Solver Communications: Options and Results

Horand Gassmann · Jun Ma · Kipp Martin

the date of receipt and acceptance should be inserted later

Abstract Much has been written about optimization instance formats. The MPS standard for linear mixed-integer programs is well known and has been around for many years. Other extensible formats are available for other optimization categories such as stochastic and nonlinear programming. However, the problem instance is not the only piece of information shared between the instance generator and the solver. Solver options and solver results must also be communicated. To our knowledge there is no commonly accepted format for representing either solver options or solver results. In this paper we propose a framework and theory for solver option and solver result representation in a modern distributed computing environment. A software implementation of the framework is available as an open-source COIN-OR project.

Keywords optimization · software · solver options · solver communication · result representation

Mathematics Subject Classification (2000) 90C99 · 65K05 · 49N99 · 68N01

1 Introduction

Instance representation for optimization models is well understood and documented. Instance generators, particularly algebraic modeling languages (see, e.g., [8,3,17,15]), communicate with solvers using descriptions such as the MPS format [18], AMPL nl format [13], OSiL [10], etc. Considerable

H. Gassmann (corresponding author)  
Rowe School of Business,  
Dalhousie University, Halifax, NS B3H 4R2, Canada, e-mail: Horand.Gassmann@dal.ca  

J. Ma  
JTechnologies, LLC, Arlington Heights, Illinois  
e-mail: majxuh@hotmail.com

K. Martin  
Booth School of Business, The University of Chicago, Chicago, Illinois  
e-mail: kmartin@chicagobooth.edu

Address(es) of author(s) should be given
effort has gone into making these instance representations sparse, comprehensive, extensible, and portable among many different solvers.

However, the problem instance is not the only piece of information shared between the instance generator and the solver. Solver options such as the maximum number of iterations, convergence criteria, or a starting point for the optimization algorithm may also have to be communicated to the solver. And once the solver has finished execution, the results, including solution status, statistics and values of the decision variables, must be reported back so that proper reports can be generated. Often, the results can also serve as the starting point for subsequent phases of optimization.

Unlike problem instance representations, solver options and solver results are not generally shared among solvers, and to our knowledge there is no commonly accepted format for representing either solver options or solver results. The MPS System that gave rise to the well-established and widely used description format for linear programs also featured a format for solver options (using so-called agenda cards) and LP results — see, e.g., the book by Murtagh [20]. These proposed formats were not generally accepted, however, and are not widely used or easily extensible.

The COIN-OR Optimization Services (OS) project [9] is a framework designed to support the solution of a wide variety of optimization problems in a service-oriented distributed computing environment, as well as a single standalone computer. A key aspect of the OS project is communicating options and results between client software on a local machine and solvers on a server. Along the way, and especially in a distributed environment, options and results may also be used/written by intermediate agents. One result of our research is a set of guiding principles that make it easier to develop and maintain stable, platform-independent interfaces. In this paper we summarize our discoveries, design principles, and logic behind the designs, developed through years of practical work and wide exposure to a large set of optimization systems and software. We propose a standard for communicating solver options and a standard for communicating solver results. These standards are implemented as two XML schemas, Optimization Services option Language (OSoL) for recording and communicating solver options to the solver, and Optimization Services result Language (OSrL) for communicating solver results back to the calling program and ultimately to the user.

This paper is organized as follows. In Section 2 we present the major design recommendations and principles for solver options and solver results. These include separation of instance standard functionality; the need to specify a relationship between the internal memory representation and the standard; the need to consider solver hierarchies when designing a standard; the level of representation detail; and the use of XML schemas to specify a standard. In Section 3 we describe the OSoL solver option standard. We outline the design principles for this option standard and describe the XML schema that implements the design principles. We also describe an in-memory representation of the solver options along with an associated API and discuss how OSoL is used to communicate with solvers and modeling languages. In Section 4 we describe the OSrL solver result standard, following the same format as in Section 3. We outline the design principles, describe the XML schema that implements the principles, describe the in-memory representation and associated API, and finally discuss the use of OSrL in solver and modeling language communication. We conclude the paper in Section 5 with a brief description of the OS COIN-OR open-source project along with information about obtaining and using the software that implements the concepts described in this paper.
2 Theory and Design Principles

The optimization paradigm consists of three basic functions: instance generation — creating a problem instance to be optimized; instance consumption — finding an optimal or near-optimal solution (or determining unboundedness or infeasibility); and reporting the result of instance consumption (i.e., the solution) back to the user.

Within this paradigm it is useful to distinguish between a tightly coupled system and a loosely coupled system. In a tightly coupled system the first two functions, instance generation and consumption, are performed by a single piece of software. An instance is generated for consumption by a specific solver. A good example of a tightly coupled system is LINGO [17]. In LINGO, a user generates an instance with the LINGO modeling language and this model is optimized using an internal LINDO solver. But even in such tightly coupled systems, often there is a need to export internal problem structures into some standard formats. In a loosely coupled system the instance generation is “solver agnostic”. The instance is generated without requiring knowledge of which solver will be used. Hence there is a need to create a generic and portable instance representation; likewise for solver options and solver results. AMPL, GAMS, and MPL are examples of modeling languages designed for a loosely coupled system. A loosely coupled system is more consistent with modern service-oriented IT architectures than is a tightly coupled system. We focus on loosely coupled systems in this paper.

The basic principle of a loosely coupled system is illustrated in Figure 1: The problem instance is created (somehow) by the client (or “consumer”) and packaged into a format suitable for transmitting it to the server or service, which typically includes a solver. This is often accompanied by a set of solver options that control the behavior of the solver. After optimizing the instance, the server passes back the solution to the client.

Typically the client, such as an algebraic modeling language, produces an optimization instance in its own format, which then has to be translated into a form understandable by the server, and similarly for solver options and solver results (where the latter are of course transmitted in the opposite direction, from server to client). This leads to a large number of possible translators (for every possible combination of client and server), unless there is a common intermediate exchange
format. The Optimization Services (OS) Project provides this common format along with other higher level standards for how information should be exchanged. Such a common intermediate format immediately results in far fewer links: one translator from each type of client to the common format, and another translator that makes the common format understandable to the server — independent of the original source.

Having a common format for instance, options and results is even more important in distributed computing. In a distributed environment, the translation step involves not just a change of the representation of the information, it may also involve physical transmission over the Internet, which may require packaging the information with suitable wrappers such as a SOAP envelope [26]. Almost all of the complexity of this can be hidden from the user. For instance, Optimization Services appears to an algebraic modeling language such as AMPL like a black box; all the user has to know is one thing: the URL of the computer on which this solver resides. There is an additional option to select the final optimization solver that is to be used (e.g., Cbc [4], Ipopt [29], Cplex [6]), but an intelligent instance-solver matching service is used to infer a suitable default solver based on characteristics of the optimization instance. And by embedding the service in a compute cloud, it should be possible with appropriate registry and discovery mechanisms to automate even the selection of the solver location.

A further exciting possibility is to use the same or similar mechanisms to communicate not just with optimization solvers, but with other analysis tools such as analyzers or simulation software.

In the remainder of this section we present the major design recommendations and principles for representing and exchanging solver options and solver results.

2.1 Separation of Standard Functionality

We found it important and beneficial to distinguish among the following components in a loosely coupled optimization environment.

- **Instance representation**: An instance representation is a complete description of the optimization problem that is to be solved. It should not contain extraneous information; in particular, it should not be convoluted with starting points, solver options, solver results, etc. A model instance should be completely independent of software, solvers, or algorithms that are used to solve the problem.

- **Option representation**: It is often necessary to pass options to instruct the solver how the problem should be solved, e.g., where to start the solution from some initial variable values, or which pricing mechanism to use in an LP solver. Representing options is a challenge, given the wide variety of solvers and a complete lack of a standard for option formats.

- **Result representation**: After a problem is solved (or terminated otherwise) results must be passed as output back from the solver to the client. There may be one or more solutions for the same input problem instance, even from the same solver. Each solution may contain a minimal amount of information such as optimal objective function value(s) and the optimal values of the decision variables, or more detailed information such as range information in the case of a linear programming solution. For instances with multiple objectives, the output may even represent an entire set of efficient solutions, etc.

- **Modification representation**: There are many solution procedures such as column generation and cutting plane algorithms that require modifying the original problem instance. It is impractical to separately generate numerous large-scale problem instances that only vary slightly from
the initial instance or from each other. Aside from the possibility of specifying multiple objectives and RHS in MPS, and switching between them, there is no standard instance modification format that the authors are aware of. This is a good topic for further research and will not be treated in this paper.

Having a separate representation standard for each of these four components allows for maximum flexibility, modularity, and reusability in software choices and design. Indeed, as pointed out later in the paper, the design objectives for option and result standards are quite different from the design objectives for an instance representation standard. It is equally important to keep in mind that the representation of instances should be separate and completely unaware of how the instances are communicated between the client and server [9]. We do not deal with instance communication in this paper.

2.2 In-Memory Representation and API Design

Representation of instances, options, and results necessarily requires the description of a file format, something that can be written out, reused, archived, or transmitted over a network. However, what is convenient as a file format may not work very well in computer memory. Hence there is a need to convert the file representation into in-memory objects and vice versa. This process is aided greatly by formal mapping rules that state in an abstract way how pieces of the file representation correspond to elements of the in-memory representation. In Section 3.3 we describe a set of formal mapping rules between an XML-based file representation for solver options and the corresponding in-memory representation. Section 4.3 provides an analogous description for solver results.

In addition, there is a need for an application programming interface (API), by which a user (or an intermediate layer of software) can build or interpret either the file or the in-memory representation in whole or in pieces. The API should contain conventional get() and set() methods for various parts of the format. Most of these methods are pure convenience methods, however, as the description of the format should be sufficient to enable users with sufficient knowledge of programming to write such methods for themselves.

2.3 Solver Hierarchies

As stated in the introduction, the COIN-OR Optimization Services project is a framework for implementing optimization in a distributed environment, i.e., over a network. In this regard it is similar to the well-known NEOS (Network Enabled Optimization System) project [7]. From a user’s perspective, OS and the NEOS server act like solvers, or to be more specific, meta-solvers. The user submits an optimization instance, and receives back a result. However, in a distributed environment, in addition to the regular optimization solver on the server computer, there is a communication layer. The client cannot directly invoke a regular optimization solver, and needs to communicate with a meta-solver, or a remote solver “façade”, that acts as an intermediate proxy. An instance and options are sent to the remote solver proxy, which then further delegates the instance to the real “optimization solver” at its back end. Figure 2 depicts the different layers, based on the actual implementation in the OS system.

There could in fact be many “optimization solvers”, but the crucial point is that all of them communicate through the same solver proxy. In the OS project, such a proxy is technically implemented as a Web service, whose APIs are standardized in the OS communication protocol. Results
Fig. 2 The different software layers involved in a distributed environment
from the optimization solver are returned first to the remote “solver proxy” or directly to the client. See [9] for a description of how this works in the OS framework.

In this distributed environment, it is useful to be able to distinguish between options meant to control the remote solver proxy (or Web service) and options intended for the real optimization solver. In other words there are options that pertain strictly to the solver proxy and there are options that pertain strictly to the optimization of a specific instance. For example, there should be an option that tells the solver proxy which optimization solver to call. Moreover, the solver proxy must be able to pass options to the optimization solver.

There might also be other options describing the suitability of the remote system to tackle the problem at hand, e.g., minimum disk space and CPU requirements. These options should be intercepted even earlier in the communication chain.

A feature of good design in option communication is that it be easy to determine at which level in the communication chain an option is to be deciphered and acted upon. This is a challenge that can arise quite frequently. For instance, a user might be interested in setting a time limit for a job. This may be controlled by the operating system at the job level or by a solver for a particular instance. In our system we therefore provide two separate mechanisms for setting time limits. This is explained in more detail in Section 3.2.

There may also be a hierarchy of optimization solvers. Consider the COIN-OR global optimizer Couenne [1]. The Couenne solver makes use of and calls the COIN-OR Bonmin [2] and Ipopt [29] solvers. It may be necessary to send options to Couenne that are specific to controlling the behavior of Bonmin or Ipopt. Therefore, a good option representation design allows for an optimization solver to know which options should get passed to the other optimization solvers that it is linked with. This issue is discussed in further detail in Section 3.2.

As with hierarchies in the solver option representation, there are results that will be specific to a solver proxy and results that are specific to a real optimization solver. For example, the solver proxy might report the total time the job associated with this instance was in the system. However, an optimization solver might report the CPU time required to optimize a specific model instance. There are system specific job results versus the instance optimization result, and an important design feature is to make this delineation both clear and comprehensive. How this is done in our system is described in Section 4.2.

2.4 Representation Detail in a Standard

In Section 2.1 we stated that it is important to have distinct representation standards. One reason for doing this is that the amount of detail required to specify the information may vary considerably depending upon functionality. For example, a linear program is a well-defined entity, but the solution of a linear program is not. We cannot have a linear program without constraints or variables, but we can have a linear programming solution without reduced costs or right-hand side sensitivity information. In general, different optimization solvers may present their results in different formats, and some may include more detail than others. The level of solution detail is up to the solver developer and would be difficult to standardize. The same applies in the world of options, where the meanings and formats of options vary significantly between solvers. Thus, whereas with our instance description we have tried to be as encompassing and complete as possible, with our option
and result descriptions we have taken a minimalist but highly flexible approach. This minimalist philosophy, and the logic behind it, is discussed further in Section 3.1.

2.5 Use of XML and Schemas to Specify a Standard

In this paper we specify standards for representing solver options and solver results. By “standard” we mean that there is a specific and unambiguous format for representing the options for a solver or the results returned by it. Specifying a format is greatly facilitated by using XML. The key benefit of XML is an XML schema that provides a rigorous and unambiguous way to specify the syntax for an XML document. Indeed, the schema specification is the standard. In Sections 3.2 and 4.2 we describe the schemas for the option standard (OSoL) and the result standard (OSrL). There are other reasons why XML is the best way to impose a standard.

- By using an XML schema it is possible to specify additional constraints and rules governing the content of an XML document. This includes requiring specific data types, specifying default values, constraining the order of elements, specifying enumeration lists, etc.
- The availability of a schema, which is also an XML document, facilitates the writing of parsers. In fact, there are open source libraries that take a schema as an input and automatically parse and validate the XML document.
- There are easy and natural transcription rules to convert the content of the XML files into internal data structures (such as the Document Object Model (DOM) [25]). There are open source libraries for creating these internal data structures (see, e.g., [24]).
- XML is an open W3C standard that is nonproprietary, unencumbered by copyright, patent, trade secret, or any other intellectual property restriction.
- XML-based Extensible Stylesheet Language (XSL) [27] offers a convenient way to specify translations of XML documents. For example, if an optimization solution is formatted in OSrL, XSL can be applied to the solution instance to easily produce an HTML document that displays the solution data in a user-friendly form.

3 The OS Option Standard

In this section we describe our solver option format OSoL in more detail.

The separation between what constitutes an instance element versus a solver option is not clear-cut and is not universally agreed upon. For example, initial values can reasonably be argued to fall into either category. Some modeling languages like AMPL and GAMS do indeed treat initial values as part of the optimization instance and transmit initial values to the solver as part of the instance representation. In fact, AMPL treats all data items indexed over the variables, constraints or objectives of the problem in this way, including branching priorities, basis status, reduced costs and slack values, even user-defined items. The dichotomy in AMPL is therefore not between instance and options, but between array-valued data items and scalar options. Only the latter are communicated to the solver in an option string, all the array-valued items are contained in the .nl file that holds the optimization instance [13].

GAMS uses a very similar mechanism but it allows a very small number of array-valued options to be put into the options file. (In the case of the GAMS/Cplex interface these consist of two
array-valued options dealing with sensitivity ranging on objective coefficients and right-hand side ranges, respectively [11, pp.35,41].

After lengthy discussions and careful reflection, the authors decided to deviate from this practice and to treat initial values, initial basis information, branching weights for integer variables and all similar information as part of the option set. This reflects our view on the separation of functionality described in Section 2.1. One advantage of this approach is that it allows indexing of vector-valued options over arbitrary index sets not necessarily associated with the problem instance, e.g., seeds for a random number generator.

3.1 Philosophy of Option Design

Solver options are not only vast and constantly growing in number, but also vary greatly and lack standardization even among the most common ones such as feasibility tolerances, maximum number of iterations or even “time” limits. This influences the design of any standard for representing options. It is further useful to distinguish between syntax (how to represent the options) and semantics (how to interpret their meaning).

There is another aspect that influences the design of option and result formats: A solver is 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. Finally, particularly in a distributed environment, additional facilities are useful 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.

These considerations led us to the following basic principles that aided us in our design of the OSoL schema.

1. Unless universally accepted and commonly used, we do not try to enumerate or hard code options, especially those related with optimization solution and solver algorithms, into an API or standard. There are too many solvers, each with their own needs and functionality. Finding common options across integer programming, linear, nonlinear, stochastic, SDP solvers etc. is hopeless. In addition, solvers may interpret the same option differently.
2. The user should be able to specify any option that a solver or analysis tool will support at the file level and the API level, in the most natural way possible.
3. The option API can parse the options into generic data structures, but should not have to interpret options. The option interface deals with the syntax only; the semantics are left to the solver.
4. The option interface should support multiple data types — even special types defined by the solver developers.
5. The format and API should be extensible, so that most changes that solver developers may make to the options do not require changes to the option interface.
6. The interface should be able to handle sparsity. Option files are usually small compared with problem instances, because they mostly contain only scalar-valued options. But an option can be array-valued (though usually just one-dimensional, e.g., initial variable values), and in practice,
most values can be zero. Therefore, an option format should be able to represent array-valued options in a sparse format.

7. It should be kept in mind that there are not only optimization specific options, but also options related to the environment where the optimization solution is carried out. If possible, options of different layers should be distinguished, as not all options are carried all the way through a system hierarchy to be used by the end optimization solvers.

8. In some instances it is desirable to link the option file with a particular problem instance. This feature should be supported if possible.

9. Finally, the option instance should be easy and quick to parse.

3.2 Description of the OSoL schema

Working in a distributed environment one may have to send information not just to the solver, but to the entire environment (such as solver proxy, computer system) at the remote location. For instance, the operating system may need to be instructed to move files around before the solution can even begin, there may be provisions as to how the user is notified when the job finishes, etc. Many of these items are not seen by the solver, but we chose to include them in the same file, mostly to limit the number of option files that need to be sent.

To make it easier to distinguish the level at which an option is needed, the OSoL schema is broken down into several sections, as shown in Figure 3. None of these sections is required.

![Fig. 3 The top-level OSoL elements](image-url)
- a header section that can be used to document basic information about the file itself, its creation date, its source if appropriate, authorship information, etc. This section is for documentation purposes only and is not used by the solver.
- general information about which solver is to be used, location of input files, how the user is to be notified of completion of the process, etc.
- system options, which are especially important in a distributed environment or when cloud computing is involved. These options include minimum requirements of disk space and memory availability, number of CPU cores, etc.
- service options that allow the same option file format to be used for communication not just with optimization solvers but also other software such as analyzers, schedulers, simulation software, etc.
- job options, including preparatory and cleanup steps to be used prior to, respectively, after the solver phase. These include files and directories that may have to be created, moved, or destroyed. This information would be intercepted by the operating system through appropriate system actions, and would not be passed on to the solver directly.

One very useful application of this principle is illustrated in the following snippet from an osol file:

```xml
<job>
    <otherOptions numberOfOtherOptions="1">
        <other name="get_stdout" value="true"/>
    </otherOptions>
</job>
```

This option instructs the solver server running on a remote system to capture the output generated by the solver and to return this output as part of the result file. (An example of the output generated by this option is shown in section 4.)

- optimization options, which are the options seen by the optimization solver. These can further be subdivided into options acting on (a subset of) the variables, such as initial values, basis status or the like, options acting on the constraints or objectives, and general options. (See Figure 4.)

The most important realization is that all such options can be transmitted by the communication layer without knowledge of the underlying semantics; only syntax information is required. For instance, the element `<solverOption>` expands as follows (see Figure 5).

- The `name` attribute identifies this option to the solver. There is no requirement on the parser to check the name or to match it against a list of valid names. It is presumed that the user supplied a name in the osol file that is valid in the sense that the solver will understand it once it is passed in, but error detection is left entirely to the solver.
- The `value` attribute is used to communicate the option value to the solver. For full flexibility all values are stored as strings — both in the osol file and in the internal representation once the file has been parsed into memory.
- The `type` attribute is used to record the type of variable. Since solvers are not uniform in their language (e.g., Ipopt uses the term “numeric” to describe floating point or “double” values), there is no assumption made about the types that can be expected; the value of this attribute is again an ordinary string.
- The `solver` attribute is used to limit the option to one particular solver. If the solver attribute is missing, the option will be passed to whichever solver is selected at runtime.
The description attribute can be used for annotation about the purpose of this option.

- The category attribute allows additional information to be passed to the solver regarding the option. In LINDO [16], for instance, it is possible to set attributes affecting either the entire session (or "environment" in the parlance of LINDO), or just one individual model. Both use cases are illustrated in the example below.

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

- The optional attribute NumberOfItems and the optional sequence of <item> elements can be used to provide values for array-valued options. These complement other constructs in the variables, constraints and objectives elements, which provide array-valued options when the array elements are indexed over the columns, constraint rows and objectives of the problem. Sometimes, however, an array of options must be specified that is indexed over a different set, such as an array of seed values for a random number generator.

We end this section with a small file in the OSoL format (Figure 6). This file should be self-explanatory. It first gives the URL of a remote server, specifies that the job is to be run only if the
server resides on a computer with at least 24 Gb of memory, provides initial values for two decision variables, and gives four solver options. Two of these options are intended for the Ipopt solver [29]; the other two are for the LINDO solver [30]. 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 the maximum number of iterations; 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.

3.3 Internal representation — the OSOption class

The OSoL schema defines an XML vocabulary to describe optimization options. The options instance may persist on a file system, or may temporarily be encapsulated in a SOAP envelope for use in a distributed system. However, at some point before the instance is solved by a solver, the options must be stored as objects in memory.

Our OSOption class is the in-memory representation of an OSoL file or string. This class has an API defined by a collection of get() methods for extracting various components from problem options, and a collection of set() methods for modifying or generating optimization options.

Internally the option file is represented as a C++ class exhibiting the same tree structure as the XML file. In particular, the solver options are represented as an array of 6-tuples, each tuple consisting of a name, solver, type, value, description, and category. No attempt is made by the
parser to interpret an option in any way. Changes to the solver options therefore do not require any changes to the parser, the storage scheme, or the API.

The mapping between the OSoL schema and the OSOption class follows rules that were described previously [10,12] and are modified here to illustrate their use with OSoL.

- Each complexType in an OSoL schema corresponds to a class in OSOption. Thus the OSoL schema’s complexType SolverOptions corresponds to the class SolverOptions. Elements in an actual OSoL file then correspond to objects in OSOption; for example, the <solverOptions> element that is of type SolverOptions in an OSoL file corresponds to a solverOptions object, which is of class SolverOptions.
- An attribute or element used in the definition of a complexType is a member of the corresponding OSOption class, and the type of the attribute or element matches the type of the member. In
Figure 7, for example, name is an attribute of the OSoL complexType named SolverOption, and name is a member of the OSOption class SolverOption; both have type string.

A sequence of identical schema elements corresponds to an array. For example, in Figure 7 the complexType SolverOptions has a sequence of <solverOption> elements that are of type SolverOption, and the corresponding SolverOptions class has a member solverOption that is an array of type SolverOption.

**Schema complexType**

```xml
<xs:complexType name="SolverOptions">  
    <xs:sequence>  
        <xs:element name="solverOption" type="SolverOption" minOccurs="0" maxOccurs="unbounded"/>  
    </xs:sequence>  
    <xs:attribute name="numberOfSolverOptions" type="xs:nonNegativeInteger" use="required"/>  
</xs:complexType>
```

**In-memory class**

```cpp
class SolverOptions{
    public:
        SolverOptions();
    SolverOption **solverOption;
    int numberOfSolverOptions;
}; // SolverOptions
```

**OSoL elements**

```xml
<osol ...>
    <optimization>
        osoption->optimization = new OptimizationOption();
        osoption->optimization->solverOptions = new SolverOptions();
    </optimization>
    <solverOptions numberOfSolverOptions="2">
        osoption->optimization->solverOptions->numberOfSolverOptions = 2;
        name="OsiDoReducePrint" osoption->optimization->solverOptions->solverOption[0]->name = "OsiDoReducePrint";
        solver="osi" osoption->optimization->solverOptions->solverOption[0]->solver = "osi";
        type="OsiHintParam" osoption->optimization->solverOptions->solverOption[0]->type = "OsiHintParam";
        value="true" osoption->optimization->solverOptions->solverOption[0]->value = true;
    </solverOptions>
    <solverOptions name="OsiHintTry" type="OsiHintStrength"/>
</solverOptions>
</optimization>
<osol>
```

**In-memory objects**

```cpp
OSOption* osoption = new OSPOption();
osoption->optimization = new OptimizationOption();
osoption->optimization->solverOptions = new SolverOptions();
osoption->optimization->solverOptions->numberOfSolverOptions = 2;
osoption->optimization->solverOptions->solverOption[0]->name = "OsiDoReducePrint";
osoption->optimization->solverOptions->solverOption[0]->solver = "osi";
osoption->optimization->solverOptions->solverOption[0]->type = "OsiHintParam";
osoption->optimization->solverOptions->solverOption[0]->value = true;
osoption->optimization->solverOptions->solverOption[1]->name = "OsiHintTry";
osoption->optimization->solverOptions->solverOption[1]->type = "OsiHintStrength";
```

Fig. 7 The <solverOptions> element and its representation as an in-memory object.
3.4 Solver Communication

Eventually a framework such as OS will have to pass the information to the solver in a form that the solver can understand. This depends on the solver and on the API. We illustrate three available methods in increasing order of rigidity.

3.4.1 Ipopt

Communication with Ipopt is very flexible, because the Ipopt API contains three methods that can be used to communicate options as name/value pairs. The great advantage of this approach is its extensibility. The solver developer is free to change the internal options without any modifications to the OSoL parser or the solver interface. Let’s say, for instance, that the solver developer wishes to implement a new option we will call NewRule that affects the performance of Ipopt’s interior point algorithm, and assume NewRule can take an optional integer parameter. This is recorded by the user (upon receiving the pertinent information from the solver developer) as

\[\text{<option name="NewRule" type="integer" solver="ipopt" value="5">}\]

The parser reads and stores this into the data structure and passes it on to Ipopt as a name/value pair — no interpretation of the semantics takes place, and neither the parser nor the Ipopt interface needs to be modified. Ipopt decodes the option, checks its validity, and acts appropriately. Another big advantage of using the API to set solver options is that the set() methods give instant feedback as to whether an option was declared properly. If any error condition is detected, it is possible to stop the program gracefully.

3.4.2 Cbc

Slightly less flexible is the way in which OS communicates with Cbc. Cbc exposes as part of its API a command line parser. This makes it possible to string together all the options selected in an osol file into a long string and to have Cbc parse that string. Again, there is no need for OS to know the meaning of the Cbc options. However, the feedback mechanism available with Ipopt cannot be used. Cbc either terminates or continues execution with a faulty option, and this behavior cannot be controlled from the calling program. This is not very attractive, particularly if a simple typing error causes an integer program to run for several days on a remote computer before the error is detected.

3.4.3 Osi

The Osi interface is used by a number of open source (e.g., Clp [5], Dylp [14], Symphony [21], Glpk [19]) and commercial (e.g., Cplex [6]) linear programming solvers. Unlike the other interfaces just described, it relies on an enumeration of commonly available options. This limits its use, as only very common options are supported. Options not covered by the enumeration (even very basic ones, such as whether to use a simplex or interior point method) thus cannot be communicated. Extending the interface is also quite cumbersome, and the interface must be aware of the options that can be passed, as well as their semantics. However, the developers of Osi are aware of these shortcomings and have recently announced a major overhaul to address these issues [22].
3.4.4 Other interfaces

Other interfaces would be possible, based on exposing more or fewer methods in the API. In the extreme, the user could be directed to put all the solver options into a file (with solver-specific syntax) and to supply only the location of the file to the solver. This would again limit the knowledge needed by the OS interface concerning the options, but it would be challenging to implement in a distributed environment.

3.5 Modeling Language Communication

It is important to keep in mind that the OS framework is not intended to be used in stand-alone fashion exclusively; often it is called from a modeling language such as AMPL or GAMS. We implemented an AMPL solver interface called OSAmplClient [12, section 5.1] which looks to AMPL like an ordinary solver and can be selected using the AMPL command

\begin{verbatim}
option solver OSAmplClient;

AMPL communicates with the OSAmplClient in two ways: the AMPL command
\end{verbatim}

\begin{verbatim}
option OSAmplClient_options "...";
\end{verbatim}

\begin{verbatim}
can be used to construct a command line for the OSAmplClient. The command line options are intercepted and acted upon only locally, but they may include the name and location of an OSoL file, which can be used to pass options through the OSAmplClient to the solver.
\end{verbatim}

On a subsequent
\begin{verbatim}
solve;
\end{verbatim}

command the optimization instance is constructed by AMPL and passed to OSAmplClient as a string in .nl format. OSAmplClient translates this string into OSiL and passes it on the OSSolverService. Some care must be exercised, since the .nl file may contain array-valued options (indexed over variables, constraints and objectives) that may have to be combined with other options found in the OSoL file.

A similar mechanism is available to call OS from GAMS [28], and other interfaces are forthcoming.

A new COIN-OR project, CMPL [23], is one of the first mathematical programming languages to output OSiL and OSoL files directly. It also calls the Optimization Services solver directly and reads OSrL files internally. In fact, one could consider CMPL and OS to be a tightly coupled system.

4 The OS Result Standard

It is little known that the venerable MPS format for linear program instances has a corresponding format for the results of solving linear programs [20]. All of today’s major optimization modeling systems have distinct nonstandard formats in which they report the results of their computations. In a successful distributed optimization framework, a standard for this purpose is as important as a standard for reporting problem instances. Thus another part of our project has been to design OSrL, an XML-based protocol for representing the solutions of large-scale optimization problems of all kinds.
4.1 Philosophy of Solver Result Design

Different classes of optimization problems have different types of results. For example, in linear programming, allowable increases and decreases are often reported for constraint right-hand sides and objective function coefficients. This information is not necessary for a nonlinear program since there these values are typically zero. In the case of semidefinite programming, the result may be a matrix of values, rather than just a vector, and so on. Hence, like with solver options flexibility is critical.

Moreover, since there are different layers of the software that act upon different portions of the options selected by the user, it is possible that each layer produces output corresponding to the actions it was directed to perform by the options. The output from all layers should be transmitted back to the user in a common container (e.g., a file) and should adhere to the same divisions. Another consideration is that it should be easy to reuse the results from one optimization as options in a subsequent run, or to pass the output to other software such as analyzers or report generators.

For instance, it is quite natural to want to use the optimal values of one optimization as the starting point for the solution of a slightly modified problem. Common formats for solution vectors and starting points make this process easier.

In addition, it is useful to adhere to the other design principles formulated in section 3.1, namely, the result format should

– be extensible,
– handle sparsity (for vector-valued results such as variable values or dual values),
– respect the separation of syntax and semantics,
– and be quick and easy to parse.

4.2 Description of the OSrL schema

In reusing the solution of one problem in a different context (perhaps with slightly different parameter values) it is entirely possible to want to define the starting values of the next problem to be equal to the optimal values of the previous problem. There is a parallel to be drawn between option and result standards, even though on the face of it they seem to serve very different purposes.

In keeping with the previous paragraph we consciously designed OSrL to have the same top level structure as OSoL. (See Figure 8.) This includes the following main elements:

– a header section that gives the same information as the OSoL header, that is, information about the file itself, its creator and creation date, licensing information, as well as any other documentation desired;
– general information, which includes the status of the process (i.e., whether the solver terminated normally or the process was stopped due to one of several possible error conditions);
– system results containing information about the computer system on which the solver is running;
– service results; these are statistics such as the time since the service was last restarted, the number of jobs completed, the server utilization rate, etc.;
– job results, including timing information, resource usage statistics, etc., of jobs launched by the service;
– optimization results; these are the results from the optimization, including information about the solution or solutions (values of primal and dual variables, range information, and whatever other information the solver developers decided to make available to the user).
Our design goal has again been to maximize flexibility in reporting optimization results but to keep the design simple.

4.3 Internal representation — the OSResult class

The internal representation, a C++ class called OSResult, follows the same mapping rules as the OSOption class described in section 3.3. We dispense with an example in order to preserve space.

4.4 Solver Communication

Each solver driver must retrieve from the solver all results that the solver chooses to report. Typically these include the optimal value of the objective function along with the values of the optimal decision variables. Other items that may be returned — either as a matter of course, or upon demand — include dual variable values, range information, basis status, etc. These items are then stored into the appropriate slots within the OSResult class.

However, the solver results could be subject to a similar proliferation of items as the solver options, and typically there is less attention paid to the API. In order to avoid having to maintain
complete enumerations of solver results we allow the user to request the return of the complete solver-generated output. This information is returned as a single element in the OSrL file. Figure 9 shows selected portions of the OSrL file generated from a call to the COIN-OR solver Cbc.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<osrl xmlns="os.optimizationservices.org" ... />
<general>
  <generalStatus type="normal"/>
  <instanceName>P0033</instanceName>
  <solverInvoked>COIN-OR cbc</solverInvoked>
</general>
<job>
  <timingInformation numberOfTimes="1">
    <time type="elapsedTime" unit="second" category="total">
      4.9999999999999996e-2
    </time>
  </timingInformation>
  <otherResults numberOfOtherResults="1">
    <other name="stdout_capture">
      Welcome to the CBC MILP Solver
      Version: Trunk (unstable)
      Build Date: Jun 1 2012
      Revision Number: 1782
      Cbc0038I Pass 1: suminf. 0.93926 (3) obj. 2896.64 iterations 6
      Result - Stopped on node limit
      Objective value: 3347.00000000
      Lower bound: 2819.357
      Gap: 0.19
      ...
    </other>
  </job>
<optimization numberOfSolutions="1" numberOfVariables="33" numberOfConstraints="16">
  <solution targetObjectiveIdx="-1">
    <status type="feasible" description="node limit reached"/>
  </solution>
<variables>
  <values numberOfVar="33">
    <var idx="0">1</var> <-- x[0] = 1
    ...
</variables>
</optimization>

Fig. 9 Portions of a sample .osrl file (with annotations)

4.5 Modeling Language Communication

The contents of the .osrl file may have to be communicated back to the algebraic modelling language. This can take place via the writing and reading of solution files or in memory, by accessing the API of either the AML or the OSSolverService. The most important data items are undoubtedly the
values of the primal and dual variables, but if the solver computes other information, such as range
information or basis status, this must also be transferred.

4.5.1 AMPL

AMPL has an interactive shell that takes commands from the keyboard sequentially. Once the user
types

\texttt{solve;}

as in section 3.5, several things happen behind the scenes: AMPL instantiates the current problem
from the model and data specified by the user, writes the instance into an nl file, along with any
array-valued information such as starting points and starting basis information; starts a shell process
and builds a command line for the solver to run inside the shell; starts a listener that waits until
the solver process has finished and has returned a file in the AMPL .sol format; and finally reads
the .sol file and transfers its contents into the internal AMPL data structure. Once the control is
returned to the user — assuming there were no errors along the way — the user can inspect the
returned solution, make modifications to the model, start another solve, etc. Some of these steps
can even be automated using the AMPL command language and looping constructs.

From the point of view of the OS software, the translation process from solver output to AMPL
.sol file is a two-step process: First the solver interface converts the solver result to OSrL format,
and then the AMPL interface processes the OSrL data and writes the .sol file.

4.5.2 GAMS

A very similar mechanism is employed by GAMS. Since GAMS release 23.6, an OSSolverService
is available for remote solving of GAMS problems over the internet. On Unix systems only a
command line interface is provided. The command line executable reads GAMS input, prepares the
optimization instance, starts the solver, and processes the output into a report file in a GAMS-
specific format. If OSSolverServices is used as the solver, an optional solution file in OSrL format can
also be retained. On Windows systems GAMS also provides an interactive development environment
(IDE), which produces the same output and in addition parses the solution file.

5 Software Implementation and availability

The OSSolverService system is implemented as a COIN-OR project. It links to several COIN-
OR solvers (Clp, Cbc, Ipopt, Bonmin, Couenne, SYMPHONY, DyLP) and has been successfully
hooked to Glpk, Lindo and Cplex. It supports both local and remote access. Several remote servers
exist; the most stable of these is the one maintained by Kipp Martin at \url{https://74.94.100.129:8080/OSServer/services/OSSolverService}.
Source and executables are available from the website \url{https://projects.coin-or.org/OS}.
The web site also contains a user's manual and links to further resources, such as descriptions of
the schemas alluded to in this paper, an overview of the OSSolverServices project and related items.
References