Noname manuscript No.
(will be inserted by the editor)

Communication Protocols for Options and Results in a Distributed Optimization Environment

Horand Gassmann · Jun Ma · Kipp Martin

July 23, 2014

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 · framework · design pattern · system · software · solver options · solver communication · result representation · Optimization Services

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., [12,3,25,23]), communicate with solvers

H. Gassmann (corresponding author)
Rowe School of Business,
Dalhousie University, Halifax, NS B3H 4R2, Canada, e-mail: Horand.Gassmann@dal.ca
ph:+1-902-494-1844, fax:+1-902-494-1107

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

using representations such as the MPS format [26], AMPL .nl format [20], OSiL [15], etc. Considerable 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 [30]. These proposed formats were not generally accepted, however, and are not widely used or easily extensible.

Optimization Services [14,27] is a framework in the sense of Riehle [34] (see also Gamma et al. [16]) designed to support the solution of a wide variety of optimization problems in a service-oriented distributed computing environment, as well as on a single standalone computer. Optimization Services is intended to standardize from an overall specification level how a set of components (e.g. libraries, classes, interfaces, executables and services) should be designed and implemented cooperatively in order to solve an optimization problem.

Along the way of designing and implementing such a framework, and especially in a distributed environment, options and results may also be used and 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 describe 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. The case for XML has been made elsewhere (see [13,14]). We repeat some of the arguments in section 2.5.

A proof-of-concept implementation of some of the key aspects of our framework is hosted at COIN-OR [18]. We will make frequent reference to this implementation throughout this paper. We shall be careful to distinguish between the framework, Optimization Services, and the COIN-OR implementation, `OS`. In particular, we would like to point out that the framework could be implemented completely independently of electronic computers and networks. The "modeling system" could be a professor writing down the XML-formatted files on paper, the communication layer could be based on interdepartmental or even "sneaker" mail, and the "solver" could be a graduate student sitting in an office three doors down the hall. This distinguishes our efforts from initiatives such as the Open Grid Services Architecture (OGSA) [50,32] or the Globus Toolkit (see [45] and other publications at `toolkit.globus.org`). OGSA is a communication protocol, specifically designed to support grid requirements. In contrast OSxL is about standardization of domain representation, in this case optimization. It can be transported or communicated using the Web service or OGSA or any other mechanism.

Previous work on Optimization Services [27,14] laid out the theoretical basis of the framework design (specifically abstract and conceptual components) at a high level and described how these abstractions should work together in a collaborative way. These documents do not talk in any detail about how the abstractions themselves should be designed. The current paper drills down into two of these abstractions, namely options and results, and proposes the design principles and theories behind OSoL and OSrL. In addition, OSoL and OSrL have undergone substantial modification since the initial proposal in these documents.

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 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 end the paper in Section 5 with some conclusions and a brief description of the `OS` COIN-OR open-source project [18] along with information about obtaining and using the software that implements the concepts described in this paper.

2 Theory and Design Principles

One of the main objectives of Optimization Services as a framework is to generate optimization design patterns [46]. A design pattern is a repeatable solution that has been extracted and formalized from best practices. Effectively generalized design patterns have proven highly influential in various fields such as software engineering. In certain situations, they have helped revolutionize how researchers, analysts and software designers approach solutions efficiently.

Frameworks further generalize and piece together related design patterns in a collaborative manner. Systems, libraries and tools have been implemented to facilitate the use of such proven frameworks, in which people only need to deal with some high level business requirements as specified and concentrate on implementing their own components, leaving the entire framework to deal with the rest of the work. The system then is an implemented solution based on the framework that effectively reuses many good design patterns.

In the context of this paper, a pattern means a tried and true way to deal with an optimization process — from the model context to the problem instance and to the final solution — that appears over and over again. From the software perspective, the optimization process almost always 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. Such a pattern justifies the generation of an effective and standard means of communication between OR software components, therefore realizing one of the goals of Optimization Services as a framework to bring order into chaos. The `OS` project implements many of the libraries and tools

following the specifications of the Optimization Services framework. Using the tools we have also set up systems that solve real-life optimization problems.

Within this paradigm it is useful to distinguish between *tightly coupled* systems and *loosely coupled* systems. 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 [25]. In LINGO, a user generates an instance with the LINGO modeling language and this model is optimized using an internal LINDO solver. 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 by the client 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. At this point the solution could be sent to an analyzer or similar software, the instance could be modified, and the communication cycle can be repeated, as often as necessary.

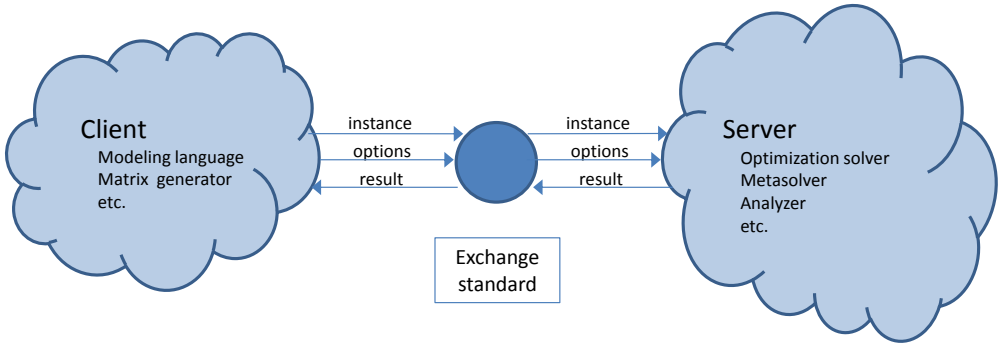

Fig. 1 Client and server in an optimization system

Typically, the client, such as an algebraic modeling language, produces an optimization instance in its own format, which then must be translated into a form understandable by the server. Solver options, as well as solver results, must also be communicated between the client and server, as depicted in Figure 1. 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 framework 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 by requiring just one translator for each client and one for each server, each of which can communicate directly with the universal intermediate format.

Having a common format for instances, options and results is even more important in distributed computing. In a distributed environment, the translation step may involve more than just a change of the representation of the information, as it may also involve physical transmission over the Internet. This may require packaging the information with suitable wrappers such as a SOAP envelope [40]. Almost all of the complexity of this can be hidden from the user. All the user has to know is one thing: the location of the service, for example, the URL of the computer on which the solver resides. There is an additional option to select the final optimization solver that is to be used (e.g., Cbc [8], Ipopt [44], Cplex [4]), but an intelligent instance-solver matching service is used to infer a suitable default solver based on characteristics of the optimization instance. The service may even be embedded in a compute cloud, and with appropriate registry and discovery mechanisms (see, e.g., [36,52,1,51]) it is possible to automate even the selection of the solver location. In addition our approach allows the same options and result standards to be used for communication with other services providing access to different analysis tools, such as analyzers or simulation software.

A database connectivity framework such as ODBC [49] is a close analogue to Optimization Services. As part of the Optimization Services framework, the OSiL language communicates to the solver the optimization instance that needs to be solved, just as the SQL query language communicates (via ODBC) to the database management system (DBMS) which records need to be retrieved. In this analogy the DBMS takes the place of an optimization solver, with a different database driver that conforms to the ODBC standard for each DBMS. In the same fashion, the `OS` project also implements drivers for many different solvers. A large DBMS would allow numerous directives as to how the retrieval is to take place, where the database is located, etc. This is analogous to the various solver options, for which Optimization Services uses the common language OSoL. Finally, the retrieved record set plays the role of the result. Typically only the records matching the query are presented, but many systems can include information about the number of records searched, the time spent in the search, etc. In the Optimization Services framework, the results returned from the various solvers are standardized into the OSrL format.

Languages and communication protocols have also been studied in other domains, such as KQML [7], an instance format in the domain of Artificial Intelligence, and ACL [10], a competing format designed by the Foundation for Intelligent Physical Agents and endorsed by the IEEE Computer Society [11]. When designing standards in different domains, the critical design factors differ. For instance, large optimization problems must be able to handle sparse vectors efficiently. This is addressed in our schema designs and is very different from languages such as PSL [21] that are designed to describe a relatively small number of objects at a time. In this paper we try to explicitly and comprehensively lay out the material factors in the domain of optimization, more specifically, solver options and results, which has never been researched in the context of standardization.

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 Functionality

Based on observations and abstractions from many different examples of implementations and optimization models, 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. This narrow interpretation is taken from the book by Williams [53]. There often are different equivalent representations of a problem, but we would consider these separate instances. Unlike some medium-level languages such as Zinc and MiniZinc [6,29,31], Optimization Services is not concerned with automatic reformulations to allow different instances of a problem to be sent to different solvers. We freely acknowledge that not every solver can handle every possible representation of a problem. Our only concern is that the capabilities of the instance representation language be broad enough to allow *the user* to choose a representation suitable for their needs.
- **Option representation:** It is often necessary to pass options to instruct the solver *how* the problem should be solved; e.g., a starting solution (given perhaps by initial values for the decision variables in the problem), 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. This is highly solver-dependent. 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 solver output may contain the description of an efficient frontier, 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 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. Just as there may be multiple instances for a given problem, there may be multiple option sets associated with a particular instance. However, there is a unique result representation corresponding to each combination of instance and options. 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, options and results should be separate and completely unaware of how they are *communicated* between the client and server [14]. 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 a formal mapping rules between an XML-based file representation for solver options and the corresponding in-memory representation. In our in-memory representation we can take advantage of everything we know about the XML and parse directly into our data structure, because all elements and types are known at compile time. This is much more efficient than using the generic document object model (DOM—see [47]). 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. Examples include methods such as `setInitialVariableValues`, `getNumberOfVariables`, etc. Most of these methods are pure convenience methods, however. 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 `OS` project is an implementation of a framework for 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 [5]. From a user's perspective, the `OS` and NEOS servers act like solvers, or to be more specific, like *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 intermediary or *proxy*. An instance and options are sent to the remote solver proxy, which then further delegates the instance to the real solver at its back end.

Figure 2 is a generic architecture depicting the different layers, along with the various formats used for representing the inputs and outputs. Also indicated in Figure 2 are examples of the software components found in the `OS` implementation. In Figure 2 the proxy is the server communication agent (implemented in `OS` as the executable `OSServer`).

The modeling language environment (such as AMPL or GAMS) produces output in its own output format. The local solver server translates this output into OSiL and OSoL format and passes this to the client communication agent, which wraps everything in a SOAP envelope and transmits it to the communication agent on the server side. The server communication agent (i.e., the solver proxy) unwraps the SOAP envelope and passes the file(s) to the remote solver server. If necessary or so directed by the user, an instance may be added from the server's files or even retrieved from a third location. The remote solver server then translates the OSiL and OSoL representations into a format that the solver understands, and invokes the solver.

There could in fact be many 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 Optimization Services communication protocol. Results from the optimization solver are returned first to the remote solver proxy and then to the client. See [14] for a description of how this works in the Optimization Services framework.

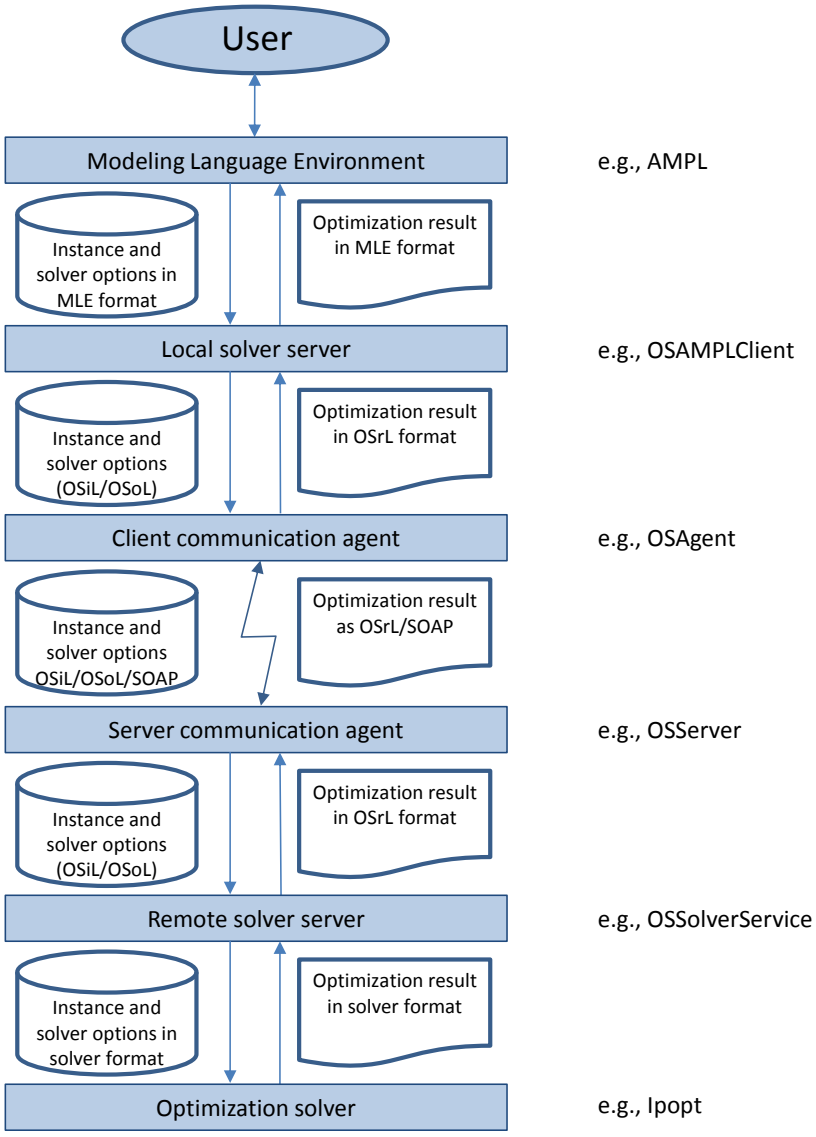

Fig. 2 The different software layers involved

In this distributed environment, it is useful 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. The solver proxy must be able to pass the latter options to the optimization solver. There are several types of options intercepted by the solver proxy. For example, there should be an option that tells the solver proxy which optimization solver to call. 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, and to control the environment in other ways (such as preparatory file movement and cleanup operations).

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 arises 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.

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. One cannot have a linear program without constraints or variables, but one 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, we have taken a minimalist but highly flexible approach in our proposed option and result descriptions. 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, a markup language that can be read by both human and machine. (For further information about XML see [41].)

One element of an XML file might look like this:

```
<minMemorySize unit="gigabyte">24.0</minMemorySize>
```

Text items enclosed in angle brackets ('<' and '>') are markup and usually come in pairs, a *start-tag* and an *end-tag*. (There are also empty-element tags such as `<paragraph/>`.)

Text contained between a start-tag and an end-tag is considered the *content* of the element, here the number string '24.0'. Attributes (such as `unit="gigabyte"`) included in a start-tag can be used to record *metadata*, that is, information *about* the content.

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), respectively. Other reasons why XML is the best way to impose a standard include:

- 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.
- There are easy and natural transcription rules to convert the content of the XML files into generic tree structures (such as the Document Object Model (DOM) [39]). There are open source libraries for creating these internal data structures (see, e.g, Xerces [38]).
- The availability of a schema, which is also an XML document, further simplifies the writing of parsers. In fact, there are open source libraries that take a schema as input and automatically parse and validate the XML document (see, e.g., [48]).
- XML is an open W3C standard that is nonproprietary, unencumbered by copyright, patent, trade secret, or any other intellectual property restriction.
- The design goals of XML emphasize simplicity, generality, and usability over the Internet — see `http://www.w3.org/TR/REC-xml/#sec-origin-goals`.
- XML-based Extensible Stylesheet Language (XSL) [42] 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.

Even though XML has a reputation of being wordy, the OSiL versions of the Netlib LP problems (`www.netlib.org`) are actually smaller than their MPS counterparts — especially for the larger problems. For these reasons, as well as consistency, XML schemas are the proper design choice for the representation of options and results.

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 [20].

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 [17, 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 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 software. 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 process.

These considerations led us to the following basic principles that aided us in our design of the OSoL schema. (The same principles also guided our design of result formats — see Section 4.1.)

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 solvers for integer, linear, nonlinear, stochastic, semi-definite programming etc., is hopeless. In addition, solvers may use different interpretions for the same option, such as the setting of a print level to control solver output. At present only initial values and initial basis information are mentioned specifically.
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 directly into generic data structures that are used to interface with the solver API, but should not have to interpret options. The option interface deals with the syntax only, the semantics are left to the solver.
4. Solvers support a wide variety of data types, so we should be in a position to support all of them — 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. By default the OSoL option file is independent of any optimization instance. However, sometimes it is desirable to link the option file with a particular instance, such as when initial variable values are given. This feature should be supported if possible.
9. Finally, the option file should be easy and quick to parse.

This section talks about the general design philosophy, not implementation specifics. We will turn to some of these in the next section.

3.2 Description of the OSoL schema

In the following we illustrate some schema elements using diagrams generated by the XML editor XMLSpy [2] (see, e.g., Figure 3). The small rectangle at the left of each figure gives the name of an XML tag and the large rectangle at the right describes the content. Solid lines and boxes are used for required content, dotted lines and boxes indicate optional content. Additionally, the diagrams show typing information in the shaded rectangles. For instance, in Figure 3 the top level element `osol` is of type `OSoL`.

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 process 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.

In order 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. As the figure shows, all six children of the root node `osol` are optional and could be omitted.

- 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 the 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.

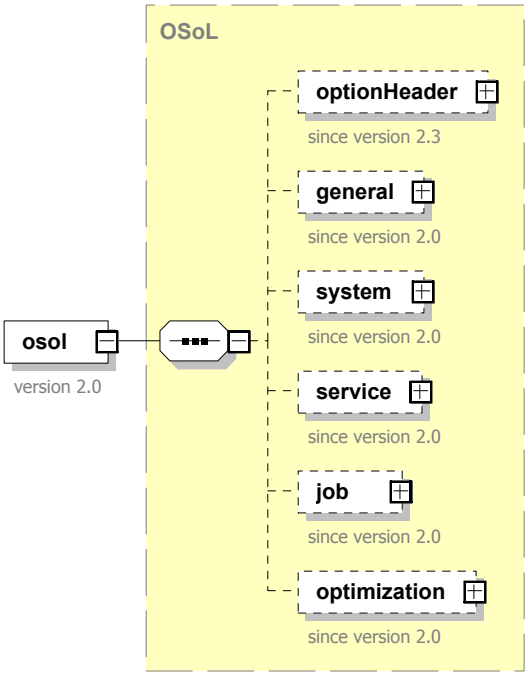

Fig. 3 The top-level OSoL elements (printed from Altova XMLSpy®)

- **service** options, which 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 and after the solver phase, respectively. 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 do not need to be passed to the solver. Unlike the Globus Toolkit [45] or OGSA [50], the OSoL schema is silent on *how* these functions should be implemented on the server. The options simply declare what needs to be done in order to complete the optimization task successfully. This includes the supporting environment (external) as well as the optimization job (internal). It is useful to think of a job as a solver process launched by a service in order to solve an optimization problem. A service can launch multiple jobs, and a job can involve multiple solver calls.
- `optimization` options, which are the options intended for 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.)

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).

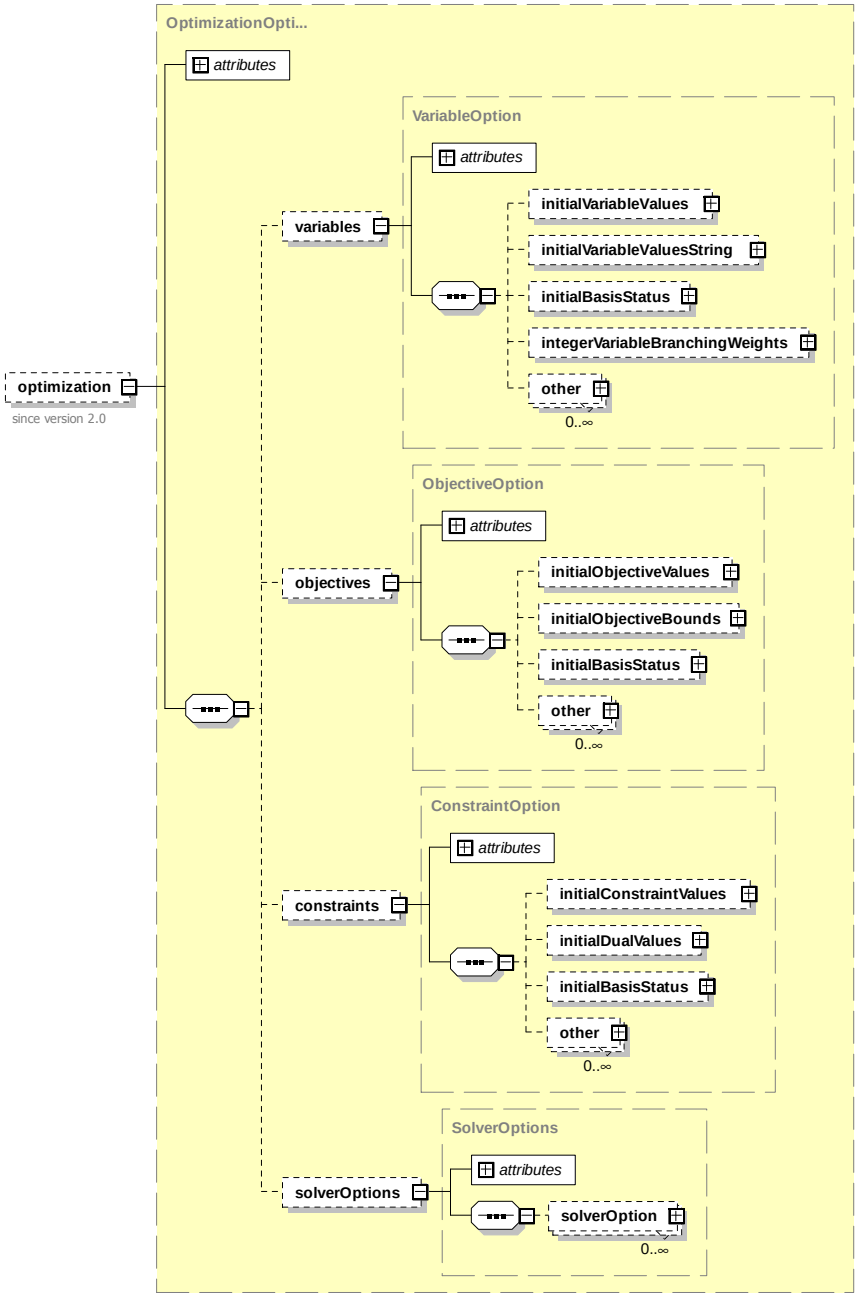

Fig. 4 The Optimization Element in OSoL(printed from Altova XMLSpy®)

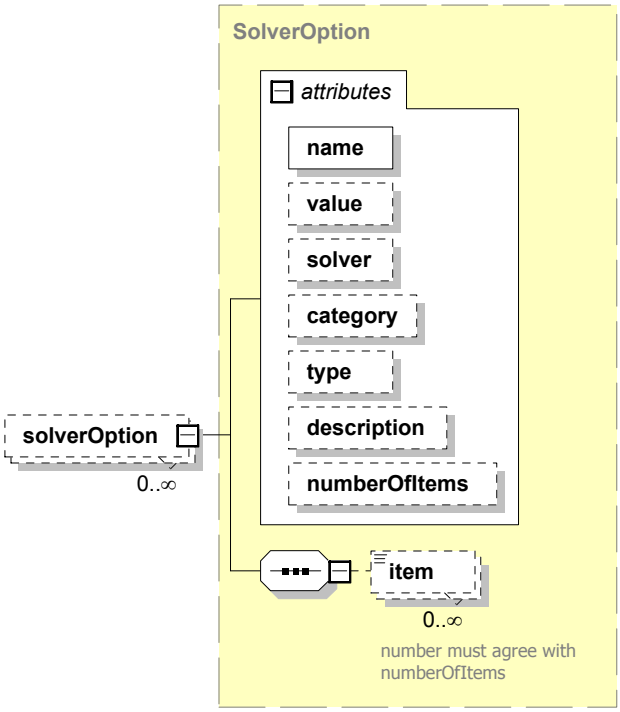

Fig. 5 Option Element (printed from Altova XMLSpy®)

- 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 `description` attribute can be used for annotation about the purpose of this option.
- 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 value. 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 list of supported solvers is implementation dependent and subject to registry and discovery. (A service might, for instance, publish a list of solvers that can be searched through the Internet.)
- The `category` attribute allows additional information to be passed to the solver regarding the option. In LINDO [24], for instance, it is possible to set attributes affecting either an entire interactive session (or "environment" in the parliance of LINDO), or just one individual model solved during such a session. Both use cases are illustrated in the example below.

```
<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.

Figure 4 shows the flexibility and extensibility of the OSoL schema. Many lists, such as the options for variables, objectives and constraints, as well as the top-level options in Figure 3 end with a sequence of <other> elements that can be used to allow implementation-specific options to be added. For instance, because the user can directly influence the amount, and to some extent the form, of the output, we have found the following declaration to be very useful:

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

This option was added to our implementation of the remote communication agent. It instructs the solver server running on a remote system to capture all 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.)

(The attribute `numberOfOtherOptions` and similar `numberOf...` atributes are used to improve the efficiency of the parser. Knowing the size of arrays beforehand makes it possible to allocate sufficient memory in a single operation as opposed to doing it one object at a time.)

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, as explained on page 15.

3.3 Internal representation — the OSOption class and its members

The OSoL schema defines an XML vocabulary to describe optimization options. The options instance may persist on a file system, or it 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. There are multiple tools that can generate the in-memory representations automatically, but sometimes ambiguity

```
<?xml version="1.0" encoding="UTF-8"?>
<osol xmlns="os.optimizationservices.org"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="os.optimizationservices.org
          http://www.optimizationservices.org/schemas/2.0/OSoL.xsd">
    <optionHeader>
        <name>Example.osol</name>
        <fileCreator>
          Horand Gassmann, Jun Ma and Kipp Martin
        </fileCreator>
        <licence>
          This file is licensed under the Eclipse Public License.
          Please see the accompanying LICENSE file in root directory for terms.
        </licence>
    </optionHeader>
    <general>
        <serviceURI>
            http://74.94.100.129:8080/OSServer/services/OSSolverService
        </serviceURI>
    </general>
    <system>
        <minMemorySize unit="gigabyte">24.0</minMemorySize>
    </system>
    <optimization>
        <variables numberOfOtherVariableOptions="0">
            <initialVariableValues numberOfVar="2">
                <var idx="0" value="1.5"/>
                <var idx="1" value="1.5"/>
            </initialVariableValues>
        </variables>
        <solverOptions numberOfSolverOptions="4">
            <solverOption name="max_iter" solver="ipopt" type="integer" value="2000"/>
            <solverOption name="print_level" solver="ipopt" type="integer" value="5"/>
            <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>
```

Fig. 6 A sample OSoL file

arises in binding certain elements from the XML schema to the in-memory objects. Our implementation is intended to give a guideline on how schemas should be bound into in-memory objects explicitly in a more unambiguous way.

The `OSOption` 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 can be 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 by the solver developer 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 [15,19] and are modified here to illustrate their use with OSoL (see Figure 7).

- Each *complexType* (that is, an element in the XML schema that may contain other elements or attributes) 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`.

3.4 Solver Communication

Eventually a framework such as Optimization Services 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, as they were encountered in the `OS` implementation.

3.4.1 Ipopt

Communication with Ipopt is very flexible, because the Ipopt API contains three methods, named `SetIntegerValue`, `SetNumericValue` and `SetStringValue`, 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

```
<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, e.g., the call

```
bool success = setIntegerValue("NewRule", 5);
```

gives instant feedback as to whether the 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 pass that string to Cbc for parsing. 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 name, and this

Schema complexType **In-memory class**

```
<xs:complexType name="SolverOptions">  <-------------------------------------->  class SolverOptions{
                                                                                   public:
                                                                                     SolverOptions();
  <xs:sequence>
    <xs:element name="solverOption" type="SolverOption"
            minOccurs="0" maxOccurs="unbounded"/>  <--------------------------->      SolverOption **solverOption;
  </xs:sequence>
  <xs:attribute name="numberOfSolverOptions" type="xs:nonNegativeInteger"
            use="required"/>  <----------------------------------------------->      int numberOfSolverOptions;
</xs:complexType>  <------------------------------------------------------------>  }; //SolverOptions


<xs:complexType name="SolverOption">  <-------------------------------------->  class SolverOption{
                                                                                   public:
                                                                                     SolverOption();
  <xs:sequence>
    <xs:element name="item" type="xs:string"
            minOccurs="0" maxOccurs="unbounded">  <--------------------------->      std::string *item;
    </xs:element>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required"/>  <-------------->     std::string name;
  <xs:attribute name="value" type="xs:string" use="optional"/>  <------------->     std::string value;
  <xs:attribute name="solver" type="xs:string" use="optional"/>  <------------>     std::string solver;
  <xs:attribute name="category" type="xs:string" use="optional"/>  <---------->     std::string category;
  <xs:attribute name="type" type="xs:string" use="optional"/>  <-------------->     std::string type;
  <xs:attribute name="description" type="xs:string" use="optional"/>  <------->     std::string description;
  <xs:attribute name="numberOfItems"
            type="xs:nonNegativeInteger" use="optional" default="0"/>  <------>     int numberOfItems;
</xs:complexType>                                                                 }; //SolverOption
```

OSoL elements **In-memory objects**

```
<osol ...>                          OSOption* osoption = new OSOption();
<optimization>                      osoption->optimization = new OptimizationOption();
<solverOptions                      osoption->optimization->solverOptions = new SolverOptions();
    numberOfSolverOptions="2">      osoption->optimization->solverOptions->numberOfSolverOptions = 2;
                                    osoption->optimization->solverOptions->solverOption = new SolverOption*[2];
  <solverOption                     osoption->optimization->solverOptions->solverOption[0] = new SolverOption();
    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";
  <solverOption                     osoption->optimization->solverOptions->solverOption[1] = new SolverOption();
    name="OsiHintTry"               osoption->optimization->solverOptions->solverOption[1]->name = "OsiHintTry";
    solver="osi"                    osoption->optimization->solverOptions->solverOption[1]->solver = "osi";
    type="OsiHintStrength"/>        osoption->optimization->solverOptions->solverOption[1]->type = "OsiHintStrength";
</solverOptions>
</optimization>
</osol>
```

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

behavior cannot be controlled from the calling program. This is not very attractive, particularly if a simple typing error (e.g., `-maxNd` instead of `-maxN`) 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 [9], Dylp [22], Symphony [33], Glpk [28]) and commercial (e.g., Cplex [4]) linear and integer programming solvers. Unlike the other interfaces described above, 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 announced a major overhaul to address these issues [35].

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` system 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` [19, section 5.1] which looks to AMPL like an ordinary solver and can be selected using the AMPL command

```
option solver OSAmplClient;
```

AMPL communicates with the `OSAmplClient` in two ways: the AMPL command

```
option OSAmplClient_options "...";
```

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.

On a subsequent

```
solve;
```

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 `OSAgent` or the `OSSolverService` (see Figure 2). 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 [43], and other interfaces are forthcoming.

A new COIN-OR project, CMPL [37], is one of the first mathematical programming languages to output OSiL and OSoL files directly. It also calls the `OSSolverService` 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

At least since the late 1950s linear programming solvers were able to send their solutions to a printer, using a set format for the output they created. However, not even the result format of the venerable MPS format (see, e.g., [30]) was adopted as a standard. 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 communicating problem instances. Thus another part of our project has been to design OSrL, an XML-based protocol that allows the efficient representation of solutions to large-scale optimization problems of all kinds. This is complicated further by the fact that both the solver developer and the user can influence the appearance of the solver output.

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 nonlinear programs since in this case 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, as in the case of 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 keeping with the previous section we consciously designed OSrL to have the same top level structure as OSoL. (See Figure 8.) This includes the following main elements:

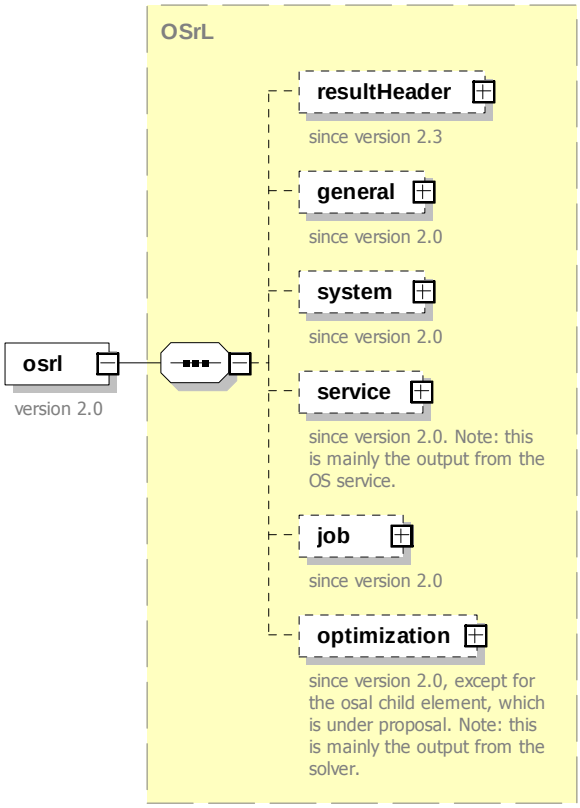

Fig. 8 The top level elements in OSrL (printed from Altova XMLSpy®)

- a **header** section that gives the same information as the OSoL header: 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, which 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 geared towards simplicity while maximizing flexibility in reporting optimization results.

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 omit an example in order to preserve space.

4.4 Solver Communication

The local solver server (see Figure 2) must retrieve from the remote system all the results and output that the user requested and that the solver chooses to report. Typically these include the optimal value of the objective function along with the values assigned to the decision variables in the optimal decision variables. Other items that may be returned (always or on demand) include dual variable values, range information, basis status, to name a few. 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.

4.5 Modeling Language Communication

The contents of the OSrL file may have to be communicated back to the algebraic modeling 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 `OS` system 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

```
solve;
```

as in section 3.5, AMPL must do several things: it instantiates the current problem from the model and data specified by the user; writes the instance into a .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 — 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.

```
<?xml version="1.0" encoding="UTF-8"?><?xml-stylesheet type="text/xsl"
    href="http://www.coin-or.org/OS/stylesheets/OSrL.xslt"?>
<osrl xmlns="os.optimizationservices.org"   ... >
<general>
    <generalStatus type="normal"/>         <-- solver terminated without error
    <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">   <--- solver-generated output starts here
        ...
        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>
    </otherResults>
</job>
<optimization numberOfSolutions="1" numberOfVariables="33" numberOfConstraints="16">
    <solution>
        <status type="feasible" description="node limit reached"/>
        <variables >
            <values numberOfVar="33">
                <var idx="0">1</var>      <--   x[0] = 1
                ...
            </values>
        </variables>
        <objectives >
            <values numberOfObj="1">
                <obj idx="-1">3347</obj>
            </values>
        </objectives>
    </solution>
</optimization>
</osrl>
```

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

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

We have presented two XML schemas for describing solver options and solver results in a standard way. The formats can communicate with several modeling languages as well as a growing number of solvers. Flexibility, extensibility, scalability, portability, conciseness and generality have all been emphasized in the design. We have shown that our approach is practical and is especially useful in the context of remote computing. We implemented our proof-of-concept system `OS` as a COIN-OR project. The system can be called from the modeling systems AMPL, GAMS and CMPL. It links to several COIN-OR solvers (Clp, Cbc, Ipopt, Bonmin, Couenne, SYMPHONY, DyLP) and has been successfully hooked to Glpk, Lindo, Cplex, Gurobi and Mosek. The system supports both local and remote access. Several remote servers exist although none of them is particularly stable at this point. We maintain a list of valid URLs in the OS User's Manual (`https://projects.coin-or.org/svn/OS/trunk/OS/doc/osUsersManual.pdf`).

Source and executables are available from the website `https://projects.coin-or.org/OS`. The website also contains links to further resources, such as descriptions of the schemas alluded to in this paper, an overview of the `OS` project and related items.

Work is ongoing to extend the system to incorporate other classes of problems (second-order cone programming, matrix programming, stochastic programming). We are also studying the use of the same or similar mechanisms to communicate not just with optimization solvers, but with other analysis tools such as analyzers or simulation software.

Acknowledgments

The authors are grateful to the associate editor and three anonymous referees for their handling of the paper. Their careful reading of and detailed comments on a previous version have vastly improved the presentation. Any remaining shortcomings are the sole responsibility of the authors.

References

1. Eyhab Al-Masri and Qusay H. Mahmoud. Investigating web services on the world wide web. In *WWW '08: Proceedings of the 17th international conference on the World Wide Web*, pages 795–804, 2008.
2. Altova. XMLSpy XML Editor. http://www.altova.com/xmlspy.html, accessed 2 May 2014.
3. Anthony Brooke, David Kendrick, Alexander Meeraus, and Ramesh Raman. *GAMS: A User's Guide*. GAMS Development Corporation, Washington, DC, USA, 2011. also available electronically as http://www.gams.com/dd/docs/bigdocs/GAMSUsersGuide.pdf.

4. Cplex. IBM ILOG CPLEX Optimizer. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/, accessed 14 June 2013.
5. J. Czyzyk, M.P. Mesnier, and J.J. Moré. The NEOS server. *Computational Science & Engineering, IEEE*, 5:68–75, 1998.
6. Maria Garcia de la Banda, Kim Marriott, Reza Rafeh, and Mark Wallace. The modelling language Zinc. In *Principles and Practice of Constraint Programming - CP 2006*, volume 4204 of *Lecture Notes in Computer Science Volume 4204*, pages 700–705. Springer Verlag, 2006.
7. Tim Finin, Don McKay, Rich Fritzson, and Robin McEntire. KQML: An information and knowledge exchange protocol. In Kazuhiro Fuchi and Toshio Yokoi (Ed.), editors, *Knowledge Building and Knowledge Sharing*. Ohmsha and IOS Press, 1994.
8. John Forrest. Cbc, a COIN-OR project. https://projects.coin-or.org/Cbc, accessed 14 June 2013.
9. John Forrest. Clp project wiki. https://projects.coin-or.org/Clp, accessed 14 June 2013.
10. Foundation for Intelligent Physical Agents. FIPA ACL Message Structure Specification. http://www.fipa.org/specs/fipa00061/SC00061G.html, 2002.
11. Foundation for Intelligent Physical Agents. Welcome to FIPA! http://www.fipa.org/, 2014.
12. Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Brooks/Cole Publishing Company/Cengage Learning, second edition, 2002.
13. Robert Fourer, Leo Lopes, and Kipp Martin. LPFML: A W3C XML schema for linear and integer programming. *INFORMS Journal on Computing*, 17:139–158, 2005.
14. Robert Fourer, Jun Ma, and Kipp Martin. Optimization Services: A framework for distributed optimization. *Operations Research*, 58:1624–1636, 2010.
15. Robert Fourer, Jun Ma, and Kipp Martin. OSiL: An instance language for optimization. *Computational Optimization and Applications*, 45(1):181–203, 2010.
16. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
17. GAMS Development Corporation. Cplex solver manual. http://www.gams.com/dd/docs/solvers/cplex.pdf, accessed 14 June 2013.
18. Horand Gassmann, Jun Ma, and Kipp Martin. Os project wiki. https://projects.coin-or.org/OS, accessed 4 May 2014.
19. Horand Gassmann, Jun Ma, Kipp Martin, and Wayne Sheng. Optimization Services 2.6 User's Manual. Technical report, COIN-OR, 2013. available electronically at https://projects.coin-or.org/svn/OS/releases/2.6.0/OS/doc/osUsersManual.pdf.
20. David M. Gay. Hooking Your Solver to AMPL. Technical Report 97-4-06, Computing Sciences Research Center, Bell Laboratories, Murray Hill, NJ 07974, 1997. available electronically at http://www.ampl.com/REFS/hooking2.pdf.
21. Michael Grüninger and Christopher Menzel. The Process Specification Language (PSL): Theory and applications. *AI Magazine*, 24:63–74, 2003.
22. Lou Hafer. DyLP project wiki. https://projects.coin-or.org/DyLP, accessed 14 June 2013.
23. Bjarni Kristjánsson. http://www.maximalsoftware.com/mpl/, accessed 14 June 2013.
24. LINDO Systems Inc. http://www.lindo.com/, accessed 14 June 2013.
25. LINDO Systems Inc. An overview of LINGO. http://www.lindo.com/index.php?option=com_content&view=article&id=2&Itemid=10, accessed 29 November 2011.
26. lp_solve development group. MPS file format. http://lpsolve.sourceforge.net/5.1/mps-format.htm, accessed 14 June 2013.
27. Jun Ma. *Optimization Services (OS)*. PhD thesis, Industrial Engineering and Management Sciences, Northwestern University, 2005.
28. Andrew Makhorin. GLPK (GNU Linear Programming Kit). http://www.gnu.org/s/glpk/,accessed 14 June 2013, October 2008.
29. Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, September 2008.
30. Bruce A. Murtagh. *Advanced Linear Programming*. McGraw-Hill, 1981.
31. Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming CP 2007*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007.
32. Open Grid Forum. https://www.ogf.org/ogf/doku.php/about, accessed 23/May/2014.
33. Theodore K. Ralphs. SYMPHONY project wiki. https://projects.coin-or.org/SYMPHONY, accessed 14 June 2013.
34. Dirk Riehle. *Framework Design: A Role Modeling Approach*. PhD thesis, ETH Zürich, Switzerland, 2000.
35. Matthew Saltzman and Lou Hafer. Personal communication, 15 November 2011.

36. SAP AG. Discovering services in the services registry. http://help.sap.com/saphelp_nwpi711/helpdata/en/2e/8526937af346a0bc446905ea964ceb/content.htm, 2014.
37. Mike Steglich. CMPL - COIN Mathematical Programming Language. https://projects.coin-or.org/Cmpl, accessed 14 June 2013.
38. The Apache Software Foundation. The Apache Xerces Project. http://xerces.apache.org/, accessed 14 June 2013, 2011.
39. The World Wide Web Consortium (W3C). Document Object Model (DOM). http://www.w3.org/DOM/, accessed 14 June 2013, January 2005.
40. The World Wide Web Consortium (W3C). SOAP Version 1.2 Part 1. http://www.w3.org/TR/soap12-part1/, accessed 14 June 2013, April 2007.
41. The World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0 (Fifth edition). http://www.w3.org/TR/REC-xml, accessed 14 June 2014, November 2008.
42. The World Wide Web Consortium (W3C). The Extensible Stylesheet Language Family (XSL). http://www.w3.org/Style/XSL/, accessed 14 June 2013, May 2011.
43. Stefan Vigerske. GAMSlinks project wiki. https://projects.coin-or.org/GAMSlinks, accessed 14 June 2013.
44. Andreas Waechter. Ipopt project wiki. https://projects.coin-or.org/Ipopt, accessed 14 June 2013.
45. Von Welch (ed.). *Globus Toolkit Version 4 Grid Security Infrastructure: A Standards Perspective*. The Globus Alliance, September 2005.
46. Wikipedia. Design pattern. http://en.wikipedia.org/wiki/Design_pattern, accessed 23/May/2014.
47. Wikipedia. Document object model. http://en.wikipedia.org/wiki/Document_Object_Model.
48. Wikipedia. Java Architecture for XML Binding. http://en.wikipedia.org/wiki/JAXB.
49. Wikipedia. Open database connectivity. http://en.wikipedia.org/wiki/Open_Database_Connectivity.
50. Wikipedia. Open grid services architecture. http://en.wikipedia.org/wiki/Open_Grid_Services_Architecture, accessed 23/May/2014.
51. Wikipedia. Universal description discovery and integration. http://en.wikipedia.org/wiki/Universal_Description_Discovery_and_Integration.
52. Wikipedia. Web services discovery. http://en.wikipedia.org/wiki/Web_Services_Discovery.
53. H. Paul Williams. *Model Building in Mathematical Programming*. John Wiley & Sons, 5th edition edition, 2013.