child element <value>, a corresponding array of row indices or column indices in a child element <rowIdx> or <colIdx>, and in a child element <start> an array that indicates where each row or column begins within the previous two arrays.

There are other ways of specifying a constraint coefficient matrix, and we considered designing OSiL to offer a choice of alternatives. In the end we chose to include only the three-array scheme, because it is simple, compact, and easily converted to and from other schemes by auxiliary software.

Linear constraints have the general form *lb-const* $\leq$ *linear-expr* $\leq$ *ub-const*, allowing for the special cases *lb-const* $= -\infty$, *ub-const* $= +\infty$, and *lb-const* $=$ *ub-const* (an equality constraint). The *lb-const* and *ub-const* values that aren't infinite are naturally stored in OSiL's <constraints> section.

Finally, a decision must be made about the handling of the linear objective coefficients. One or more objectives may be included with the constraints, identified by *lb-const* and *ub-const* values that are both infinite. This design is attractive for its simplicity; but the objective coefficients are potentially scattered among the constraint coefficients, and are hard to extract for the purposes of some solvers. Instead, OSiL places the nonzero objective coefficients in the separate <objectives> section. This approach has the further advantage of generalizing cleanly to the nonlinear case.

Optimization problems may incorporate expressions that are not linear, of course, and a

true standard must accommodate these as well. There is a natural way to represent general expressions in XML, however, by defining an element for each operator or function, with child elements specifying the operands or arguments. Each child may itself be an element for an operator or function, so that the entire graph of the expression is defined recursively in a natural way. As an example, Figure 10 shows the OSiL representation of the term $\ln(x_0 x_1)$ that appears in (3). These representations are placed in a `<nonlinearExpressions>` section of the OSiL file, which is of course defined by a `NonlinearExpressions` complexType in the OSiL schema.

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

Figure 10: The OSiL representation of $\ln(x_0 x_1)$.

It developed that one of the greatest challenges of our project was to figure out how to define this schema, to permit efficient parsing of the many elements representing operators and functions that can reasonably be considered fundamental to optimization problems. There are over 200 such elements, as shown in Figure 11. In addition to the usual smooth nonlinear functions such as log and cosine, there are comparison and logical operators, statistical functions and distributions, and common spreadsheet operators.

The key to writing an efficient schema to encompass all of the potential elements of an

Category	**Examples**	**Number in Category**
Arithmetic	`Plus`, `Sum`, `Power`	10
Elementary	`Log`, `Factorial`, `Round`	22
Trigonometric	`Cos`, `Arccos`, `Cosh`	24
Statistics	`Mean`, `Percentile`, `Npv`	30
Probability	`BetaDist`, `GammaCum`, `NormalInv`	92
Boolean	`Leq`, `Or`, `InSet`, `AllDiff`	26
Variable	`Variable`, `RandomVariable`	5
Terminal	`Number`, `String`, `Identifier`	3
Constant	`PI`, `INF`, `TRUE`, `NAN`	8
Other	`Quadratic`, `XPath`, `Complements`	12

Figure 11: Number of element types in OSiL's general expression graphs.

expression is to first define a general complexType `OSnLNode`, and an *abstract* element of that type, also named `OSnLNode`, as shown in Figure 12. Then each element corresponding to a different node of the expression graph is defined by a complexType that *extends* `OSnLNode` and by an element of that complexType. Figure 13 shows as an example the definition of the element `<times>` from Figure 10, via the complexType `<OSnLNodeTimes>`. The line

```
<xs:sequence minOccurs="2" maxOccurs="2">
```

specifies that the element has exactly two children, corresponding to the two operands of any multiplication operator.

```
<xs:complexType name="OSnLNode" mixed="false">
</xs:complexType>

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

Figure 12: Definition of a generic expression-graph node is OSiL.

```
<xs:complexType name="OSnLNodeTimes">
    <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="times" type="OSnLNodeTimes" substitutionGroup="OSnLNode">
</xs:element>
```

Figure 13: Definition of the `<times>` element in OSiL.

This arrangement — analogous to the use of virtual classes in object-oriented programming languages — permits a parser to validate an OSiL file against the schema without a complex conditional or switch statement to find each element among the many possibilities. A detailed explanation of the schema and parser issues is beyond the scope of this paper, but can be found in Fourer, Ma, and Martin (2007).

Linear expressions are a special case of nonlinear ones, and so in principle OSiL requires no special features for linear objectives and constraints. It makes sense to handle the linear

case specially, however, because there is a particularly efficient way to write linear expressions (in terms of coefficient lists) and because there are many optimization algorithm prepared to take advantage of this efficiency. We considered other cases that could be handled specially, but in the end decided that only quadratic expressions merited special handling in OSiL, because they can also be viewed in a simple way as coefficient lists, and are the target of several specialized algorithms that are widely implemented. OSiL represents each quadratic term by four values in a `<quadraticCoefficients>` section: an identifying number for the objective or constraint in which it lies, indices of its two variables, and a coefficient. An example is given in Fourer, Ma, and Martin (2007).

All of the OS schemas are available at `www.optimizationservices.org`. Although schemas can be read as text files, large ones are easier to navigate with the help of software such as XML Spy (`www.altova.com`) or Oxygen (`www.oxygenxml.com`). These packages present schemas in a convenient graphical environment.

3.3 OSrL: Optimization Services result Language

It is little known that the venerable MPS format for linear program instances has a corresponding format for the results of solving linear programs. All of the major optimization modeling systems have distinct nonstandard formats in which they expect results to be reported. In a successful distributed optimization framework, a standard for this purpose is as important as a standard for reporting problem instances. Thus another part of our project has been to design OSrL, an XML-based protocol for representing the solutions of large-scale optimization problems of all the kinds that can be described using OSiL. As with OSiL, OSrL is defined by an XML schema.

In conceiving OSrL our design goal has been to maximize flexibility in reporting optimization results, but to keep the design simple. A linear program is a well-defined entity, for example, but the solution of a linear program is not. We cannot have a linear program without constraints or variables, but we can have a linear programming solution without reduced costs or right-hand side sensitivity information. In general, different optimization solvers may present their results in different formats, and some may include more detail than others. The level of solution detail is up to the solver developer and would be difficult to standardize.

Thus whereas with OSiL we have tried to be as encompassing and complete as possible, with OSrL we have taken a minimalist approach. The objective of OSrL is to the allow the

solver developer to report as much or little detail as desired.

We illustrate this design philosophy in Figure 14 with an example of OSrL's `<variables>` element. A solver developer will almost certainly want to report the values of the decision variables in a solution, so OSrL provides a `<values>` element containing a `<var>` element for each variable (with variables at zero optionally omitted). However, aside from the values of the variables, it is not so clear what other solution information associated with variables the solver developer will wish to report. Hence, we provide an `<other>` element with attributes `name` and `description`. The `<variables>` element can have none or an unlimited number of `<other>` children. This allows for complete flexibility in reporting.

```
<variables>
    <values>
        <var idx="0">539.984</var>
        <var idx="1">252.011</var>
    </values>
    <other name="reduced costs" description="the variable reduced costs">
        <var idx="0">0</var>
        <var idx="1">0</var>
    </other>
</variables>
```

Figure 14: An OSrL `<variables>` element for results from a linear program.

There are similar constructs for, among others, `<constraints>` and `<objectives>`, as shown in the full OSrL schema accessible at `www.optimizationservices.org`.

3.4 OSoL: Optimization Services option Language

In addition to a model instance, a solver is often sent a list of option settings to guide its algorithms. We have designed OSiL's representation of instances to be solver-independent, so that the same representation can be sent to any appropriate solver. But a description of options is unavoidably solver-dependent. Thus our OS framework incorporates a separate XML-based language, OSoL, for representing solver options.

OSoL's design philosophy is analogous to that of OSrL: maximize flexibility in specifying solver options but keep the details simple. Solver options vary greatly and lack standardization even among the most common ones such as "iteration limit" or "feasibility tolerance." Thus like OSrL, OSoL provides an `<other>` element that solver developers can adapt as needed.

Figure 15 illustrates the flexibility of OSoL. Within the <optimization> element are two <other> tags, the first telling the solver to use a barrier solution algorithm, and the second telling the algorithm to use a particular feasibility tolerance. The name attributes of the <other> tags identify particular options and are specific to individual solvers. If a solver does not recognize an <other> tag's name attribute then it is free to respond accordingly, such as by issuing a warning message and ignoring the tag.

```
<osol xmlns="os.optimizationservices.org">
    <general>
        <contact transportType="smtp">
            kipp.martin@chicagogsb.edu
        </contact>
        <instanceLocation locationType="http">
           http://www.coin-or.org/OS/rosenbrockmod.osil
        </instanceLocation>
    </general>
    <optimization>
        <other name="solverAlg">barrier</other>
        <other name="feasTol">1.0e-6</other>
    </optimization>
</osol>
```

Figure 15: An example of an OSoL file.

An optional <general> element may be included to make certain associations between an options file and a model instance. In our example, the <general> element has two children. One is a <contact> child whose attribute transportType has value smtp, which tells the solver to email the results of the optimization to the address contained in the element. There is also an <instanceLocation> child with locationType attribute equal to http. This indicates the location of the model instance, enabling a client machine to tell a solver server where the model instance can be obtained. The client then does not need to actually send the model instance, only its location.

The OSoL protocol is also used to pass options to registry servers, discussed in Section 3.6.

3.5 In-Memory Representation

Thus far we have only discussed XML file standards for representing optimization model instances. A modeling system would normally first create an instance within internal data structures, however, and then write it to an OSiL file for transmission over the network

to a solver service as in Figure 5. Subsequently the solver service would read this OSiL file, extract the optimization instance, and put that information into its own internal data structures for use by the optimizing algorithm.

One of our major design decisions was to provide some standardization of this process, by designing an `OSInstance` class whose structure exactly parallels that of the OSiL schema. Thus there is a standard way to represent OSiL instances in memory as well as in XML files.

To support the `OSInstance` class, we have written open-source C++ libraries that provide an interface to the model instance. This interface consists of a set of methods (functions) that allow the user of the OS library to perform tasks of three kinds:

- ▷ Extract information about an instance in memory — number of variables, lower and upper bounds on constraints, and indeed any information that may be represented in an OSiL file — through a collection of `get()` methods.
- ▷ Similarly, create or modify an instance in memory though a collection of `set()` methods.
- ▷ Provide function, gradient, and Hessian evaluations, and Jacobian and Hessian sparsity patterns, through a collection of `calculate()` methods.

The `get()` and `set()` methods are similar in concept to what might be found in a library for interfacing to any XML format. The `calculate()` methods are specific to optimization, however; they are designed for nonlinear solvers that require function and derivative computations to be carried out externally. Currently these methods are implemented by linking to CppAD (`www.coin-or.org/CppAD/Doc/cppad.xml`), an open-source package for algorithmic differentiation that is hosted by COIN-OR.

We have similarly designed `OSResult` and `OSOptions` classes to parallel the OSrL and OSoL schemas, and have written analogous `get()` and `set()` methods. The availability of our C++ libraries is discussed further in Section 5.

3.6 Representation Protocols for Registry Servers

When an optimization service registers with a registry service it must provide detailed information about the problems it can solve. For example, is it only a linear programming solver, or does it allow integer variables? If it is a nonlinear solver, does it seek a globally optimal solution, or does it search for only locally optimal points? In the OS framework, the information that the optimization service must provide to the registry service is specified using the OSeL (Optimization Services entity Language) protocol. Like the other protocols we have

introduced, OSeL is specified by a schema. A key element of OSeL is `<optimizationType>`, which contains numerous children such as `<contraintType>` and `<variableType>` that spell out the optimization problems that the solver can handle.

If a client is to communicate with a suitable optimization server in peer-to-peer fashion, it must have the address (URL) of the server. In order to determine which servers support the appropriate solver type, the client will query the OS registry server. To query the database on the registry server, clients use the Optimization Services query Language (OSqL). The OSqL schema specifies, for example, an `<optimizationType>` element matching that of the OSeL schema. This allows for the symmetric registration and querying of optimization problem types.

It is important to observe that the OS Framework does not specify how the registry service should parse the OSqL query and use the information to query the registry database. If the registry database is in XML format, one possibility is to parse the OSqL file to build a query in XQuery, which is then executed against the registry database. XQuery is a standard specified by the W3C for querying XML databases; it is to an XML database what SQL is to a relational database. It is attractive for its consistency with the XML orientation of the OS Framework. The registry service could also be implemented using a standard kind of relational database, however.

How the OSqL query is communicated to the OS registry is specified in the Optimization Services discover Language (OSdL), a WSDL document; this protocol is discussed in Section 4.2. Clients get location information about optimization service solvers from the registry as a sequence of URIs (or URLs), whose syntax is specified in the Optimization Services uri Language (OSuL).

4. Optimization Services Communication

We have described OS protocols for representing model instances, optimization results, and solver options. In a distributed computing environment these representations must be *communicated* between the service consumers and service providers. Sections 4.1 and 4.2 describe the OS protocols for communicating with solver servers and registry servers, respectively.

4.1 Communication Protocols for Solver Servers

For effective communication between a consumer and a provider under the SOA paradigm (Figure 4), service consumers must tell service providers exactly "what to do." Similarly, service providers must tell service consumers what "they can do."

In a Web Services implementation of a service oriented architecture, a service provider communicates to consumers its *capabilities* — the set of functions or methods that it can perform — using WSDL (Web Services Discovery Language). A WSDL document is written in XML and provides a listing of these methods along with their inputs (arguments) and outputs. In the OS project the WSDL communication protocol for this purpose is OShL (Optimization Services hookup Language), which describes the methods to be used in communication between solvers and clients. For example, when communicating with the provider of an optimization service, a natural method is `solve()`. Thus the service consumer will request a `solve()` service from the solver service provider. Figure 16 shows the WSDL that defines the `solve()` method. The method takes a `solveRequest` and responds with a `solveResponse`.

```
<operation name="solve" parameterOrder="osil osol">
     <input name="solveRequest" message="os:solveRequest"/>
     <output name="solveResponse" message="os:solveResponse"/>
</operation>
```

Figure 16: WSDL defining the `solve()` method in the OShL protocol.

Figure 17 shows the WSDL that provides the details of the `solveRequest` and of the `solveResponse`. A `solveRequest` requires two string arguments: the first argument `osil` is the model instance in OSiL format and the second argument `osol` is the solver options in OSrL format. The `solveResponse` is `osrl` which is the solution in an OSrL string.

```
<message name="solveRequest">
     <part name="osil" type="xsd:string"/>
     <part name="osol" type="xsd:string"/>
</message>
<message name="solveResponse">
     <part name="osrl" type="xsd:string"/>
</message>
```

Figure 17: Details of Figure 16's `solveRequest` and `solveResponse`.

Figure 18 shows the inputs and outputs of the six methods that constitute OShL. The purposes of these methods are as follows:

- ◇ `solve()` performs *synchronous* communication with the server. It submits a model instance and waits for the solution.
- ◇ `send()` performs *asynchronous* communication with the server. This method requires a `<jobID>` element in the `osol` string. It returns true if the problem was successfully submitted, false otherwise; it can be used with `knock()` to see if a job is ready and with `retrieve()` to get the results back.
- ◇ `getJobID()` is used to maintain session and state on a distributed system. The JobID returned can be used as input in the `osol` string for `send()`.
- ◇ `knock()` requests process and job status information from the remote server in OSpL (Optimization Services process Language) format. This method can be used to see if a job is complete, and if so, `retrieve()` can be used to get the result.
- ◇ `retrieve()` gets results from a solver. Like `send()`, this method requires a `<jobID>` element in the `osol` string..
- ◇ `kill()` terminates a job on the server. This method can be used to abort long-running jobs or jobs for which there was input error. It is particularly important to an SOA for optimization on account of the pronounced unpredictability of solver performance.

Refer to this paper's online supplement, Fourer, Ma, and Martin (2008), for a detailed example illustrating the OShL protocols.

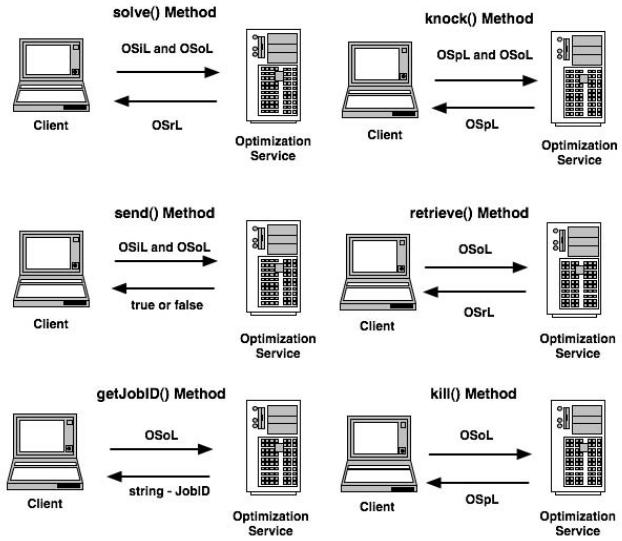

Figure 18: The OS hookup Language (OShL) communication methods.

As we discussed in Section 2.3 and illustrated in Figure 6, service providers and service consumers communicate with each other through SOAP. If, for example, a service consumer wished to retrieve the result of an optimization, the consumer could send a SOAP envelope in the body of an HTTP message to the solver server. The SOAP envelope would contain a `<retrieve>` element. The `retrieve()` method has a single argument, `osol`. Thus `<retrieve>` has a child element, `<osol>`. The `<osol>` element then contains the actual solver options in XML format conforming to the OSoL schema. Therefore we have a string in the OSoL protocol packed inside an element in the OShL protocol in a SOAP envelope in an HTTP body.

An additional benefit of using WSDL is that when one implements an OS Web service on the server side, one can use OShL as a reference and take advantage of software tools that automatically generate much of the needed server-side code.

4.2 Communication Protocols for Registry Servers

In the OS framework, clients must be able to *discover* the locations of optimization solvers in order to initiate direct peer-to-peer communication with them. Optimization solvers must be able to *register* their services. The Optimization Services discover Language (OSdL) specifies a protocol for communicating with the registry server in order to both register and discover optimization services. Like OShL, OSdL is specified using WSDL. The two key methods described by OSdL are `find()` and `register()`, whose inputs and outputs are shown in Figure 19:

- ◇ `find()` is used to discover an optimization service. Its arguments are an OSqL string that contains the query commands for finding an appropriate optimization solver (for example, one that can solve nonlinear optimization problems), and an OSoL string that specifies options to the query (for example, a limit on the number of results returned). The method returns an OSuL string that contains the URLs for solvers capable of handling the specified problems.
- ◇ `register()` is used by the optimization service to register itself with the registry service. The information about the optimization service is passed to the registry service using the entity (OSeL) protocol.

OSqL, OSuL, and OSeL are all representation protocols that were described in Section 3.6.

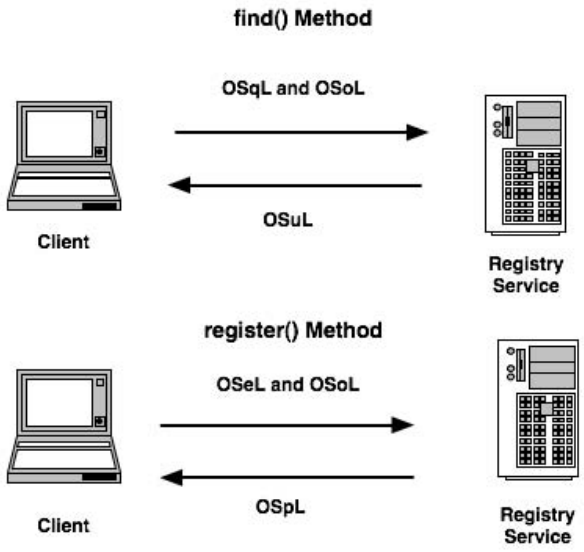

Figure 19: The OS discovery Language (OSdL) communication methods.

5. The COIN-OR Open Source OS Project

In order to provide a reference implementation and test the OS framework, we have implemented many of the protocols described in this paper in C++ and Java libraries. This code is the basis of an open-source project within COIN-OR (COmputational INfrastructure for Operations Research, `projects.coin-or.org/OS`). The COIN-OR OS project provides the following:

- ◇ A library of classes for reading and writing files in the OSiL, OSrL, and OSoL formats.
- ◇ A library that can be used to create Web Services SOAP packages containing OSiL instances and to contact a server for solution.
- ◇ A robust solver and modeling language interface for linear and nonlinear optimization problems, including the `get()`, `set()`, and `calculate()` methods described in Section 3.5.
- ◇ A command-line executable, `OSSolverService`, for reading problem instances — in OSiL format, AMPL nl format, or MPS format — and calling a solver either locally or on a remote server. The `OSSolverService` implements the six OShL methods described in Section 4.1.
- ◇ Server software that works with Apache Tomcat and Apache Axis to provide a Web Services implementation, `OSSolverService.jws`, that acts as middleware between the remote client that submits the instance and the server on which a solver optimizes the instance and returns the result. This software implements the six OShL methods on the server end.

- ◇ A program `OSAmplClient` that appears as a "solver" to the AMPL modeling environment and, based on options given in AMPL, contacts OS solvers either remotely or locally to solve instances created in the AMPL modeling language. The optimization result in OSrL format is then translated back into a format understandable by AMPL for displaying results.
- ◇ Utilities that convert MPS format (`www.mcs.anl.gov/OTC/Guide/OptWeb/continuous/constrained/linearprog/mps.html`) and AMPL nl format (`www.ampl.com/hooking.html`) into the OSiL XML format.
- ◇ A lightweight version of the project, `OSCommon`, for modeling language and solver developers who want to use the OS API, readers, and writers without the overhead of other COIN-OR projects or any third-party software.

This paper's online supplement shows how the COIN-OR software can be used to call a remote server. It includes a detailed illustration of using the `OSSolverService` in conjunction with the OShL methods.

6. Conclusions and Future Work

We have completed a framework for applying optimization as a software service. Ongoing research is directed into two areas.

First, we are working to extend the libraries to new classes of optimization problems, including extensions for semidefinite and cone programming, robust optimization, disjunctive programming, constraint programming, and stochastic programming (see Fourer et al. (2007)).

Second, we are working to gain acceptance of the OS standards. We hope to use the COIN-OR project as springboard to get both solver developers and modeling language developers to adopt the OS framework.

The COIN-OR OS libraries currently support the commercial solvers CPLEX, KNITRO, and LINDO, in addition to the open-source COIN-OR solvers CBC, CLP, DyLP, IPOPT, SYMPHONY, and VOL. The GNU GLPK solver is also supported by the OS libraries. The MOSEK ApS optimization solver and the Frontline Systems Solver Platform SDK currently support OSiL for problem instance representation of mixed integer linear programs.

Support is also being developed for modeling languages. The COIN-OR project includes an executable `OSAmplClient` that can be called from inside the AMPL modeling language (Fourer et al. (2003), `www.ampl.com`), much as Kestrel (Dolan et al. (2008)) allows NEOS

solvers to be invoked from within AMPL. A similar feature is available for the GAMS modeling language (Brooke et al. (1988), `www.gams.com`) through the COIN-OR project GAMSlinks (`projects.coin-or.org/GAMSlinks`); a prototype currently supports mixed-integer linear programming. We illustrate the use of OS with AMPL and GAMS in the Online Supplement to this paper. We also plan to develop similar features for the LINGO modeling language (`www.lindo.com`).

LogicBlox (`www.logicblox.com`), a developer of online predictive and optimization software, is currently developing a product based on Optimization Services. This product allows users to develop optimization models through a Web-based graphical user interface. A model instance is converted to OSiL and then sent to a solver on a local or remote machine; the underlying result is returned as OSrL where it is then converted into a more user-friendly solution report. A browser is the only required software on the client. This is a true example of optimization as a service.

Acknowledgments

This paper has benefited greatly from careful reading and comments by Gail Honda.

References

Brooke, A., Kendrick, D., Meeraus, A. 1988. *GAMS, A User's Guide*. Scientific Press, Redwood City, CA. Updated at `www.gams.com/docs/gams/GAMSUsersGuide.pdf`.

Czyzyk, J., Owen, J.H., Wright, S.J. 1997. Optimization on the Internet. *OR/MS Today* **24**(5) 48–51.

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

Dolan, E.D., Fourer, R., Goux, J.-P., Munson, T.S., Sarich, J. 2008. Kestrel: An Interface from Modeling Systems to the NEOS Server. Published ahead of print in *INFORMS Journal on Computing.*

Fourer, R., Gassmann, H.I., Ma, J., Martin, K. 2007. An XML-Based Schema for Stochastic Programs. To appear in *Annals of Operations Research.*

Fourer, R., Gay, D.M., Kernighan, B.W. 2003. *AMPL: A Modeling Language for Mathematical Programming,* 2nd edition. Brooks/Cole, Cengage Learning, Florence, KY.

Fourer, R., Goux, J.-P. 2001. Optimization as an Internet Resource. *Interfaces* **31**(2) 130–150.

Fourer, R., Ma, J., Martin, K. 2007. OSiL: An Instance Language for Optimization. Forthcoming in *Computational Optimization and Applications.* `www.springerlink.com/content/34jx62447n33w634`.

Fourer, R., Ma, J., Martin, K. 2008. Optimization Services: A Framework for Distributed Optimization. Online Supplement. `gsbkip.chicagogsb.edu/optimizationServicesOnline.pdf`.

Ma, J. 2005. Optimization services (OS), a General Framework for Optimization Modeling Systems. Ph.D. Dissertation, Department of Industrial Engineering & Management Sciences, Northwestern University, Evanston, IL. `https://www.coin-or.org/OS/publications/Thesis2005.pdf`.

Rosenbrock, H.H. 1960. An Automatic Method for Finding the Greatest or Least Value of a Function. *Computer Journal* **3** 175–184.

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