Instance Formats

Horand Gassmann*, Jun Ma†, and Kipp Martin‡

Version 1: August 11, 2009

Abstract

In order to use an optimization solver, it is necessary to communicate an optimization instance to the solver. Additionally, solvers typically take options to specify tolerances, set algorithmic parameters, etc. Finally, the solver must communicate the result instance back to the user. In this article we review existing formats for optimization instances, solver option instances, and result instances and describe the research issues involved in designing these instance formats.

*School of Business Administration, Dalhousie University, Halifax, Canada, e-mail: Horand.Gassmann@dal.ca

†Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, Illinois, e-mail: majxuh@hotmail.com

‡Booth School of Business, The University of Chicago, Chicago, Illinois, e-mail: kmartin@chicagobooth.edu (corresponding author)

In order to use an optimization solver, it is necessary to communicate a model instance to the solver. Early users of optimization solvers often created a specific model instance from input data by writing a *matrix generator*. Mixed-integer linear programming solvers such as IBM's MPSX/370 package and LINDO provided an application program interface that allowed a user to write a matrix generator in a procedural programming language such as PL/I or Fortran. The matrix generator was then linked to the solver library to produce an executable program that solved the specific instance. The term matrix generator arose because, initially, the most widely used optimization solvers were for mixed-integer linear programming, and specifying the constraint matrix and objective function vector was sufficient to describe a model instance (along with a characterization of variable types and bounds and constraint right-hand-sides).

Matrix generators were tightly coupled with the solver because the model instance was communicated directly in-memory to the solver. However, modelers still needed to store the model instance in a persistent state for archival, communication, and debugging purposes. The classic MPS format developed at IBM was the first widely adopted instance format. Even today, most linear or MIP solvers read and write files in MPS format. See [1].

The development of *algebraic modeling languages* represented a major breakthrough for the field of Operations Research because they greatly reduced the effort required to produce an optimization model instance. Modeling languages such as GAMS [2] and AMPL [3], that communicate with solvers by writing an instance to a file, have resulted in additional instance formats. Indeed, both GAMS and AMPL have their own instance representation formats. There now exits a plethora of proposed instance formats, depending on the optimization type. In Table 1 we list some key optimization types and corresponding formats. This table is illustrative, but by no means complete.

Table 1: Optimization Classes and Proposed Formats

Optimization Type	**Instance Format**
Mixed-Integer Linear Program	MPS, lpsolve-lp, CPLEX-lp, AMPL nl, OSiL, OptML, GAMS dat
Nonlinear	xMPS, MOI, SIF, OSiL, AMPL nl, GAMS dat, LINDO MPI
Stochastic Linear Programming	SMPS, OSiL
Semidefinite and Second Order Cone	sparse SDPA, SDPLR, OSiL
Linear Network Optimization	NETGEN, NETFLO, DIMACS, RELAX4
Traveling Salesman Problem	tsp

It is crucial to understand the difference between an algebraic model and a specific instance of that model. The description of a specific optimization instance is very different from the description of a generic algebraic model. An algebraic model description is *symbolic, general, concise,* and *understandable*. See [4]. An algebraic model contains declarations of high-level algebraic components such as sets, parameters, variables, objectives, and constraints. By contrast, a model instance is *explicit, specific, verbose,* and *convenient* for the solver rather than for the human modeler. An algebraic modeling language takes a model and compiles it into an instance using a specific data set. A

linear programming model instance is illustrated in Figure 1. (The details of this representation are discussed in Section 2.) An appropriate analogy from object-oriented programming is that the algebraic model corresponds to a class and the model instance corresponds to an *instantiated* object in that class.

In this article we review existing instance formats for optimization problems, solver options, and solver results. We describe the research issues involved in designing instance formats. We do not address algebraic models. See the companion article [5] in this Volume.

In the next section we discuss key issues that must be considered when designing an instance format. In Section 2 we describe formats for representing optimization instances. In addition to optimization instances, it is necessary to have formats for solver options (Section 3) and results (Section 4). There are obviously numerous types of optimization models. Extensions to some of the more recently studied optimization areas are the topic of Section 5.

1 Key Design Issues

There are several key design issues when designing an instance representation. First is the decision of which instance to represent. Is the proposed format only for the model instance, or does it include instances of solver options, solver results, or instance modifications? This is discussed more thoroughly in Section 1.1. Then there is the key issue of how to format or represent optimization instance information. This is discussed in Section 1.2. The next issue is how comprehensive and extensible should the representation be. This is discussed in Section 1.3. Finally there is the issue of how the instance representations map to in-memory objects on the solver side. This is discussed in Section 1.4.

1.1 Key Issue 1: Separation of Functionality

In a modeling language-solver infrastructure an instance passed to the solver must contain information on the problem to be optimized and must specify solver options. Likewise a result is returned from the solver. There are actually four distinct instance formats that can be specified.

1. Optimization *problem* instance format – this is discussed briefly in Section 1.2 and in more detail in Section 2.
2. Solver *option* instance format – it is often necessary to pass options to a solver. This is a challenge given the wide variety of solvers and a complete lack of a standard for option formats. See Section 3 for a proposed standard.
3. Solver *result* instance format – after a problem is optimized (or terminated) results must be passed back from the solver to the client. See Section 4 for more discussion on result instance formats.
4. Instance *modification* format – there are many solution procedures such as column generation and cutting plane algorithms that require modifying the original problem instance. To date there is no standard instance modification format that the authors are aware of. This is a good topic for further research.

1.2 Key Issue 2: Optimization Instance Format

A typical optimization instance might include millions of nonzero coefficients in linear terms and hundreds, if not thousands, of nonlinear terms. A central problem in representing instances is managing these entries and ensuring that they are handled correctly and efficiently. There are three basic styles or formats that have been used to represent a model instance.

1.2.1 Natural Algebraic Format

Consider the simple constraint $.5x_0 + .8333x_1 \leq 600$. One way to represent this constraint instance is in its natural algebraic format. For example, in the LINDO solver, this constraint is expressed as

```
R0000001)   +0.5 x0  + 0.8333 x1  < +600.
```

Other solvers have similar algebraic formats, for example CPLEX LP file format, lpsolve lp-format, and Xpress lp format. The primary advantage of this format is that it is easily read by a human and convenient for debugging purposes. This format is also verbose and will not be parsed as quickly as more compact schemes.

1.2.2 Packed

Modeling languages take an algebraic model and compile that with the data to produce a model instance. In the case of modeling languages such as AMPL and GAMS, the model instance is temporarily written to a file which is then read by the solver. In order to make this process efficient, modeling languages use a packed instance representation. For example, the body of the constraint $.5x_0 + .8333x_1 \leq 600$ has the following representation in the AMPL nl format [6].

```
J1 2
0 0.5
1 0.8333
```

For more on how to interpret these lines and on representing the right-hand side see figure 2.

1.2.3 Markup Language

Yet another option for model instance representation is to use a markup language such as Extensible Markup Language (XML). Two proposed XML based instance formats are OSiL [7] and OptML [8]. In OSiL XML markup the constraint matrix is represented in packed matrix format [9], either by row or by column. In row representation the constraint $.5x_0 + .8333x_1 \leq 600$ might appear as follows.

The `<start>` element contains the children

```
...<el>2</el><el>4</el>...
```

The `<colIdx>` element contains the children

```
...<el>0</el><el>1</el>...
```

The `<values>` element contains the children

```
...<el>.5</el><el>0.8333</el>...
```

Although verbose, there are good reasons to use XML for an instance format. (See [10] for a discussion.)

1.3 Key Issue 3: Design of Format: Level of Detail and Flexibility

An optimization instance is a well-defined entity. A certain amount of detail is required and it is not possible to be flexible in deciding whether or not problem components should be present. For example, 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. Similar logic applies to solver options. Thus, when representing an optimization instance, it is necessary to express the instance in a very rigorous fashion. However, when designing option or result instances, flexibility may be more desirable. We elaborate on this issue more in Sections 3 and 4.

1.4 Key Issue 4: Design of an In-Memory Instance Object

In various situations, e.g. when using a modeling language such as GAMS or AMPL, the optimization instance may be serialized (e.g. written to a file) and then deserialized (e.g read back into memory) by the solver. The way that an optimization instance is represented in memory may look very different from the format that persists in a string or file. There is no "standard way" to represent, for example, an instance in the MPS format as an in-memory object. The Optimization Services framework is the first to specify a set of mapping or binding rules on how to transform the persistent instance format into in-memory objects. The OS format (OSiL) for representing an instance is based on a W3C XML Schema. The use of a schema to specify the XML instance format is key to providing a binding between the format and the the `OSInstance` class which is the in-memory representation of the optimization instance. For example, each XML schema complexType corresponds to a class in `OSInstance.` For more details on this binding see [7].

2 Optimization Instance Format

Instance formats are used to store problem instances in a file or string before sending them to the solver. These can be created by hand, or be output from a matrix generator or an algebraic modeling language. Formats can be either text or binary.

An instance format must be able to represent all components of a problem instance. Special purpose formats for narrow problem classes such as network problems might have more restricted requirements than formats for general ones, but some problem components are universal to all problem classes, such as decision variables, objectives, and constraints.

Information about the number of variables and constraints must be contained in the instance format, but there are at least two ways to accomplish this goal. For

example, the AMPL nl format [6] has a header section that contains explicit information about the number of continuous, integer, and binary variables and breaks this down further into variables that appear in nonlinear expressions in the objective, constraints, both, or neither. The MPS format [21], on the other hand, contains the information implicitly and requires the reader to keep count of the variables as the instance is being processed. Similarly for constraints. Many instance formats (one notable exception being the MPS format) assume that the problem has a unique objective, so that there is no need to record the number of objectives. Obviously, formats designed to handle multi-objective problems must be able to express the number of objectives found in the problem instance.

Along with the number of decision variables, it is necessary to store information about them, such as upper and lower bounds. Finally, for report generation purposes and communication of the optimization result to the user, it is desirable that each variable have a unique identifier, which may or may not be communicated to the solver. For that reason some instance formats refer to variables as well as constraints by number, while others, including the MPS format, refer to them by name. The latter allows more flexibility in the ordering of certain problem components but is more costly to parse.

The biggest challenge for a problem instance format is handling nonzero elements in the constraints. In problems of realistic size, it is essential to exploit sparsity in the linear, quadratic, and nonlinear components of the problem. Sparse representations of linear constraint matrices can be achieved in several different ways. MPS records the matrix coefficients as triples of (*row*, *column*, *value*), where *row* and *column* are names instead of numbers. AMPL nl format stores the matrix column by column with markers separating one column from another and with a row number and coefficient value for every nonzero value in the column. OSiL uses a combination of three arrays, giving the column starts, row indices and coefficient values. Row-wise storage schemes are also possible.

As an illustration of the above ideas, we offer the following small linear programming problem.

$$\begin{array}{lll} \max & 10x_0 + 9x_1 & \\ \text{s.t.} & 0.7x_0 + 1.0x_1 & \leq 630 \\ & 0.5x_0 + 0.8333x_1 & \leq 600 \\ & 1.0x_0 + 0.6667x_1 & \leq 708 \\ & 0.1x_0 + 0.25x_1 & \leq 135 \end{array} \tag{1}$$

Figure 1 is a representation of instance (1) in MPS format. Lines 3–7 set up the objective and constraint rows, the number being determined implicitly. Lines 9–11 give the objective and constraint coefficients associated with the first column, x_0; lines 12–14 give the coefficients for the second column, x_1. Lines 16–17 specify the right-hand side values associated with the four constraints.

The AMPL nl format can be stored in either binary or text format; we give the text form for instance (1) in Figure 2. The annotated section of the problem (lines 1–10) is used to set up the problem dimension; the next ten lines are used to show any nonlinear expressions (of which there are none, as indicated by the special code `n0` in lines 12, 14, 16, 18 and 20) in the constraints and the objective. The right-hand sides are set up in lines 21–25. (The number 1 preceding the right-hand side values mark them as upper bounds.) Lines 26–28 give lower bounds on the decision variables. The constraint

```
 1   NAME  ParInc. From Anderson, Sweeney, and Williams
 2   ROWS
 3    N       OBJ
 4    L  R0000000
 5    L  R0000001
 6    L  R0000002
 7    L  R0000003
 8   COLUMNS
 9       x0             OBJ           -10
10       x0        R0000000           0.7   R0000001           0.5
11       x0        R0000002             1   R0000003           0.1
12       x1             OBJ            -9
13       x1        R0000000             1   R0000001        0.8333
14       x1        R0000002        0.6667   R0000003          0.25
15   RHS
16       RHS1      R0000000           630
17       RHS1      R0000001           600
18       RHS1      R0000002           708
19       RHS1      R0000003           135
20   BOUNDS
21   ENDATA
```

Figure 1: A sample MPS file

matrix is given in row format starting in line 31. Line 31 specifies that the first row contains two entries, and the following two lines contain index-value pairs. This pattern repeats for the remaining three constraints. A similar format is used in lines 43–45 to specify the objective row.

Finally we give the OSiL format for the same problem (see figure 3). OSiL is an extremely general problem instance format for representing linear, integer, quadratic and nonlinear problem instances as well as a large number of special features, such as multi-objectives, stochastic programs, cone programs, special ordered sets, and others. The OSiL format is largely self-documenting, so we only highlight a few main points. Header information is given in lines 3 to 5. The actual optimization data start in line 6. Separate sections give information about variables (line 7–10: bounds, variable type and an optional name), objective function (direction of optimization and objective coefficients) and constraints (right-hand side values). The start values in line 26 give the column starts in the linear constraint matrix, which allow the computation of the number of nonzeroes in each column, and the indices and values appear in lines 29–32, and 35–38, respectively.

Due to the availability of fast linear programming codes, linear models were initially of primary interest. However, as progress is made with nonlinear solution algorithms, there is increased interest in formats for nonlinear instances. Quadratic programs are the most natural extension of linear programs. A quadratic term is easily stored as a 4-tuple: *(row_index*, *variable_1_index*, *variable_2_index*, *coefficient value)*. The QPS extension to the MPS format allows quadratic terms only in the objective and achieves

```
 1   g3 0 1 0   # problem parinc
 2   2 4 1 0 0  # vars, constraints, objectives, ranges, eqns
 3    0 0       # nonlinear constraints, objectives
 4    0 0       # network constraints: nonlinear, linear
 5    0 0 0     # nonlinear vars in constraints, objectives, both
 6    0 0 0 1   # linear network variables; functions; arith, flags
 7    0 0 0 0 0 # discrete variables: binary, integer, nonlinear (b,c,o)
 8    8 2       # nonzeros in Jacobian, gradients
 9    0 0       # max name lengths: constraints, variables
10    0 0 0 0 0 # common exprs: b,c,o,c1,o1
11   C0
12   n0
13   C1
14   n0
15   C2
16   n0
17   C3
18   n0
19   O0 1
20   n0
21   r
22   1 630
23   1 600
24   1 708
25   1 135
26   b
27   2 0
28   2 0
29   k1
30   4
31   J0 2
32   0 0.7
33   1 1
34   J1 2
35   0 0.5
36   1 0.8333
37   J2 2
38   0 1
39   1 0.6667
40   J3 2
41   0 0.1
42   1 0.25
43   G0 2
44   0 10
45   1 9
```

Figure 2: A sample nl file

```
<?xml version="1.0" encoding="UTF-8"?>
<osil xmlns="os.optimizationservices.org">
    <instanceHeader>
            <name>Par Inc. </name>
    </instanceHeader>
    <instanceData>
            <variables numberOfVariables="2">
                    <var name="x0" lb="0" type="C" />
                    <var name="x1" lb="0" type="C" />
            </variables>
            <objectives numberOfObjectives="1">
                    <obj name = "Par, Inc. Objective Function"
                           maxOrMin="max"  numberOfObjCoef="2">
                            <coef idx="0">10</coef>
                            <coef idx="1">9</coef>
                    </obj>
            </objectives>
            <constraints numberOfConstraints="4">
                    <con name="cutanddye" ub="630" />
                    <con name="sewing"    ub="600"/>
                    <con name="finishing" ub="708"/>
                    <con  ub="135"/>
            </constraints>
            <linearConstraintCoefficients numberOfValues="8">
                    <start>
                            <el>0</el>     <el>4</el>      <el>8</el>
                    </start>
                    <rowIdx>
                            <el>0</el>     <el>1</el>
                            <el>2</el>     <el>3</el>
                            <el>0</el>     <el>1</el>
                            <el>2</el>     <el>3</el>
                    </rowIdx>
                    <value>
                            <el>0.7</el>   <el>.5</el>
                            <el>1.</el>    <el>.1</el>
                            <el>1.0</el>   <el>0.8333</el>
                            <el>0.6667</el><el>0.25</el>
                    </value>
            </linearConstraintCoefficients>
    </instanceData>
</osil>
```

Figure 3: A sample OSiL file

this by adding a QSECTION where each record lists both variables in the quadratic term and the coefficient.

There are several ways in which general nonlinear expressions can be captured in an instance format. We illustrate with a modification of the Rosenbrock function [11]. The model is:

$$\begin{array}{lll} \min & (1-x_0)^2 + 100 * (x_1 - x_0^2)^2 + 9 * x_1 & \\ \text{s.t.} & x_0 + 10.5 * x_0^2 + 11.7 * x_1^2 + 3 * x_0 * x_1 & \leq 25 \\ & \ln(2 * x_0 * x_1) + 7.5 * x_0 + 5.25 * x_1 & \geq 10 \end{array} \tag{2}$$

In instance (2) the terms $100 * (x_1 - x_0^2)^2$ and $\ln(2 * x_0 * x_1)$ are not quadratic and must use a more general representation. We illustrate approaches to modeling general nonlinear terms using the $\ln(2{*}x_0{*}x_1)$ term. The standard algebraic format $\ln(2{*}x_0{*}x_1)$ is called *infix* and is ambiguous with regard to the order of operations. The ambiguity is resolved by using parentheses (or by convention with regard to operator precedence and left or right associativity), where for example, $\ln((2 * x_0) * x_1)$ specifies an order of operations. Due to the infix ambiguity, most instance formats store each nonlinear term as a *prefix* or *postfix* sequence of operands and operators. The list of operators and operands is also called an *instruction list.* The prefix and postfix sequence for $\ln((2 * x_0) * x_1)$ is

```
ln * * 2 x_0 x_1
```

and

```
x1 2 x0 * * ln,
```

respectively. A prefix or postfix instruction list is trivial to evaluate using a stack data structure and is also convenient for algorithmic or automatic differentiation. The postfix or prefix instruction list is a key part of several instance formats. For example, the AMPL nl format uses prefix combined with numerical codes for the operators `ln` and `*` and writes

```
o43
o2
o2
n2
v0
v1
```

where `o43` is the code for the logarithm operator, `o2` is the code for the multiplication operator and `n` and `v` denote references to numbers and variables, respectively.

The OSiL instance format also corresponds to a prefix instruction list. The term $\ln((2 * x_0) * x_1)$ in OSiL is:

```
<nl idx="1">
    <ln>
        <times>
            <times>
                <number type="real" value="2.0"/>
                <variable coef="1.0" idx="0"/>
            </times>
```

```
            <variable coef="1.0" idx="1"/>
        </times>
    </ln>
</nl>
```

The sequence of open tags is

```
<ln><times><times><number><number><variable><variable>
```

and corresponds directly to the prefix instruction list.

A nonlinear extension of MPS based on postfix is described by Halldórsson et al. [12]. It puts the postfix instruction list into the MPS record structure. For example, $\ln((2 * x_0) * x_1)$ is written as:

```
NONLINEAR
    CON1      V1        mult          2         X0
    CON1      V2        mult          V1        X1
    CON1      RES       ln            V2
```

Lines 2 and 3 compute the product $2x_0x_1$ and store it into the intermediate object `V2`; the last line computes the natural logarithm of `V2` and stores into the reserved object `RES`, which is used to designate the final result of the computation, which gets added to the linear and quadratic terms for constraint `CON1`.

A completely different format, SIF (standard input format), was developed by Conn et al. [13, 14]. A SIF file consists of two pieces, a declarative portion in which a number of declarations of auxiliary quantities are interspersed with declarations of variables, objectives and constraints, and a procedural portion where the nonlinear functions are defined in terms of the original variables and coefficients as well as the auxiliary quantities previously defined. SIF [14] is a very complex and far-reaching proposal that did not achieve universal acceptance.

Formats for specialized problems can occasionally streamline the information needs and reduce, often greatly, the size of the resulting instance file. For instance, a stochastic programming problem may need to record values for problem coefficients that take different values in a number of different scenarios — along with a number of deterministic coefficients that do not change from one scenario to the next. If the random variables are time-staged, this can lead to a geometric explosion of the event tree. The SMPS format for stochastic programs was designed to use redundancy and only store the stochastic components explicitly. When the random variables are independent from one stage to the next, the SMPS format can store the event tree in linear space. See [15] for more detail. Similar features are available in OSiL. See [16]. Compact specialized instance formats exist for other problem classes, such as semi-definite cone programming, network problems and traveling salesman problems.

3 Solver Option Instance Format

Solver options are an integral part of the solution process and are usually tied to the problem instance. Traditionally there have not been any standards for the communication of options, since each solver tended to have its own API with which the user communicates either through files, command line options or interactive keyboard inputs.

Standardizing the input format for solver options is hard since the options tend to be very specific to the solver and algorithm. In designing a common format it is useful to break down the process into syntax (how to represent options) and semantics (how to interpret their meanings). While the former is relatively easy to standardize, the latter is nearly impossible, even though a number of common options may be identified that are implemented in the majority of solvers. Standard option formats therefore need to be both rigid in format and flexible in content, and must be extensible to leave room for future solver development.

The development of option formats is driven by other considerations as well. Since solver options form only part of the input sent to a solver, it is desirable to have some way to ensure that the options are linked to the correct problem instance. This is necessary particularly where the options provide values for the elements of an instance component, such as initial variable values. Some solvers such as LINDO allow solver options to have different scopes, applying either to one particular problem only, or to the entire solver environment. This is useful, because it allows the user to set an output level once for a series of problems instead of having to repeat the information for each separate problem.

Solvers are not the only component of a mathematical programming system; options could be sent to an optimization analyzer, simulation tool, or similar. If a large environment has such diverse tools, it seems advantageous to design option formats that can be shared among all components. In order to establish the correct match of option file and analysis tool, the intended target must be recorded in the option file. Additionally, especially in a commercial environment, basic security features such as license information, username and password may be needed in the options file. Finally, especially in a distributed environment, additional facilities are often needed to monitor the progress of the solution process on the remote computer, to verify capabilities of the computer system on which the solver is to be run, and perhaps to perform ancillary file operations before and after the solution.

In order to describe the design of an options format, we distinguish several classes of solver options. First, options may apply to the problem instance. For example, the MPS input format does not allow specifying the direction of optimization (min or max). Solver options (so called *agenda cards*) [21] can be used to communicate this information.

A second category of options controls the flow of the solution algorithm. In this category there are options to select a particular algorithm class (e.g., primal simplex, dual simplex or interior point method in a linear program) or algorithm variant (for instance a particular pricing scheme). Options such as whether to scale a coefficient matrix or not, what crashing and preprocessing procedures to use, as well as limits on the number of iterations or the solution time, would also fall into this category. For nonlinear problems one can also specify an initial point.

Other solver options include various tolerances, the amount and type of output generated (including the level of diagnostic messages when parsing the input), the location and name of input and output files, whether the solution needs to be saved for subsequent runs, perhaps with modified data, etc. It does not stop there. Especially in distributed computing it may be necessary to control the environment in which the solution is obtained. For example, it may be known that the solution of an instance places space requirements on a server, which may limit the number of servers that could potentially be used.

The Optimization Services option Language (OSoL) schema is to our knowledge the first attempt at standardizing the specification of solver and process options. See [19] and [20]. We end this section with a small file in the OSoL format. This file should be self-explanatory. It first specifies that the job is to be run only on computers with at least 1Gb of memory, provides initial values for two decision variables, and gives four solver options. Two of these options are intended for the Ipopt solver [17]; the other two are for the LINDO solver [18]. This demonstrates that an option file can be shared between different solvers. The two Ipopt options control the amount of output to be generated and where the output is to be directed; the LINDO options illustrate that the same option can be used with different scope, applying to a single model in the first case and to the entire session in the second case.

```
<?xml version="1.0" encoding="UTF-8"?>
<osol>
    <system>
        <minMemorySize unit="gigabyte">1.0</minMemorySize>
    </system>
    <optimization>
        <variables numberOfOtherVariableOptions="2">
            <initialVariableValues numberOfVar="2">
                <var idx="0" value="1"/>
                <var value="4.742999643577776e-2" idx="1"/>
            </initialVariableValues>
        </variables>
        <solverOptions numberOfSolverOptions="4">
            <solverOption name="print_level" solver="ipopt"
                type="integer" value="5"/>
            <solverOption name="output_file" solver="ipopt"
                type="string" value="ipopt.out"/>
            <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"/>
        </solverOptions>
    </optimization>
</osol>
```

4 Solver Result Instance Format

It is necessary for an optimization solver to communicate the solution result back to the user. Most solvers have their own result format or API for communicating the solution result. Although not nearly as widespread as the MPS instance format, there is also an MPS format for reporting solver results. See Murtagh [21]. The MPS result format is table-based and separated into three sections: `header`, `rows` and `columns`. The `header` section lists such information as problem instance name, objective value, status and number of iterations. The `rows` section provides results on the constraints such as constraint values, slacks and duals. The `columns` section provides results on

the variables such as solution values and reduced costs. The two major problems with the MPS result instance format are that it is `optimization` centric and is not easily extensible.

An alternative result instance format is OSrL (Optimization Services result Language). See [19] and [20]. OSrL supports multiple solutions with multi-objectives. Each solution contains a separate set of variable, objective, constraint and other results. For results that are uncommon, or solver specific, or subject to different interpretations, OSrL provides the mechanism of `otherSolutionResult` and `otherSolverOutput` that can be either scalar values or vector-based result values optionally indexed over each objective, variable, or constraint. We show below a small segment of OSrL from an actual application that illustrates "non-standard" solver output that is important to communicate.

```
<otherSolutionResult name="ReqDocPairings" numberOfItems="55">
     <item>778,Liberty,Surgical,Pod1,1,M,AM,1</item>
     <item>778,Liberty,Surgical,Pod2,1,T,AM,1</item>
     <item>551,Liberty,Surgical,Pod3,1,M,AM,1</item>
                  . . .
</otherSolutionResult>
```

In addition to the `optimization` category, which communicates output directly related to the optimization, the OSrL format contains four further categories: `general`, `system`, `service`, and `job`. The `general` result section contains such information as general status and message, service name and uri, instance name, job ID, solver invoked and time stamp. The `system` section contains information about available disk space, memory and CPU. The `service` section contains information about the host optimization service, such as its current state (busy or idle) and service utilization. The `job` section contains information about the particular optimization job process like its submit/scheduled/start/end time, used CPU, disk and memory usage. Each of these sections provides the "`other`" mechanism for custom results.

5 Extensions

If an optimization instance format is well designed, it should be flexible enough to extend the standard *"core"* optimization types such as continuous and mixed integer linear, quadratic, and general nonlinear programs. In this section we list potential extensions to the core instance format. It is critical that the core instance format is not affected by the extensions and that whenever a new optimization type is added, the core format remains unchanged, keeping the standards backward compatible.

1. Logic and Constraint Programs. If the instance format has a built-in nonlinear expression tree, it should be quite natural for it to be extended to this type of programs, as logic and relational operators are expressed in the same way as all regular nonlinear operators. For example `if` is simply an operator that takes three operands. Constraint programs uses other special operators such as `and`, `or`, `xor`, `not`, `if`, `implies`, `eq`, `neq`, `geq`, `gt`, `leq`, and `lt`.
2. Complementarity Problems. Similar to extending the logic and constraint programs, we can add a `complements` operator to express such problems. The

`complements` operator takes two operands, which must both be inequality operators. `complements` evaluates to true if both operands are true and at least one inequality is tight.

3. Special Ordered Sets (SOS). There are several types of SOS [22, 23, 24, 25], but each type is essentially expressed as a set of indices of variables already defined in the core programs, plus optional information such as a convexity row.
4. Cone Programs. There are four widely used cone types including nonnegative cone, quadratic cone, rotated quadratic cone, and semidefinite cone.
5. Disjunctive Programs. Among all the possible extensions, the disjunctive program extension is probably the one that benefits the most from building on the core, as almost all the data have already been represented in the core, and each disjunction usually just alters a tiny piece of information from the core, e.g. change of a matrix coefficient or constraint bound. Each disjunction is essentially trying to construct a separate optimization instance with the least amount of extra data.
6. Robust Optimization and Stochastic Optimization. These are two ways for dealing with uncertainty in the problem parameters. Robust optimization can be thought of as a special case of stochastic programming, and it is a design choice whether to define it as a separate extension or just subsume it under the more general heading of stochastic programming.
7. Instance Modification. Popular algorithms such as branch-cut-and-price often require adding or deleting constraints and variables. Currently there is no instance format for specifying model changes.
8. Real-time Data. In a tightly coupled modeling environment, a modeler creates a problem instance using a modeling language. The instance file is self contained and has all of the parameters necessary for optimization by a solver. However, as data are updated in real time it is desirable from an efficiency standpoint to update only the data that have changed and not regenerate the entire instance.
9. User-defined functions. No matter how many nonlinear operators and functions an instance format supports, it cannot exhaust all of them and should support custom defined functions, to be used in the nonlinear expression tree along with other standard functions.
10. Optimization via Simulation. Not all functions (constraints or objectives) have explicit forms and if the function is a black box simulation that cannot be easily modeled by any user function, one needs the support of optimization via simulation.
11. Network and Graph Problems. Network and graph problems traditionally are built around the concepts of *vertices* and *arcs*, instead of *variables* and *constraints*. Special formats may be needed to deal with them effectively.
12. Instance Communication Over a Network. Given the trend toward cloud computing and computing on demand there will almost certainly be a need to communicate instance formats over a network. The optimization instance, solver options and solver results, along with header information, must all be packed in the body of a larger "instance" that is communicated over a network. For a discussion of higher level network instance formats for optimization see [19].

13. Optimization over Vector Spaces. Discrete time optimal control problems are often merely an approximation of an underlying continuous time problem. The discretization step is arbitrary and should occasionally be allowed to be refined by the solution algorithm. It would therefore be useful to have a way to express the underlying continuous time problem directly.

References

[1] http://wiki.mcs.anl.gov/NEOS/index.php/Linear_Programming_FAQ\#What_is_MPS_format.3F http://wiki.mcs.anl.gov/NEOS/index.php/Linear_Programming_FAQ#What_is_MPS_format.3F

[2] http://www.gams.com.

[3] http://www.ampl.com.

[4] Fourer, R. Modeling languages versus matrix generators for linear programming. ACM Trans. Math. Software. 1983 June; 9(2):143–183.

[5] cross reference to modelling languages

[6] Gay, DM. Hooking your solver to AMPL. Technical Report 97-4-06. 1997. Computing Sciences Research Center, Bell Laboratories, Murray Hill, NJ.

[7] Fourer, R, Ma, J, Martin, RK. OSiL: An instance language for optimization. Comput. Optim. Appl. To appear. DOI 10.1007/s10589-008-9169-6.

[8] http://www.maximal-usa.com/slides/Svna01Max/sld036.htm.

[9] Duff, IS, Erisman, AM, Reid, JK. Direct Methods for Sparse Matrices. 1986. Oxford University Press, Oxford.

[10] Fourer, R, Lopes, L, Martin, RK. LPFML: A W3C XML schema for linear and integer programming. INFORMS J. Comput. 2005 Apr; 17(2):139–158.

[11] Rosenbrock, HH. An Automatic Method for Finding the Greatest or Least Value of a Function. Computer J. 1960; 3: 175–184.

[12] Halldórsson, BV, Thorsteinsson, ES, Kristjánsson, B. A modelling interface to nonlinear programming solvers: An instance — xMPS, the extended MPS format. 2000. available at http://www.maximalsoftware.com/resources/xmps/xmps.pdf.

[13] Conn, AR, Gould, NIM, Toint, PL. LANCELOT: A Fortran Package for Large-scale Nonlinear Optimization (Release A). 1992. Springer Series in Computational Mathematics Vol. 17. Springer Verlag, Berlin.

[14] Conn, AR, Gould, NIM, Toint, PL. The SIF Reference Document. http://www.numerical.rl.ac.uk/lancelot/sif/sifhtml.html.

[15] Gassmann, HI. The SMPS format for stochastic linear programs. http://myweb.dal.ca/gassmann/smps2.htm.

[16] Fourer, R, Gassmann, HI, Ma, J, Martin, RK. An XML-Based Schema for Stochastic Programs. Ann. Oper. Res. 2009 Feb;166:313–337.

[17] Waechter, A. Introduction to Ipopt. http://www.coin-or.org/Ipopt/documentation/.

[18] http://www.lindo.com.

[19] http://www.optimizationservices.org.

[20] Fourer, R, Ma, J, Martin, RK. Optimization Services: A Framework for Distributed Optimization. Oper. Res. To appear.

[21] Murtagh, BA. Advanced Linear Programming: Computation and Practice. 1981. McGraw-Hill Publishers, New York.

[22] Beale, EML, Forrest, JJH. Global optimization using special ordered sets. 1976. Mathematical Programming. 10; 52–69.

[23] Tomlin, JA. A suggested extension of special ordered sets to non-separable non-convex programming problems. pp. 359–370 in P. Hansen, ed. Studies on Graphs and Discrete Programming. 1981. North-Holland Publishing Company, Amsterdam.

[24] Escudero, LF. S3 sets, an extension of the Beale-Tomlin special ordered sets. Math. Prog. Ser. A and B. 1988 July; 42(1):113–123.

[25] `http://lpsolve.sourceforge.net/5.0/SOS.htm`.