Using the CoinEasy Software

Horand Gassmann, Jun Ma, Kipp Martin

May 27, 2012

Abstract

This is the User's Manual for the CoinEasy project which is designed to help new users of COIN-OR get up and running. It is intended for users of COIN-OR who are interested in solving optimization problems but *do not* wish to compile source code. In particular, we show how the user can send optimization problems to a COIN-OR server and get results back. The COIN-OR server, `webdss.ise.ufl.edu`, is 2x Intel(R) Xeon(TM) CPU 3.06GHz 512MiB L2 1024MiB L3, 2GiB DRAM, 4x73GiB scsi disk 2xGigE machine.

Contents

List of Figures

List of Tables

1 The Optimization Services (OS) Project

The objective of Optimization Services (OS) is to provide a general framework consisting of a set of standards for representing optimization instances, results, solver options, and communication between clients and solvers in a distributed environment using Web Services. This COIN-OR project provides source code for libraries and executable programs that implement OS standards. See the COIN-OR Trac page `http://projects.coin-or.org/OS` or the Optimization Services Home Page `http://www.optimizationservices.org` for more information.

Like other COIN-OR projects, OS has a versioning system that ensures end users some degree of stability and a stable upgrade path as project development continues. The current stable version of OS is 2.5, and the current stable release is 2.5.0, based on trunk version 4490.

This document provides descriptions for the following components of the OS project:

1. A set of XML based standards for representing optimization instances (OSiL), optimization results (OSrL), and optimization solver options (OSoL). There are other standards, but these are the main ones. The schemas for these standards are described in Section 6.

2. A command line executable `OSSolverService` for reading problem instances (OSiL format, AMPL nl format, MPS format) and calling a solver either locally or on a remote server. This is described in Section 4.

3. Standards that facilitate the communication between clients and optimization solvers using Web Services.

4. An executable program `OSAmplClient` that is designed to work with the AMPL modeling language. The `OSAmplClient` appears as a "solver" to AMPL and, based on options given in AMPL, contacts solvers either remotely or locally to solve instances created in AMPL. This is described in Section 5.1.

2 Quick Roadmap

If you want to:

- Download the OS binaries (executables and libraries) – see Section 3.
- Use the OSSolverService to read files in nl, OSiL, or MPS format and call a solver locally or remotely – see Section 4.
- Use modeling languages to generate model instances in OSiL format – see Section 5.
- Use AMPL to solve problems either locally or remotely with a COIN-OR solver, Cplex, GLPK, or LINDO – see Section 5.1.
- Use GAMS to solve problems either locally or remotely – see Section 5.2.

3 Downloading the CoinAllBinaries

The CoinAll project is actually a meta-project consisting of most of the COIN-OR solvers and supporting utility projects. We describe how to download this project.

Most users will only be interested in obtaining the binaries, which we describe next. It is also possible to obtain the source code for the project, which will be of interest mostly to developers. If binaries are not provided for a particular operating system, it may be possible to build them from the source. For details it is best to start reading the OS web page at `http://projects.coin-or.org/OS/`.

The repositoryof the binaries is at `http://www.coin-or.org/download/binary/OS/`.

The binary distribution for the OS library and executables follows the following naming convention:

```
OS-version_number-platform-compiler-build_options.tgz (zip)
```

For example, OS Release 2.1.0 compiled with the Intel 9.1 compiler on an Intel 32-bit Linux system is:

```
OS-2.1.0-linux-x86-icc9.1.tgz
```

For more detail on the naming convention and examples see:

`https://projects.coin-or.org/CoinBinary/wiki/ArchiveNamingConventions`

After unpacking the `tgz` or `zip` archives, the following folders are available.

bin – this directory has the executables `OSSolverService` and `OSAmplClient`.

include – the header files that are necessary in order to link against the OS library.

lib – the libraries that are necessary for creating applications that use the OS library.

share – license and author information for all the projects used by the OS project.

Files are also provided for an Apache Tomcat Web server along with the associated Web service that can read SOAP envelopes with model instances in OSiL format and/or options in OSoL format, call the `OSSolverService`, and return the optimization result in OSrL format. The naming convention for the server binary is

```
OS-server-version_number.tgz (.zip)
```

For example, the files associated with OS server release 2.0.0 are in the binary distribution

```
OS-server-2.0.0.tgz
```

There is no platform information given since the server and related binaries were written in Java.

Finally for Windows users we provide Visual Studio project files (and supporting libraries and header files) for building projects based on the OS library and libraries used by the OS project. The binary for this is named

```
OS-version_number-VisualStudio.zip
```

For example, the necessary files associated with OS stable 2.4 are in the binary distribution

```
OS-2.4-VisualStudio.zip
```

The binaries provided are based on Visual Studio Express 2008.

4 The OSSolverService

The OSSolverService is a command line executable designed to pass problem instances in either OSiL, AMPL nl, or MPS format to solvers and get the optimization result back to be displayed either to standard output or a specified browser. The OSSolverService can be used to invoke a solver locally or on a remote server. It can communicate with a remote solver both synchronously and asynchronously. At present six service methods are implemented, solve, send, retrieve, getJobID, knock and kill. These methods are explained in more detail in Section 4.4. Only the solve method is available locally.

There are two ways to use the OSSolverService executable. The first way is to use the interactive shell. The interactive shell is invoked by either double clicking on the icon for the OSSolverService executable, or by opening a command window, connecting to the directory holding the executable, and then typing in OSSolverService with no arguments. Using the interactive shell is fairly intuitive and we do not discuss in detail. The second way to use the OSSolverService executable is to provide arguments at the command line. This is discussed next. The command line arguments are also valid for the interactive shell.

4.1 OSSolverService Input Parameters

At present, the OSSolverService takes the following parameters. The order of the parameters is irrelevant. Not all the parameters are required.

osil xxx.osil This is the path information and name of the file that contains the optimization instance in OSiL format. It is assumed that this file is available on the machine that is running OSSolverService. This parameter can be omitted, as there are other ways to specify an optimization instance. Although we endorse the convention that OSiL schema files have the extension .osil, OSoL files have the extension .osol, etc., it is not required. Any other path and file name could be substituted for xxx.osil.

osol xxx.osol This is the path information and name of the file that contains the solver options. It is assumed that this file is available on the machine that is running OSSolverService. It is not necessary to specify this parameter.

osrl xxx.osrl This is the path information and name of the file to which the solver solution will be written upon return. A valid file path must be given on the machine that is running OSSolverService. It is not necessary to specify this parameter. If this parameter is not specified then the solver solution is displayed to the screen.

osplInput xxx.ospl The name of an input file in the OS Process Language (OSpL); this is used as input to the knock method. If serviceMethod knock is specified, this parameter is also needed.

osplOutput xxx.ospl The name of an output file in the OS Process Language (OSpL); this is the output string from the knock and kill method. If not present, the ouput is displayed to the terminal screen.

serviceLocation url This is the URL of the solver service. It is not required, and if not specified it is assumed that the problem is solved locally.

serviceMethod methodName This is the service method to be invoked. The options are solve, send, kill, knock, getJobID, and retrieve. The use of these options is illustrated in the examples below. This parameter is not required, and the default value is solve.

Table 1: Solver configurations

	binaries (Section 3)	UNIX build (See OS User's Manual)	MSVS build (See OS User's Manual)
Bonmin	x	x[1]	x[1,2]
Cbc	x	x	x
Clp	x	x	x
Couenne	x	x[1]	—
DyLP	x	x	—
Ipopt	x	x[1]	x[1,2]
SYMPHONY	x	x	x
Vol	x	x	x

Explanations:
[1]Requires third-party software to be downloaded
[2]Requires Fortran compiler

Table 2: Default solvers

Problem type	Default solver
Linear, continuous	Clp
Linear, integer	Cbc
Nonlinear, continuous	Ipopt
Nonlinear, integer	Bonmin

`mps xxx.mps` This is the path information and name of the MPS file if the problem instance is in MPS format. It is assumed that this file is available on the machine that is running `OSSolverService`. The default file format is OSiL so this option is not required.

`nl xxx.nl` This is the path information and name of the AMPL nl file if the problem instance is in AMPL nl format. It is assumed that this file is available on the machine that is running `OSSolverService`. The default file format is OSiL so this option is not required.

`solver solverName` **Note that this option only has effect for local calls.** For a remote solve or send, put the solver name into the field `<solverToInvoke>` in an OSoL file and specify this file with `osol ...`. Possible values of this parameter depend on the installation. The default configurations can be read off from Table 1. Other solvers supported (if the necessary libraries are present) are `cplex` (Cplex through COIN-OR Osi), `glpk` (GLPK through COIN-OR Osi) and `lindo` (LINDO). If no value is specified for this parameter, then a default solver is used for the (local) `solve` method. The default solver depends on the problem type and can be read off from table 2.

`browser browserName` This parameter is a path to the browser on the local machine. If this optional parameter is specified then the solver result in OSrL format is transformed using XSLT into HTML and displayed in the browser.

`config xxx.config` This optional parameter specifies a path on the local machine to a text file containing values for the input parameters. This is convenient for the user not wishing to constantly retype parameter values.

--help This parameter prints out the list of available options (in essence, this list). Synonyms for `--help` are `-h` and `-?`.

--version This parameter prints version and licence information. `-v` is an acceptable synonym.

The input parameters to the `OSSolverService` may be given entirely in the command line or in a configuration file. We first illustrate giving all the parameters in the command line. The following command will invoke the `Clp` solver on the local machine to solve the problem instance `parincLinear.osil`.

Remark. When invoking the commands below involving `OSSolverService` we assume that the user is connected to the directory where the `OSSolverService` executable is located. If the binary download was successful, the `OSSolverService` is in the `bin` directory, and the relative path to the data directory is `../examples/data`. There are several subdirectories corresponding to different file types used and illustrated in the following examples. The user may wish to execute `OSSolverService` from the `bin` directory so that all that follows is correct in terms of path definitions.

```
./OSSolverService solver clp osil ../examples/data/osilFiles/parincLinear.osil
```

Alternatively, these parameters can be put into a configuration file. Assume that the configuration file of interest is `testlocalclp.config`. It would contain the two lines of information

```
osil ../examples/data/osilFiles/parincLinear.osil
solver clp
```

Then the command line is

```
./OSSolverService config ../examples/data/configFiles/testlocalclp.config
```

Windows users should **note** that the folder separator is always the forward slash ('/') instead of the customary backslash ('\').

Parameters specified in the configure file are overridden by parameters specified at the command line. This is convenient if a user has a base configure file and wishes to override only a few options. For example,

```
./OSSolverService config ../examples/data/configFiles/testlocalclp.config solver lindo
```

or

```
./OSSolverService solver lindo config ../examples/data/configFiles/testlocalclp.config
```

will result in the LINDO solver being used even though `Clp` is specified in the `testlocalclp` configure file.

Some things to note:

1. The default `serviceMethod` is `solve` if another service method is not specified. The service method cannot be specified in the OSoL options file.

2. The command line parameters are intended to only influence the behavior of the local `OSSolverService`. In particular, only the service method is transmitted to a remote location. Any communication with a remote solver other than setting the service method **must** take place through an OSoL options file.

3. Only the `solve()` method is available for local calls to `OSSolverService`.
4. If the options `send`, `kill`, `knock`, `getJobID`, or `retrieve` are specified, a `serviceLocation` must be specified.
5. When using the `send()` or `solve()` methods a problem instance must be specified.

4.2 The Command Line Parser

The top layer of the local OSSolverService is a command line parser that parses the command line and the config file (if one is specified) and passes the information on to a local solver or a remote solver service, depending on whether a `serviceLocation` was specified. If a `serviceLocation` is specified a call is made to a remote solver service, otherwise a local solver is called.

If a local solve is indicated, we pass to a solver in the OSLibrary two things: an OSoL file if one has been specified and a problem instance. The problem instance is the instance in the OSiL file specified by the `osil` option. If there is no OSiL file, then it is the instance specified in the nl file. If there is no nl file, it is the instance in the mps file. If no OSiL, nl or mps file is specified, an error is thrown.

The OSoL file is simply passed on to the OSLibrary; it is not parsed at this point. The OSoL file elements `<solverToInvoke>` and `<instanceLocation>` cannot be used for local calls. One can specify which solver to use in the OSLibrary through the `solver` option. If this option is empty, a default solver is selected (see Table 2).

If the `serviceLocation` parameter is used, a call is placed to the remote solver service specified in the `serviceLocation` parameter. Two strings are passed to the remote solver service: a string which is the OSoL file if one has been specified, or the empty string otherwise, and a string containing an instance if one has been specified. The instance can be specified using the `osil`, `nl`, or `mps` option. If an OSiL file is specified in the `osil` option, it is used. If there is no OSiL file, then the instance specified in the nl file is used. If there is no nl file, the mps file is used. If no file is given, an empty string is sent.

For remote calls, the solver can only be set in the osol file, using the element `<solverToInvoke>`; the `solver` option has no effect.

4.3 Solving Problems Locally

Generally, when solving a problem locally the user will use the `solve` service method. The `solve` method is invoked synchronously and waits for the solver to return the result. This is illustrated in Figure 1. As illustrated, the `OSSolverService` reads a file on the hard drive with the optimization instance, usually in OSiL format. The optimization instance is parsed into a string which is passed to the `OSLibrary` (see the OS User's Manual), which is linked with various solvers. Similarly an option file in OSoL format is parsed into a string and passed to the `OSLibrary`. *No interpretation of the options is done at this stage*, so that any `<solverToInvoke>` or `<instanceLocation>` directives in the OSoL file will be ignored for local solves. The result of the optimization is passed back to the `OSSolverService` as a string in OSrL format.

Here is an example of using a configure file, `testlocal.config`, to invoke `Ipopt` locally using the `solve` command.

```
osil ../examples/data/osilFiles/parincQuadratic.osil
solver ipopt
serviceMethod solve
```

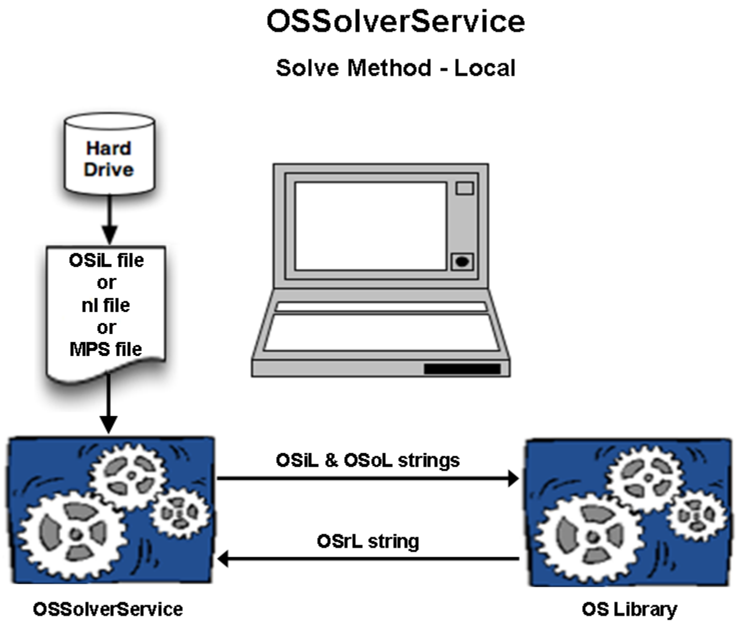

Figure 1: A local call to `solve`.

```
browser /Applications/Firefox.app/Contents/MacOS/firefox
osrl /Users/kmartin/temp/test.osrl
```

The first line of `testlocal.config` gives the local location of the OSiL file, `parincQuadratic.osil`, that contains the problem instance. The second parameter, `solver ipopt`, is the solver to be invoked, in this case COIN-OR Ipopt. The third parameter `serviceMethod solve` is not really needed, since the default solver service is `solve`. It is included only for illustration. The fourth parameter is the location of the browser on the local machine. The fifth parameter is `osrl`. The value of this parameter, `/Users/kmartin/temp/test.osrl`, specifies the location on the local machine where the OSrL result file will get written.

4.4 Solving Problems Remotely with Web Services

In many cases the client machine may be a "weak client" and using a more powerful machine to solve a hard optimization instance is required. Indeed, one of the major purposes of Optimization Services is to facilitate optimization in a distributed environment. We now provide examples that illustrate using the `OSSolverService` executable to call a remote solver service. By remote solver service we mean a solver service that is called using Web Services. The OS implementation of the solver service uses Apache Tomcat. See `tomcat.apache.org`. The Web Service running on the server is a Java program based on Apache Axis. See `ws.apache.org/axis`. This is described in greater detail in the OS User's Manual. This Web Service is called `OSSolverService.jws`. It is not necessary to use the Tomcat/Axis combination.

See Figure 2 for an illustration of this process. The client machine uses `OSSolverService` executable to call one of the six service methods, e.g., `solve`. The information such as the problem instance in OSiL format and solver options in OSoL format are packaged into a SOAP envelope

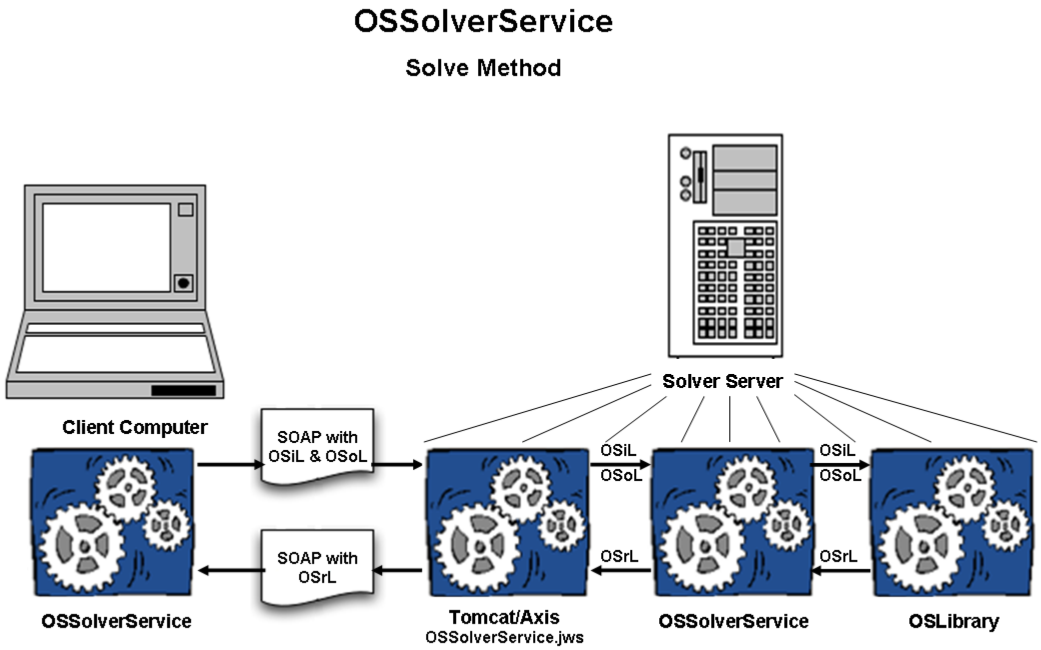

Figure 2: A remote call to `solve`.

and sent to the server. The server is running the Java Web Service `OSSolverService.jws`. This Java program running in the Tomcat Java Servlet container implements the six service methods. If a `solve` or `send` request is sent to the server from the client, an optimization problem must be solved. The Java solver service solves the optimization instance by calling the `OSSolverService` on the server. So there is an `OSSolverService` on the client that calls the Web Service `OSSolverService.jws` that in turn calls the executable `OSSolverService` on the server. The Java solver service passes options to the server's `OSSolverService` in form of two strings, an osil string representing the instance and an osol string representing the options (if any).

For remote calls the instance location can be specified either as a command parameter (on the command line or in a config file) or through the `<instanceLocation>` element in the OSoL options file. OSiL files specified in the `<instanceLocation>` element must be converted to an osil string by the solver service. If two instance files are specified in this way — one through the local command interface, the other in an options file — the information on the command line takes precedent.

In the following sections we illustrate each of the six service methods.

4.4.1 The `solve` Service Method

First we illustrate a simple call to `OSSolverService.jws`. The call on the client machine is

```
./OSSolverService config ../examples/data/configFiles/testremote.config
```

where the `testremote.config` file is

```
osil ../examples/data/osilFiles/parincLinear.osil
serviceLocation http://kipp.chicagobooth.edu/os/OSSolverService.jws
```

No solver is specified and by default the `Clp` solver is used by the `OSSolverService`, since the problem is a continuous linear program. If, for example, the user wished to solve the problem with the `SYMPHONY` solver then this is accomplished either by using the `solver` option on the command line

```
./OSSolverService config ../examples/data/configFiles/testremote.config solver symphony
```

or by adding the line

```
 solver symphony
```

to the `testremote.config` file. When solver information is given both on the command line and in the config file, the command line information supercedes the config file.

Next we illustrate a call to the remote SolverService and specify an OSiL instance that is actually residing on the remote machine that is hosting the `OSSolverService` and not on the client machine.

```
./OSSolverService osol ../examples/data/osolFiles/remoteSolve1.osol
     serviceLocation  http://kipp.chicagobooth.edu/os/OSSolverService.jws
```

where the `remoteSolve1.osol` file is

```
<?xml version="1.0" encoding="UTF-8"?>
<osol xmlns="os.optimizationservices.org"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="os.optimizationservices.org
      http://www.optimizationservices.org/schemas/2.0/OSiL.xsd">
    <general>
        <instanceLocation locationType="local">c:\parincLinear.osil</instanceLocation>
        <contact transportType="smtp">kipp.martin@chicagogsb.edu</contact>
        <solverToInvoke>ipopt</solverToInvoke>
    </general>
</osol>
```

If we were to change the `locationType` attribute in the `<instanceLocation>` element to `http` then we could specify the instance location on yet another machine. This is illustrated below for `remoteSolve2.osol`. The scenario is depicted in Figure 3. The OSiL string passed from the client to the solver service is empty. However, the OSoL element `<instanceLocation>` has an attribute `locationType` equal to `http`. In this case, the text of the `<instanceLocation>` element contains the URL of a third machine which has the problem instance `parincLinear.osil`. The solver service will contact the machine with URL `http://www.coin-or.org/OS/parincLinear.osil` and download this test problem. So the `OSSolverService` is running on the server `kipp.chicagobooth.edu` which contacts the server `www.coin-or.org` for the model instance.

```
<?xml version="1.0" encoding="UTF-8"?>
<osol xmlns="os.optimizationservices.org"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="os.optimizationservices.org
      http://www.optimizationservices.org/schemas/2.0/OSiL.xsd">
    <general>
        <instanceLocation locationType="http">
```

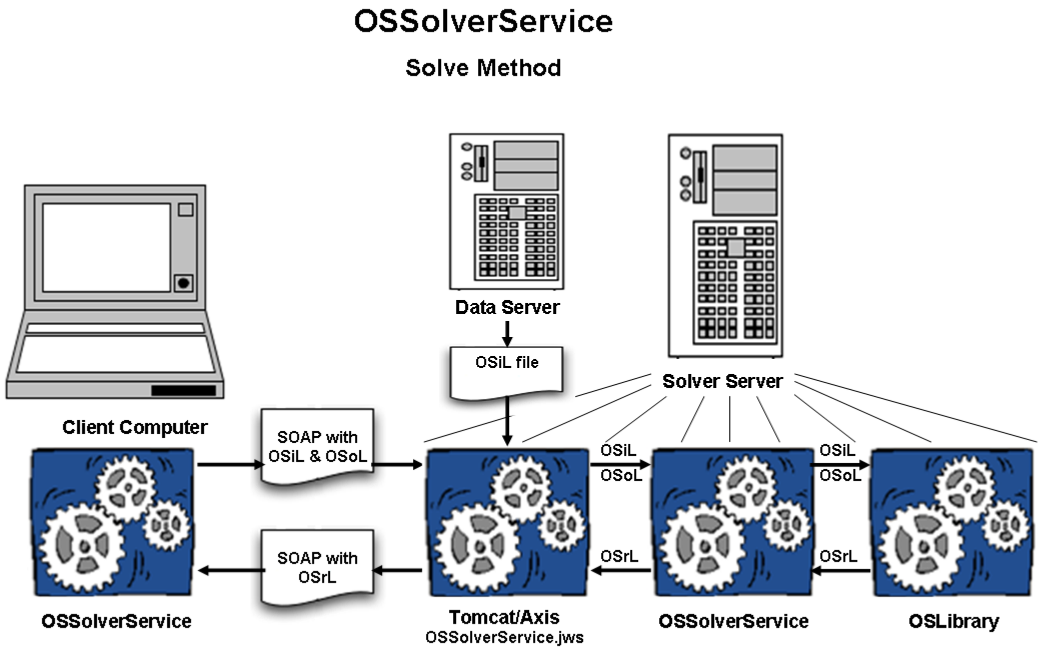

Figure 3: Downloading the instance from a remote source.

```
            http://www.coin-or.org/OS/parincLinear.osil
        </instanceLocation>
        <solverToInvoke>ipopt</solverToInvoke>
    </general>
</osol>
```

Note: The `solve` method communicates synchronously with the remote solver service and once started, these jobs cannot be killed. This may not be desirable for large problems when the user does not want to wait for a response or when there is a possibility for the solver to enter an infinite loop. The `send` service method should be used when asynchronous communication is desired.

4.4.2 The `send` Service Method

When the `solve` service method is used, then the `OSSolverService` does not finish execution until the solution is returned from the remote solver service. When the `send` method is used, the instance is communicated to the remote service and the `OSSolverService` terminates after submission. An example of this is

```
./OSSolverService config ../examples/data/configFiles/testremoteSend.config
```

where the `testremoteSend.config` file is

```
-nl ../examples/data/amplFiles/hs71.nl
serviceLocation http://kipp.chicagobooth.edu/os/OSSolverService.jws
serviceMethod send
```

In this example the COIN-OR `Ipopt` solver is specified. The input file `hs71.nl` is in AMPL nl format. Before sending this to the remote solver service the `OSSolverService` executable converts the nl format into the OSiL XML format and packages this into the SOAP envelope used by Web Services.

Since the `send` method involves asynchronous communication the remote solver service must keep track of jobs. The `send` method requires a `JobID`. In the above example no `JobID` was specified. When no `JobID` is specified the `OSSolverService` method first invokes the `getJobID` service method to get a `JobID`, puts this information into an OSoL file it creates, and sends the information to the server. More information on the `getJobID` service method is provided in Section 4.4.4. The `OSSolverService` prints the OSoL file to standard output before termination. This is illustrated below,

```
<?xml version="1.0" encoding="UTF-8"?>
<osol xmlns="os.optimizationservices.org"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="os.optimizationservices.org
      http://www.optimizationservices.org/schemas/2.0/OSiL.xsd">
    <general>
        <jobID>
            gsbrkm4__127.0.0.1__2007-06-16T15.46.46.075-05.00149771253
        </jobID>
        <solverToInvoke>ipopt</solverToInvoke>
    </general>
</osol>
```

The `JobID` is one that is randomly generated by the server and passed back to the `OSSolverService`. The user can also provide a `JobID` in their OSoL file. For example, below is a user-provided OSoL file that could be specified in a configuration file or on the command line.

```
<?xml version="1.0" encoding="UTF-8"?>
<osol xmlns="os.optimizationservices.org"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="os.optimizationservices.org
      http://www.optimizationservices.org/schemas/2.0/OSiL.xsd">
    <general>
        <jobID>123456abcd</jobID>
        <solverToInvoke>ipopt</solverToInvoke>
    </general>
</osol>
```

The same `JobID` cannot be used twice, so if `123456abcd` was used earlier, the result of `send` will be an error condition.

In order to be of any use, it is necessary to get the result of the optimization. This is described in Section 4.4.3. Before proceeding to this section, we describe two ways for knowing when the optimization is complete. One feature of the standard OS remote SolverService is the ability to send an email when the job is complete. Below is an example of the `OSoL` that uses the email feature.

```
<?xml version="1.0" encoding="UTF-8"?>
<osol xmlns="os.optimizationservices.org"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
      xsi:schemaLocation="os.optimizationservices.org
      http://www.optimizationservices.org/schemas/2.0/OSiL.xsd">
    <general>
        <jobID>123456abcd</jobID>
        <contact transportType="smtp">
            kipp.martin@chicagogsb.edu
        </contact>
        <solverToInvoke>ipopt</solverToInvoke>
    </general>
</osol>
```

The remote Solver Service will send an email to the above address when the job is complete. A second option for knowing when a job is complete is to use the `knock` method. (See Section 4.4.5.)

Note that in all of these examples we provided a value for the `<solverToInvoke>` element. A default solver is used (see Table 2) if another solver is not specified.

4.4.3 The `retrieve` Service Method

The `retrieve` method is used to get information about the instance solution. This method has a single string argument which is an OSoL instance. Here is an example of using the `retrieve` method with `OSSolverService`.

```
./OSSolverService config ../examples/data/configFiles/testremoteRetrieve.config
```

The `testremoteRetrieve.config` file is

```
serviceLocation http://kipp.chicagobooth.edu/os/OSSolverService.jws
osol ../examples/data/osolFiles/retrieve.osol
serviceMethod retrieve
osrl /home/kmartin/temp/test.osrl
```

and the `retrieve.osol` file is

```
<?xml version="1.0" encoding="UTF-8"?>
<osol xmlns="os.optimizationservices.org"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="os.optimizationservices.org
      http://www.optimizationservices.org/schemas/2.0/OSiL.xsd">
    <general>
        <jobID>123456abcd</jobID>
    </general>
</osol>
```

The OSoL file `retrieve.osol` contains a tag `<jobID>` that is communicated to the remote service. The remote service locates the result and returns it as a string. The `<jobID>` should reflect a `<jobID>` that was previously submitted using a `send()` command. The result is returned as a string in OSrL format. The user must modify the line

```
osrl /home/kmartin/temp/test.osrl
```

to reflect a valid path for their own machine. (It is also possible to delete the line in which case the result will be displayed on the screen instead of being saved to the file indicated in the `osrl` option.)

4.4.4 The getJobID Service Method

Before submitting a job with the `send` method a `JobID` is required. The `OSSolverService` can get a `JobID` with the following options.

```
serviceLocation http://kipp.chicagobooth.edu/os/OSSolverService.jws
serviceMethod getJobID
```

Note that no OSoL input file is specified. In this case, the `OSSolverService` sends an empty string. A string is returned with the `JobID`. This `JobID` is then put into a `<jobID>` element in an OSoL string that would be used by the `send` method.

4.4.5 The knock Service Method

The OSSolverService terminates after executing the `send` method. Therefore, it is necessary to know when the job is completed on the remote server. One way is to include an email address in the `<contact>` element with the attribute `transportType` set to `smtp`. This was illustrated in Section 4.4.1. A second way to check on the status of a job is to use the `knock` service method. For example, assume a user wants to know if the job with `JobID 123456abcd` is complete. A user would make the request

```
./OSSolverService config ../examples/data/configFiles/testRemoteKnock.config
```

where the `testRemoteKnock.config` file is

```
serviceLocation http://kipp.chicagobooth.edu/os/OSSolverService.jws
osplInput ../examples/data/osolFiles/demo.ospl
osol ../examples/data/osolFiles/retrieve.osol
serviceMethod knock
```

the `demo.ospl` file is

```
<?xml version="1.0" encoding="UTF-8"?>
<ospl xmlns="os.optimizationservices.org">
    <processHeader>
        <request action="getAll"/>
    </processHeader>
    <processData/>
</ospl>
```

and the `retrieve.osol` file is as in Section 4.4.3.

The result of this request is a string in OSpL format, with the data contained in its `processData` section. The result is displayed on the screen; if the user desires it to be redirected to a file, a command should be added to the `testRemoteKnock.config` file with a valid path name on the local system, e.g.,

```
osplOutput ./result.ospl
```

Part of the return format is illustrated below.

```
<?xml version="1.0" encoding="UTF-8"?>
<ospl xmlns="os.optimizationservices.org">
  <processHeader>
    <serviceURI>http://localhost:8080/os/ossolver/CGSolverService.jws</serviceURI>
    <serviceName>CGSolverService</serviceName>
    <time>2006-05-10T15:49:26.7509413-05:00</time>
  <processHeader>
  <processData>
     <statistics>
        <currentState>idle</currentState>
        <availableDiskSpace>23440343040</availableDiskSpace>
        <availableMemory>70128</availableMemory>
        <currentJobCount>0</currentJobCount>
        <totalJobsSoFar>1</totalJobsSoFar>
        <timeServiceStarted>2006-05-10T10:49:24.9700000-05:00</timeServiceStarted>
        <serviceUtilization>0.1</serviceUtilization>
        <jobs>
        <job jobID="123456abcd">
            <state>finished</state>
            <serviceURI>http://kipp.chicagobooth.edu/ipopt/IPOPTSolverService.jws</serviceURI>
            <submitTime>2007-06-16T14:57:36.678-05:00</submitTime>
            <startTime>2007-06-16T14:57:36.678-05:00</startTime>
            <endTime>2007-06-16T14:57:39.404-05:00</endTime>
            <duration>2.726</duration>
          </job>
        </jobs>
     </statistics>
  </processData>
</ospl>
```

Notice that the `<state>` element in `<job jobID="123456abcd">` indicates that the job is finished.

When making a `knock` request, the OSoL string can be empty. In this example, if the OSoL string had been empty the status of all jobs kept in the file ospl.xml is reported. In our default solver service implementation, there is a configuration file `OSParameter` that has a parameter `MAX_JOBIDS_TO_KEEP` . The current default setting is 100. In a large-scale or commercial implementation it might be wise to keep problem results and statistics in a database. Also, there are values other than `getAll` (i.e., get all process information related to the jobs) for the OSpL `action` attribute in the `<request>` tag. For example, the `action` can be set to a value of `ping` if the user just wants to check if the remote solver service is up and running. For details, check the OSpL schema.

4.4.6 The `kill` Service Method

If the user submits a job that is taking too long or is a mistake, it is possible to kill the job on the remote server using the `kill` service method. For example, to kill job `123456abcd`, at the command line type

```
./OSSolverService config  ../examples/data/configFiles/kill.config
```

where the configure file `kill.config` is

```
osol ../examples/data/osolFiles/kill.osol
serviceLocation http://kipp.chicagobooth.edu/os/OSSolverService.jws
serviceMethod kill
```

and the `kill.osol` file is

```
<?xml version="1.0" encoding="UTF-8"?>
<osol xmlns="os.optimizationservices.org"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="os.optimizationservices.org
      http://www.optimizationservices.org/schemas/2.0/OSiL.xsd">
    <general>
        <jobID>123456abcd</jobID>
    </general>
</osol>
```

The result is returned in OSpL format.

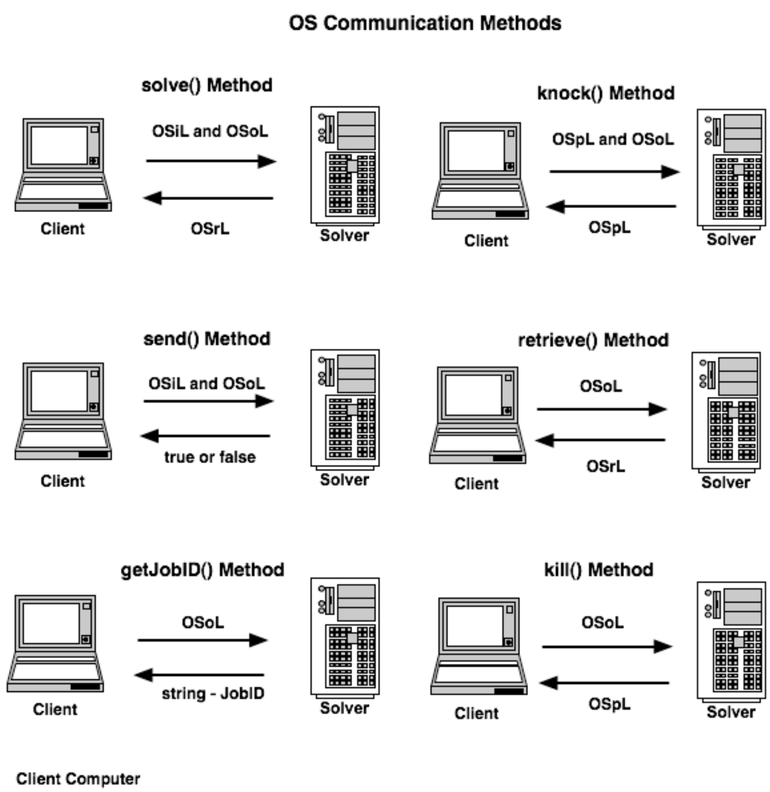

Figure 4: The OS Communication Methods

4.5 Passing Options to Solvers

The OSoL (Optimization Services option Language) protocol is used to pass options to solvers. When using the `OSSolverService` executable this will typically be done through an OSoL XML file by specifying the `osol` option followed by the location of the file. However, it is also possible to write a custom application that links to the OS library and to build an OSOption object in memory and then pass this to a solver. We next describe the feature of the OSoL protocol that will be the most useful to the typical user.

In the OSoL protocol there is an element `<solverOptions>` that can have any number of `<solverOption>` children. (See the file `parsertest.osol` in OS/data/osolFiles.) Each `<solverOption>` child can have six attributes, all of which except one are optional. These attributes are:

- **name:** this is the only required attribute and is the option name. It should be unique.
- **value:** the value of the option.
- **solver:** the name of the solver associated with the option. At present the values recognized by this attribute are `"ipopt"`, `"bonmin"`, `"couenne"`, `"cbc"`, and `"osi"`. The last option is used for all solvers that are accessed through the Osi interface, which are `clp`, `DyLP`, `SYMPHONY` and `Vol`, in addition to `Glpk` and `Cplex`, if the latter are included in the particular build of OSSolverService.
- **type:** this will usually be a data type (such as integer, string, double, etc.) but this is not necessary.
- **category:** the same solver option may apply in more than one context (and with different meaning) so it may be necessary to specify a category to remove ambiguities. For example, in LINDO an option can apply to a specific model or to every model in an environment. Hence we might have

  ```
  <solverOption name="LS_IPARAM_LP_PRINTLEVEL"
     solver="lindo" category="model"  type="integer" value="0"/>
  <solverOption name="LS_IPARAM_LP_PRINTLEVEL"
     solver="lindo" category="environment" type="integer" value="1"/>
  ```

  where we specify the print level for a specific model or the entire environment. The category attribute should be separated by a colon (':') if there is more than one category or additional subcategories, as in the following hypothetical example.

  ```
  <solverOption name="hypothetical"
     solver="SOLVER" category="cat1:subcat2:subsubcat3"
        type="string" value="illustration"/>
  ```

- **description:** a description of the option; typically this would not get passed to the solver.

As of trunk version 2164 the reading of an OSoL file is implemented in the `OSCoinSolver`, `OSBonmin` and `OSIpopt` solver interfaces. The `OSBonmin`, and `OSIpopt` solvers have particularly easy interfaces. They have methods for integer, string, and numeric data types and then take options in form of `(name, value)` pairs. Below is an example of options for `Ipopt`.

```
<solverOption name="mu_strategy" solver="ipopt"
     type="string" value="adaptive"/>
<solverOption name="tol" solver="ipopt"
     type="numeric" value="1.e-9"/>
<solverOption name="print_level" solver="ipopt"
```

```
     type="integer" value="5"/>
<solverOption name="max_iter" solver="ipopt"
     type="integer" value="2000"/>
```

We have also implemented the `OSOption` class for the `OSCoinSolver` interface. This can be done in two ways. First, options can be set through the Osi Solver interface (the OSCoinSolver interface wraps around the Osi Solver interface). We have implemented all of the options listed in `OsiSolverParameters.hpp` in `Osi` trunk version 1316. In the Osi solver interface, in addition to string, double, and integer types there is a type called `HintParam` and a type called `OsiHintParam`. The value of the `OsiHintParam` is an `OsiHintStrength` type, which may be confusing. For example, to have the following Osi method called

```
setHintParam(OsiDoReducePrint, true, hintStrength);
```

the user should set the following `<solverOption>` tags:

```
<solverOption name="OsiDoReducePrint" solver="osi"
    type="OsiHintParam"  value="true" />
<solverOption name="OsiHintIgnore" solver="osi"
     type="OsiHintStrength" />
```

There should be only one `<solverOption>` with type `OsiHintStrength` and if there are more than one in the OSoL file (string) the last one is the one implemented.

In addition to setting options using the Osi Solver interface, it is possible to pass options directly to the `Cbc` solver. By default the following options are sent to the `Cbc` solver,

```
-log=0  -solve
```

The option `-log=0` will keep the branch-and-bound output to a minimum. Default options are overridden by putting into the OSoL file at least one `<solverOption>` tag with the `solver` attribute set to `cbc`. For example, the following sequence of options will limit the search to 100 nodes, cut generation turned off.

```
<solverOption name="maxN" solver="cbc" value="100" />
<solverOption name="cuts" solver="cbc" value="off" />
<solverOption name="solve" solver="cbc"  />
```

Any option that `Cbc` accepts at the command line can be put into a `<solverOption>` tag. We list those below.

```
Double parameters:
  dualB(ound)  dualT(olerance)  primalT(olerance)  primalW(eight)
Branch and Cut double parameters:
  allow(ableGap)  cuto(ff)  inc(rement)  inf(easibilityWeight)  integerT(olerance)
  preT(olerance)  ratio(Gap)  sec(onds)
Integer parameters:
  cpp(Generate)  force(Solution)  idiot(Crash)  maxF(actor)  maxIt(erations)
  output(Format)  slog(Level)  sprint(Crash)
Branch and Cut integer parameters:
  cutD(epth)  log(Level)  maxN(odes)  maxS(olutions)  passC(uts)
  passF(easibilityPump)  passT(reeCuts)  pumpT(une)  strat(egy)  strong(Branching)
  trust(PseudoCosts)
```

```
Keyword parameters:
  chol(esky)  crash  cross(over)  direction  dualP(ivot)
  error(sAllowed)  keepN(ames)  mess(ages)  perturb(ation)  presolve
  primalP(ivot)  printi(ngOptions)  scal(ing)
Branch and Cut keyword parameters:
  clique(Cuts)  combine(Solutions)  cost(Strategy)  cuts(OnOff)  Dins
  DivingS(ome)  DivingC(oefficient)  DivingF(ractional)  DivingG(uided)  DivingL(ineSearch)
  DivingP(seudoCost)  DivingV(ectorLength)  feas(ibilityPump)  flow(CoverCuts)  gomory(Cuts)
  greedy(Heuristic)  heur(isticsOnOff)  knapsack(Cuts)  lift(AndProjectCuts)  local(TreeSearch)
  mixed(IntegerRoundingCuts)  node(Strategy)  pivot(AndFix)  preprocess  probing(Cuts)
  rand(omizedRounding)  reduce(AndSplitCuts)  residual(CapacityCuts)  Rens  Rins
  round(ingHeuristic)  sos(Options)  two(MirCuts)
Actions or string parameters:
  allS(lack)  barr(ier)  basisI(n)  basisO(ut)  directory
  dirSample  dirNetlib  dirMiplib  dualS(implex)  either(Simplex)
  end  exit  export  help  import
  initialS(olve)  max(imize)  min(imize)  netlib  netlibD(ual)
  netlibP(rimal)  netlibT(une)  primalS(implex)  printM(ask)  quit
  restore(Model)  saveM(odel)  saveS(olution)  solu(tion)  stat(istics)
  stop  unitTest  userClp
Branch and Cut actions:
  branch(AndCut)  doH(euristic)  miplib  prio(rityIn)  solv(e)
  strengthen  userCbc
```

The user may also wish to specify an initial starting solution. This is particularly useful with interior point methods. This is accomplished by using the `<initialVariableValues>` tag. Below we illustrate how to set the initial values for variables with an index of 0, 1, and 3.

```
<initialVariableValues numberOfVar="3">
   <var idx="0" value="1"/>
   <var idx="1" value="4.742999643577776" />
   <var idx="3" value="1.379408293215363"/>
</initialVariableValues>
```

As of trunk version 2164 the initial values for variables can be passed to the `Bonmin` and `Ipopt` solvers.

When implementing solver options in-memory, the typical calling sequence is:

```
solver->buildSolverInstance();
solver->setSolverOptions();
solver->solve();
```

5 OS Support for Modeling Languages, Spreadsheets and Numerical Computing Software

Algebraic modeling languages can be used to generate model instances as input to an OS compliant solver. We describe two such hook-ups, `OSAmplClient` for AMPL, and `CoinOS` for GAMS (version 23.8 and above).

5.1 AMPL Client: Hooking AMPL to Solvers

It is possible to call all of the COIN-OR solvers listed in Table 1 (p.7) directly from the AMPL (see `http://www.ampl.com`) modeling language. In this discussion we assume the user has already obtained and installed AMPL. The binary download described in Section 3 contains an executable, `OSAmplClient.exe`, that is linked to all of the COIN-OR solvers listed in Table 1. From the perspective of AMPL, the `OSAmplClient` acts like an AMPL "solver". The `OSAmplClient.exe` can be used to solve problems either locally or remotely.

5.1.1 Using OSAmplClient for a Local Solver

In the following discussion we assume that the AMPL executable `ampl.exe`, the `OSAmplClient`, and the test problem `eastborne.mod` are all in the same directory.

The problem instance `eastborne.mod` is an AMPL model file included in the OS distribution in the `amplFiles` directory. To solve this problem locally by calling `OSAmplClient.exe` from AMPL, first start AMPL and then open the `eastborne.mod` file inside AMPL. The test model `eastborne.mod` is a linear integer program.

```
model eastborne.mod;
```

The next step is to tell AMPL that the solver it is going to use is `OSAmplClient.exe`. Do this by issuing the following command inside AMPL.

```
option solver OSAmplClient;
```

It is not necessary to provide the `OSAmplclient.exe` solver with any options. You can just issue the `solve` command in AMPL as illustrated below.

```
solve;
```

Of the six methods described in Section 4 only the `solve` method has been implemented to date.

If no options are specified, the default solver is used, depending on the problem characteristics (see Table 2 on p.7). If you wish to specify a specific solver, use the `solver` option. For example, since the test problem `eastborne.mod` is a linear integer program, `Cbc` is used by default. If instead you want to use `SYMPHONY`, then you would pass a `solver` option to the `OSAmplclient.exe` solver as follows.

```
option OSAmplClient_options "solver symphony";
```

Valid values for the `solver` option are installation-dependent. The solver name in the `solver` option is case insensitive.

5.1.2 Using OSAmplClient to Invoke an OS Solver Server

Next, assume that you have a large problem you want to solve on a remote solver. It is necessary to specify the location of the server solver as an option to OSAmplClient. The `serviceLocation` option is used to specify the location of a solver server. In this case, the string of options for `OSAmplClient_options` is:

```
serviceLocation  http://xxx/OSServer/services/OSSolverService
```

where xxx is the URL for the server. This string is used to replace the string 'solver symphony' in the previous example. The serviceLocation option will send the problem to the location http://xxx and, assuming the remote executable is indeed found in the indicated folder, will start the executable.

However, each call

```
option OSAmplClient_options
```

is memoryless. That is, the options set in the last call will overwrite any options set in previous calls and cause them to be discarded. For instance, the sequence of option calls

```
option OSAmplClient_options "solver symphony";
option OSAmplClient_options "serviceLocation
    http://xxx/OSServer/services/OSSolverService";
solve;
```

will result in the default solver being called.

If the intent is to use the SYMPHONY solver at the remote location, the option must be declared as follows:

```
option OSAmplClient_options "solver symphony
    serviceLocation http://xxx/OSServer/services/OSSolverService";
solve;
```

For brevity we will omit the AMPL instruction

```
option OSAmplClient_options
```

the double quotes and the trailing semicolon in the remaining examples.

Finally, the user may wish to pass options to the individual solver. This is done by specifying an options file. (A sample options file, solveroptions.osol is provided with this distribution). The name of the options file is the value of the osol option. The string of options to OSAmplClient_options is now

```
serviceLocation http://xxx/OSServer/services/OSSolverService
osol solveroptions.osol
```

This solveroptions.osol file contains four solver options; two for Cbc, one for Ipopt, and one for SYMPHONY. You can have any number of options. Note the format for specifying an option:

```
<solverOption name="maxN" solver="cbc" value="5" />
```

The attribute name specifies that the option name is maxN which is the maximum number of nodes allowed in the branch-and-bound tree, the solver attribute specifies the name of the solver that the option should be applied to, and the value attribute specifies the value of the option. As a second example, consider the specification

```
<solverOption name="max_iter" solver="ipopt" type="integer" value="2000"/>
```

In this example we are specifying an iteration limit for Ipopt. Note the additional attribute type that has value integer. The Ipopt solver requires specifying the data type (string, integer, or numeric) for its options. Different solvers have different options, and we recommend that the user look at the documentation for the solver of interest in order to see which options are available. A good summary of options for COIN-OR solvers is http://www.gams.com/dd/docs/solvers/coin.pdf.

If you examine the file solveroptions.osol you will see that there is an XML tag with the name <solverToInvoke> and that the solver given is symphony. **This has no effect on a local solve!** However, if this option file is paired with

```
serviceLocation http://xxx/OSServer/services/OSSolverService
osol solveroptions.osol
```

then in our reference implementation the remote solver service will parse the file solveroptions.osol, find the <solverToInvoke> tag and then pass the symphony solver option to the OSSolverService on the remote server.

5.1.3 AMPL Summary

1. Tell AMPL to use the OSAmplClient as the solver:

   ```
   option solver OSAmplClient;
   ```

2. Specify options to the OSAmplClient solver by using the AMPL command

   ```
   option OSAmplClient_options "(option string)";
   ```

3. There are three possible options to specify:

   - the location of the options file using the osol option;
   - the location of the remote server using the serviceLocation option;
   - the name of the solver using the solver option; valid values for this option are installation-dependent. For details, see Table 1 on page 7 and the discussion in Section 4.1.

   These three options behave *exactly like* the solver, serviceLocation, and osol options used by the OSSolverService described in Section 4.2. Note that the solver option only has an effect with a local solve; if the user wants to invoke a specific solver with a remote solve, then this must be done in the OSoL file using the <solverToInvoke> element.

4. The options given to OSAmplClient_options can be given in any order.

5. If no solver is specified using OSAmplClient_options, the default solver is used. (For details see Table 2).

6. A remote solver is called if and only if the serviceLocation option is specified.

5.2 GAMS and Optimization Services

This section pertains to GAMS version 23.8 (and above) that now includes support for OS. Here we describe the GAMS implementation of Optimization Services. We assume that the user has installed GAMS.

In GAMS, OS is implemented through the `CoinOS` solver that is packaged with GAMS. The GAMS `CoinOS` solver is really a *solver interface* that links to the OS library. At present the GAMS `CoinOS` solver does not support local calls, but it can be used to make remote calls to an `OSSolverService` executable on a remote server. How this is done is the topic of the next section.

5.2.1 Using GAMS to Invoke a Remote OS Solver Service

We now describe how to call a remote OS solver service using the GAMS `CoinOS`. Before proceeding, it is important to emphasize that when calling a remote OS solver service, different sets of solvers may be supported, even for the same version of the OS solver service. For example, the remote implementation may provide access to solvers such as `SYMPHONY`, `Couenne`, `Glpk` and `DyLP`. There are several reason why you might wish to use a remote OS solver service.

- Have access to a faster machine.
- Be able to submit jobs to run in asynchronous mode – submit your job, turn off your laptop, and check later to see if the job ran.
- Call several additional solvers (e.g., `SYMPHONY`, `Couenne`, `Glpk` and `DyLP`). Note, however, that not all solvers may be available available locally (especially Glpk) may not be available for a remote call.

We will illustrate several possible calls with the sample GAMS file `eastborne.gms` which found in the `data/gamsFiles` directory. We assume that this file exists in the current directory and that the GAMS executable is found in the search path. The command to execute at the command line would then be

```
gams eastborne.gms MIP=CoinOS optfile=1
```

The server name (`CoinOS`) is case-insensitive and could equally well have been written as "`MIP=coinos`" or "`MIP=COINOS`". Moreover, the file `eastborne.gms` contains the directive

```
Option MIP = CoinOS;
```

and hence the option `MIP=CoinOS` could have been omitted from the command line.

Since the solver is named `CoinOS`, the options file pointed to by the last part of the command (`optfile=1`) should be named `CoinOS.opt`. In general multiple option files are possible, and the GAMS convention is as follows:

`optfile=1` corresponds to `CoinOS.opt`

`optfile=2` corresponds to `CoinOS.op2`

...

`optfile=99` corresponds to `CoinOS.o99` It is important to distinguish between the option files

for GAMS just mentioned and the option file (in OSoL format) passed to the OS solver server (see below). We now explain the valid options that can go into a GAMS option file when using the CoinOS solver. The options are `service (string)`: Specifies the URL of the COIN-OR solver service. This option is required in order to direct the remote call appropriately.

Use the following value for this option.

```
service http://74.94.100.129:8080/OSServer/services/OSSolverService
```

`writeosil (string)`: If this option is used, GAMS will write the optimization instance to file `(string)` in OSiL format.

`writeosrl (string)`: If this option is used, GAMS will write the result of the optimization to file `(string)` in OSrL format.

The options just described are options for the GAMS modeling language. It is also possible to pass options directly to the COIN-OR solvers by using the `OS` interface. This is done by passing the name of an options file that conforms to the OSoL standard. The option

`readosol (string)` specifies the name of an OS option file in OSoL format that is given to the solver. **Note well:** The file `CoinOS.opt` is an option file for GAMS but the GAMS option `readosol` in the GAMS options file is specifying the name of an OS options file.

The file `solveroptions.osol` is contained in the OS distribution in the `osolFiles` directory in the `data` directory. This file contains four solver options; two for `Cbc`, one for `Ipopt`, and one for `SYMPHONY` (which is available for remote server calls, but not locally). You can have any number of options. Note the format for specifying an option:

```
<solverOption name="maxN" solver="cbc" value="5" />
```

The attribute `name` specifies that the option name is `maxN` which is the maximum number of nodes allowed in the branch-and-bound tree, the `solver` attribute specifies the name of the solver to which the option should be applied, and the `value` attribute specifies the value of the option.

Default solver values are present, depending on the problem characteristics. For more details, consult Table 2 (p.7). In order to control the solver used, it is necessary to specify the name of the solver inside the XML tag `<solverToInvoke>`. The example `solveroptions.osol` file contains the XML tag

```
<solverToInvoke>symphony</solverToInvoke>
```

Valid values for the remote solver service specified in the `<solverToInvoke>` tag are installation dependent; the solver service at `http://74.94.100.129:8080/OSServer/services/OSSolverService` accepts `clp`, `cbc`, `dylp`, `glpk`, `ipopt`, `bonmin`, `couenne`, `symphony`, and `vol`.

By default, the call to the server is a *synchronous* call. The GAMS process will wait for the result and then display the result. This may not be desirable when solving large optimization models. The user may wish to submit a job, turn off his or her computer, and then check at a later date to see if the job is finished. In order to use the remote solver service in this fashion, i.e., *asynchronously*, it is necessary to use the `service_method` option.

`service_method (string)` specifies the method to execute on a server. Valid values for this option are `solve`, `getJobID`, `send`, `knock`, `retrieve`, and `kill`. We explain how to use each of these.

The default value of `service_method` is `solve.` A `solve` invokes the remote service in synchronous mode. When using the `solve` method you can optionally specify a set of solver options in an OSoL file by using the `readosol` option. The remaining values for the `service_method` option are used for an asynchronous call. We illustrate them in the order in which they would most logically be executed.

`service_method getJobID`: When working in asynchronous mode, the server needs to uniquely identify each job. The `getJobID` service method will result in the server returning a unique job ID. For example if the following `CoinOS.opt` file is used

```
service http://74.94.100.129:8080/OSServer/services/OSSolverService
service_method getJobID
```

with the command

```
gams.exe eastborne.gms optfile=1
```

the user will see a rather long job ID returned to the screen as output. Assume that the job id returned is `coinor12345xyz`. This job ID is used to submit a job to the server with the `send` method. Any job ID can be sent to the server as long as it has not been used before.

`service_method send`: When working in asynchronous mode, use the `send` service method to submit a job. When using the `send` service method a job ID is required. An options file must be present and must specify a job ID that has not been used before. Assume that in the file `CoinOS.opt` we specify the options:

```
service http://74.94.100.129:8080/OSServer/services/OSSolverService
service_method send
readosol sendWithJobID.osol
```

The `sendWithJobID.osol` options file is identical to the `solveroptions.osol` options file except that it has an additional XML tag:

```
    <jobID>coinor12345xyz</jobID>
```

We then execute

```
gams.exe eastborne.gms optfile=1
```

If all goes well, the response to the above command should be: "Problem instance successfully sent to OS service". At this point the server will schedule the job and work on it. It is possible to turn off the user computer at this point. At some point the user will want to know if the job is finished. This is accomplished using the `knock` service method.

`service_method knock`: When working in asynchronous mode, this is used to check the status of a job. Consider the following `CoinOS.opt` file:

```
service http://74.94.100.129:8080/OSServer/services/OSSolverService
service_method knock
readosol sendWithJobID.osol
readospl knock.ospl
writeospl knockResult.ospl
```

The knock service method requires two inputs. The first input is the name of an options file, in this case sendWithJobID.osol, specified through the readosol option. In addition, a file in OSpL format is required. You can use the knock.opsl file provided in the binary distribution. This file name is specified using the readospl option. If no job ID is specified in the OSoL file then the status of all jobs on the server will be returned in the file specified by the writeospl option. If a job ID is specified in the OSoL file, then only information on the specified job ID is returned in the file specified by the writeospl option. In this case the file name is knockResult.ospl. We then execute

```
gams.exe eastborne.gms optfile=1
```

The file knockResult.ospl will contain information similar to the following:

```
<job jobID="coinor12345xyz">
    <state>finished</state>
    <serviceURI>http://192.168.0.219:8443/os/OSSolverService.jws</serviceURI>
    <submitTime>2009-11-10T02:13:11.245-06:00</submitTime>
    <startTime>2009-11-10T02:13:11.245-06:00</startTime>
    <endTime>2009-11-10T02:13:12.605-06:00</endTime>
    <duration>1.36</duration>
</job>
```

Note that the job is complete as indicated in the <state> tag. It is now time to actually retrieve the job solution. This is done with the retrieve method.

service_method retrieve: When working in asynchronous mode, this method is used to retrieve the job solution. It is necessary when using retrieve to specify an options file and in that options file specify a job ID. Consider the following CoinOS.opt file:

```
service http://74.94.100.129:8080/OSServer/services/OSSolverService
service_method retrieve
readosol sendWithJobID.osol
writeosrl answer.osrl
```

When we then execute

```
gams.exe eastborne.gms optfile=1
```

the result is written to the file answer.osrl.

Finally there is a kill service method which is used to kill a job that was submitted by mistake or is running too long on the server.

service_method kill: When working in asynchronous mode, this method is used to terminate a job. You should specify an OSoL file containing the job ID by using the readosol option.

5.2.2 GAMS Summary:

1. In order to use OS with GAMS you can either specify CoinOS as an option to GAMS at the command line,

   ```
   gams eastborne.gms MIP=CoinOS
   ```

or you can place the statement `Option ProblemType = CoinOS;` somewhere in the model *before* the `Solve` statement in the GAMS file.

2. If no options are given, then the model will be solved locally using the default solver (see Table 2 on p.7).

3. In order to control behavior (for example, whether a local or remote solver is used) an options file, `CoinOS.opt`, must be used as follows

```
gams.exe  eastborne.gms optfile=1
```

4. The `CoinOS.opt` file is used to specify *eight potential options*:

   - `service (string)`: using the COIN-OR solver server; this is done by giving the option

     ```
     service  http://74.94.100.129:8080/OSServer/services/OSSolverService
     ```

   - `readosol (string)`: whether or not to send the solver an options file; this is done by giving the option

     ```
     readosol  solveroptions.osol
     ```

   - `solver (string)`: if a local solve is being done, a specific solver is specified by the option

     ```
     solver solver_name
     ```

     Valid values are `clp`, `cbc`, `glpk`, `ipopt` and `bonmin`. When the COIN-OR solver service is being used, the only way to specify the solver to use is through the `<solverToInvoke>` tag in an OSoL file. In this case the valid values for the solver are `clp`, `cbc`, `dylp`, `glpk`, `ipopt`, `bonmin`, `couenne`, `symphony` and `vol`.

   - `writeosrl (string)`: the solution result can be put into an OSrL file by specifying the option

     ```
     writeosrl  osrl_file_name
     ```

   - `writeosil (string)`: the optimization instance can be put into an OSiL file by specifying the option

     ```
     writeosil  osil_file_name
     ```

   - `writeospl (string)`: Specifies the name of an OSpL file in which the answer from the `knock` or `kill` method is written, e.g.,

     ```
     writeospl  write_ospl_file_name
     ```

   - `readospl (string)`: Specifies the name of an OSpL file that the `knock` method sends to the server

     ```
     readospl  read_ospl_file_name
     ```

   - `service_method (string)`: Specifies the method to execute on a server. Valid values for this option are `solve`, `getJobID`, `send`, `knock`, `retrieve`, and `kill`.

5. If an OS options file is passed to the GAMS `CoinOS` solver using the GAMS `CoinOS` option `readosol`, then GAMS does not interpret or act on any options in this file. The options in the OS options file are passed directly to either: i) the default local solver, ii) the local solver specified by the GAMS `CoinOS` option `solver`, or iii) to the remote OS solver service if one is specified by the GAMS `CoinOS` option `service`.

6 OS Protocols

The objective of OS is to provide a set of standards for representing optimization instances, results, solver options, and communication between clients and solvers in a distributed environment using Web Services. These standards are specified by W3C XSD schemas. The schemas for the OS project are contained in the `schemas` folder under the `OS` root. There are numerous schemas in this directory that are part of the OS standard. For a full description of all the schemas see Ma [2]. We briefly discuss the standards most relevant to the current version of the OS project.

6.1 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.

OSiL stores optimization problem instances as XML files. Consider the following problem instance, which is a modification of an example of Rosenbrock [3]:

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

$$\text{s.t.} \quad x_0 + 10.5x_0^2 + 11.7x_1^2 + 3x_0x_1 \leq 25 \tag{2}$$

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

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

There are two continuous variables, x_0 and x_1, in this instance, each with a lower bound of 0. Figure 5 shows how we 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 (continuous, binary, general integer).

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

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

Figure 5: The `<variables>` element for the example (1)–(4).

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 6 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 Figure 5. In particular, Figure 6 defines the complexType named `Variables`, which comprises an element named `<var>` and an attribute named `numberOfVariables`. The `numberOfVariables` attribute is of a standard type `positiveInteger`, whereas the `<var>` element is a user-defined complexType named `Variable`. Thus the complexType `Variables` contains a sequence of `<var>` elements that are of complexType `Variable`. OSiL's schema must also provide a specification for the `Variable` complexType, which is shown in Figure 7.

```
<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 6: 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 7: The `Variable` complexType in the OSiL schema.

In OSiL the linear part of the problem is stored in the `<linearConstraintCoefficients>` element, which stores the coefficient matrix using three arrays as proposed in the earlier LPFML schema [1]. There is a child element of `<linearConstraintCoefficients>` to represent each array: `<value>` for an array of nonzero coefficients, `<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. This is shown in Figure 8.

```
<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>
```

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

The quadratic part of the problem is represented in Figure 9.

```
<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>
```

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

The nonlinear part of the problem is given in Figure 10.
The complete OSiL representation can be found in the Appendix (Section 7.1).

6.2 OSnL (Optimization Services nonlinear Language)

The OSnL schema is imported by the OSiL schema and is used to represent the nonlinear part of an optimization instance. This is explained in greater detail in the OS User's Manual. Also refer to Figure 10 for an illustration of elements from the OSnL standard. This figure represents the nonlinear part of the objective in equation (1), that is,

$$(1 - x_0)^2 + 100(x_1 - x_0^2)^2.$$

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

Figure 10: The <nl> element for the nonlinear part of the objective (1).

6.3 OSrL (Optimization Services result Language)

OSrL is an XML-based language for representing the solution of large-scale optimization problems including linear programs, mixed-integer programs, quadratic programs, and very general nonlinear programs. An example solution (for the problem given in (1)–(4)) in OSrL format is given below.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type = "text/xsl"
  href = "/Users/kmartin/Documents/files/code/cpp/OScpp/COIN-OSX/OS/stylesheets/OSrL.xslt"?>
<osrl xmlns="os.optimizationservices.org"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="os.optimizationservices.org
      http://www.optimizationservices.org/schemas/2.0/OSiL.xsd">
    <general>
        <generalStatus type="normal"/>
        <serviceName>Solved using a LINDO service</serviceName>
        <instanceName>Modified Rosenbrock</instanceName>
    </general>
    <optimization numberOfSolutions="1" numberOfVariables="2" numberOfConstraints="2"
        numberOfObjectives="1">
        <solution targetObjectiveIdx="-1">
            <status type="optimal"/>
            <variables>
                <values numberOfVar="2">
                    <var idx="0">0.87243</var>
```

```
                <var idx="1">0.741417</var>
            </values>
            <other numberOfVar="2" name="reduced costs" description="the variable reduced costs">
                <var idx="0">-4.06909e-08</var>
                <var idx="1">0</var>
            </other>
        </variables>
        <objectives>
            <values numberOfObj="1">
                <obj idx="-1">6.7279</obj>
            </values>
        </objectives>
        <constraints>
            <dualValues numberOfCon="2">
                <con idx="0">0</con>
                <con idx="1">0.766294</con>
            </dualValues>
        </constraints>
    </solution>
</optimization>
```

6.4 OSoL (Optimization Services option Language)

OSoL is an XML-based language for representing options that get passed to an optimization solver or a hosted optimization solver Web service. It contains both standard options for generic services and extendable option tags for solver-specific directives. Several examples of files in OSoL format are presented in Section 4.4.

6.5 OSpL (Optimization Services process Language)

This is a standard used to enquire about dynamic process information that is kept by the Optimization Services registry. The string passed to the `knock` method is in the OSpL format. See the example given in Section 4.4.5.

7 Appendix – Sample OSiL files

7.1 OSiL representation for problem given in (1)–(4) (p.30)

```
<?xml version="1.0" encoding="UTF-8"?>
<osil xmlns="os.optimizationservices.org"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="os.optimizationservices.org
      http://www.optimizationservices.org/schemas/2.0/OSiL.xsd">
    <instanceHeader>
        <name>Modified Rosenbrock</name>
        <source>Computing Journal 3:175-184, 1960</source>
        <description>Rosenbrock problem with constraints</description>
    </instanceHeader>
```

```
&lt;instanceData&gt;
    &lt;variables numberOfVariables="2"&gt;
        &lt;var lb="0" name="x0" type="C"/&gt;
        &lt;var lb="0" name="x1" type="C"/&gt;
    &lt;/variables&gt;
    &lt;objectives numberOfObjectives="1"&gt;
        &lt;obj maxOrMin="min" name="minCost" numberOfObjCoef="1"&gt;
            &lt;coef idx="1"&gt;9.0&lt;/coef&gt;
        &lt;/obj&gt;
    &lt;/objectives&gt;
    &lt;constraints numberOfConstraints="2"&gt;
        &lt;con ub="25.0"/&gt;
        &lt;con lb="10.0"/&gt;
    &lt;/constraints&gt;
    &lt;linearConstraintCoefficients numberOfValues="3"&gt;
        &lt;start&gt;
            &lt;el&gt;0&lt;/el&gt;&lt;el&gt;2&lt;/el&gt;&lt;el&gt;3&lt;/el&gt;
        &lt;/start&gt;
        &lt;rowIdx&gt;
            &lt;el&gt;0&lt;/el&gt;&lt;el&gt;1&lt;/el&gt;&lt;el&gt;1&lt;/el&gt;
        &lt;/rowIdx&gt;
        &lt;value&gt;
            &lt;el&gt;1.&lt;/el&gt;&lt;el&gt;7.5&lt;/el&gt;&lt;el&gt;5.25&lt;/el&gt;
        &lt;/value&gt;
    &lt;/linearConstraintCoefficients&gt;
    &lt;quadraticCoefficients numberOfQuadraticTerms="3"&gt;
        &lt;qTerm idx="0" idxOne="0" idxTwo="0" coef="10.5"/&gt;
        &lt;qTerm idx="0" idxOne="1" idxTwo="1" coef="11.7"/&gt;
        &lt;qTerm idx="0" idxOne="0" idxTwo="1" coef="3."/&gt;
    &lt;/quadraticCoefficients&gt;
    &lt;nonlinearExpressions numberOfNonlinearExpressions="2"&gt;
        &lt;nl idx="-1"&gt;
            &lt;plus&gt;
                &lt;power&gt;
                    &lt;minus&gt;
                        &lt;number type="real" value="1.0"/&gt;
                        &lt;variable coef="1.0" idx="0"/&gt;
                    &lt;/minus&gt;
                    &lt;number type="real" value="2.0"/&gt;
                &lt;/power&gt;
                &lt;times&gt;
```

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

References

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

[2] J. Ma. Optimization Services (OS), A General Framework for Optimization Modeling Systems, 2005. Ph.D. Dissertation, Department of Industrial Engineering & Management Sciences, Northwestern University, Evanston, IL.

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

Index