NORTHWESTERN UNIVERSITY

Optimization Services (OS)

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Industrial Engineering and Management Sciences

By

Jun Ma

EVANSTON, ILLINOIS

June, 2005

ABSTRACT

Optimization Services (OS)

Jun Ma

This doctoral thesis presents a general optimization system design introduced under our new concept of Optimization Services (OS) along with its Optimization Services Protocol (OSP). Optimization Services is intended to be a unified *framework* for the next generation distributed optimization systems, mainly optimization over the Internet. Thus Optimization Services can be regarded as the Operations Research Internet. The corresponding Optimization Services Protocol is intended to be a set of industrial *standards*.

Optimization Services framework is an XML-based, service-oriented, optimization-centered, distributed and decentralized architecture. The Optimization Services Protocol is an application level networking protocol that includes over 20 sub-protocols of Optimization Services x Languages (OSxL). Optimization within a local environment is treated as a special case; issues within a local environment are mostly addressed under the distributed case.

Although large-scale optimization has been under research for over half a century now, the challenge of making it useful in practice has continued to the present day. Initially, the greatest difficulties were posed by solution computation and model building, but the primary impediment to broader use of optimization models and methods today is now more of *communication*. Currently there exists an abundance of optimization solvers and other supporting tools, various formats to represent optimization problems, and heterogeneous mechanisms to communicate with optimization components. Moreover different optimization components are implemented in different programming and modeling languages and located on different platforms locally or over the network. Even if a prospective user is *not* puzzled by such a plethora of combinations, the trouble of obtaining, installing, and configuring the software does not justify the benefits from using it.

Through standardization of representation, communication, discovery and registration, the framework provides an open infrastructure for all optimization system components including modeling language environments, servers, registries, communication agents, interfaces, analyzers, solvers and simulations. The goal is that all the algorithmic codes are implemented as services under this framework and customers use these computational services similar to daily utility services. Optimization Services also facilitates a healthier development environment for research and development in the general area of Operations Research and Management Sciences.

ACKNOWLEDGEMENT

I thank Professor Robert Fourer, my advisor from the Industrial Engineering and Management Sciences department at Northwestern University, for bringing me into this wonderful and significant project and providing the vital vision and direction. I thank Professor Kipp Martin, my co-advisor from Graduate School of Business at University of Chicago for his enthusiasm and constant support of Optimization Services. I thank the whole Optimization Technology Center team at Argonne National Lab for bring NEOS into the world. I thank Tom Tirpak, my manager at Motorola’s Advanced Technology Center, for providing the perfect environment, opportunity and motivation. Tom and my fellow researchers at the Motorola lab have sparked my interests in a lot of other fields, notably Virtual Prototyping in Electrical Engineering and Machine Learning. I thank my other committee members, Professor John Birge from Graduate School of Business of University of Chicago, Professor Wei Chen from the Mechanical Engineering Department at Northwestern University and Professor Sanjay Methrotra from the Industrial Engineering and Management Sciences Department at Northwestern University for sharing their precious time and providing valuable suggestions. Professor Methrotra and his students Wayne Sheng and Michael Chen make me feel like doing research in a big and warm family.

I also want to thank my wife, Haiyan Xu– staying up the nights with me, just not to let me feel working there alone and sleepy. I joked with her that in “OSxL”, one of the 4-letter acronyms that we coined in this project, the letter “x” is reserved for her. By now, as a non-operations researcher, she is all too familiar with my babbles of the acronyms. I thank my parents Shiming Ma and Hao Li. Without their support, I would not have completed my Ph.D. degree. This thesis is also for my little daughter Angela, who brings me happiness and enlightenment throughout my doctoral work.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

This doctoral thesis presents a general optimization system design introduced under our new concept of Optimization Services (OS) along with its Optimization Services Protocol (OSP). Optimization Services is a pioneering effort in building a unified *framework* for the next generation of distributed optimization systems, mainly involving optimization over the Internet. The phrase "next generation" emphasizes the fact that Optimization Services is a state-of-the-art design and is *not* adapted from any existing system. Thus Optimization Services can be regarded as a new Operations Research Internet. The corresponding Optimization Services Protocol is intended to be a set of industrial *standards*. We are also developing our own *system* according to this standard OS framework (see http://www.optimizationservices.org [92] or http://www.optimizationservices.net [93]). Optimization Services is the first systematic approach to addressing and solving general issues in optimization system and software development. Optimization Services Protocol is the first approach to standardizing all major instance representations and communications in distributed optimization systems.

Technically, the newly introduced Optimization Services framework is an XML-based, service-oriented, optimization centered, distributed and decentralized architecture. The Optimization Services Protocol is an application level networking protocol that includes over 20 specifications or sub-protocols of Optimization Services x Languages (OSxL[1]). Optimization within a local environment is treated as a special case. Therefore issues that exist within a local environment are mostly addressed under the distributed case.

Although large-scale optimization has been a subject of research for over half a century now, the challenge of making it useful in practice has continued to the present day. Initially, the greatest difficulties were posed by solution computation and model building, but the primary impediment to broader use of optimization models and methods today is now more of *communication*. Currently there exists an abundance of optimization solvers, various formats to represent optimization problems, and heterogeneous mechanisms to communicate with optimization components. There are also plentiful research initiatives in developing supporting

[1] The third small letter "x" in the acronym can be replaced with any of the other 25 letters to represent a concrete sub-protocol. For example, OSiL stands for Optimization Services instance Language, which is an XML language for representing any optimization instance including general nonlinear programming. OSxL is used in this thesis to generically mean all such concrete languages or sub-protocols specified in the Optimization Services Protocol (OSP).

tools to analyze and benchmark optimization problems, and solvers. Moreover different optimization components are implemented in different programming and modeling languages and located on different platforms locally or all over the network. Even if a prospective user is *not* to be puzzled by such a plethora of combinations, the trouble of obtaining, installing, and configuring the Operations Research (OR) software does not justify the benefits from using it. A wider level of collaboration to move toward some agreements is an imminent necessity. The research in Optimization Services originated with the motivation to start a wider level of cooperation to move toward a final standardization and facilitate a healthier development environment for research and development in the general area of Operations Research and Management Sciences.

The research in Optimization Services is technologically timely. In the areas of Computer Science and Electrical Engineering, distributed technologies such as XML and Web services are growing rapidly in importance in today's computing environment and are already widely accepted as industrial standards. It is our vision that by combining Operations Research and modern computing technologies, Optimization Services will make a wider audience aware of, and benefit from, an increasing amount of OR software that is implemented increasingly well.

The advent of Optimization Services is also timely with the current efforts undertaken by the Operations Research and Management Sciences community to market the area as the Science of Better, to promote practice and to create demand. Through standardization of modeling representation, communication, discovery and registration, the framework provides an open infrastructure for all optimization system components including modeling language environments, servers, registries, communication agents, interfaces, analyzers, solvers and simulation engines. The goal is that all the algorithmic codes will be implemented as services under this framework and customers use these computational services similar to daily utility services (therefore the name Optimization Services). Special knowledge of optimization algorithms, problem types, and solver options required of users should be minimized. A supply chain modeler, for example, should just concentrate on writing a good supply chain model. Everything else that involves detecting the problem structure, finding the right solver, invoking the software, solving the instance, providing the computing resources and presenting the solution should be automatically taken care of by Optimization Services. It is the combination of distributed system embedded intelligence, smooth coordination of all the tasks, and effortless human involvement in the whole seamless integration process that makes Optimization Services unique and significant.

A "service" is intended to serve customers. For Optimization Services, there are mainly three categories of customers:

- **Application developers** create and build system components such as modeling language environments and solvers as part of a larger optimization system. The components together take care of such generic functions as managing data, solving optimization problems, and presenting solutions in a graphical interface. Optimization Services provides a set of specific guidelines for application developers to implement their part of an optimization system. The "state-of-the-art" design and the resulted standardization extensively and drastically reduce the development time and effort for the developers while significantly improving software and system design qualities. Application providers are the major intended audience of this thesis.
- **Modelers** work in a modeling language environment or in an environment provided by some graphical user interface to build optimization models and get acceptable solutions. From the perspective of Optimization Services, modelers are the immediate customers and beneficiaries; they can now solely concentrate on building more robust models by letting Optimization Services take care of the interfacing and solution parts. Modelers should not read this thesis in detail, but they should be aware of what Optimization Services is and how Optimization Services can benefit them.
- **Users** run application packages that perform optimization at some stage through the optimization system. Users are usually the ultimate customers of any optimization system. With Optimization Services, users may not even realize that they are running optimization system components such as solvers, although they are often aware of optimization goals, such as minimizing costs or maximizing profits. Although not directly interfacing with Optimization Services, users will experience higher quality performance and results from of the application packages that they are using. Solutions are more likely and more quickly to be found as the application developers now have a much wider range of optimization resources to reply upon and modelers can concentrate on building better optimization models that more accurately reflect the users' business problems.

A side effect of Optimization Services is that although the OS framework is intended to be an infrastructure for the area of OR/MS, the design concept and philosophy is general enough to be learned and adopted by designers of distributed systems and architectures in many other domains.

CHAPTER 1 INTRODUCTION TO OPTIMIZATION SERVICES

This chapter gives a general non-technical description of Optimization Services (OS) and the corresponding Optimization Services Protocols (OSP). Optimization Services is a unified *framework* for the next generation distributed optimization systems, mainly optimization over the Internet. The corresponding Optimization Services Protocol is intended to be a set of industrial *standards*. The phrase “next generation” emphasizes the fact that Optimization Services is a state-of-the-art design and is *not* adapted from any existing system. It also suggests that the OS framework fits well in the general picture of the “Future of Computing.”

1.1 Future of Computing – A General Background

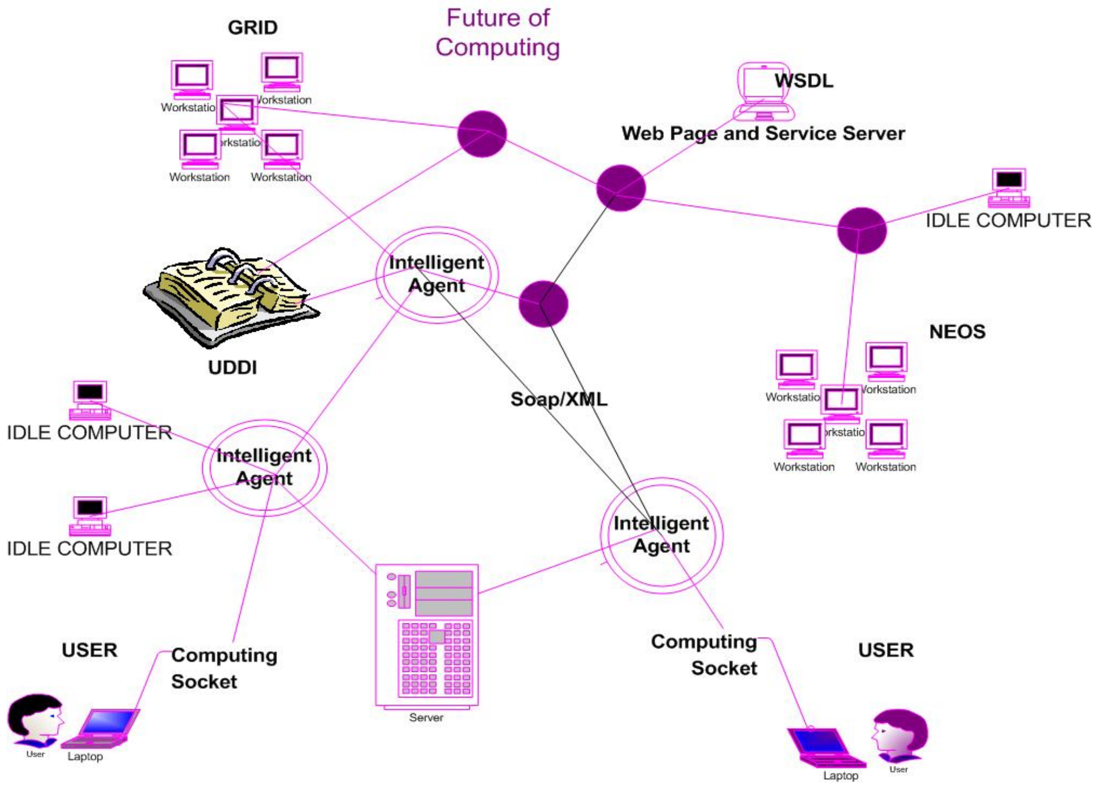


Figure 1-1: Future of computing.

Figure 1-1 depicts a future computing framework in which semantic Web services and software agents interact with each other. A “consumer” plugs his computer into a so-called “computing socket” (or a wireless access point), which is presumably next to the electrical and phone outlets. Computing then is solely viewed as part of the daily utilities that are ubiquitously available (thus the coined name Optimization Services).

The corresponding utility or power company is the consumer's application service provider that rents computing power and resources and charges a monthly bill. As soon as the consumer starts his computer, a network connection is instantly established. Software agents will help find where the consumer's requested services are, *automatically*, based on the request time, the computing socket location, and the consumer's own needs. The software agents are themselves software services. The consumer is not aware of the existence of these agents. "Computing power companies" keep registries of these agents and contact them on behalf of the consumer. The consumer does not need to know which computer or grid of computers his requested services are finally run, just as he does not need to know where his electric power is generated or where the water flows in from.

To locate services, software agents usually coordinate with each other and with registries. Some registries are general ones that keep information of all kinds of Web services, such as Universal Description, Discovery and Integration (UDDI, see Chapter 4). Others are specialized ones like the Optimization Services Registry (see Chapter 8) that only serves registration and discovery of Optimization software. Facilities such as Condor [38][72] can help in finding computers to provide idle computing power.

Admittedly most of these tasks could be achieved by an arrangement of customized software tools using existing technologies, but that would be an enormous human effort. Think of the early Yahoo search engine for Web pages with human categorization.

Listed below are the major components used to achieve the tasks described in the above scenario. Some are mature enough to be commercialized, whereas others are still in different research phases:

- Peer to Peer (P2P) [87]
- Software Agents [1][39]
- Ontology and the Semantic Web [18]
- Grid Computing [41]
- Embedded Web services [17]

Although it is true that many of the technologies already exist, it is the combination of distributed system embedded intelligence, smooth coordination of all the tasks, and effortless human involvement in the whole integration process that makes these scenarios significant. In this case, think of the Google search engine[14], with its automated web crawlers and state-of-the-art file storage design.

The above-mentioned lack of automation and heavy human involvement is true of the current status of Operations Research (OR) software and system development. A lot of time is spent in solving issues such as programming language compatibility, format compatibility, interface compatibility, platform compatibility, and system compatibility. Although most of the OR related tasks can be done by a combination of manual labor and custom tools using existing OR technologies, the OR community needs a combination of software and systems with embedded intelligence, seamless integration and no human involvement.

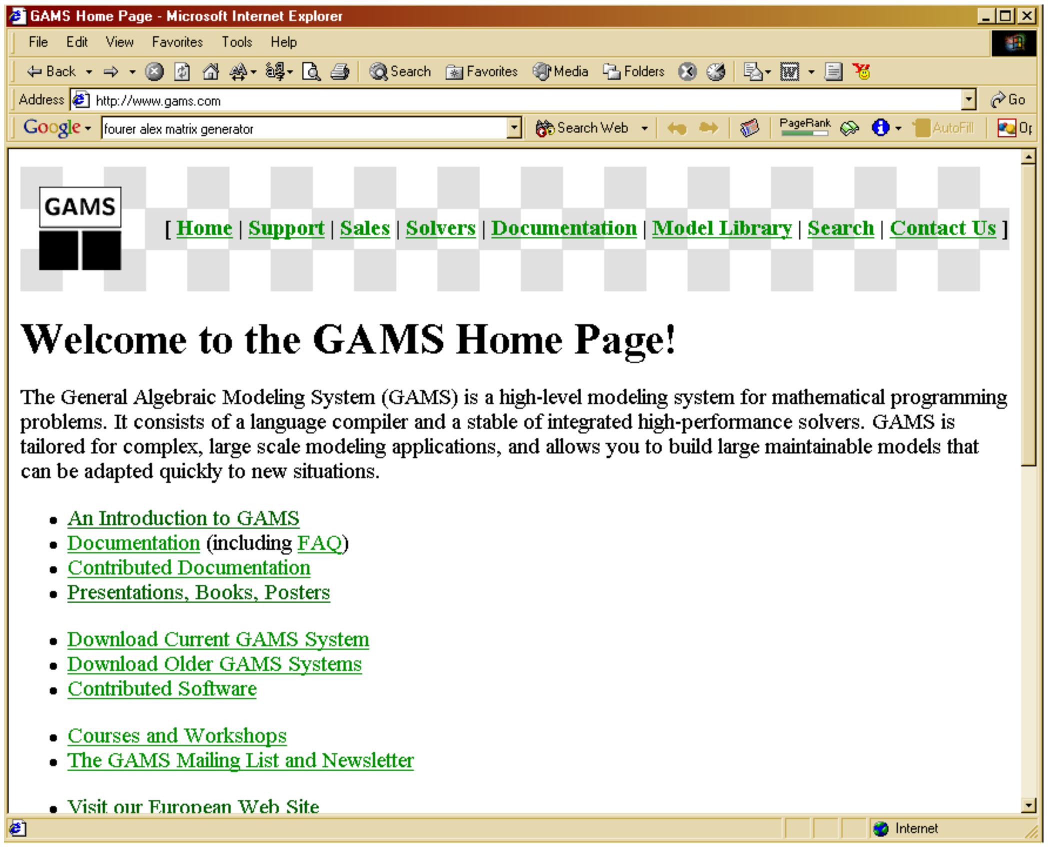
Our research in Optimization Services is also motivated by the fact that although large-scale optimization has been a subject of research for over half a century now, the challenge of making it useful in practice is still a problem. Initially the greatest difficulties were posed by solution computation and model building, but the primary impediment to broader use of optimization models and methods today is one of communication.

Currently there are many optimization solvers, various formats to represent optimization problems, and heterogeneous mechanisms to communicate with optimization components. There are also numerous research initiatives in developing supporting tools to analyze and benchmark optimization problems and solvers. Moreover, different optimization components are implemented in different programming and modeling languages and located on different platforms locally or all over the network. Even if a prospective user is not puzzled by such a plethora of combinations, the trouble of obtaining, installing and configuring the OR software does not justify the benefits from using it.

1.2 Optimization Services (OS)

In the early history of solving the mathematical programs, the translation of an optimization model to a format required by a linear program solver involved intensive human labor and human labor alone. The first major attempt to provide an environment to help the solution of a mathematical program was the matrix generator. A matrix generator is a computer code that creates input in the form of coefficient matrices for a linear program solver. The task of translation from the modeler's form to the algorithmic code's form is thus divided and shared between human and computer. The task is shared because what the matrix generator takes is not a modeler's form. A modeler still has to convert a symbolic model to a special instance representation and then the matrix generator code translates this representation to the format that the solver desires. But the dominance of matrix generator continued to the early 1980's.

Then there was a big breakthrough with the development of modeling languages (the first major one being GAMS, see Figure 1-2), which entirely shifted the human labor of translation to computer. In 1983, Robert Fourer articulated a contrast between the modeler's view and the algorithm's view. He described new design considerations that would combine strength of general, high level languages with special-purpose languages [45]. Modeling languages introduced two key ideas: separation of the data from the model and separation of modeling language from the solver. They addressed the issues of verifiability, modifiability, documentability, independence, simplicity, and other special drawbacks of matrix generators. As modeling languages began to be packaged with other auxiliary tools that assist in model construction, people started to call them modeling systems.


GAMS Home Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media Folders
Address http://www.gams.com
Go
Google fourer alex matrix generator
Search Web PageRank AutoFill
GAMS
[Home | Support | Sales | Solvers | Documentation | Model Library | Search | Contact Us]
Internet

Welcome to the GAMS Home Page!

The General Algebraic Modeling System (GAMS) is a high-level modeling system for mathematical programming problems. It consists of a language compiler and a stable of integrated high-performance solvers. GAMS is tailored for complex, large scale modeling applications, and allows you to build large maintainable models that can be adapted quickly to new situations.

- An Introduction to GAMS
- Documentation (including FAQ)
- Contributed Documentation
- Presentations, Books, Posters

- Download Current GAMS System
- Download Older GAMS Systems
- Contributed Software

- Courses and Workshops
- The GAMS Mailing List and Newsletter

- Visit our European Web Site

Figure 1-2: Home page of GAMS, the first major modeling language (http://www.gams.com).

It has become increasingly common to separate modeling languages and systems from optimization solvers. In fact, the modeling language software, solver software, and data used to generate the model instance might reside on different machines using different operating systems. The next great leap forward happened in the mid 1990's when large-scale optimization was brought onto the Internet. The NEOS Server [29] for Optimization is the most ambitious realization to date of the optimization *server* idea. A cooperative effort of over 40 designers, developers, collaborators, and administrators at the Optimization Technology Center of Northwestern University and Argonne National Laboratory, NEOS provides access to dozens of solvers. Modelers can submit problems with representations of many kinds and through networking mechanisms based on nearly all major protocols.

By using distributed computing technologies such as XML and Web services, we envision the Optimization Services approach as the next step in the evolution of optimization technologies.

The Optimization Services framework is an XML-based, service-oriented, optimization centered, distributed and decentralized architecture. By using Optimization Services Protocols, Optimization Services enable OR software to integrate with partners and clients in a fashion that is loosely coupled simple and platform-independent. In the next four sections, we illustrate the Optimization Services framework from different perspectives – from the viewpoint of OS as a framework for optimization systems, from the viewpoint of OS as a middle computational infrastructure for Operations Research (OR), from the viewpoint of OS as a next generation Network Enabled Optimization System (NEOS), and from the viewpoint of OS as the OR Internet.

1.2.1 OS as a framework for optimization systems

Optimization Services is a framework that specifies how a set of cooperative classes and interfaces should be designed and implemented in order to solve an optimization problem. The Optimization Services framework has the following properties:

- It consists of multiple classes or components, each of which may provide an abstraction of some particular optimization concept.
- It defines how these abstractions work together to solve an optimization problem.
- Its optimization-related components are reusable, which is what makes Optimization Services a good framework, since it provides generic behavior that many different types of OR applications can make use of.
- It organizes patterns at a higher level. By "pattern" we mean a tried and true way to deal with an optimization process, from the whole context to the problem and to the final solution that appears over and over again. Thus the adopted patterns in the Optimization

Services should be an effective means of communication between OR software developers, therefore bringing order into chaos.

There are many definitions of a framework. In Appendix B, we list the classes and interfaces provided by Optimization Services. Some may regard these as a framework, but these are really a library. There is a key difference between a library and a framework. A library contains functions or routines that an application or a user can invoke. A framework provides generic, cooperative components that software can follow and extend. Figure 1-3 shows the difference between a framework and a library. The Optimization Services framework provides a foundation upon which OR applications, software, and libraries are built, whereas an OR library is a piece of software used by other OR applications.

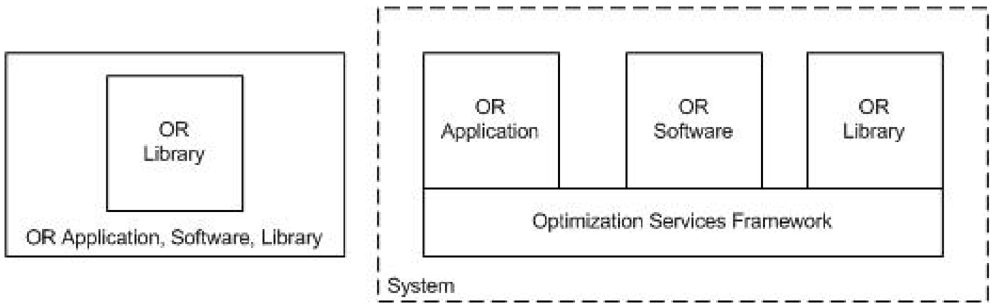

OR Library
OR Application, Software, Library
OR Application
OR Software
OR Library
Optimization Services Framework
System

Figure 1-3: Difference between an OR library and the Optimization Services framework.

Figure 1-3 also shows that the Optimization Services framework, *per se*, is not a system. It becomes a system (the dashed part in the figure) when implemented with the components (applications, software, libraries) built upon the framework. All the components work together to solve an optimization problem.

As an analogy, think of the Optimization Services as a constitution. A constitution itself is really not a government or a court system. Rather it is a documented framework that specifies the components and the nature of a government, its powers and responsibilities. Likewise Optimization Services is a “constitution” that specifies how such optimization components as modeling languages and solvers should be built and how they should interact with each other, only that the OS “constitution” is written in the XML language. Some specific examples in this “OS constitution” are specifications for the format of the instance output of a modeling language or the process for discovering and invoking a solver.

Although Optimization Services is intended to be a standard framework, *not* a system, we are also developing the optimization system according to this framework (see http://www.optimizationservices.org [92] or http://www.optimizationservices.net [93]) and building libraries for other people to put up their OS software and components (see Chapter 8).

1.2.2 OS as a computational infrastructure for Operations Research (OR)

Operations Research, as a branch of applied mathematics, has its foundations in mathematics, computing and economic theories, on which basic tools in optimization and simulation are built. We apply these tools to model problems in such areas as manufacturing, distribution, finance, and marketing.

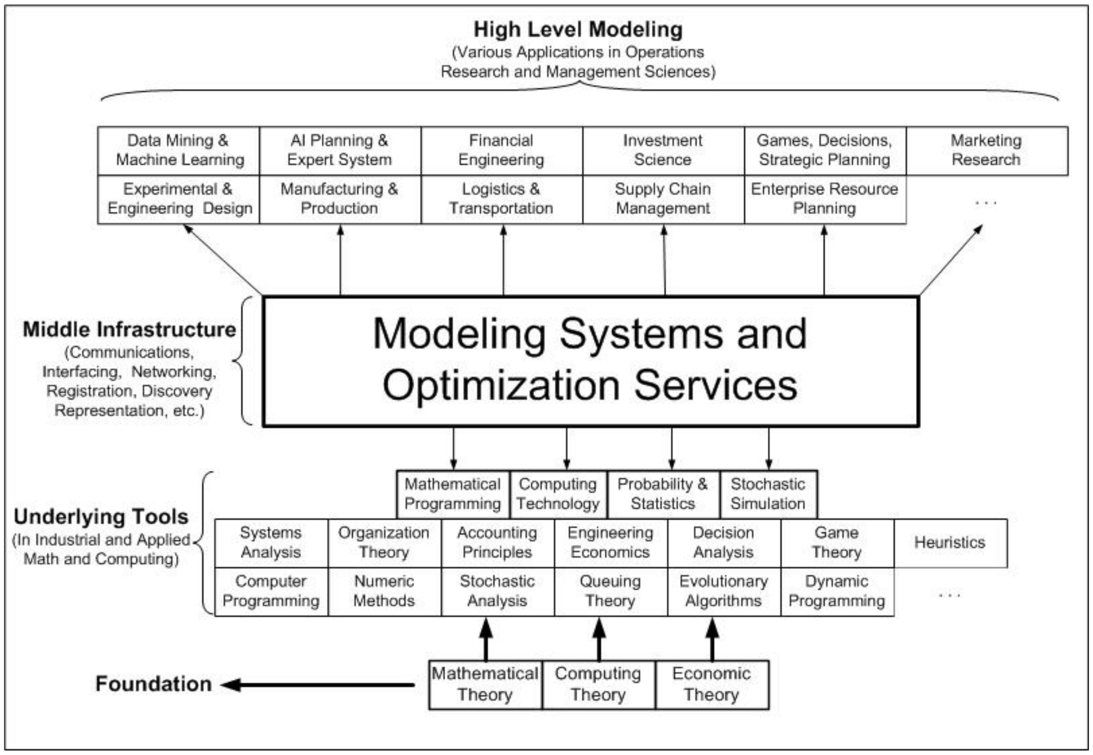


Figure 1-4: Positioning of OS in the hierarchy of Operations Research (OR).

Figure 1-4 shows a hierarchy of operations research activities. The highest level in the hierarchy is concerned with modeling and is the part that directly interfaces to consumers who use models for daily analysis.

The level of “Underlying Tools” comprises such core areas as mathematical programming, stochastic simulation, and statistics. This level is typically regarded as what uniquely defines Operations Research.

Optimization Services' position is in the middle of the Operations Research hierarchy. It is concerned with things like communication infrastructures, modeling languages and systems. It is an interface part that bridges OR modeling with OR tools. When implemented smoothly, it is the part that is not noticed by modelers or users.

1.2.3 OS as the next generation Network Enabled Optimization System (NEOS)

Figure 1-5: NEOS Server for Optimization at http://www-neos.mcs.anl.gov.

The NEOS server of the Optimization Technology Center of Northwestern University and Argonne National Laboratory makes more than 50 solvers available through several network mechanisms. Because the Server has evolved along with the Web and the Internet from their early times, it is limited to some degree by initial design decisions and is facing growing communication difficulties.

Optimization Services, with all the OR applications, software and libraries built upon the OS framework, is intended to be the next-generation NEOS. It addresses many outstanding design and implementation challenges faced by the current NEOS under the large-scale and distributed optimization environment. For example, the benefit to the optimization community of a common format for instance representation and an accepted application programming interface (API) for solvers is clear. If modeling languages support a common format (addressed by our Optimization Services instance Language – OSiL), and solvers support a common API that operates on the instance format, then solver developers do not have to worry about supporting multiple model formats and modeling language developers do not have to worry about supporting varied solver input formats. As stated in the original National Science Foundation (NSF) proposal [44] for this research, titled *Next-Generation Servers for Optimization as an Internet Resource*:

"The planned research is motivated by a vision of a next-generation NEOS Server that addresses outstanding challenges of communication in large-scale optimization. This work will address design as well as implementation issues posed by standardizing problem representations, automating problem analysis and solver choice, working with new web-service standards, scheduling computational resources, benchmarking solvers, and verification of results — all in the context of the special requirements of large-scale computational optimization."

Considering the fact that the NEOS Server has over the past decade shown significant value in helping users of all kinds, Optimization Services can have widespread benefits to practitioners inside and outside of the Operations Research community. The continuing goal of Optimization Services as the next generation NEOS should stay the same as the current NEOS, to “make optimization a part of the worldwide software infrastructure that supports science and commerce.”

There is one fundamental difference between NEOS and Optimization Services. NEOS is based on a tightly coupled centralized structure. All the solvers are connected with the server, and all the optimization job requests have to go through it. Therefore, the system does not scale well.

On the other hand, Optimization Services adopts a decentralized Service-oriented Architecture (SOA, see Chapter 4). There is still in some sense a “central” server in the middle, but it functions as a lightweight “registry server,” or just “registry.” Such a registry knows all the solvers and other Operations Research software that exist in the whole decentralized system by keeping metadata files. Metadata here means that the registry contains information *about* the software, but not the software itself. No solvers are actually executed by this registry; instead users directly contact the solvers in a peer-to-peer mode. The advantages of a decentralized Service-oriented Architecture are significant and are elaborated throughout this thesis. The Internet has become popular because it is a decentralized architecture. There is no such thing as a “central repository server” that hosts all the Web pages. Development and maintenance all happen spontaneously. It is our vision that a decentralized architecture can better promote research and development in Operations Research.

1.2.4 OS as the Operations Research (OR) Internet

Optimization Services and the Internet are closely related because of the decentralized architecture.

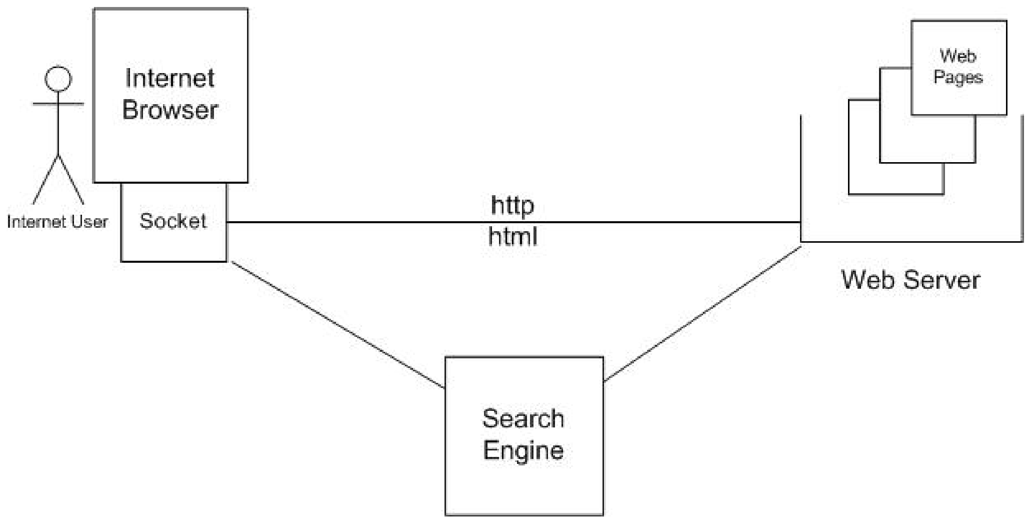


Figure 1-6: A simplified sketch of Internet for purpose of illustration.

In order to "surf the Internet" (Figure 1-6), a user uses an Internet browser to view the Web pages, which usually contain interactive links and forms. Clicking the links and filling in forms are what we call the user inputs. In the scenario of Optimization Services (Figure 1-7), the user is a modeler and his inputs are a model and the model's data. Instead of the browser, the modeler constructs the model in a Modeling Language Environment (MLE[1]) or in a Graphical User Interface (GUI[2]) environment and instead of sending the model inputs to a web server, the MLE or GUI sends the inputs to an OS server. The OS server hosts solvers rather than Web pages (although Web pages can still be hosted along with the solvers on an OS server). Although the Internet existed long before it became popular, the entertaining Web pages were what made the Internet successful. The same can be said about Optimization Services. Without the actual "contents" provided by the solvers, OS is just an empty skeleton that can never be widely used no matter how well the skeleton is designed.

[1] Modeling Language Environment is more traditionally called Modeling System. In this thesis, we prefer to use the term Modeling Language Environment and abbreviate it as MLE to avoid the potential confusion on the use of the term "system," because MLE is just one of the many components considered in a more general Optimization Services system.
[2] The difference between MLE and GUI will be explained in **www.optimizationservices.org (or http**.

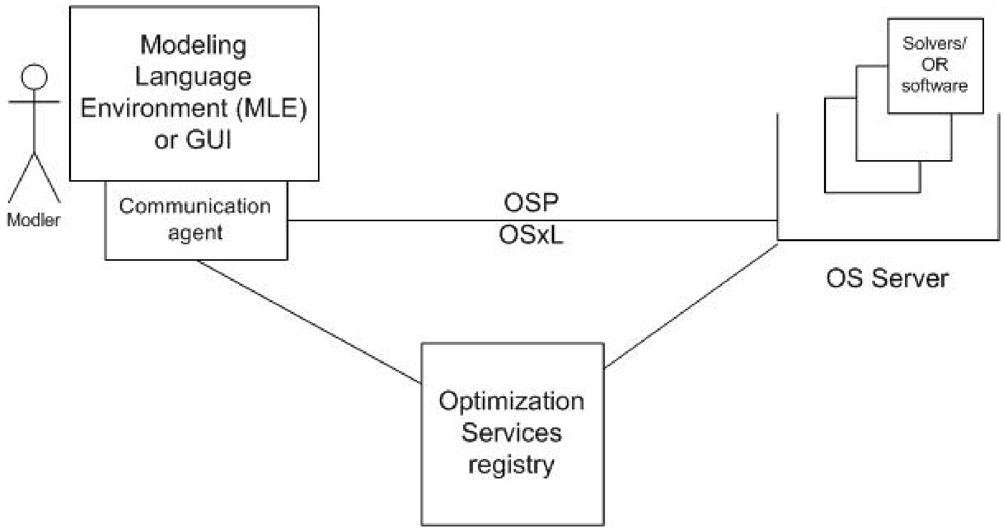

Modeling Language Environment (MLE) or GUI
Modler
Communication agent
OSP
OSxL
Solvers/ OR software
OS Server
Optimization Services registry

Figure 1-7: Analogy between Optimization Services and the Internet.

To further apply the analogy, it is never the browser that contacts a web server. Rather the browser opens a socket, and through the socket, the browser sends the request and waits for the response. These all happen without the user's knowledge. The exact equivalent of the socket in Optimization Services is the communication agent. The MLE or GUI delegates the agent to send an optimization instance to the remote OS server that hosts the solver. Like the socket, the agent understands all the communication protocols in order to establish the connection. But instead of using the HTTP protocol and sending/receiving HTML instances, the agent uses the OSP communication protocol and sends/receives OSP representation instances.

Nowadays people heavily rely on search engines to find Web pages. The Optimization Services registry serves the function of a search engine. But unlike the Internet search engines, there has to be a unique registry in the whole Optimization Services system to ensure Quality of Service (QoS). Communication agents always know where the registry is, as there is only one. This registry has complete information of available services, as this is the only place that the services can register. The OS registry will not be overburdened as no software is connected through it.

When a certain query is sent to the OS registry, usually from an MLE or GUI, the OS registry returns the locations of the found software and the MLE or GUI makes a peer-to-peer contact with the software at the provided location. This discovery process is similar to the search engine process, with the exception that everything in the OS system happens automatically between the software components, without user interaction.

On the opposite side of the discovery process is the registration process. In the case of the Internet, it is usually the search engine “crawlers” that automatically collect the contents of all the Web pages. In the Optimization Services case, it is the OR software developer’s responsibility to send the required information to, and get approved by, the OS registry, possibly through a mixture of automatic and manual procedures. This is primarily due to two reasons. One is that the quantity of OR software packages is not nearly large enough to be crawled efficiently. A second, and more important reason, is that the requirement of QoS on the OS registry is much stricter in order to ensure smooth functioning between OS components. The mechanism of “wantonly” crawling and storing “unwarranted” things found on the hyperlink paths degrades the Optimization Services.

1.3 Optimization Services Protocol (OSP)

A protocol is an agreed upon format for transmitting data between two devices, hardware or software. The Optimization Services Protocol determines how optimization related data are *represented* and *communicated* between two Optimization Services compatible software components. Just like the Internet Protocol (IP), OSP can also be used by organizations sharing private networks.

OSP is a rapidly evolving set of standards that consists of over 20 sub-protocols, all described by an abbreviation of in the form of “OSxL”, meaning some Optimization Services x Language. For example, OSiL stands for Optimization Services instance Language, which is a language expressed in XML to specify the structure and format of general optimization instances. As a core of the Optimization Services framework, OSP has great promise for the world of Operations Research applications, optimization systems and distributed computing.

1.3.1 OSP as an application level protocol in protocol layering

In modern protocol design, protocols are “layered.” Layering is a design principle that divides the protocol design into a number of smaller parts, each of which accomplishes a particular sub-task, and interacts with the other parts of the protocol only in a small number of well-defined ways. For example, one layer might describe how to encode text (with ASCII, say), while another may detect and retry errors (with TCP, the Internet's Transmission control protocol), another handles addressing (with IP, the Internet Protocol). Layering allows the parts of a protocol to be designed and tested without a combinatorial explosion of cases, keeping each design relatively simple.

As illustrated in Figure 1-8, the *reference* model usually used for layering is the Open Systems Interconnection (OSI) seven layer model -- physical, link, network, transport, session, presentation, and application layers from bottom to top. The Internet protocols (TCP/IP) can be analyzed using the OSI model, even though TCP/IP has only four distinct layers -- network access (e.g. Ethernet), internet (e.g. IP), transport (e.g. TCP), and application layers (e.g. HTTP). All protocols layered above the HTTP protocol (e.g. SOAP, briefly described in the next section) are also called application level protocols. Thus OSP, being a protocol based on SOAP (Chapter 4), is classified as an application level networking protocol.

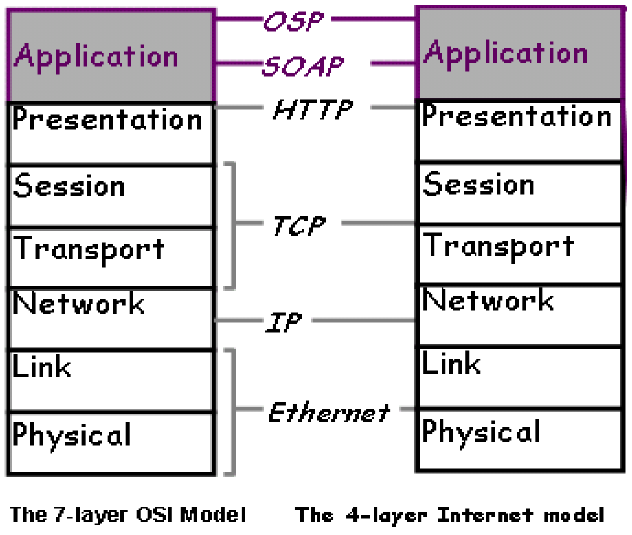


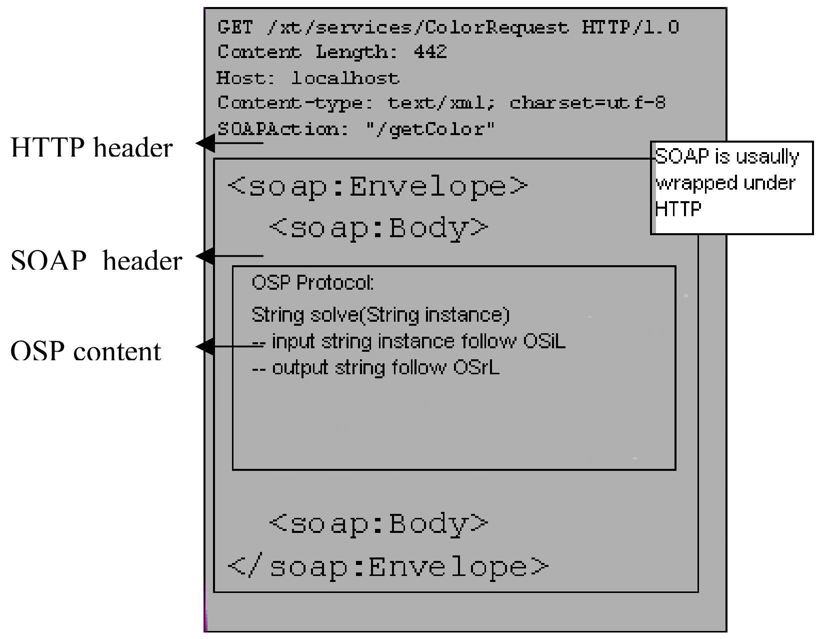
Figure 1-8: Layering of Internet protocols.

1.3.2 OSP as an interdisciplinary protocol between CS and OR

The Optimization Services Protocol is entirely based on SOAP[1]. Short for Simple Object Access Protocol, SOAP is a lightweight XML-based messaging protocol used to encode the information in Web service request and response messages. SOAP messages are independent of any operating system or protocol and may be transported using a variety of Internet protocols, including SMTP, MIME, and HTTP, although nearly always it is using HTTP. Generally, the protocols under the network layer belong to the area of Electrical Engineering, and the

[1] More exactly, it is our implementation of the Optimization Services Protocol (OSP) that is entirely based on SOAP. Theoretically OSP can be built on any networking protocol, but the XML nature of OSP and SOAP make them a natural pair.

protocols above the network layer belong to the area of Computer Science. In this regard, SOAP is naturally a Computer Science protocol.


HTTP header
SOAP header
OSP content
SOAP is usaully wrapped under HTTP
OSP Protocol:
String solve(String instance)
-- input string instance follow OSiL
-- output string follow OSrL

```
GET /xt/services/ColorRequest HTTP/1.0
Content Length: 442
Host: localhost
Content-type: text/xml; charset=utf-8
SOAPAction: "/getColor"
<soap:Envelope>
   <soap:Body>

   <soap:Body>
</soap:Envelope>
```

Figure 1-9: OSP inside SOAP, which, in turn, is usually inside HTTP.

Although SOAP defines a set of rules for *structuring* messages, it does not specify the actual *content* of the messages. In that sense SOAP is a generic and domain-independent protocol. OSP takes on the task of specification of the content in the domain area of Operations Research. The nature of bridging protocols in two separate areas – Computer Science and Operations Research – classifies OSP as an interdisciplinary protocol.

In an actual data packet, all the contents specified in OSP are inside a SOAP envelope. As both OSP and SOAP are XML based protocols, this is equivalent to saying that OSP contents are child elements of a SOAP parent element (Figure 1-9). For example, the Optimization Services hookup Language (OShL) sub-protocol of OSP specifies that OS compatible solvers should provide an invocation in the form:

```
String solve(String instance);
```

in which the input string "`instance`" has to follow the representation format specified by the Optimization Services instance Language (OSiL) and the output string has to follow the representation format specified by the Optimization Services result Language (OSrL).

1.3.3 OSP sub-protocols

There are mainly two categories of OSP sub-protocols, one that deals with representation (Chapter 6) and the other that deals with communication (Chapter 7). All the sub-protocols described in Chapter 8 - Optimization Services Registry, actually belong to either the representation or communication category. Since the registry is one of the most significant parts of Optimization Services and there are numerous corresponding sub-protocols, all the registry related representation sub-protocols are listed separately in Chapter 8. Most of the representation sub-protocols are specified in XML schema, a mechanism for defining a vocabulary specifying the structure of XML documents (Chapter 4). Most of the communication sub-protocols are specified in Web Services Description Language (WSDL, Chapter 4), a mechanism to describe the technical invocation syntax of a Web service, such as an optimization service.

The principle of the representation related OSP sub-protocols is that they concentrate on *content structure* rather than *presentation appearance*, making the files more reusable and leaving the visual details to the end-user software, like Modeling Language Environments.

Representation sub-protocols

Here is the list of names and brief descriptions of the OSP sub-protocols for representations (non-registry-related) that are covered in detail in Chapter 6:

- Optimization Services general Language (OSgL) – definitions of general data structures used by all other OSxL schemas.
- Optimization Services instance Language (OSiL) – a general optimization instance format specification, including general nonlinear, constraint and logic, network and graph, stochastic and other extensions.
- Optimization Services linear Language (OSlL) – reserved in honor of the original LP-FML[53]. LP-FML is among the first XML initiatives to standardize linear optimization instance formats.
- Optimization Services nonlinear Language (OSnL) – definitions of all the nonlinear, combinatorial, and other nodes (e.g. operators, operands, etc.) used in other OSxL's, mainly OSiL.
- Optimization Services result Language (OSrL) – a general optimization result format specification, mainly outputted by solvers that can include analyses as well as solutions.

- Optimization Services option Language (OSoL) – a general OR software option format specification.
- Optimization Services analysis Language (OSaL) – an optimization analysis format specification of analyzer output.
- Optimization Services simulation Language (OSsL) – a specification of input and output format of a simulation engine.
- Optimization Services transformation Language (OStL) – a standard transformation style sheet used to present other instance representations.

Communication sub-protocols

Communication sub-protocols deal with the general areas of optimization access, operations and flows. No mechanisms such as encoding and security are addressed in OSP. OSP leverages the mechanisms provided by its underlying protocols, for example the encoding scheme from SOAP and the security support from HTTP. Here is the list of names and brief descriptions of the OSP sub-protocols for communication (non-registry-related) that are covered in detail in Chapter 7:

- Optimization Services hookup Language (OShL) – a description of how to hook up with OS software, mainly solvers and analyzers.
- Optimization Services call Language (OScL) – a description of how to call simulation engines.
- Optimization Services flow Language (OSfL) – an XML document of predefined standard flows of optimization services invocations.

Registry sub-protocols

The following are registry-related representation and communication sub-protocols and are covered in detail in Chapter 8:

- Optimization Services query Language (OSqL, representation) – a specification of the query language format used to discover the optimization services in the OS registry.
- Optimization Services uri Language (OSuL, representation) – a specification of the discovery result (in uri) sent back by the OS registry.
- Optimization Services entity Language (OSeL, representation) – a specification of entity information used to describe the *static* information of an optimization service (such as name, type, and description).

- Optimization Services process Language (OSpL, representation) – a specification of process information to describe the *dynamic* information of an optimization service (such as number of jobs being solved).
- Optimization Services benchmark Language (OSbL, representation) – a specification of benchmark information used to partly describe an optimization service.
- Optimization Services yellow-page Language (OSyL, representation) – a specification of the organization of the registry database information.
- Optimization Services discover Language (OSdL, communication) – a description of how to discover optimization services in the OS registry.
- Optimization Services join Language (OSjL, communication) – a description of how an optimization service can join the OS registry.
- Optimization Services knock Language (OSkL, communication) – a description of how the OS registry can "knock" on remote OS services to check their run time information.
- Optimization Services validate Language (OSvL, communication) – a description of how the OS registry can be used to validate any OS instance.

A brief outline of the thesis follows. In Chapter 2, we describe optimization systems and components in general. Any optimization system that is built on the Optimization Services framework is called an Optimization Services (OS) system and the system components are called OS-compatible components. In Chapter 3, we discuss two real world distributed optimization systems. They initially served as motivations to the research in Optimization Services. In fact, the implementation of Optimization Services is intended to be a next-generation system of the first example -- NEOS. Chapter 4 provides the necessary background on modern computing and distributed technologies in order to read the thesis. Chapter 5 formally introduces the concept of Optimization Services. Chapter 6, Chapter 7, and Chapter 8 describe respectively the representation, communication, discovery and registration parts of the OS framework and the corresponding OSP protocols. Although Optimization Services is intended to be a standard framework, NOT a system, we are also developing the Optimization Services system according to this framework and building libraries for other people to put up their OS software and components. A derived research product from the Optimization Services is a modeling language that natively supports the OSP protocols. We generically named this modeling language Optimization Services modeling Language (OSmL, see Chapter 9). Unlike other OSxL's, OSmL is the component that directly faces a modeler. So it is *not* intended to be a standard. What is natural for one modeler may not be for another, so user flexibility is the order of the computing world today. OSmL is invented to illustrate an original idea of designing modeling languages and to facilitate the adoption of Optimization Services. We end the thesis with Chapter 10 with a discussion of additional research and business models based on Optimization Services. Appendix A lists some of the extensions of optimization representations covered in Chapter 6. The design and implementation of Optimization Services libraries are covered in Appendix B.

CHAPTER 2 OPTIMIZATION SYSTEMS AND COMPONENTS

There are different definitions of an optimization system. The chapter is not intended to add another one. Rather our purpose is to describe the scope of the Optimization Services framework and show the system components that are targeted in the OS framework's standardization process. We are mainly interested in the more general distributed optimization systems. Optimization within a local environment is treated as a special case. Issues that exist within a local environment are mostly addressed under the distributed case.

First we clarify certain terminology usage in this thesis. Most modeling language software starts with a *core* modeling language along with a language compiler. Gradually the core evolves to include other *auxiliary* software such as preprocessors and graphical user interfaces (GUIs). By "auxiliary" we mean tools that *help* in constructing, preprocessing and compiling a modeling language, but *not* solving the model, which is the function of a solver. Modeling languages are eventually packaged with solvers in distribution. The whole package is usually called a modeling system. In this thesis, however, we stay away from using the term "modeling system" to avoid its potential confusion with the more general optimization system shown in Figure 2-1.

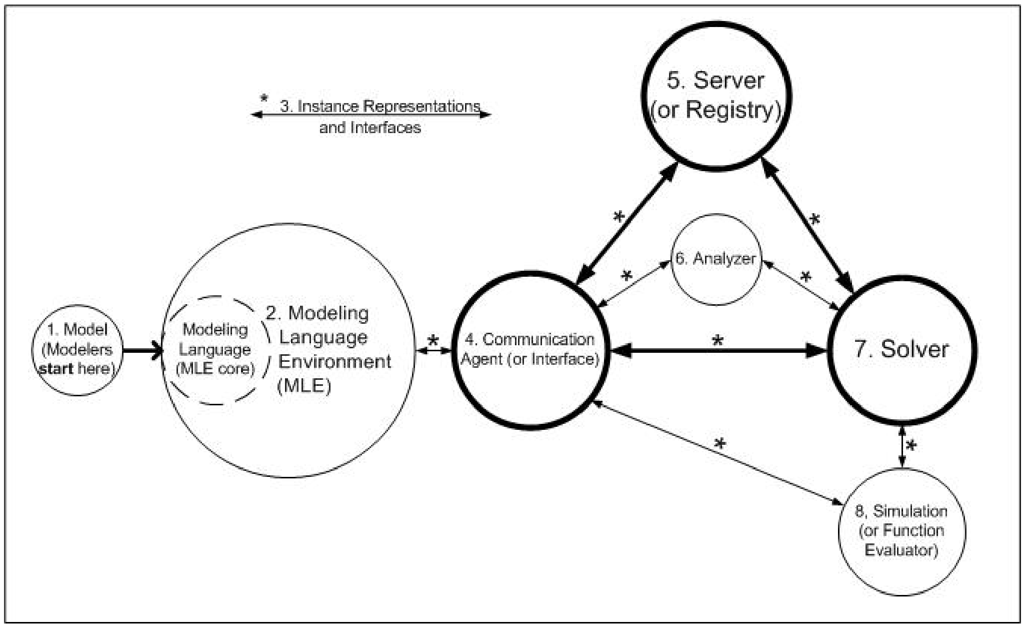

* 3. Instance Representations and Interfaces
5. Server (or Registry)
6. Analyzer
1. Model (Modelers start here)
Modeling Language (MLE core)
2. Modeling Language Environment (MLE)
4. Communication Agent (or Interface)
7. Solver
8, Simulation (or Function Evaluator)

Figure 2-1: A typical optimization system and component interaction.

Although a modeling package without any solvers is sometimes called a modeling system, we call a modeling language without any solvers a Modeling Language Environment (MLE), that is, a modeling language core with only auxiliary tools. An MLE is one of the components in an optimization system. An optimization system contains most of the following components:

1. Model. This is where a modeler starts. The model component differs from the rest of the optimization components in that it is an abstraction of an input problem rather than a physical piece.
2. Modeling Language Environment (MLE). The core of the MLE is the modeling language, in which an abstract model is defined. The MLE helps in the implementation process. Often a modeling environment may not have a modeling language, but just a spreadsheet or some graphical user interfaces with implicitly defined models. We call it a GUI. From the perspective of the general optimization system, the functions of MLEs and GUIs are the same.
3. Instance Representation. An instance Representation is also called an instance. It is generated by various optimization system components and exchanged among them. For example, an MLE parses a model and generates a problem instance. This problem instance is then sent to a solver to be solved. The instance component differs from other physical components in that it is a data piece rather than software.
4. Communication Agent/Interface. A communication agent is also called an agent. Agents are in charge of communication in a distributed system. No agents are needed in a local environment, in which case interfaces and objects are instantiated in memory and methods are invoked locally. Communication agents are used to send and receive instances. Instance representations and communication agents are least visible to system users, although they constitute the backbones of an optimization system.
5. Server/Registry. Think of a registry as a lightweight server for now. A server or registry is the heart of a distributed system. An agent usually communicates with a server or a registry before invoking a solver.
6. Analyzer. This is as an important auxiliary component in the whole system. Without analyzers, an optimization system can potentially involve much human interaction. So analyzers play a key role in an automated optimization system.
7. Solver. Being the real "contents" of an optimization system, solvers make the whole system meaningful and are what users really need.

8. Simulation. Think of a simulation as a black box function evaluator. The simulation engine may or may not reside with the solver. If the simulation is a simple function that stays locally with the solver, it is usually called a function evaluator, a function pointer, an evaluation routine, or an expression tree. In an optimization system, simulations are usually invoked by a solver. Most of the optimization solver algorithms involve some iterative schemes and each iteration may potentially involve an invocation of the simulation.

There may be many other components in the optimization system. We mention some of them in the following chapters, such as problem libraries, and solver benchmarkers. However the above eight components are the key ones. They are the main targets to be "regulated" by the Optimization Services framework (Chapter 5). When all these components are built according to the Optimization Services (OS) framework, we call them "OS-compatible," and we call the optimization system an Optimization Services system. Each of the components is explained in detail in the subsequent sections.

The process of the optimization system in Figure 2-1 is self-explanatory. Typically the process starts from a modeler who has a model (1) to be solved. He constructs the model in an MLE (2). The MLE in turn compiles the model and generates an instance representation (3). The MLE then delegates a communication agent (4) to send the instance to a solver (7). In a local environment, where there is no agent, the link between 4 and 7 is an interface through which the MLE instantiates the solver in memory. In a distributed environment, the MLE may access the solver through a server or a registry (5), therefore the respective links between 4 and 5 and between 5 and 7. The communication with the analyzer (6) is similar to that with a solver (7). The link between communication agent (4) and simulation (8) means that the agent may call the simulation to get a function value, although in an optimization scenario, it is usually the solver (7) that calls the simulation (8) iteratively.

As will become clear, the triangle between communication agent (4), registry (5) and solver (7) will later evolve into the Service-oriented Architecture (SOA). The design philosophy of SOA serves as the basis of our Optimization Services framework. A "service" is intended to serve customers. For our optimization system, there are mainly three categories of "human" customers:

- **Application developers** create and build system components such as modeling language environments and solvers as part of a larger optimization system. The components together take care of such generic functions as managing data, solving optimization problems, and presenting solutions in a graphical interface.

- **Modelers** work in a modeling language environment or GUI to build optimization models and get acceptable solutions.
- **Users** run application packages that perform optimization at some stage through the optimization system. Users are usually the ultimate customers of any optimization system.

Modelers and application developers may see optimization in different ways. For modelers, a mathematical program is an abstract representation to be analyzed and understood; for application developers, a mathematical program is a concrete instance to be represented, communicated and solved. Modelers benefit most immediately from innovations that help people to choose and experiment with optimization software. Some application developers are also modelers, while others deal mainly with the inputs and outputs of optimization models set up by modelers. Users may not even realize that they are running optimization system components such as solvers, although they are often aware of optimization goals, such as minimizing costs or maximizing profits.

2.1 Model

In general, models are abstractions of reality. Although a model can take forms such as a graph or a flow chart, the models we discuss in this thesis are mainly high-level mathematical representations of problems that people find reasonably natural or convenient. The solved models are often used to assist in decision-making, an essential principle of Operations Research and Management Sciences. With the help of a range of quantitative techniques, models are tested, manipulated, and hopefully solved. Thus the models must be accurately formulated. For example, most of the time, the goals, the decisions, and the constraints of the problem must be clearly defined. The term "mathematical programming[1]" is often used as a synonym for "optimization" to mean the minimization or maximization of an objective function of many variables subject to constraints on the variables. A typical example is a linear program,

$$\begin{array}{ll} \underset{x}{\text{minimize}} & cx \\ \text{subject to} & Ax = b \\ & x \geq 0 \end{array} \qquad (2\text{-}1)$$

[1] The word "programming" was first used in the 1940's to mean planning or scheduling of related activities within a large operation; the necessary relationship to computer programming was incidental to the choice of name. There is no direct connection between the two, although indirectly some computer programming must be done in order to solve mathematical programs **://www.optimizationservices.net)**.

where x is an n-vector of variables, c is an n-vector of objective coefficients, A is an $m \times n$ matrix of constraint coefficients, and b is an m-vector of constraint right hand side coefficients. The expression cx is called the objective function and the equations $Ax = b$ are called the constraints. Usually A has more columns than rows, and $Ax = b$ is therefore quite likely to be under-determined, leaving great latitude in the choice of variables x with which to minimize cx. The expression of the math program can have many variations. All these affect the design of a modeling language discussed in the next section.

- **Language variation**. People may state the model in a different language to mean the same model. One common example would be using "s.t." instead of "subject to."
- **Algebraic variation**. A simplest example would be to use y instead of x to represent variable vectors. Also in (2-1), instead of using $Ax = b$, we can use a double-sided constraint $b \le Ax \le b$ to express the same math program.
- **Mathematical program type variation**. The above example illustrates a linear math program. We can have many more math program types by allowing different variable types (continuous, integer, binary), function types (linear, nonlinear), constraint types (unconstrained, bound constrained, generally constrained) and other special properties (special objectives, special operators, special structures, special parameter, variable or function behaviors). The analysis and categorization of optimization can be a daunting task (Figure 2-2). Much effort has been dedicated to designing a good categorization of optimization problems such as the one shown in Figure 2-3. A good categorization with unambiguousness and practicality will facilitate automated analysis thus directing optimization system development in a healthy and robust way (more in Chapter 6).

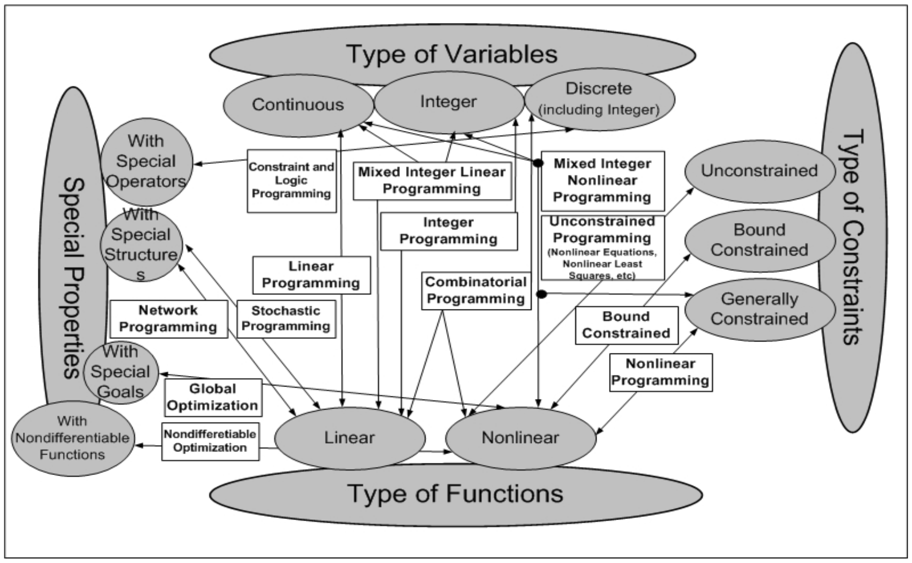


Figure 2-2: A daunting task of optimization categorization.

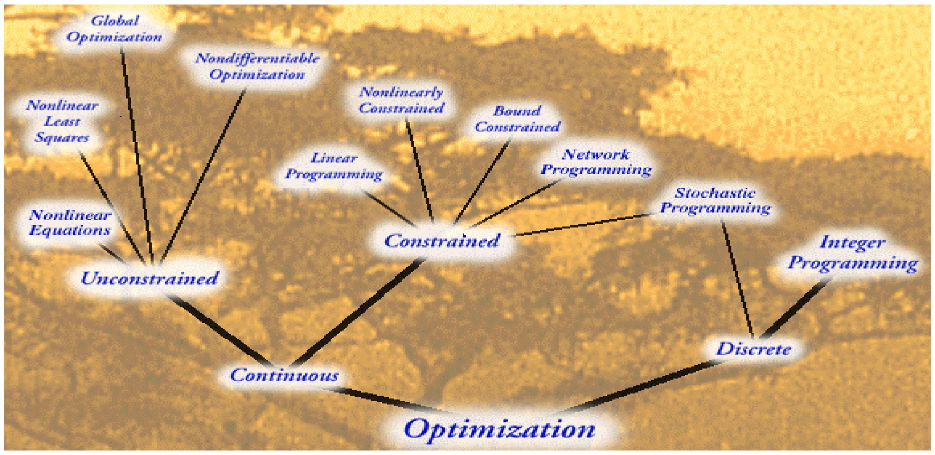


Figure 2-3: NEOS Optimization Tree to help users manually choose connected solvers.

We described three types of customers of optimization systems. A modeler is the person to formulate a model in a mathematical program. The modeler needs to express both what the mathematical program is and how it relates to the situation being modeled. Models expressed in mathematical programs should have the following common characteristics [45]:

- **Symbolic**. They represent most of the problem data by symbols, which are usually mnemonic in nature.
- **General**. They can define an entire class of mathematical programs together, each particular mathematical program corresponding to some choice of data.

- **Concise**. They describe a mathematical program nearly as briefly as possible, in such a way that the description's length depends on the complexity of the model rather than on the quantity of data or on particular data values.
- **Understandable**. They present a mathematical program in a form that is easily read and comprehended by people.

Different people have different preferences of expressing a mathematical model. The Optimization Services framework is *not* intended to standardize the expression of the model at the user or modeler level. The Optimization Services framework mainly regulates the communication between machine and software components. In fact, through standardizing the underlying communication, the Optimization Services framework promotes the flexibility for users to express models differently with various modeling languages, as they will no longer be limited by the choices of software due to interface compatibility issues.

2.2 Modeling Language Environment (MLE)

Modeling languages are a standard tool in the development of mathematical programming applications. A modeling language environment is designed to help people formulate mathematical programs and analyze their solutions. A modeling language environment takes as input the above described "model," and translates the model to the forms required by solvers automatically. Figure 2-4 shows an example of the classic diet problem expressed in the AMPL modeling language. The goal of a modeling language is to express a mathematical programming problem in much the same way that a modeler does, which is to describe mathematical programs in a readable and symbolic form, such as the familiar algebraic notation for variables, constraints, and objectives.

```
set NUTR ordered;
set FOOD ordered;

param cost {FOOD} >= 0;
param f_min {FOOD} >= 0, default 0;
param f_max {j in FOOD} >= f_min[j], default Infinity;

param n_min {NUTR} >= 0, default 0;
param n_max {i in NUTR} >= n_min[i], default Infinity;

param amt {NUTR,FOOD} >= 0;

# ------------------------------------------

var Buy {j in FOOD} integer >= f_min[j], <= f_max[j];

# --------------------------------------------------------

minimize Total_Cost:  sum {j in FOOD} cost[j] * Buy[j];

# --------------------------------------------------------

subject to Diet {i in NUTR}:
n_min[i] <= sum {j in FOOD} amt[i,j] * Buy[j] <= n_max[i];
```

```
param:  FOOD:                    cost  f_min  f_max :=
"Quarter Pounder w/ Cheese"    1.84     .      .
"McLean Deluxe w/ Cheese"      2.19     .      .
"Big Mac"                      1.84     .      .
"Filet-O-Fish"                 1.44     .      .
"McGrilled Chicken"            2.29     .      .
"Fries, small"                  .77     .      .
"Sausage McMuffin"             1.29     .      .
"1% Lowfat Milk"                .60     .      .
"Orange Juice"                  .72     .      . ;

param:  NUTR:   n_min  n_max :=
Cal      2000      .
Carbo     350    375
Protein    55      .
VitA      100      .
VitC      100      .
Calc      100      .
Iron      100      . ;

param amt (tr):
Cal  Carbo Protein   VitA   VitC  Calc   Iron :=
"Quarter Pounder w/ Cheese"  510     34     28     15      6     30     20
"McLean Deluxe w/ Cheese"    370     35     24     15     10     20     20
"Big Mac"                    500     42     25      6      2     25     20
"Filet-O-Fish"               370     38     14      2      0     15     10
"McGrilled Chicken"          400     42     31      8     15     15      8
"Fries, small"               220     26      3      0     15      0      2
"Sausage McMuffin"           345     27     15      4      0     20     15
"1% Lowfat Milk"             110     12      9     10      4     30      0
"Orange Juice"                80     20      1      2    120      2      2 ;
```

Figure 2-4: The AMPL model and data on the classic diet problem (http://www.ampl.com).

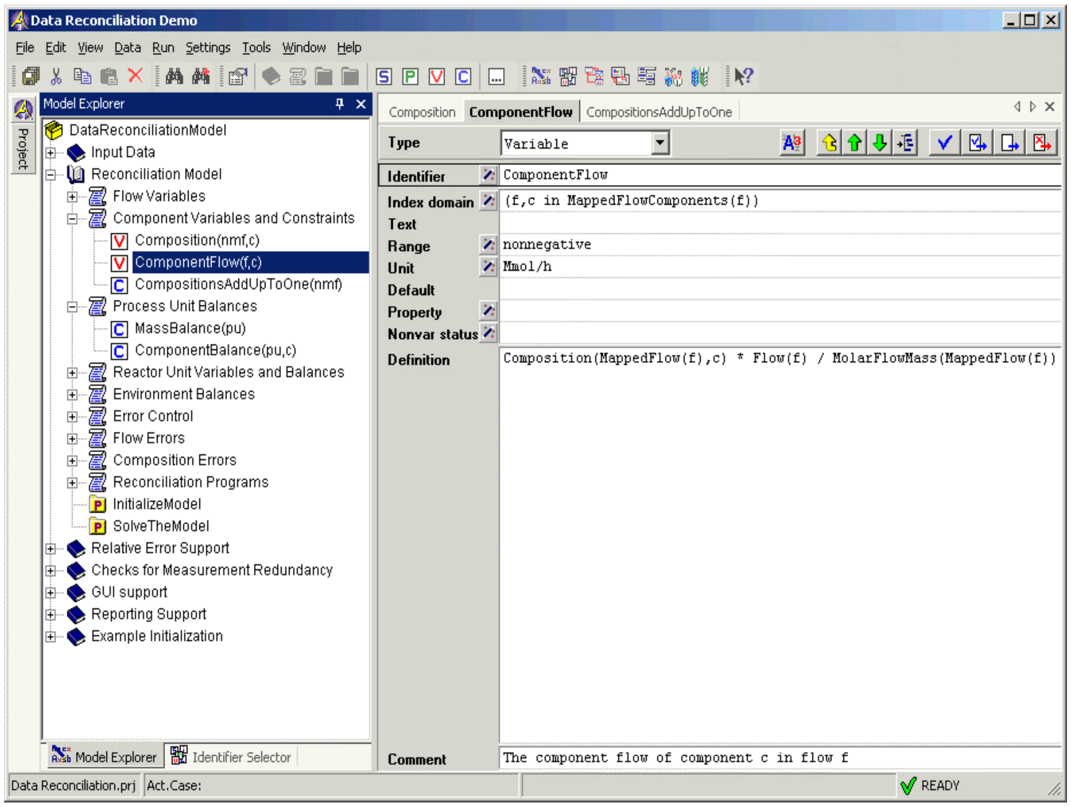
A modeling language is, however, not a general-purpose programming language; rather, it is a special purpose declarative language that expresses the mathematical programming model in a notation that computer system can interpret. Be aware that there are modeling languages for other areas in Operations Research, such as simulation, statistical regression and differential equations. But mathematical models in these areas tend to have a fairly small number of equations that may be written out in full without undue effort.

Mathematical programming models, by contrast, are almost always too big to be represented without symbolic combinatorial devices, such as index sets. Modeling languages for mathematical programs are consequently somewhat harder to design and implement.

In general, any modeling language for mathematical programs must satisfy two opposing sets of requirements. One set is imposed by the needs of customers discussed early in this chapter and the other by the nature of computers. Customers want a modeling language that is easy to use and to understand. Thus a modeling language needs to enforce an organization and terminology that modelers find convenient and natural.

Computer systems, on the other hand, require a modeling language that can be processed and translated at reasonable cost. Above all, this means that the specification of a modeling language – both syntax and semantics – must be unambiguous and not overly complicated. Additionally, the language's notation must be simple and precise enough to be read, stored, and printed by machines. A practical modeling language is a compromise between the above requirements. In principle this compromise can be carried out in many ways, and workable modeling languages are designed in many ways. For example, in general, a flexible and powerful modeling language is most logically based on a variable-and-constraint (row-wise) form. However, a modeling language can also be derived from any activity-and-requirement (column-wise) modeler's form with no greater difficulty. There are other attractive forms for more specialized applications; as an example, a modeling language for network linear programs could be based on a node-and-arc form of description.

A modeling language is the core of a modeling language environment. The most important module in an MLE is a compiler that translates a model into an instance. Among the other auxiliary software modules are graphical user interfaces (GUIs) (e.g. Figure 2-5) and preprocessors. MLEs usually lead to more reliable application of mathematical programming at lower overall cost.



Type: Variable
Identifier: ComponentFlow
Index domain: (f,c in MappedFlowComponents(f))
Text:
Range: nonnegative
Unit: Mmol/h
Default:
Property:
Nonvar status:
Definition: Composition(MappedFlow(f),c) * Flow(f) / MolarFlowMass(MappedFlow(f))
Comment: The component flow of component c in flow f

Figure 2-5: AIMMS Modeling Environment with model explorer and property windows (http://www.aimms.com) .

Common alternatives to algebraic modeling language environments include spreadsheet front ends to optimization, and custom optimization applications written in general-purpose programming languages that are usually equipped with some GUIs. Matrix generators may be used behind the GUIs to generate optimization instances.

In the Optimization Services framework, the MLE is required to output an instance in Optimization Services instance Language (OSiL, Chapter 6). Also if the MLE is to invoke OS software on a distributed system, it must either carry out the communication following exactly the OS communication protocols or invoke an OS agent to do the job instead (Chapter 7).

2.3 Instance Representation

If a model is to express a mathematical program in a modeler's form, an instance representation is to express it in a computer algorithm's or solver's form. These two forms of mathematical programs are not much alike. Most applications of mathematical programming involve translating one form to the other and communicating the translated form to other system components through some complex interfaces.

There are three main reasons we emphasize the roles of instance representations and later interfaces (or agents in a distributed environment):

- Regular users and modelers (as versus developers) do not see low-level representations and interfaces. But in certain situations, awareness of low level operations helps make more appropriate judgments and decisions.
- The low level representations and interfaces are the biggest obstacle for the development of optimization in the Internet age. A good design of a low level component is essential for efficient and effective communication between different optimization system components. This is the key to building a simple, standard, scalable and smooth system infrastructure.
- The principal advantage of an instance representation as a separation between a modeling language environment and a solver lies in its flexibility. Writing software for interfacing, or drivers, does not require access to proprietary information about either the MLE or the solver. Thus the writing of drivers is encouraged. Some may be written by a modeling language developer and others by a solver developer. Driver source code can be made public, providing useful examples for writers of additional drivers.

Upon receiving a “solve” request from the modeler, the modeling language environment compiles the current *high-level* model/data into a particular *low-level*[1] optimization problem, or instance representation, in a format that has been designed to be flexible and easy to be parsed to the input data structure required by a solver. Such a generic process is described in Figure 2-6.

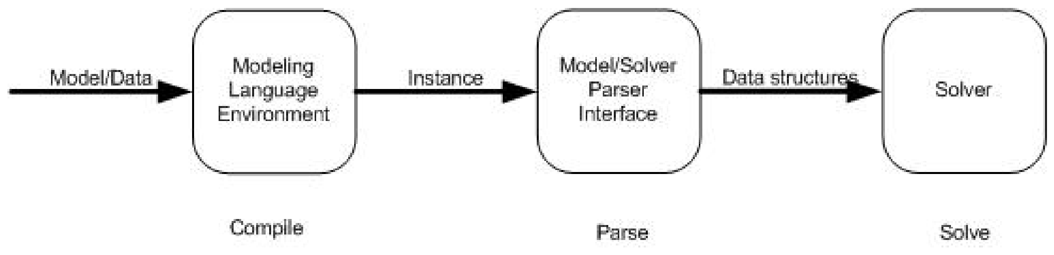


Figure 2-6: A generic process of instance generation and parsing.

[1] In this thesis, we mean the low-level instance representation when we simply say representation or instance.

Without the modeling language environment, a modeler must formulate his model directly in an instance representation, which is tedious and difficult to understand and adapt to similar models. Instance representations are distinctly different from models expressed in mathematical programs [45] in that they are:

- **explicit** rather than symbolic – they incorporate numerical problem data directly in the model;
- **specific** rather than general – they describe just one particular mathematical programs;
- **redundant** rather than concise – they describe a mathematical program more extensively than necessary, and the length of their description depends on the number and size of the data values;
- **convenient** rather than understandable – they organize a mathematical program so that it can be stored and operated upon most efficiently by the computer.

There are many acceptable instance representation formats just as there are many modeling languages. Table 2-1 lists different optimization types and major corresponding input formats. Many solvers also take binary inputs directly from general programming languages such as C, C++, Java, Matlab and FORTRAN. The Optimization Services instance Language (OSiL) in the OS framework supports all the major optimization types.

Linear Programming Quadratic Programming Mixed Integer Linear Programming	MPS, xMPS, LP, CPLEX, GMP, GLP, PuLP, LPFML, MLE instances
Nonlinearly Constrained Optimization Bounded Constrained Optimization Mixed Integer Nonlinearly Constrained Optimization Complementarity Problems Nondifferentiable Optimization Global Optimization	MLE instances SIF (only for Lancelot solver)
Semidefinite & Second Order Cone Programming	Sparse SDPA, SDPLR
Linear Network Optimization	NETGEN, NETFLO, DIMACS, RELAX4
Stochastic Linear Programming	sMPS
Stochastic Nonlinear Programming	None
Combinatorial Optimization	None (except for TSP input, only intended for solving Traveling

	Sales Person problems.
Constraint and Logic Programming	None
Optimization with Distributed Data	None
Optimization via Simulation	None

Table 2-1: Major optimization types and corresponding input formats; Optimization Services instance Language (OSiL) supports all the listed optimization types.

As seen in Table 2-1, there is not a widely accepted format for nonlinear programs. Solvers usually take the instance generated from a modeling language, and use the library provided by the MLE to parse the instance. Also many optimization types do not have any standard format.

One widely used format for representing linear and quadratic math programs is the MPS format that originated from IBM. See Figure 2-7 for the MPS representation of the quadratic program in 2-2.

$$\begin{aligned} &\text{minimize} && -x_2 + 1/2(2x_1^2 - 3x_1x_3 + 4x_2^2 + 5x_3^2) \\ &\text{subject to} && 6x_1 + 7x_2 - 8x_3 \geq 9 \\ & && x_1 \geq 0, x_2 \geq 0, x_3 \geq 0 \end{aligned} \qquad 2\text{-}2$$

```
NAME          qpEx
ROWS
 N  obj
 G  c1
COLUMNS
    x1        c1        6
    x2        obj       -1
    x2        c1        7
    x3        c1        -8
RHS
    rhs       c1        9
QSECTION      obj
    x1        x1        2
    x1        x3        -3
    x2        x2        4
    x3        x3        5
ENDATA
```

Figure 2-7: MPS representation of the quadratic math program in 2-2.

The original MPS only supported pure linear programming. The `QSECTION` in Figure 2-7 for quadratic programming was added in a later MPS extension. The reason that MPS has become widely accepted is not due to its flexibility or powerfulness. In fact, MPS was originally set up by an IBM user group for data representation using punch cards. It remained the only choice for decades until the 1980's. MPS already falls behind the current needs of operations researchers. Nowadays, many outdated formats like MPS and LP, mainly serve for submitting bug reports and for communicating benchmark problems. Optimization systems use much more general and efficient formats for communicating problem instances to solvers and for retrieving results.

2.4 Interface/Communication Agent

The instance representation created by a general-purpose modeling language environment is usually not directly input into a solver. Instead the instance is sent through an interface, a driver that converts between the generated instance and the data structures required by the solver (Figure 2-6). Each solver for a modeling language environment has its own driver, tailored to its particular requirements. The driver handles a variety of solver-specific information. The most important tasks are the processing of instance representations, handling of algorithmic directives (or “options”), evaluation of expressions at given points, and generation of solution reports (or “results”).

In the case of the AMPL modeling language environment, the AMPL language compiler converts the current problem instance to an AMPL “.nl” format, which is specific to AMPL, but not used by other MLEs. Each solver’s AMPL driver transforms this representation as necessary, passes the transformed instance to the solver itself, and retrieves the reported solution. Finally the driver converts the solution information to a “.sol” result format that the AMPL language environment is able to read and present to the user. The interfacing process is illustrated in Figure 2-8.

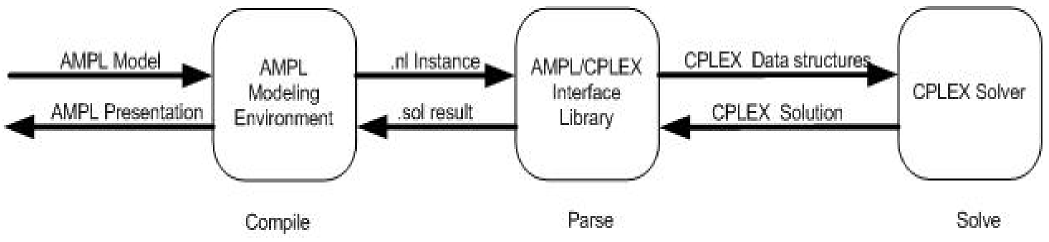
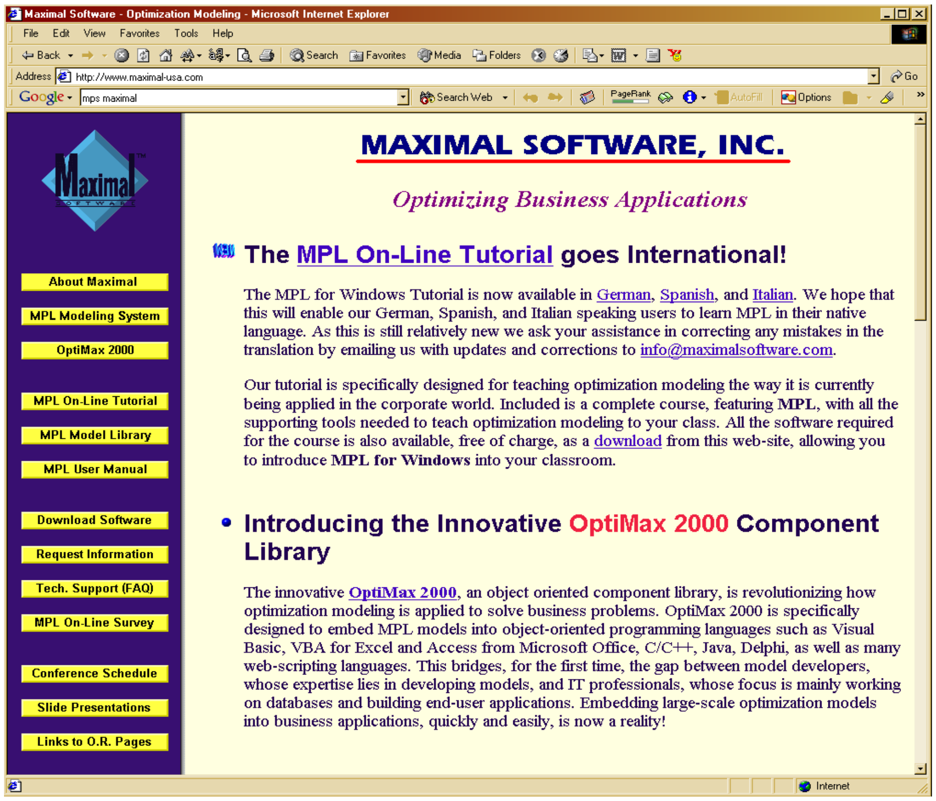


Figure 2-8: Interface between AMPL and CPLEX solver.

Optimization models are usually developed in the context of some larger algorithmic scheme or application. The ability of optimization software to be embedded through smooth interfaces is often a key consideration. Although most optimization software packages are built to be run in stand-alone mode, many are available in callable library form, and an increasing number can be accessed as class libraries in an object-oriented framework. Solver systems have long been available in this form, with the application-specific calling program taking the place of a general purpose modeling language environment. Modeling language environments have gradually also become available for embedding (Figure 2-9), so that the considerable advantages of developing and maintaining a modeling language formulation can be carried over into application software that solves instances of a model. Unfortunately, due to lack of standards, each interface requires a different implementation. Needs for standardization are receiving increasingly serious attention.



MAXIMAL SOFTWARE, INC.

Optimizing Business Applications

The MPL On-Line Tutorial goes International!

The MPL for Windows Tutorial is now available in German, Spanish, and Italian. We hope that this will enable our German, Spanish, and Italian speaking users to learn MPL in their native language. As this is still relatively new we ask your assistance in correcting any mistakes in the translation by emailing us with updates and corrections to info@maximalsoftware.com.

Our tutorial is specifically designed for teaching optimization modeling the way it is currently being applied in the corporate world. Included is a complete course, featuring **MPL**, with all the supporting tools needed to teach optimization modeling to your class. All the software required for the course is also available, free of charge, as a download from this web-site, allowing you to introduce **MPL for Windows** into your classroom.

- **Introducing the Innovative OptiMax 2000 Component Library**

The innovative **OptiMax 2000**, an object oriented component library, is revolutionizing how optimization modeling is applied to solve business problems. OptiMax 2000 is specifically designed to embed MPL models into object-oriented programming languages such as Visual Basic, VBA for Excel and Access from Microsoft Office, C/C++, Java, Delphi, as well as many web-scripting languages. This bridges, for the first time, the gap between model developers, whose expertise lies in developing models, and IT professionals, whose focus is mainly working on databases and building end-user applications. Embedding large-scale optimization models into business applications, quickly and easily, is now a reality!

Figure 2-9: MPL Modeling Language's component library for embedding in larger applications (http://www.maximal-usa.com).

In a distributed system, the generic process illustrated in Figure 2-6 needs the addition of a communication agent as shown in Figure 2-10. The separation of an instance from a model provides the valuable flexibility needed by the agent. Once a modeling language environment has translated a model and data to an instance, the MLE delegates further solver invocation process to the agent. The agent in turn sends the instance to the solver side through complex networking mechanisms. The agent knows everything about how to invoke a remote solver, which arguments to pass, and in which input formats to write the arguments.

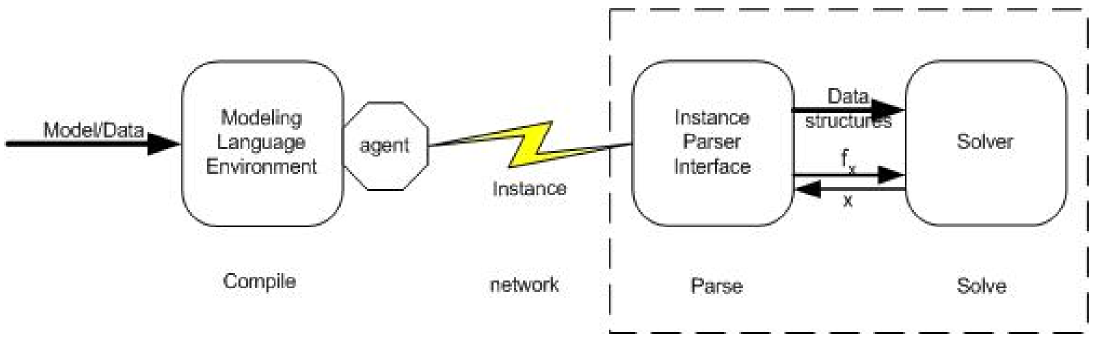


Figure 2-10: A generic process of instance generation and parsing.

Notice that the instance parsers usually reside together with the solver for purpose of computational efficiency. The communication between the two can be highly iterative, such as sending function and gradient values at each iteration.

The solution-finding process runs independently of the MLE. Thus the MLE developers can concentrate on high-level language design and parsing and the agent developers can concentrate on low-level communication. The communication can involve both distributed networking and invocation of local drivers. In either situation, the MLE and the agent do not need to remain an active process to wait for the response while the solver is running. For an MLE to retrieve a job result later, some mechanisms need to be established to keep the networking "stateful," i.e. matching a request from a later period (e.g. retrieval) with a request from a previous period (e.g. job submission).

The separation of communication agent also allows the agent to be used not only by MLEs but also others components in the optimization system. This is a key in the Optimization Services framework as many components can potentially be clients of others. For example a solver can be a simulation client to request function values from the simulation (see §2.8).

2.5 Optimization Server and Registry

In the mid-1990s, developers of optimization software began to use Internet services so that their users could try their software without installing it on their local computers. The initial optimization servers tended to use email or ftp to move problem files in one or both directions, with the associated Web pages advertising and explaining the service. Designs soon evolved to make use of Web Wide Web forms that were integral parts of server operations. The NEOS Server [29] for Optimization is the biggest and most successful realization to date of the optimization server idea. More is discussed in Chapter 3.

For the optimization servers, people's main interest is in the server side. The client is generally a browser (or a "thin" client) that sends problems and receives results via ordinary Web pages. Data are represented in HTML and sent through HTTP, while the server-side connections to solvers are usually via such mechanisms as Common Gateway Interface (CGI) scripts (Figure 2-11).

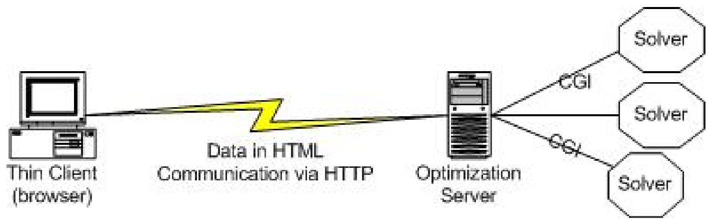


Figure 2-11: A typical optimization server with a "thin" client.

Technologies are needed to balance the work between client and server while maintaining or improving the quality of the client-server communication. For example more recent optimization servers (e.g. NEOS Kestrel [28]) allow model building on the local machine through a modeling language environment (or a "thick" client) and let the MLE conduct communications via such remote procedure call mechanisms as CORBA or XML-RPC. Such arrangements offer the greater stability and portability of established standards, together with the advantages of an object-oriented design (Figure 2-12).

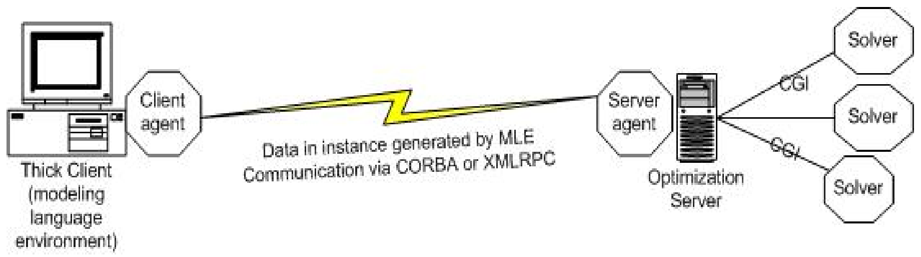

Client agent
Server agent
Solver
CGI
Solver
CGI
Solver
Thick Client (modeling language environment)
Data in instance generated by MLE Communication via CORBA or XMLRPC
Optimization Server

Figure 2-12: An optimization server with a "thick" client.

However, the fundamental server side architecture remains the same, and a user stills needs to access the optimization software through the optimization server. Therefore, an optimization server needs some way to protect itself from requests that can soak up all available resources. Some improvements have been achieved through different means. For example a time limit or a charge proportional to resource usage is sometimes imposed, but these mechanisms may discourage the use of an optimization server. More flexible strategies are implemented to take advantage of prior experience with different problems and solvers. One such strategy is to build a database of solver performance that can be automatically updated as optimization requests are carried out. An adaptive scheduler can then employ information from the database, together with specific customer preferences, in making initial allocations of computing resources to requests. For long-running jobs, such a scheduler can also monitor performance and take simple actions, such as increasing or decreasing a job's priority, moving a job to a faster or slower machine, suspending a job while querying its owner for instructions, and terminating a job.

The above strategies only achieve marginal improvements. A fundamental alternative to the traditional centralized optimization server is to replace the server with a "middle man" not to carry out the optimization jobs, but to provide connection information between clients and solvers.

The optimization Services framework takes this next step and introduces the concept of an optimization registry (Figure 2-13). An agent first contacts the registry for location information about solvers. Upon response from the registry, the agent takes a second step to contact the solver in a peer-to-peer mode. In both steps, data representation and communication follow the Optimization Services Protocol (OSP). Such an arrangement alleviates the burden of any traditional optimization server. Another direct result of the decentralization is that solver providers will correspondingly assume a more independent role to compete for customers' business. We envision decentralization as the future in distributed optimization for it provides an encouraging environment for the development of optimization systems and components. The Optimization Services registry is described in detail in Chapter 8.

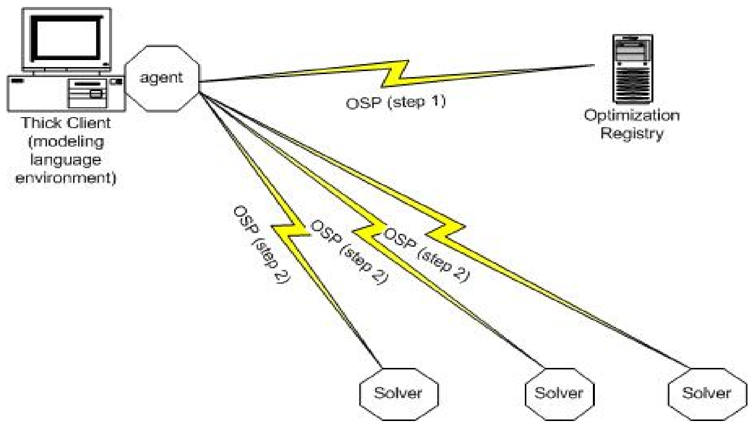


Figure 2-13: The optimization registry architecture.

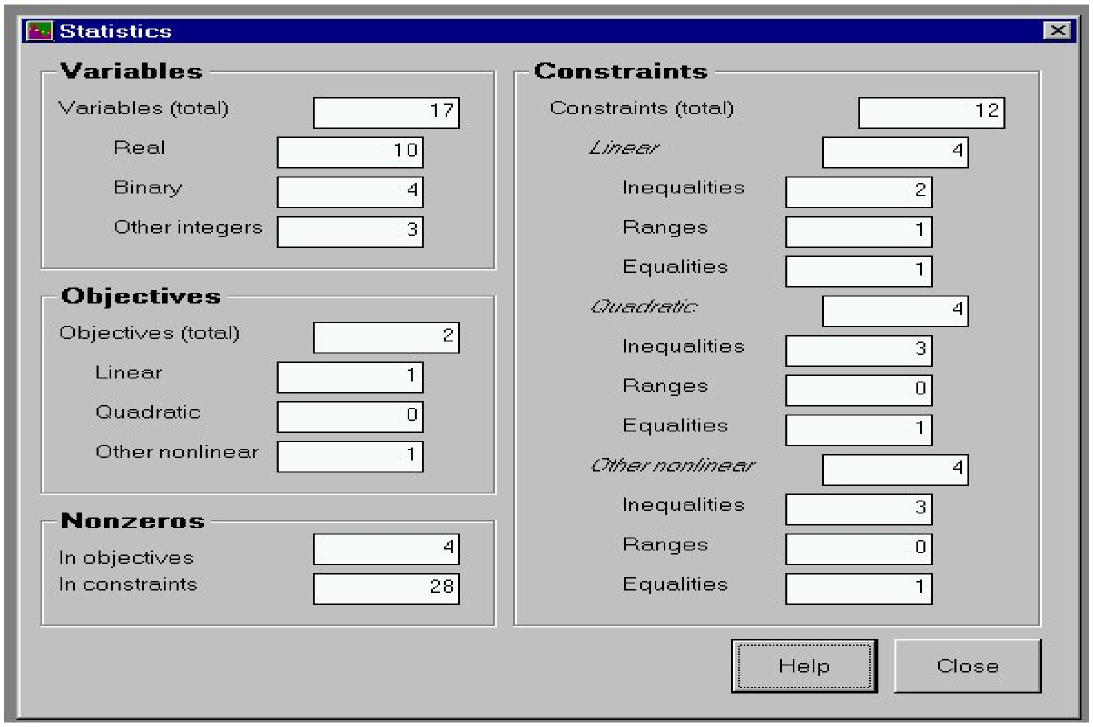
2.6 Analyzer

Programming languages such as C++ or Java are equipped with supporting tools, e.g. debuggers, in their corresponding integrated development environment (IDE[1]). Similarly analyzers are traditionally viewed as auxiliary components of a modeling language environment (or what we abbreviated as MLE[2]). They can be used in a preprocessing or a presolve phase of an optimization before the final instance is sent to a solver. If an optimization model is easy enough, it can potentially be solved by an analyzer without sending to a solver.

As shown in the Optimization Services framework, mainly with the introduction of the optimization registry and the corresponding discovery mechanism in a decentralized optimization system, analyzers become a highly integrated and critical part of the whole framework. The output of an analyzer can be used by a solver query engine to locate the appropriate solvers for the model analyzed by the analyzer. So unlike other auxiliary tools provided by a modeling language environment, analyzers are treated as a separate system component.

Conventionally, a solver query engine could communicate directly with modelers. But its usefulness would then depend on the willingness and ability of modelers to give correct lists of characteristics for the problems they want to solve. If problem characteristics could instead be automatically extracted from the modeler's submissions, the query engine could operate much more automatically and reliably. So as a basic requirement, analyzers should be able to detect the optimization types of a problem instance. If the format of an instance is well designed, in many cases conclusions can be immediately drawn by looking at the structure of the instance. For example if the instance format is separated into distinct linear and nonlinear parts, and if the analyzer does not detect the nonlinear part, it follows that the problem is a linear program. Figure 2-14 shows a basic mathematical program analysis report from the MProbe Analyzer [18], most of which can be detected by parsing the instance without further computation.

[1] An example of an IDE is Microsoft Visual Studio.
[2] The creation of the acronym "MLE" is related to the fact that people use "IDE" to stand for integrated development environment.


Statistics

Variables

Variables (total)	17
Real	10
Binary	4
Other integers	3

Objectives

Objectives (total)	2
Linear	1
Quadratic	0
Other nonlinear	1

Nonzeros

In objectives	4
In constraints	28

Constraints

Constraints (total)	12
Linear	4
Inequalities	2
Ranges	1
Equalities	1
Quadratic	4
Inequalities	3
Ranges	0
Equalities	1
Other nonlinear	4
Inequalities	3
Ranges	0
Equalities	1

Figure 2-14: MProbe Analyzer's basic analysis report.

For more advanced details, analyzers usually require further computational work such as studying nonlinear functions to discern their shapes in a region of interest. Such information is often crucial in finding the best-fit optimization solver. Again for these more advanced structure detection, there are problem characteristics that can be unambiguously determined by fast algorithms (e.g. network flow problems, quadratic problems) and there are more difficult ones that cannot be analyzed in an efficient and completely certain way (e.g. convex/non-convex problems). If the analyzer is used as a standalone tool, user interaction can help throughout the analysis process. But in the context of an optimization system, an analyzer is used as an intermediate procedure in a computerized process. There is no user available for extra input or hints. We want to make an automatic determination of problem characteristics, and of solver choice based on those characteristics. Tradeoffs between speed and reliability should be carefully considered.

From the Optimization Services framework perspective, there are more requirements on the analyzers in terms of communication with other components. For example, after a modeling language environment converts a model into an instance, the MLE will likely send the instance to the analyzer before contacting a solver. So the analyzer should take the same instance as any solver on the optimization system.

After an analysis is carried out, the analyzer's output is converted into a query that feeds into an optimization registry's solver search engine. Such a process requires a standardized analyzer output that can be converted into the query understood by the registry. An OS-compatible analyzer should take OSiL as its input and generate output in OSaL, the Optimization Services analysis Language. See Chapter 6 for more information on these representations.

2.7 Solver

Optimization solvers, or solvers, are algorithms designed and implemented to find optimal solutions to specific optimization problems. A solver takes a low-level instance of an optimization problem and produces another low-level representation of the optimization result. Any solver on an Optimization Services system should take the Optimization Services instance Language (OSiL) as its input and generate the Optimization Services result Language (OSrL, see Chapter 6) as its output.

A solver, however, does not usually carry out computation directly on the instance representation. Rather an instance reader parses the input into the internal objects or data structures required by the solver's algorithm. Optimization Services provides libraries for reading the standard OSiL input (OSiLReader) and writing the OSrL output (OSiLWriter), as shown in Figure 2-15.

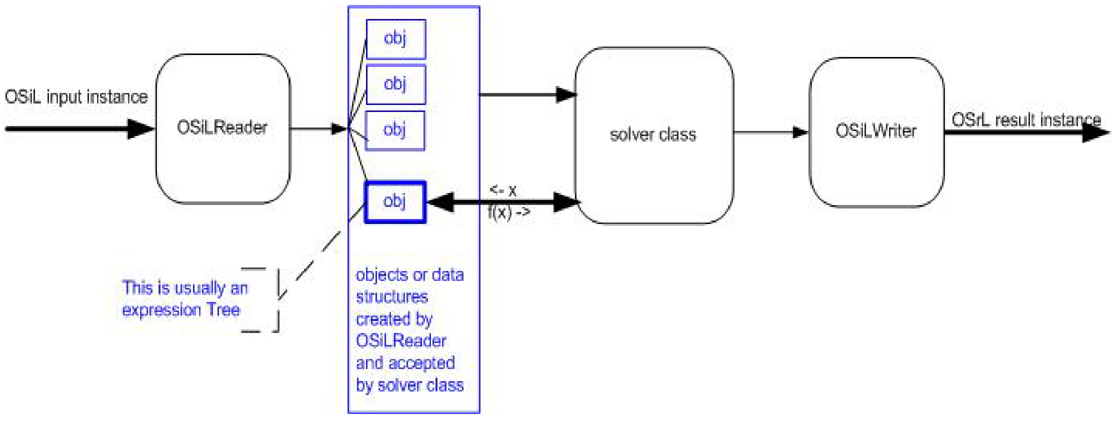

OSiL input instance
OSiLReader
obj
obj
obj
obj
<- x
f(x) ->
solver class
OSiLWriter
OSrL result instance
This is usually an expression Tree
objects or data structures created by OSiLReader and accepted by solver class

Figure 2-15: A generic input and output process of an Optimization Services compatible solver.

Not only there are optimization solvers of many types, there are also usually large differences between solvers of similar types in performance in terms of speed, numerical stability, and adaptability to computer architectures. As solvers are the ultimate need of an optimization system user, the quality of the solvers directly determines the success of the entire optimization system. This is especially important in a decentralized and automated architecture like Optimization Services. To ensure that the OS registry only send addresses of the solvers that are of reasonably high quality, regulations are imposed when an OS-compatible solver is to be registered in the OS registry. Special OSP protocols are also designed to make sure a solver is well-described, live, reliable, and robust. Information about solvers that is kept in the OS registry includes:

- entity information that is reported by solver developers at registration., e.g. solver types, solver locations, maximum problem size (Optimization Services entity Language, OSeL, see Chapter 8);
- option information that is reported by solver developers at registration, e.g. algorithm directives like maximum time, output listing (Optimization Services option Language, OSoL, see Chapter 6);
- real-time process information that is either automatically reported by the registered solver software ("push") or detected by the OS registry ("pull"), e.g. whether the solver is live online, how many optimization jobs are in the solver queue (Optimization Services process Language, OSpL, see Chapter 8);
- benchmark information that is produced separately by auxiliary software tools designated by the OS registry, e.g. general solver ratings (Optimization Services benchmark Language, OSbL, see Chapter 8).

Solver development in some areas like stochastic programming is lagging due to the lack of a good representation. It is the hope of Optimization Services that by introducing a set of universal standards, the project can help facilitate solver development in such areas.

2.8 Simulation (Function Evaluator)

A conventional iterative hill-climbing or evolutionary searching algorithm such as the Newton-based nonlinear optimization method generates a series of trial solutions, or iterates, and requires the values of the nonlinear objective and constraint functions only at each iteration. These nonlinear solvers typically require a user routine, in a programming language such as C or Java, that takes an iterate as input and returns the corresponding objective and constraint function or derivative values as output. We call such routines function evaluators. The calling conventions for user-supplied function evaluators differ from one solver to another. The function evaluators are called repeatedly throughout the optimization process and directly affect the speed of the employed algorithm.

Usually function evaluators reside locally with the solver that calls them and there are explicit mathematical formulas for the objective and constraint functions. In reality, such requirements cannot always be met due to reasons like the following:

- Function evaluators are coded in some general software, usually called a model service that calculates function values as well as doing other things. It is possible that the final objective and constraint functions consist of calculations from multiple model services.
- Many model services are located remotely. Local copies cannot be easily duplicated due to various reasons. For example, the model service may be tightly coupled with a database system.
- Some model services are so complicated that no simple mathematical representation can be formulated.
- Some of the model services are proprietary and thus their formulas cannot be revealed.
- Most importantly, some model services do not return results instantaneously. The delays make it difficult to integrate the model services into the optimization solver.

In situations like these, we refer to the function evaluators as simulations.

Optimization via simulation (or simulation optimization) is usually thought of as optimization over performance measures from outputs of *stochastic* simulations. But the simulations that we refer to here can be deterministic as well. As a matter of fact, from the Optimization Services framework point of view, function evaluators that specify a set of inputs (can be none), a set of outputs (at least one), and the invocation address are considered a simulation as illustrated in Figure 2-16. The function form is usually hidden inside the simulation. We also call such simulations as "black-box simulations," and call optimization via simulation "black-box optimization."

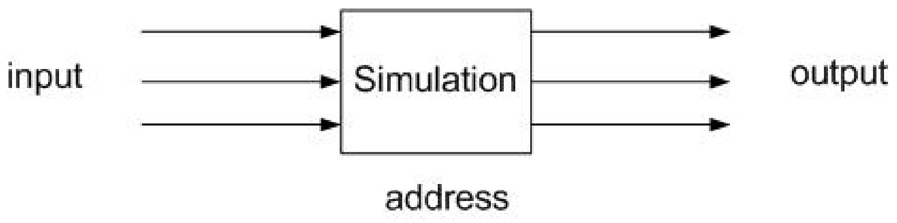


Figure 2-16: Three requirements of a simulation: input, output and address.

If a model requires an optimization via simulation, the simulation function has to contain information regarding the three required components, and it is the modeling language's job to provide natural features to support simulation definitions.

In an optimization problem, an objective or constraint function is of the form $y = f(x_1, x_2, \cdots, x_{n)}$, where the function input is a vector and the function output is a single scalar ($R^n \to R$). Our simulation box in Figure 2-16 is more general than a function since the simulation can have multiple outputs ($R^n \to R^m$). So a modeler has to specify which output or combined value of several outputs is to be taken as the function value (y).

For example suppose the objective function in (2-3) cannot be written in a closed form.

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & x_1^2 + 2x_2^2 \\ \text{subject to} \quad & 2x_1 + 3x_2 \geq 9 \\ & \mathrm{x}_1 \geq 0,\ x_2 \geq 0 \end{aligned} \tag{2-3}$$

Instead the function is calculated from the simulation at the address http://somesite.com/mySimulation. The black box simulation – "mySimulation" – at this site looks like the schema shown in Figure 2-17.

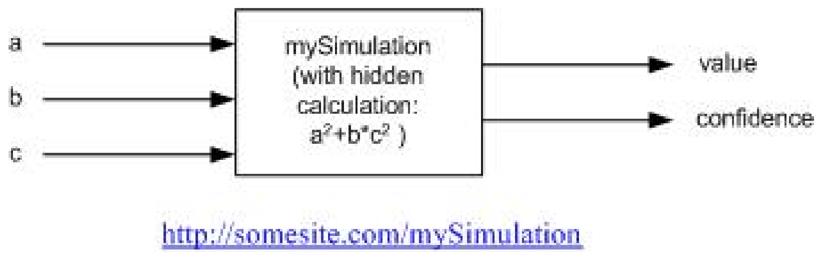

a
b
c
mySimulation (with hidden calculation: a²+b*c²)
value
confidence
http://somesite.com/mySimulation

Figure 2-17: The schema of a simulation called "mySimulation," which hides its internal calculation.

So "mySimulation" takes three inputs: "a", "b," and "c" and generates two outputs: "value" and "confidence." The internal calculation $a_1^2 + b * c_2^2$ is hidden from the user. The objective function $x_1^2 + 2x_2^2$ in (2-3) shows that we are only interested in changing two of the input parameters: "a" and "c," as variables and keep "b" constant at 2. Also the objective function ignores the "confidence" output and only takes "value" output. One way to rewrite the model in (2-3) as a simulation optimization is shown in (2-4).

$$\begin{aligned}
&\min_{x} imize \quad mySimulation(x_1, 2, x_2) \\
&subject\ to \quad 2x_1 + 3x_2 \geq 9 \\
&\qquad\qquad\quad x_1 \geq 0, x_2 \geq 0 \\
\\
&mySimulation\{ \\
&address = http://somesite.com/mySimulation \\
&input: \\
&a \\
&b \\
&c \\
&output: \\
&value + confidence * 0 \\
&\}
\end{aligned} \tag{2-4}$$

We replace the objective function with the simulation $mySimulation(x_1, 2, x_2)$, so we pass in three inputs. In the simulation definition part of the model we specify the three simulation requirements: simulation address, input and output. The inputs – a, b, and c – are listed in order and in this example take the values x_1, 2, and x_2 respectively. In a sense, "mySimulation" is no different from a user-defined function, except that in a user-defined function we write down the actual form of the function, whereas in "mySimulation" we write down the three requirements.

There are essentially two types of input in the simulation: constant input (e.g. $b = 2$) and variable input (e.g. $a = x_1$). Constant inputs remain the same throughout the simulation optimization. Variable inputs are decided by the solver and can change at each optimization iteration. This means at each iteration, the solver may call the simulation at the address by providing a new set of inputs and obtain a new function value from the simulation outputs.

Another thing to notice is that since the simulation engine is a black box, there are usually no derivative values provided. Of course the solver can numerically compute a derivative by invoking the simulation twice at two separate points and calculate a finite difference. But there are two main reasons not to do that. One is due to the long communication time just mentioned.

The other is that the behavior of the simulation function is usually unknown. For example it can be discrete. So it may not be appropriate to use a solver that requires derivatives on a simulation optimization model. These issues and others are discussed in our second example in Chapter 3.

From the Optimization Services framework point of view, if a simulation like "mySimulation" is to be invoked by an OS-compatible solver, its input ("a" "b" and "c") and output ("value" and "confidence") have to be put in a standard format. Such a standard format is specified by the Optimization Services simulation Language (OSsL). Of course when the model in (2-4) is translated by a modeling language into the Optimization Services instance Language (OSiL), OSiL should support features of simulation optimization and be able to embed OSsL.

CHAPTER 3 OPTIMIZATION SYSTEM IMPLEMENTATIONS

In this chapter we discuss two distributed optimization systems – the AMPL modeling system using remote NEOS solvers and Motorola's Virtual Prototyping (VP) Intelligent Optimization System. Issues in designing and implementing the two systems are discussed. The two different optimization systems provided us with the initial motivations for a more general design and framework. Though architecturally different, these two optimization systems can be unified under our Optimization Services framework (see Chapter 5).

3.1 AMPL and Network Enabled Optimization System (NEOS)

AMPL is an algebraic modeling language developed by Robert Fourer, David Gay and Brian Kernighan [48], in the mid 1980's. For a detailed description, refer to [49] and the AMPL Web site at http://www.ampl.com. General concepts of modeling languages and modeling language environments (MLE) are discussed in Chapter 2.

NEOS stands for Network Enabled Optimization System. The optimization system resides at Argonne National Laboratory. The project was started in 1995. NEOS system provides optimization software and services through Web, email, socket-based access and Remote Procedure Call. Development of the NEOS system has been mainly supported by the Optimization Technology Center (OTC) team, which consists of research scientists, professors, post-doctorate associates and graduate students from Northwestern University and Argonne National Laboratory. With the development of NEOS, emphasis in recent years has shifted to Internet, distributed computing, and problem-solving environments. For more information check the OTC Web page at http://www.ece.nwu.edu/OTC and the NEOS Web page at http://www-neos.mcs.anl.gov.

3.1.1 Standalone AMPL architecture

Figure 3-1 shows the standalone system architecture of AMPL with a local solver.

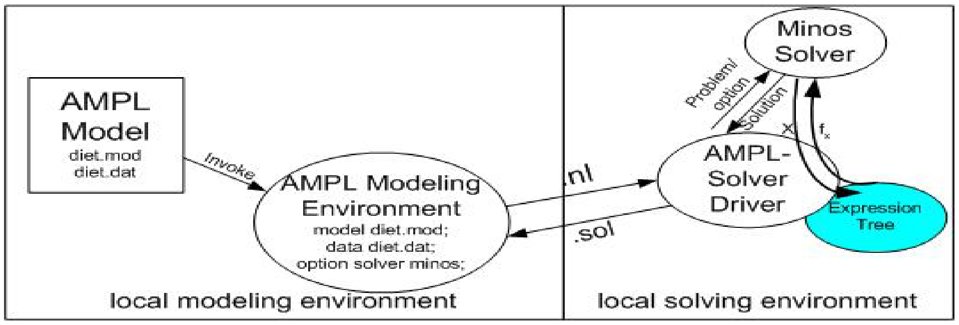


Figure 3-1: Standalone AMPL-Solver architecture (local).

A user begins in a command environment. After starting AMPL, the user sees the AMPL's prompt:

```
ampl:
```

The user communicates with AMPL in two ways: by typing commands, and by setting options that influence subsequent commands. In Figure 3-1, the user invokes a previously constructed model, which usually consist of a ".mod" file and a ".dat" file. The ".mod" file is AMPL's abstract algebraic representation of an optimization problem. The ".dat" file contains specific values of data that define a particular problem. AMPL then combines the ".mod" and ".dat" file and converts them into a low level optimization instance representation in a ".nl" format. For the diet example in Figure 2-4, that corresponds to a user typing at the prompt:

```
ampl: model diet.mod;
ampl: data diet.dat;
```

and a diet .nl file is created. Then as soon as the user types:

```
ampl: option solver minos;
ampl: solve;
```

the ".nl" instance file is sent to the Minos solver through the AMPL-Solver Driver, a piece of interface software between the AMPL modeling language and the hooked solver.

For nonlinear objectives and constraints, the AMPL-Solver Driver contains an expression tree to calculate function values, and sometimes gradients and Hessians as well. Throughout the optimization iterations, the solver calls function evaluation routines that use expression trees and obtains function values (f_x) from the expression trees by providing the current variable values (x), all through the AMPL-Solver Driver. Due to the hill-climbing nature of most nonlinear algorithms, solvers need to know the function values only at specific points.

Finally optimization results are sent back by solvers. The AMPL-Solver Driver interface converts the results into a ".sol" format and the AMPL modeling provides convenient commands for viewing the solutions.

3.1.2 AMPL-NEOS architecture

The NEOS Server currently provides more than 50 optimization solvers through many types of networking interfaces. Users send requests to a central server at Argonne National Lab, but optimization solvers can be on any workstation on the Internet that is connected with NEOS through a standard setup [27].

NEOS's Kestrel interface provides a mechanism that enables remote optimization from within the AMPL modeling environment. For example to solve the diet problem using the remote Minos solver on NEOS, the user would type at the prompt:

```
ampl: model diet.mod;
ampl: data diet.dat;
ampl: option solver kestrel;
ampl: option kestrel_options 'solver=minos';
ampl: solve;
```

So much remains the same for the user. From the user's perspective, Kestrel is just another "solver" and the real solver provided through Kestrel just appears to be a further option under the "Kestrel solver." There is no networking jargon and all the communication details are hidden by the AMPL-Kestrel interface. The user won't notice whether the model is solved locally or on a network. When the "solve" command is executed, AMPL delegates the Kestrel agent to handle all the networking protocols such as CORBA or XML-RPC to send requests and receive responses. In fact even AMPL does not know anything about the networking. To be more exact, the Kestrel communication agent hides everything from the modeling language environment. For a detailed description of Kestrel, refer to the paper by Dolan et al. [28].

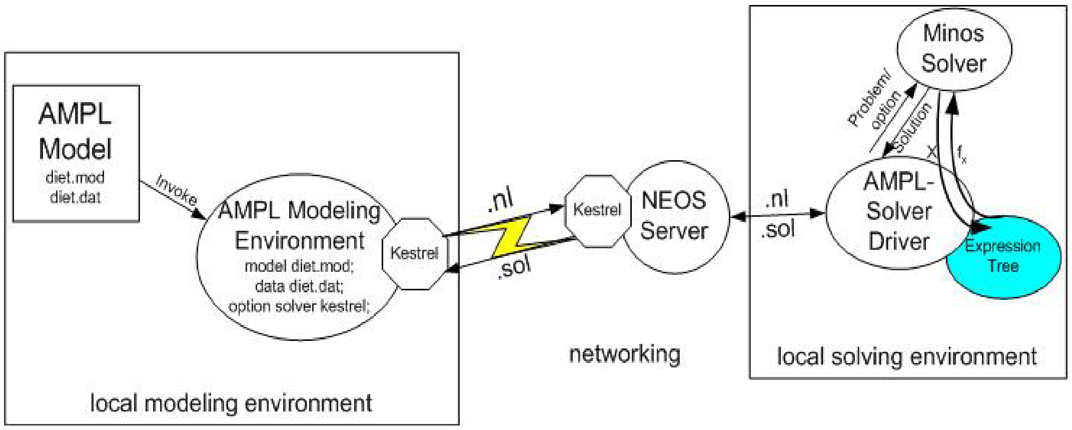


Figure 3-2: AMPL-NEOS Architecture through Kestrel.

As a result, the locally running AMPL modeling language environment can have access to a wide variety of the remote NEOS solvers. Moreover, optimization results are provided within the AMPL modeling language environment so that users do not need to parse a result file to use the generated answers. From the Kestrel perspective, we call AMPL a "thick" (or "fat") network client, as it takes away from the NEOS server many of the optimization responsibilities, such as compilation, preprocessing, and presentation. Otherwise, if an optimization request is submitted via an Internet browser, the NEOS server would do nearly all the work. In this case, we call the Internet browser a "thin" network client.

In terms of architecture, there are no major differences between the standalone AMPL and the AMPL-NEOS system shown in Figure 3-2. They are essentially the same at the two ends of the optimization process, that is, the local modeling environment and the remote solving environment. The AMPL-NEOS system adds a Kestrel client agent and a Kestrel server agent between the AMPL modeling environment and the NEOS server and connects the two Kestrel interfaces with either a CORBA or an XML-RPC interconnection. The NEOS server then further relays the information to and from the solver. The ".nl" and ".sol" files are transmitted between the Kestrel agents rather than through a local interface.

3.1.3 AMPL-NEOS optimization problem representation issues

The large number of optimization types serves as a barrier as well as a motivation toward input format standardization. Neither AMPL nor NEOS precludes any text or binary file format to be passed to a solver. If there are N solvers (or other software such as analyzers) on NEOS, then N different drivers are required to read the AMPL nl format. Besides AMPL, there are many other algebraic modeling languages and numerous GUI environments with prewritten optimization models underneath. Suppose there are M modeling languages and GUIs and N solvers, then M × N drivers are required for complete interoperability over NEOS (Figure 3-4).

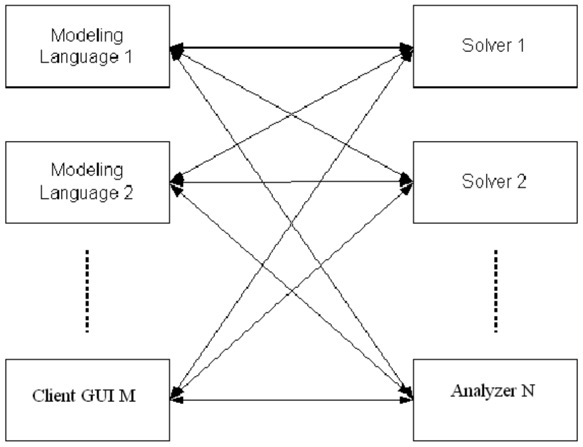


Figure 3-3: M x N drivers needed by M modeling languages (or GUIs) and N solvers (or analyzers).

Even a cursory look at the NEOS Server's list of solvers (Figure 3-5) reveals the babble of input formats recognized by current optimization software. There are more than a dozen different low-level formats recognized by one or another solver in the NEOS lineup, including MPS [86] formats for linear and integer programming, SMPS [10] extensions to the MPS format for stochastic programming, SIF [24] for nonlinear programming, formats such as SDPA specific to semidefinite programming, and DIMACS min-cost flow and other formats for network linear programming. Other solvers recognize input programmed as functions in various languages including FORTRAN, C, C++, and Matlab.

To the extent that there is any greater degree of standardization, it is through the use of input written in higher-level optimization modeling languages. Although NEOS works with the GAMS [12][16] and AMPL [48][49] languages, however, each of these supports only some of the available solvers. An arrangement that applies AMPL solvers to GAMS models is at best a stopgap, requiring execution of both the AMPL and GAMS compilers.

XML has emerged over the past few years to guide the design of standard forms for Internet communication of all kinds. The XML Schema described in Chapter 4, for example, can be used to enforce a standard for optimization and can grow in a well-defined way to accommodate new problems types. This contrasts with the current situation, where for example, parsers for the MPS Standard [86] vary in details between implementations, interpreters of the SMPS standard [10] are even more varied, and no proposal for nonlinear extensions (see, for instance [60]) has caught on at all.

In our Optimization Services framework, we propose a new low-level XML format (Optimization Services instance Language – OSiL, see Chapter 6) that currently supports all of the problem types supported through the NEOS Server, with sufficient flexibility to be extended to new types. Using the standard representation of an instance, only M + N drivers are required for complete interoperability (Figure 3-4).

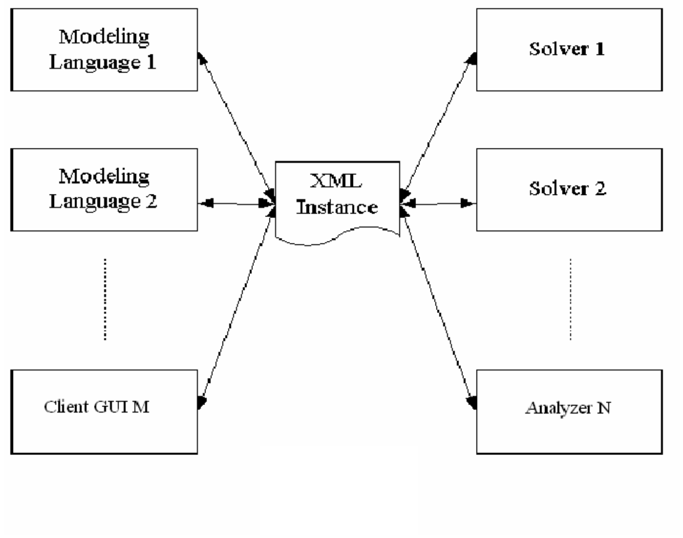
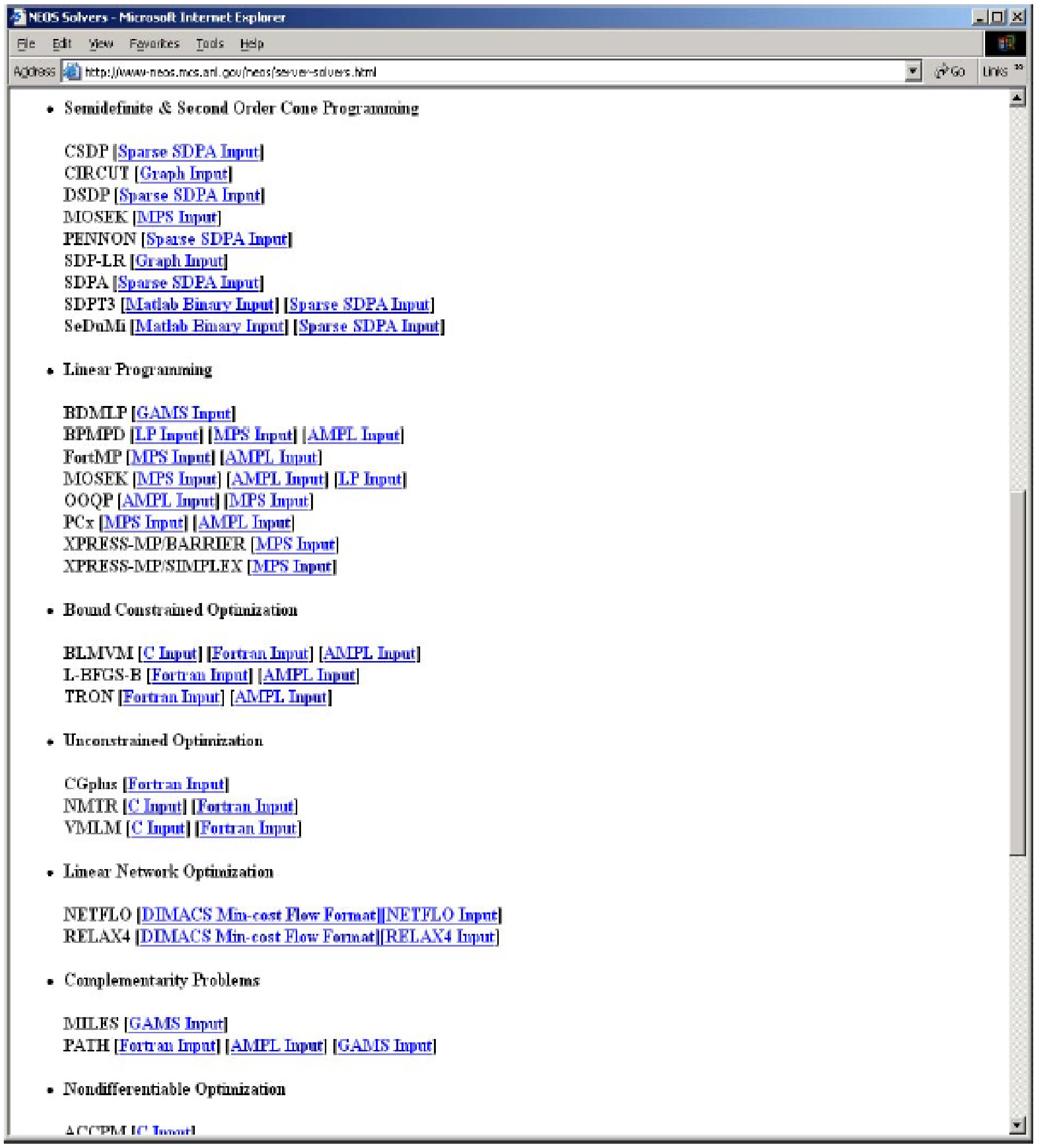


Figure 3-4: `M` + `N` drivers needed by `M` modeling languages (or GUIs) and `N` solvers (or analyzers) with a standard XML instance.

OSiL addresses problems that are not application-specific, but are as specialized as stochastic recourse problems or as generalized as nonlinear-constrained programs. The adoption of such a format by solvers will make them more universally available through Internet services. The adoption of the same format by modeling languages will enable solvers to more readily support many languages. The overall effect will be to decouple language and solver choices, letting the user pick the best tools for any project.


NEOS Solvers - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://www-neos.mcs.anl.gov/neos/server-solvers.html
Go Links

- Semidefinite & Second Order Cone Programming

  CSDP [Sparse SDPA Input]
  CIRCUT [Graph Input]
  DSDP [Sparse SDPA Input]
  MOSEK [MPS Input]
  PENNON [Sparse SDPA Input]
  SDP-LR [Graph Input]
  SDPA [Sparse SDPA Input]
  SDPT3 [Matlab Binary Input] [Sparse SDPA Input]
  SeDuMi [Matlab Binary Input] [Sparse SDPA Input]

- Linear Programming

  BDMLP [GAMS Input]
  BPMPD [LP Input] [MPS Input] [AMPL Input]
  FortMP [MPS Input] [AMPL Input]
  MOSEK [MPS Input] [AMPL Input] [LP Input]
  OOQP [AMPL Input] [MPS Input]
  PCx [MPS Input] [AMPL Input]
  XPRESS-MP/BARRIER [MPS Input]
  XPRESS-MP/SIMPLEX [MPS Input]

- Bound Constrained Optimization

  BLMVM [C Input] [Fortran Input] [AMPL Input]
  L-BFGS-B [Fortran Input] [AMPL Input]
  TRON [Fortran Input] [AMPL Input]

- Unconstrained Optimization

  CGplus [Fortran Input]
  NMTR [C Input] [Fortran Input]
  VMLM [C Input] [Fortran Input]

- Linear Network Optimization

  NETFLO [DIMACS Min-cost Flow Format][NETFLO Input]
  RELAX4 [DIMACS Min-cost Flow Format][RELAX4 Input]

- Complementarity Problems

  MILES [GAMS Input]
  PATH [Fortran Input] [AMPL Input] [GAMS Input]

- Nondifferentiable Optimization

  ACCPM [C Input]

Figure 3-5: Part of the NEOS Server's list of solvers and problems formats.

3.1.4 AMPL-NEOS optimization communication issues

Solving large optimization problems may require computational power far beyond what regular desktop workstations can offer. Due to increasing computing and networking power, typical users now have access to more resources than ever before. When NEOS project was begun in 1995, the Web was just beginning to come into widespread use. At first the NEOS supported only low-level file formats and FORTRAN programs, and input only via e-mail; successive enhancements provided the much more powerful and convenient communication options available today. To ensure reliability of the Server, this work used early and relatively mature standards, such as Web forms, TCP/IP sockets for the NEOS Submission Tool (see http://www-neos.mcs.anl.gov/neos/server-submit.html) and CORBA for the Kestrel interface [28] (see also http://www-neos.mcs.anl.gov/neos/kestrel.html).

Now a user can submit an optimization problem to NEOS via any of the above-mentioned interfaces. The NEOS Server then locates the specified solver in its data bank and schedules the user's optimization job request on a remotely connected computation resource that is currently available and equipped to process jobs of the given type. Registered solver providers must provide software and sometimes hardware. Solver administrators have to write implementations to check data consistency, solve optimization problems, and return appropriate results. The NEOS Communication Package, a Perl application, is provided to facilitate communications between the NEOS Server and solver computers. The latest version of NEOS has been re-implemented in the more powerful Python programming language and all the server-solver connections have been updated accordingly.

But still, the current NEOS Server has not fundamentally addressed the communication difficulties of large-scale optimization with respect to the combinatorial effect of the plethora of solver types, interface choices, and connection to modeling languages. As the NEOS Server has evolved along with the Web and the Internet, it is limited to some degree by early design decisions.

We are now seeing a new generation of standards that make Web services more flexible in design and easier to build and maintain. With tools such as SOAP, WSDL, Web services registries (see Chapter 4), we have developed a general and flexible Optimization Services environment for developers to make their software easily accessible on the Internet.

The effects of Optimization Services on NEOS are multifaceted:

- The NEOS server and its connected solvers will communicate using the Optimization Services framework, e.g. using standard representation for data communication.
- External optimization submissions can still be kept as flexible as possible and may become even more flexible. At least one more networking mechanism will be provided, i.e. the networking based on the Optimization Services Protocol (OSP). That means NEOS will add an interface so that it can be invoked exactly as what's specified by the Optimization Services hook-up Language (OShL, Chapter 7). It will also accept OSiL as a standard input, and may gradually deprecate the other formats.

- The entire Optimization Services system over the Internet can just be viewed as a new decentralized NEOS. In effect the old NEOS will become another OS-compatible solver in the new system. The "NEOS server" can then solve more types of optimization by delegating the job further to different solvers behind it. We therefore regard the old NEOS as a "meta-solver" registered on the new Optimization Services system.

The Optimization Services framework is also complementary to the design of OSI (Open Solver Interface), a standard procedural interface to solvers currently being implemented under the auspices of the COIN-OR project [23]. OSI provides a way of calling optimizers directly from applications, whereas OSiL is an optimization instance that can be communicated to solvers in a variety of ways. The interfacing between Optimization Services and OSI is illustrated below in Figure 3-6.

Suppose an OSiL instance (1) comes in through the Internet from another Optimization Services component. From the OSiL sender's view, an OS solver is being invoked as it accepts the standard OSiL as an input. Within the OS solver, the OSiLReader (2) first parses the OSiL instance into a set of generic objects/data structures (3) and in turn these objects/data structures are transformed into the Open Solver Interface (4), and any COIN-OR compatible solver can be hooked up behind the interface. As the OSiL is a standard instance, only one OSiLReader needs to be written to read the instance and as the Open Solver Interface is also a standard interface, both OSiLReader and Open Solver Interface can be provided in *one* library. All that a solver developer needs to do is to include this library to resolve all interface or format issues.

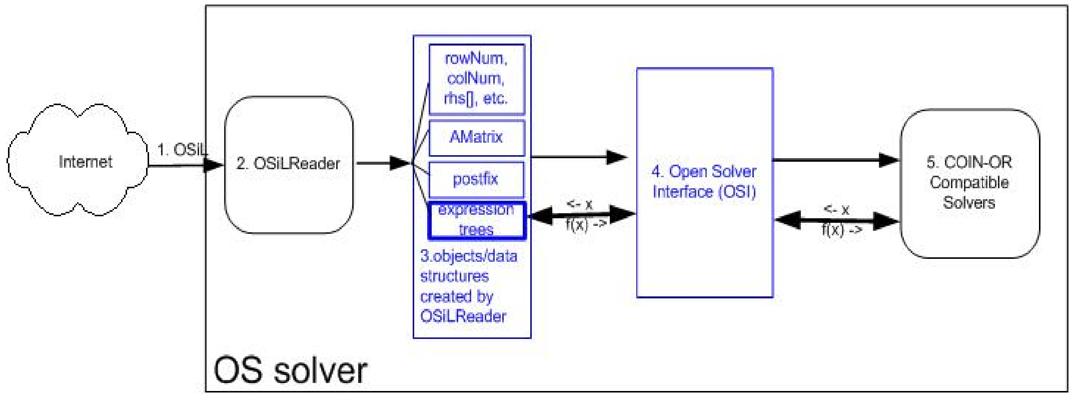

Internet
1. OSiL
2. OSiLReader
rowNum, colNum, rhs[], etc.
AMatrix
postfix
expression trees
3.objects/data structures created by OSiLReader
<- x
f(x) ->
4. Open Solver Interface (OSI)
<- x
f(x) ->
5. COIN-OR Compatible Solvers
OS solver

Figure 3-6: The Optimization Services (OS) – Open Solver Interface (OSI) connection.

The success of Optimization Services will promote the work of COIN-OR and in turn the wide acceptance of the OSI interface will allow more solvers to be easily hooked into the

Optimization Services system. To take further advantage of this, we are extending the OSI interface to more optimization types.

3.2 Motorola Labs Multidisciplinary Intelligent Optimization System

This section describes another optimization system implementation. The project finished before the first conception of Optimization Services, so it may seem a bit detached from the central theme of this thesis. However, it was through four years of experiences with this project, that we gained significant insights in designing Optimization Services. Also it provides another perspective in optimization systems used on an intranet within a corporation.

The Motorola Virtual Prototyping (VP) optimization system is a critical step in a multistage effort to develop and deploy enterprise-wide tools that drastically reduce the cycle time for new designs and technologies. The principal feature of this effort is the integration of design and development processes from various disciplines. The goal is to plan, design, construct and manage knowledge-based systems for the transfer, application and execution of highly specialized knowledge. The main economic benefit is to be realized in terms of reduced engineering effort for new product ideas, improved compliance with standard design and development rules, and more optimal design and development trade-offs.

3.2.1 Dataflow and knowledge flow

In 2001, we designed a nonlinear optimization solver using a modified feasible direction algorithm for the Virtual Prototyping (VP) group [101], led by Thomas Tirpak at the then Motorola Advanced Technology Center (MATC). Solvers of other math programming types were subsequently developed. In the second year all the solvers were integrated into the distributed Virtual Prototyping system as a new VP service. The service was intended to solve most of the general large scale nonlinear optimization problems with discrete variables encountered in Motorola's engineering design. This optimization service has since proved to be of value to the Motorola engineering community. It has been applied in areas such as printed wiring board design and embedded passives design and has helped achieve significant design and engineering cost reductions.

At the beginning, the VP optimization service was only applied on a single engineering domain simulation model. This means the entire objective function was calculated by one model service that is usually located on the same machine as the optimization engine. As the system evolved, an objective function can consist of metrics from multiple and distributed model services on remote machines, as illustrated in Figure 3-7.

$$\begin{aligned} &\min \quad f(y_1(x), y_2(x), \ldots, y_n(x)) \\ &\text{s.t. } g_j(x) <= 0 \text{ for all } j = 1,\ldots,n \end{aligned}$$

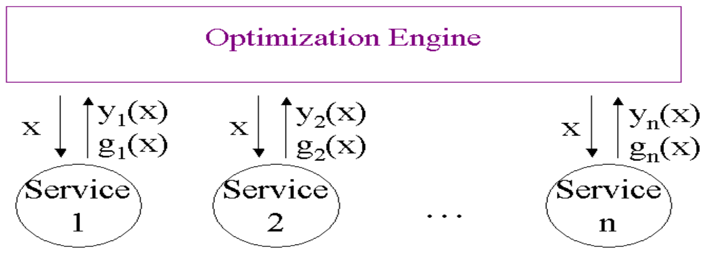


Figure 3-7: Dataflow of optimization with metrics calculated from distributed services.

The objective f of the optimization service is comprised of many y_i metrics that are calculated from the corresponding model service i. The variable set x is shared among all the services. The arrows indicate flow of information at each optimization iteration. At the higher level is the optimization engine that sends new variable values to individual model services. At the lower level are the model services that supply objective values $y_i(x)$ and constraint values $g_i(x)$. If there is an objective function, it is possible to generate an improved solution just by knowing the current function values.

This dataflow also mimics the real world knowledge flow to automate manual processes and reduce cycle time. Knowledge derives originally from customers, who express it in the form of specifications of their needed products. The specifications are likely to encompass a wide area of engineering domains such as electronic engineering, mechanical engineering, material engineering and manufacturing. These specifications are distributed to the corresponding engineering groups for proof-of-concept designing and prototyping. Without the VP Multidisciplinary Intelligent Optimization System, the engineering solutions that have been developed in a separate manner are finally combined into a complete prototype in a manual way. If the solutions have a so-called “technical interface” conflict, then they are sent back for reengineering. Such a process goes on for several rounds mainly in a time-consuming trial and error mechanism with many inter-departmental or group meetings. This manual process ends when the final complete product is free from design conflicts.

With the establishment of the automated Virtual Prototyping system, the optimization engine takes the responsibility of coordinating the design solutions that originate from separate departments, finds a feasible solution and possibly optimizes within the feasible region to find the best combination of designs. The data flow in Figure 3-7 thus leverages on knowledge flow in the real engineering world. The higher-level optimization engine replaces the “combination” process of inter-departmental meetings. The lower-level model services replace the “reengineering” process of individual departments.

3.2.2 Initial modeling of computational complication in test bed

Before we implemented our robust design (next section) in the VP system, we built a simplified test bed to simulate the true Motorola system. In our initial test bed modeling (Figure 3-8) of the data flow shown in Figure 3-7, different optimization solvers are extended from a standard optimizer interface. All solvers interact with optimization problems through a common interface. The optimization problem interface is connected with a simple accelerator. The accelerator is to simulate the behavior of remote services, and provide estimated function values to the solvers locally. Each local optimization problem has a corresponding remote service connected with it. The real computations are carried out on the remote services and the local optimization problems serve as bridges between the remote services and the optimization solvers.

Model services are simulated with relatively simple functions. All the services are initiated in separate process threads. Though the simple function value calculations take no time to complete, different time factors are realized by forcing each process thread to wait according to the parameters specified for each service. To speed up the optimization process, all the time units are scaled down to milliseconds.

The time for a model service to execute may depend on a variety of factors, e.g., the computer on which the service is running, the time of the day, the complexity of the scenario represented by the inputs (x), etc. Services may be unavailable at certain scheduled or unscheduled times. There may be a delay in transmitting the inputs to the services or the outputs from the services both due to networking and model service computation time.

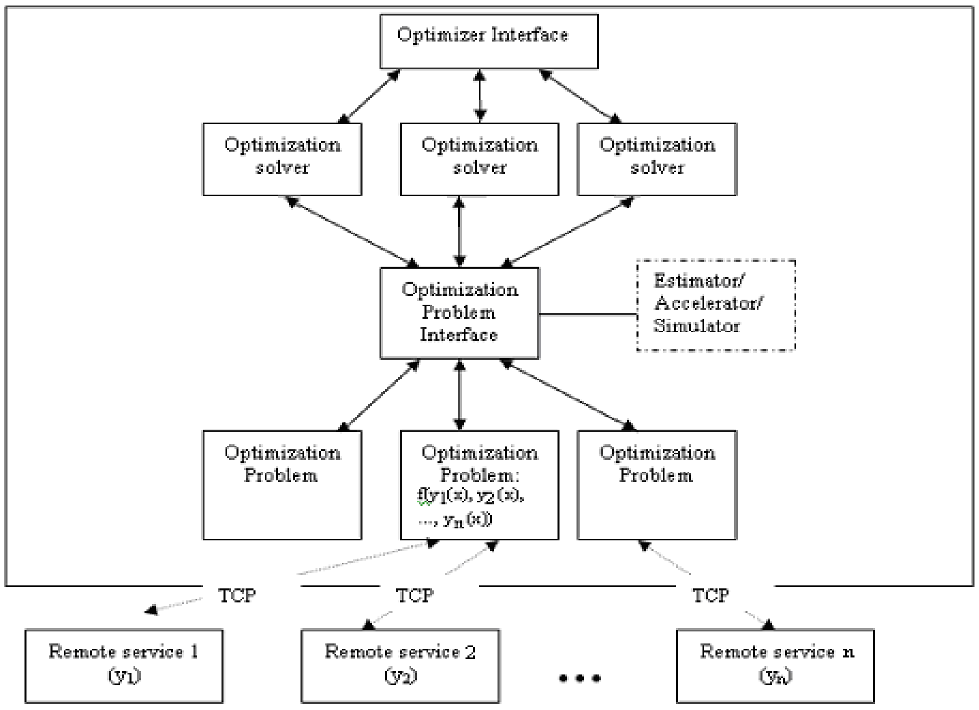


Figure 3-8: Initial modeling of optimization with metrics calculated from distributed model services.

The computational performance of the model services in Motorola's Virtual Prototyping System can be characterized by three factors: service time, server load factor, and down time. Down time includes when the server computer is down, when there is a bug in the model service software, and when there are difficulties running the service for a given set of inputs (x). Communication time between the optimization engine and remote services is insignificant.

An immediate issue is that an optimization can easily take thousands of iterations. If each iteration takes a long time due to the above factors, it may become impractical to solve the whole optimization within a reasonable amount of time.

Moreover when engineers design and construct their model services, they do not know that their models will later be used as parts of an optimization system. Therefore, these model services usually do not provide gradient information. The optimization solver often has to be based on an algorithm not using derivatives.

Benchmarking has been conducted on different optimization algorithms, and a method based on Powell's algorithm [95] with quadratic step length estimation was selected in the initial modeling system. Our initial tests have proceeded as follows.

Benchmark problems are first tested with their objective functions unbroken (i.e. the entire objective function) and statistics are collected for comparison with later tests in the distributed system. Then the objective functions are arbitrarily divided into several parts and put on different machines communicating with the TCP/IP networking protocol. The server, where the optimization solver is located, sends the current variable values to each machine for a functional evaluation and waits till it gets all the responses. It then gathers the functional values and integrates them into a whole function for the optimization solver to conduct the next iteration. Primitive estimations such as quadratic fitting and smoothing splines are used to approximate the functions. Acceleration is achieved by evaluating from the approximated functions rather than getting real values from the remote services.

Through our initial modeling, we have shown that without any estimation and acceleration techniques, the optimizations using distributed systems are solved with the same accuracies and same numbers of iterations as before. The time taken to solve each problem is, however, significantly longer, since the optimization solver always has to wait for the last and slowest machine to respond with a function value.

Simple acceleration techniques often result in less total optimization time, with relatively the same accuracies achieved. But these improvements are not guaranteed on any functions. The improvements are not even guaranteed on different starting points of the same function, since the response surfaces can behave very differently in various neighborhoods. Our primitive acceleration techniques also do not take account of networking anomalies. When a model service generates mathematical errors (e.g. divide by zero), the network becomes congested, or the server that hosts the model service crashes, our optimization process is terminated too. All these suggest further research in a better design and more robust procedure.

3.2.3 A robust implementation of distributed optimization in the real VP system

The next few sections introduce our effort to design a more advanced architecture using intelligent methods of search and acceleration. Along with optimization, special approximation procedures are being developed in the areas of statistical learning and artificial intelligence including data mining and machine learning. The challenge is how the optimization engine should simultaneously use information such as rate of improvement of the objective function and the computational performance characteristics of a set of distributed model services, so that the “best” solution can be found in the “shortest” possible time.

Due to confidentiality issues, in the following discussion actual component names of the real system are altered to more generic ones. Model services are referred to by numbers and types according to their service categorization explained above. In the benchmarks, algorithms are distinguished according to their levels of effectiveness, but the underlying optimization algorithms and various methods of function approximation are not disclosed.

The general system architecture and process flow will, however, be explained in detail. This is what the Optimization Services framework is mostly concerned with: to address the issues in system designs and provide a good infrastructure to carry out optimization of various types including optimization via distributed simulation. Optimization Services allows any optimization algorithms and approximation methods to be highly modularized and easily plugged into an optimization system.

3.2.4 Design and architecture

Figure 3-9 shows the VP Multidisciplinary Intelligent Optimization System. The upper right part of the figure is the solver architecture. The major component modules are as follows:

Central Server. This is mainly used to connect to different distributed model services or simulation engines offered by the Virtual Prototyping System, and maintain administrative routines. Requests for function values from the optimizer are always routed through the server. The optimizer is also connected with the server.

Simulation Engines. These are the major Virtual Prototyping model services in different engineering domains. When the central server relays the input (x), the simulation engine i returns value y_i.

Model Constructor. This part is used to dynamically construct multidisciplinary models that consist of services offered in the Virtual Prototyping system. It is mainly used to construct multi-objective functions: $f(y_1(x), y_2(x),...y_n(x))$, where $y_i(x)$ is calculated by simulation engine i. From the optimizer’s view, it is just another simulation engine. Instead of returning the value $y_i(x)$ from one simulation engine, it returns a combined value of several simulation engines. Of course, the optimizer can choose to call separate simulation engines and construct the multi-objective function by itself. In this way, the optimizer becomes more flexible since it can decide which simulation engine to call to get an exact value (y) and which to estimate to get an approximate value (y').

Client. This is usually any engineer who wants to use the services connected through the central server. From the client's view, the optimizer is simply another simulation engine, only the optimizer returns an optimized value. The client may not be aware that an optimization process is going on. But he may notice it takes longer to get the value, as an optimization may involve thousands of invocations of many simulation engines. All the data from are client side is sent through a communication socket.

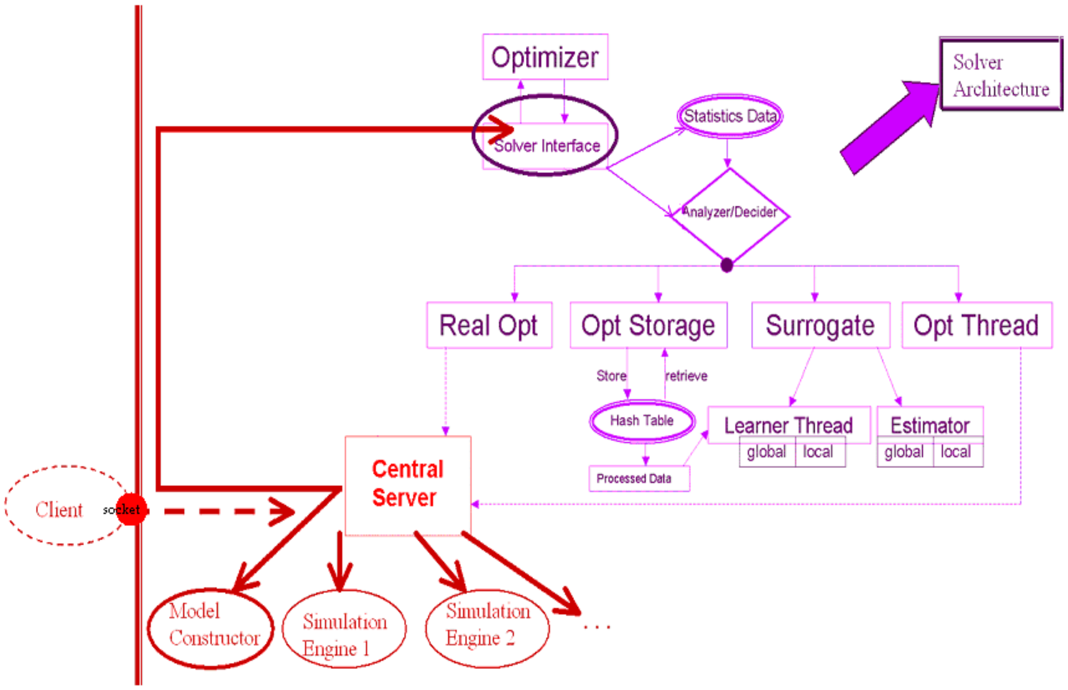


Figure 3-9: Architecture of VP Multidisciplinary Intelligent Optimization System.

Following are the modules related specifically to the solver architecture.

Optimizer. This module contains optimization solvers of different types, including linear programming, nonlinearly constrained programming, and integer programming.

Solver Interface. This is a generic interface that is connected to the remote central server. All solvers have to interact with this interface if they need function values from simulation engines connected with the central server. Notice that the solver interface separates the solvers from everything else. This means when a solver asks for a function value, it does not know where the function value comes from and it does not know whether the function value is exact or estimated. The design is critical because it allows any nonlinear solver to carry out the simulation optimization without changing its code at all. In a sense, the solver itself is not "intelligent." It is the components connected with the interface that make smart choices of function values, making tradeoffs between speed and accuracy.

Analyzer/Decider. This is the module that the solver interface uses to branch to different optimization processes. For example, it decides whether to get the real function value or approximate the value from a learned function.

Statistics Data. This module keeps track of run time information through the entire optimization process, for example, the time it takes to get a response from one of the simulation engines. The solver interface usually stores the necessary information at each iteration. The statistics data is also used by analyzer/decider for carrying out analysis.

Real Opt. This is the module that routes solver requests to real simulation engines, from which the optimizer gets an exact function value.

Opt Storage. This is an interface that provides access to retrieval and storage of online optimization data, for example the variable points and objective values on the optimization path that are needed for function learning.

Hash Table. This is basically a hashed database that stores all the evaluated variable points in a special way. In essence, it keeps a table in which a row looks like:

$$k, x_0^k, x_1^k, \cdots, x_n^k, f^k$$

where k is the iteration number, x_i^k 's are the variable values at the kth iteration and f^k is the corresponding function value.

Processed Data. This module is a data structure that processes the data stored in Hash Table into a format accessible by Learner Thread so that learning algorithms can be carried out with no format compatibility issues.

Surrogate. This is the module that acts as an approximate deputy for a simulation engine. It can be used either to learn a function through Learner or approximate the function value through Estimator.

Learner Thread. This module takes the processed data from Hash Table, and learns functions that approximate function of a simulation engine. It is a thread because it carries out function learning in a separate thread from the optimization process.

Estimator. This module takes the learned function from Learner and provides Optimizer with an estimated function value.

Opt Thread. Opt Thread is a separate thread launched by Solver Interface to get function values from the simulation engines. The purpose of Opt Thread is that the optimizer does not need to wait for a response from simulation engines because the thread is launched separately from the general optimization process. Function values are still to be returned at a later time but the optimizer can just carry on its optimization process with estimated function values. One main advantage is that more function values will later be obtained for the learning process. Another advantage is that when a simulation engine returns an error, the thread can simply be aborted without affecting the optimization process.

3.2.5 Optimization process

Figure 3-10 shows the processes of the entire intelligent optimization system. We will start with the normal flow, i.e. the flow with no learning process.

1). Normal Flow (without learning and approximation)

On the left part of the figure are processes (processes 0-10 and 11-15) with bold borders. They represent a normal nonlinear optimization flow – start with an optimization problem, alternate between finding directions/step lengths and updating variables, and finally terminate upon certain conditions. The major characteristic in this flow is that processes 2 and 5 do not get function values locally. Instead they have to go through process 11 - 15 to get function values from remote simulation engines that are connected with the central server. Function values are optionally stored in the hash table (process 15).

Processes 2 and 3 are intended to find step directions. Potentially a large number of requests are made to obtain information on function values and gradients. Thus arrows leading out of process 2 and leading into process 3 are in bold.

Processes 5 and 6 are intended to find a step length along the direction. Only a few function requests are needed. Naturally, it takes a much shorter time to do a line search than to find a direction. According to our benchmarks (§3.2.7), solvers that have a loop back mechanism from process 6 to process 4 tend to be much faster. The loop back is intended to do a very accurate line search and in practice, all the loop backs in one iteration take only a fraction of time of finding a direction.

2). Intelligent Flow (with learning and approximation)

Processes on the right part of the figure (processes 16-25) are with dotted borders and represent the intelligent components. Notice the separation between the normal flow and the intelligent flow. None of the intelligent components are built within the solver, that is, the optimization algorithm remains untouched. The idea is that any solver can leverage on the intelligent system with no code alteration and any intelligent system can be plugged in without much interfacing effort. This is the key assumption when we redesign optimization via Simulation under the Optimization Services framework (Chapter 6).

The major decision is process 12. If no intelligence is needed, it goes through a regular distributed optimization process (12-15) and back to the normal nonlinear algorithm flow (0-10). Otherwise, it leverages on the estimation and acceleration techniques in the intelligent part.

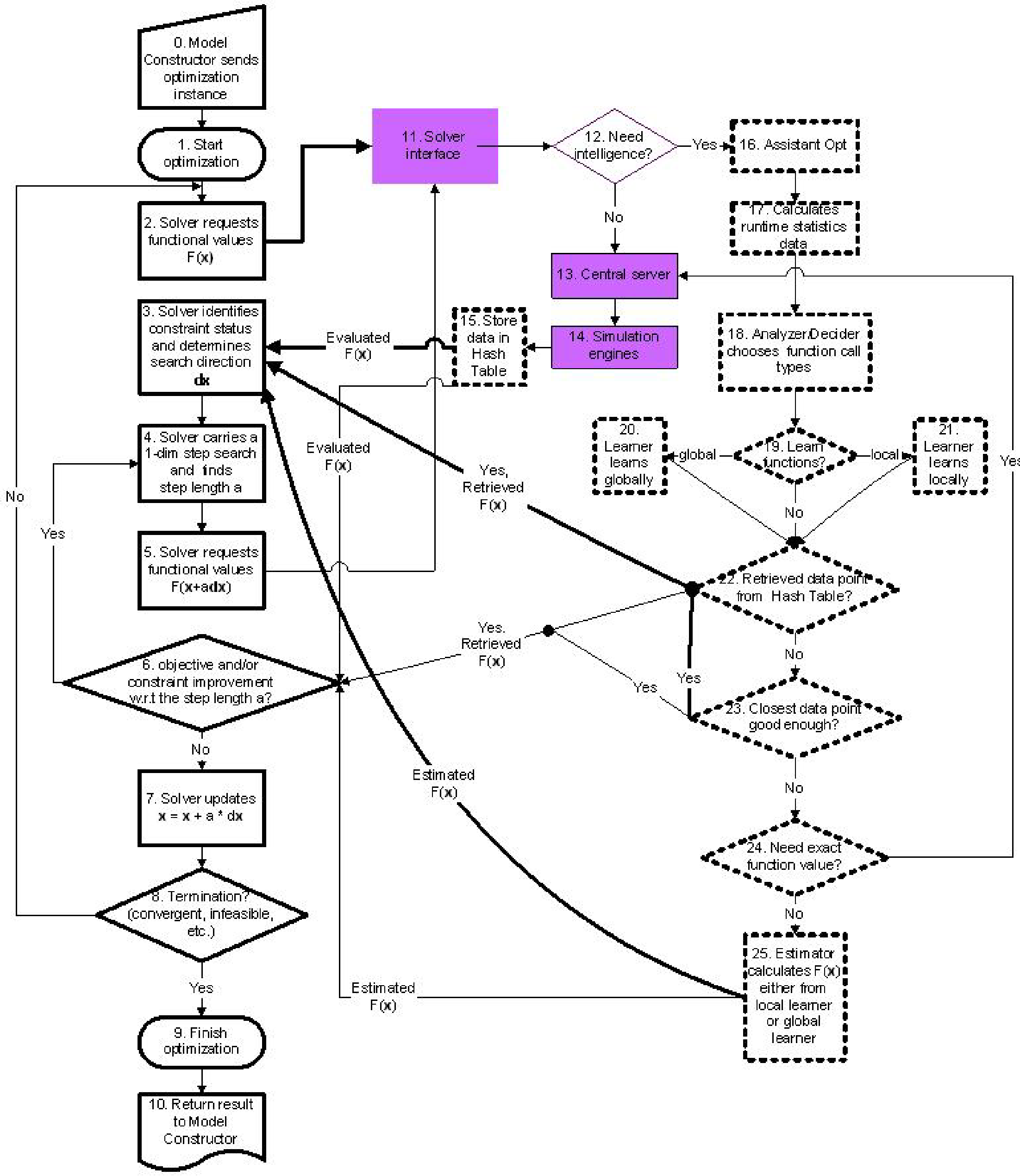

0. Model Constructor sends optimization instance
1. Start optimization
2. Solver requests functional values F(x)
3. Solver identifies constraint status and determines search direction dx
4. Solver carries a 1-dim step search and finds step length a
5. Solver requests functional values F(x+adx)
6. objective and/or constraint improvement w.r.t the step length a?
7. Solver updates x = x + a * dx
8. Termination? (convergent, infeasible, etc.)
9. Finish optimization
10. Return result to Model Constructor
11. Solver interface
12. Need intelligence?
13. Central server
14. Simulation engines
15. Store data in Hash Table
16. Assistant Opt
17. Calculates runtime statistics data
18. Analyzer/Decider chooses function call types
19. Learn functions?
20. Learner learns globally
21. Learner learns locally
22. Retrieved data point from Hash Table?
23. Closest data point good enough?
24. Need exact function value?
25. Estimator calculates F(x) either from local learner or global learner
Yes
No
global
local
Evaluated F(x)
Yes, Retrieved F(x)
Estimated F(x)

Figure 3-10: Flowchart of the intelligent optimization process; thicker lines mean more frequent data flow in the optimization process.

<u>2.1) Intelligence Flow – Analysis Stage</u>

The first thing after the intelligent optimization starts (16) is to analyze statistics of run time information (17), including:

- optimization process data, for example current iteration number, variable change rate, objective convergence rate, and constraint improvement rate;
- historical data points in database;
- finishing status of a simulation (The simulation is successfully run or it generates errors);
- time it takes between requests and responses of a simulation over recent iterations
- access types of recent function calculations – whether function values are retrieved through database, estimated through an approximated function, or evaluated by the real simulation engine;
- last global and local learning time of the function learners;
- accuracies of function learners through validation between estimated values and real values.

Statistics are constantly updated whenever new information is available.

2.2) Intelligence Flow – Learning Stage

Process 19 is a decision to learn a function from historical points. The decision to learn a function is based on one criterion, namely whether there are enough new data points. The choice of the number of data points is quite empirical and depends on the chosen learning algorithms. It can be further studied and may be changed on an adaptive basis.

Two types of learning are used. The global learning is intended to learn the entire function surface, while the local learning is used to learn the function surface in the neighborhood of the current variable point. In general learning takes various forms. Complex learning like Neural Network Training and Gene Expression Programming are potentially more accurate. For general descriptions of statistical learning, machine learning and data mining, refer to the book [67]. But they can take a long time. Motorola Advanced Technology Center has developed a number of advanced machine learning tools and several of them are both accurate and fast. The main purpose here is not to describe the algorithms inside these tools. The goal is to illustrate that with the help of well designed learning tools that are properly coordinated with an optimization algorithm, decent acceleration can be achieved (see benchmarks in §3.2.7). In addition to the proprietary tools, a range of other function fitting algorithms (e.g. quadratic fitting) are tested. But they generally do not fare well in the benchmarks. Although they are simpler and faster, they are less accurate in approximating functions in high dimensions, and tend to lead optimization in wrong directions.

In practice, global learners are relatively slower than local learners. Global learners include standard statistical regressions, neural network training, gene expression programming, etc.

Global learners are launched when an optimization first starts. The leaning or training process is stopped sooner at the beginning, but the allowed learning time gradually increases. The purpose is to generate a big picture and a roughly smooth shape (i.e. not over-fitting) of a function, so optimization can move in a generally correct direction. As data points accumulate, we increase learning time and finally as convergence slows down, we switch to local learners.

Local learners include basis expansion methods such as smoothing splines, kernel methods such as local linear or polynomial regressions, and variants of nearest-neighbor methods. By the time we switch from a global learner to a local learner, we have accumulated more points. Many algorithms in local learning need a large number of points to fit functions in high dimensional variable space.

Just as in optimization that no solver performs the best and fastest on all functions, no learners perform the best and fastest on all datasets. Not all global learners or local learners are launched, depending on factors such as the number of points and number of variables. For example certain learners simply can not be launched with a few points and other learners are only suited to fitting in low dimensions. If a learner takes an extremely long time, it may just be dropped.

The following decisions are the three choices that the solver can make to get functional values: retrieval, evaluation, and estimation.

2.3) Intelligence Flow – Retrieval Stage

Our database is in essence a hash table with the hash key being the x variable ($x_1, x_2, \ldots x_n$) and the hash value being the function value (f_x) along with an access index. The access index measures recentness of variables, which is useful in cases where only recent points are needed for learning, estimation and validation. Admittedly, a hash table takes up memory. Our reasoning is that memories are abundant, so that in practice we never have to face memory overflow. Our main concern is *speed* rather than *space*. A hash table's row indexing is based on a hash function value and record retrieval is of constant time. Thus every time we try to search for a point x, we don't have to go through the entire table, which can be time consuming with accumulation of data points. Data precision is kept to certain decimal points and digits after that are truncated, so points that are close enough are considered the same.

In our case, retrieving time is only a tiny fraction of the time it takes to calculate a function from a simulation engine. If we succeed in locating a previously calculated function value out of even thousands of retrievals, we still can save time. Function value retrieval from the database happens quite often in practice. There are three reasons that the same points are being retrieved. The first is due to the searching algorithm going back to the same region. The second is due to algorithms using finite differences to evaluate gradients. For a simple illustration, in a one variable optimization, the left point used to estimate the gradient at the current point may be the next current point if the search decides to move left to that point. The third reason is an implementation issue. Most of the time when a solver implementer codes an algorithm, he assumes that function evaluation time is negligible or about the same as retrieving from memory. So in each iteration he may just keep on requesting the same function evaluation to calculate gradient, direction, step size etc, rather than store the value in a local variable for later retrieval.

A closest point (process 23) may also be returned depending on its Euclidean distance to the current point. Because variables are normalized to the same scale before optimization, a "closeness" measure is set to a very small fraction multiplied by the number of variables. The closest point is returned if the distance between the closest point and current point is both smaller than the "closeness" measure and smaller than the distance between the closest point and the last evaluated point. The first standard is an absolute measure of closeness whereas the second standard is a relative closeness with regard to the latest movement. The second standard is also used to protect finite difference based gradient estimation, in which the last point is almost surely the closest point, thus generating a gradient value of 0.

2.4) Intelligence Flow – Evaluation Stage

If no previous data point or closest data point can be retrieved, the Analyzer/Decider may choose to get the evaluation (process 24) from the real simulation engine (process 14) through the Central Server (process 13). This process is always launched separately, but the flow does not go on until after a maximum wait time. The maximum wait time is adaptively set to some number of times larger than a moving average of the previous simulation time. If an acceptable simulation result is obtained, it is first stored in the Hash Table (process 15). If there is an error returned or the maximum wait time expires, the flow moves on to process 24 to return an estimated function value. This is a major step toward robust optimization design protecting against simulation anomalies. The simulation process is allowed to continue even after the maximum wait time. Any result the simulation eventually produces is stored in the database. This stored result is of special interest in validation and comparison of learners, because this point is both estimated by a learner and evaluated by the real simulation engine.

2.5) Intelligence Flow – Estimation Stage

If the Analyzer/Decider finally chooses to estimate a value from a learned function (process 25), it first needs to validate all the learners to measure learner effectiveness. Whether the estimation is local or global depends on whether the last learning process is global or local, because as mentioned above only one type of learner can be at launched one time. Validation is based on the sum of squared residual errors between estimated values and evaluated values. Validations are executed only on the most recent data. If not enough recent data are both evaluated and estimated, extra time will be taken to extract the most recent data from the database and estimate them with each learner. The learner that performs the best in validation is chosen to return its estimated function value to the solver.

Currently the Analyzer/Decider has an *ad hoc* mechanism to guarantee convergence or termination. There is a maximum number of times that estimations can be made in a row. When convergence rate is slow or the iteration number exceeds a certain limit, the Analyzer/Decider will always choose to get an evaluation from the real simulation. Due to the big convergence tolerance and the large iteration limit that we set, this mechanism is seldom used in practice.

3.2.6 System implementation issues and lessons learned for Optimization Services

The simulation engines in the Motorola system were not originally built to be optimized over a distributed system. In designing an intelligent multidisciplinary optimization system that involves legacy simulation software, the following major issues need to be solved for any optimization process. Due to the lack of a universal standard and framework, many of the design issues were solved on an *ad hoc* basis. Many of these serve as a motivation for our Optimization Services described in Chapter 5.

Initial Design Generation

This serves as the initial point for a nonlinear optimization. But not all the simulation engines provide such information. A set of quadruples is required for each variable in the form of:

$$(\textit{current value}, \textit{most likely value}, \textit{lower bound}, \textit{upper bound}).$$

These values are obtained by consulting with domain engineers. Current values can be customized for each optimization run by the client. When multi-start optimizations are carried out, distribution functions (for example a triangular distribution based on most likely value, lower and upper bounds) are used to generate different starting points.

Common Variable Resolution

Different simulation engines are implemented in individual domains, without exchanging information with each other. As a result, names of parameters and variables are different even though they refer to the same things. For example, the name “bsize” in simulation engine 1 may be the same as the name “board_size” in simulation engine 2. Originally, the situation was handled by constructing interdisciplinary constraints forcing different variables to be of the same values. But then the optimization problem size is made unnecessarily large due to redundant variable declarations. An overhaul has been carried out on all the simulation engine implementations to find common variables. To match all the different names to a standard naming, a static “pairing” table has been constructed, so common variables are detected and variables are declared only once. But there are still other issues. For example in the table we match “bsize” and “board_size” respectively to a common name “boardSize.”

Clients may be unaware of the common variable situations by supplying different current values to two differently named copies of the same variable, for example setting current values of “bsize” to 1.0 and “board_size” to 1.2. In cases like these, we take the average of the two values, for example, setting the current value of “boardSize” to 1.1. *Most likely values, lower bounds, and upper bounds* may also be set different when constructing a multi-domain model. In practice we choose the largest lower bound, the smallest upper bound, and the average of the mostly likely values a compromise.

Multi-objective Function Construction

A multi-objective function usually takes the form of a weighted sum. Different simulation engines are chosen by the client and corresponding weights are specified for the objective from each simulation engine. Weights are solely based on the client’s personal judgment reflecting the importance of different simulation metrics. But the client has to indicate whether a smaller value or a bigger value of a metric is better, so that the model constructor can build a consistent maximization or minimization objective function. Metrics of different simulation engines are also in different units; thus the constructed multi-objective function is unitless and only useful for relative comparisons, such as Pareto analysis.

Meaningful reports for each simulation are constructed based on optimal variable values. Normalization techniques such as arctangential transformation are taken to bring component metrics to the same scale.

Constraint Enforcement

Constraints of a multidisciplinary optimization are a combination of all constraints from each separate simulation plus interdisciplinary constraints over several simulations across domains. All the interdisciplinary constraints are hard coded in an assistant module. The assistant module first detects which simulation engines and what variables are chosen, and then returns the interdisciplinary constraints that contain the simulations and the variables.

Result Interpretation

Though the Motorola system has a proprietary data format to internally standardize results from different simulation engines, they were never intended to be combined with each other to construct a multi-objective function. Name conflict is thus one major issue. For example many simulation engines return a generic name called “result.” Efforts had to be taken in setting distinctions between the names. One way is to rename, but this caused many unforeseeable bugs because sometimes results are further analyzed by other computer systems. Another way is to group results into subsections and use combinations of simulation names, subsection names and result names.

Another issue, though encountered not as often, is that results can be discrete. During any hill-climbing type of optimization, these situations can cause optimization solvers to immediately claim a local minimum or maximum. One technique used is a smooth interpolation of the previous results. When using a learning technique that tries to estimate the function smoothly, this problem is naturally avoided. Techniques can be applied on a situation-by-situation basis. In one circumstance, we added an “interdisciplinary” objective term, as a secondary objective, to make the discrete function continuous. All the interdisciplinary objective terms are hard coded in an assistant module. The module first detects whether the simulations that have discrete objectives are chosen, and then incorporates into the optimization model the corresponding interdisciplinary objective terms. When results from an “altered” model are returned, they have to be reinterpreted and presented to the client in terms of the original model.

Process Coordination

Requests for results from distributed simulations are all launched in parallel, instead of sequentially. The simplest coordination technique is to wait for all the processes to finish before moving on to the next step. Other techniques are also employed depending on different situations. Any major textbook on designing and building parallel programs covers some most popular and practical algorithms; see [40]. For our case in Virtual Prototyping, the multi-objective function can only be constructed with the returns of all the component objectives from distributed simulations. Therefore we usually have to wait for the slowest simulation. But due to the intelligent mechanisms described above, it is also usually the slowest simulation engine whose values are estimated most often by the learners.

The client may happen to choose simulation engines that do not share variables and constraints. In situations like these, separate optimization processes are launched for each individual simulation in parallel, and results are combined finally according to the client's multi-objective construction.

Queue/Sequence Arrangement

Sometimes not all simulation processes can be launched at the same time. Some simulations may contain input parameters that are results from other simulations. Flows are hard coded when we encounter a combination of simulations that have to be invoked in sequence. Simulation processes that have to wait for results from others are put in a queue to be notified later. Some standard service flow coordination mechanism is needed here.

Input Parsing/Output Reporting

Input parsing and output reporting are specified in a proprietary format. Though standardized, the format is complicated and only understood by a few software developers in Motorola labs. Moreover the format was not built for multi-disciplinary optimization constructions. Special efforts had to be taken to extend the functionality. In the case of process sequencing, where one simulation's input takes a value from another simulation's output, the effort is extremely painstaking. In the case of generating reports of multidisciplinary results and mapping multi-dimensional space onto two-dimensional graphs, the procedure is even more laborious.

3.2.7 Simulated benchmarks

Table 3-1 through Table 3-4 show the benchmarks from applying various combinations of optimization solvers and function learning algorithms to different simulation services of different types. Results are shown in minutes taken to optimize each simulation service or combination of simulation services. An "X" in the tables means the optimization process aborted due to simulation or network errors. The ">1500" means we manually terminate the optimization process after 1500 minutes or 25 hours.

Each table shows a different learning algorithm used with a set of optimization solvers. Table 3-1 uses no learning algorithm and the optimization solvers always try to get exact function values by calling the real simulation engines. Table 3-2 uses a simple 3-layer neural network learning algorithm. Table 3-3 uses a more advanced gene expression programming learning technique. Table 3-4 uses an advanced generalized neural network learning algorithm. So the tables are ordered according to the general quality of the learning algorithm used. Qualities of these learning algorithms were intensively benchmarked within Motorola in terms of both learning accuracy and speed.

Within each table, each column indicates a different optimization solver. Four optimization solvers are used: MFD, MFD+, Direct MMFD, and Direct MMFD+. MFD is the original optimization method based on a modified feasible direction method. MFD requires gradient values. We use the finite difference method to get the gradients. MFD+ does a more intensive line search to find an accurate step length. As explained in the previous sections, finding step lengths involves much fewer function calls than finding search directions. So MFD+ is adapted especially for optimization via simulation where each simulation takes a long time. Direct MMFD modifies the MFD algorithm so that no gradients are required. Direct MMFD+ does a more intensive line search than direct MMFD.

Within each table, each row indicates a different service or combination of services. Remote service execution time is given by the following formula and data:

$$T = T_s \times LF(t) + DT \tag{3-1}$$

where:

T_s = Service time for a given server;

$LF(t)$ = Load factor as a function of time (t);

DT = Down time.

Three kinds of services with typical behaviors are identified:

Service A:

T_s = Uniform distribution [6, 30] seconds

$LF(t)$ = 2.0 from 0800 to 1700 hours; 1.0 otherwise

DT = 5% probability of the service going down for 30 seconds

This service has automatic “crash detection” and recovery; therefore, the maximum down time is 30 seconds.

Service B:

T_s = Uniform distribution [30, 60] seconds

$LF(t)$ = 1.25 from 0600 to 1400 hours; 1.0 otherwise

DT = Insignificantly small

This service runs on a dedicated server; therefore, the load factor does not change significantly during the day. The down time is insignificant, because this service runs on dual servers, and the robustness of the model service software has been proven.

Service C:

T_s = Uniform distribution [30, 90] seconds

$LF(t)$ = 2.0 from 0800 to 1700 hours; 1.0 otherwise

DT = 1% probability of the service going down for anywhere between 15 minutes and 16 hours

In short, service type A takes the shortest time and service type C takes the longest. Service B is the most stable and service A and C can malfunction, thus not able to return function values sometimes. "+" means a combination. For example "A+B" means the multi-objective function consists of metrics calculated from both service type A and service type B.

In all the tables we only show the *time* it takes for the methods. It turned out that *quality* of solutions varies little between various methods. It is partly due to final stage fine tuning and safeguards for convergence used in our intelligent optimization process, and partly due to "good" function behaviors, or normal surface shapes of our simulations. Also in practice, we have a good idea of a feasible starting point based on the existing engineering design of a product. In most cases, our goal is to improve a product that's already designed or manufactured rather than find an initial design.

service type	MFD	MFD+	Direct MMFD	Direct MMFD+
A	X	X	X	X
B	623	137	310	110
C	X	X	X	X
A+B	X	X	X	X
A+C	X	X	X	X
B+C	X	X	X	X
A+B+C	X	X	X	X

Table 3-1: Benchmark results from normal optimization without function learners (time in minutes).

service type	MFD	MFD+	Direct MMFD	Direct MMFD+
A	619	132	376	78
B	645	287	389	172
C	>1500	>1500	422	192
A+B	641	212	358	142
A+C	1231	>1500	401	>1500
B+C	908	333	385	180
A+B+C	1147	324	>1500	202

Table 3-2: Benchmark results from intelligent optimization with a simple 3-layer neural network learner (time in minutes).

service type	MFD	MFD+	Direct MMFD	Direct MMFD+
A	343	71	210	40
B	360	160	215	91
C	>1500	>1500	230	106
A+B	361	118	190	79
A+C	>1500	190	210	92
B+C	480	846	202	93
A+B+C	647	165	273	114

Table 3-3: Benchmark results from intelligent optimization with a gene expression programming learner (time in minutes).

service type	MFD	MFD+	Direct MMFD	Direct MMFD+
A	182	66	93	49
B	204	87	108	42
C	>1500	1452	105	54
A+B	165	87	92	37
A+C	1002	487	145	49
B+C	229	132	123	45
A+B+C	293	145	123	67

Table 3-4: Benchmark results from intelligent optimization with an advanced generalized neural network learning (time in minutes).

Table 3-1 shows that without “intelligence” (learning and approximation), an optimization either crashes or generally takes longer. In fact, as long as a simulation engine can malfunction, the optimization always aborts since nowhere else can it get another function value when an error or exception is returned by a simulation. The only benchmark results are from optimization that just involves service of type B, the most stable service. From the service type B row, we see that if we add an intensive line search (MFD+ vs. MFD, and Direct MMFD+ vs. Direct MMFD), we do significantly reduce optimization time. The direct method (Direct MMFD vs. MFD, and Direct MMFD+ vs. MFD+) also helps but is not as significant as the intensive line search. Combining the intensive line search and the direct method is the best choice and it also is true in the cases of intelligent optimization flow where learning algorithms are employed.

Table 3-2 uses a simple 3-layer neural network. Obviously it is more robust as we can get all the results. However if we compare the row of service type B with that in Table 3-1, we see that the optimization takes longer with each optimization algorithm. This was also true with all the other less advanced learning algorithms that we used in the early prototyping stage, e.g. quadratic fitting. If the learning is not accurate, it tends to mislead the optimization direction and as a result optimization takes more iterations to finish. The curse of dimensionality is always an issue. So learning algorithms robust in high dimension can help. In our situation, there are usually 10 to 20 variables involved. Less advanced learning algorithms tend to perform badly when the number of variables gets over 10.

Table 3-2 (as well as Table 3-3 and Table 3-4) also shows that optimization takes longer when simulations take longer. With combinations of simulations, the optimization time only correlates with the simulation that takes the longest, as we launch the simulations simultaneously and wait for the last to return before the next optimization step. One thing to notice is that optimization with a combination of simulations does not necessarily take longer than optimization with just one simulation. Occasionally they may even help to some extent. For example in Table 3-4, the MFD method on service of type “A+B” takes 165 minutes, whereas it takes 182 minutes if there is only service of type A. This is sometimes also true if other learning algorithms are used. Part of the reason is that there are usually not many extra variables involved with combination of services and function learning on more services may often be more accurate due to more data accumulation.

Table 3-3 and Table 3-4 use more advanced learning algorithms. In general the optimization takes much less time. This is most obvious in the case of using the most advanced generalized neural network learning. So speed and quality of learning algorithms matter significantly. But do notice that it is not always true. Occasionally there are some erratic behaviors. For example it takes 846 minutes for the MFD+ optimization to finish on service type B+C (Table 3-3), while it only takes 333 minutes for the simple neural network (Table 3-2). Optimization that involves service type C quite often shows these erratic behaviors. It’s possible that service type C has a more irregular function shape than the other two, thus harder to learn.

In summary, intensive line search and direct methods both help. Adoption of learning makes optimization more robust. Although not always the case, good learning algorithms can significantly speed up optimization without losing solution quality. These are all on the assumption that simulations take a long time. With simulations whose function evaluations are quick, there is no advantage of using intelligent optimization, due to the extra overhead in getting a simple function value.

CHAPTER 4 OS COMPUTING AND DISTRIBUTED TECHNOLOGIES

This chapter provides the necessary background on modern computing and distributed technologies in order to read the thesis. We explain all the technologies in the Optimization Services context.

Historically, distributed computing has been focused on the problem of distributing computation between several systems that are jointly working on a problem. The most often used distributed computing abstraction is the RPC – Remote Procedure Call. RPC allows a remote function to be invoked as if it were a local one. The history of RPC-style distributed computing is fairly complicated. More or less it started with Sun Microsystems' Open Network Computing (ONC) RPC system in 1987, as the basic communication mechanism for its Network File System (NFS). NFS is now supported on UNIX, Linux, and many other distributed operating systems. NFS is used to access directories and files located on remote computer as if those directories and files were located on the local computer.

The first major effort toward language-independent and platform-neutral distributed computing was taken by the Object Management Group (OMG) in 1989. OMG is a consortium that includes over 500 members. In 1991, OMG delivered the first version of Common Object Request Broker Architecture (CORBA), a distributed objects platform. CORBA allowed programs located in different parts of the network and written in different programming languages to communicate with each other. The term Object Request Broker (ORB) gained popularity to denote the infrastructure software that enabled distributed objects. In 1996, CORBA version 2 introduced the Internet Inter-ORB Protocol (IIOP) as major enhancements in the core distributed computing model and higher-level services that distributed objects could use. IIOP established CORBA's dominance in distributed computing for the next 5 years until the advent of Web services.

Microsoft started its own distributed computing initiative around 1990. In 1996, Microsoft delivered the Distributed Component Object Model (DCOM), which was closely tied to previous Microsoft component efforts such as Object Linking and Embedding (OLE), non-distributed COM (or OLE2), and ActiveX (lightweight components for web applications). To compete with CORBA, the next year (1997) Microsoft introduced COM+ to bring DCOM much closer to the CORBA model for distributed computing.

In the same year, Sun Microsystems added Remote Method Invocation (RMI) in its Java Development Kit (JDK 1.1). RMI is similar to CORBA and DCOM, but works only with objects written in Sun's Java programming language. In Sun's 1999 Java 2 Enterprise Edition (J2EE) platform, the company integrated RMI with CORBA's IIOP.

Unfortunately, CORBA is very complex. It requires significant effort to implement. The much simpler XML-based XML-RPC appeared in 1999 and became a strong competitor to CORBA. XML-RPC was inspired by two earlier protocols. The first is an anonymous RPC protocol designed by a person named Dave Winer. The other more important inspiration was an early draft of the SOAP protocol.

The name of Simple Object Access Protocol (SOAP) appeared for the first time around 2000, which heralded the era of Web services. Our implementation of Optimization Services is entirely based on SOAP and adopts the same architecture as that of Web services.

Although Remote Procedure Call has been the traditional approach for building distributed systems, there are other alternatives such as data-oriented or document-centric messaging (for asynchronous invocation). Rather than being focused on distributing computation by specifically invoking remote code, messaging takes a different approach. Applications that communicate via messaging run their own independent computations and communicate via messages that contain pure data. IBM released its messaging product MQSeries in 1993. Microsoft's messaging product is the Microsoft Message Queuing Server (MSMQ). Sun Microsystems' J2EE defines a set of APIs for messaging through the Java Messaging Service (JMS). There is no attempt to define a standard interoperability protocol for messaging servers.

One of the key benefits of Web services is that the core Web service protocols can support RPCs and messaging with equal ease. We define Web services and describe related technologies in the later sections (§4.5, §4.6, §4.7, §4.8). We also describe the service-oriented architecture that structures the Optimization Services framework.

4.1 Basic Computing Technologies and Terminologies

This section briefly describes basic computing technologies that help in understanding the later sections of the chapter. All of these technologies are used directly or indirectly in the design and implementation of the Optimization Services framework.

4.1.1 Java and OS design philosophies

In the Optimization Services project, we use the Java programming language to implement the Optimization Services (OS) library and build our Optimization Services system. The OS library is designed to provide a foundation of reusable objects to speed the development of OS applications and make them more reliable. Today many Operations Research applications are developed from scratch to solve a specific problem without the benefit of a foundation of tested software. This is time consuming and expensive due to the complexity and the thorough testing that is required of OR applications. By reusing the many proven classes in the OS library, OS developers can build OS applications more efficiently and the OS system will be more reliable.

In order for OS library to be accepted, it must be immediately useable and at the same time provide the depth and flexibility required for advanced applications. Our OS library provides depth and flexibility through the extensive use of interfaces to abstract methods and data structures. All methods use only these interfaces and abstract classes when accessing internal objects and this allows developers to substitute an object that provides new functionality.

Java was selected because it is platform independent and provides a rich environment for application (esp. client and server applications) development. The data structures and methods used in OS applications have a large impact on performance and using Java will make efficient components more widely available and easier to use. Moreover most current surveys and benchmarks [73] find that Java performance on numerical code is comparable to, or better than that of C++, with indications that Java's relative performance is continuing to improve.

Portability

Most programs created on a particular operating system must be converted, or ported, before they can run on a different operating system. A major advantage of the Java programming language is that users can run the same Java programs on computers using different operating systems. The phrase “write once, run anywhere” is often used to describe Java programming. This is also the goal of Optimization Services.

Java Virtual Machine (JVM)

JVM is software that runs Java programs. The virtual machine creates a simulated software environment on a computer, which allows Java programs to run outside of the computer’s operating system. This helps prevent malfunctioning Java programs from crashing a computer system and makes it possible for Java programs to run on different platforms. The

Java virtual machine also automatically handles such tasks as garbage collection, threading, security, and loading classes. The Optimization Services server that we implemented to host services like solvers relies on JVM. Thus hosted services enjoy all the support from JVM.

Free Environment and Open Source Community Support

The Microsoft .NET initiative includes a new programming language (C#) and a Common Language Runtime (CLR). C#'s language design is similar to that of Java and CLR is very much like Java's Virtual Machine – CLR components are implemented as byte code that runs in a managed environment. But .NET is Microsoft proprietary and is not free.

To design and implement an open standard framework like the Optimization Services framework, Java is a good fit and almost a necessity. Java is the most widely adopted language in the *general* Open Source and free software community. The language support has a much larger audience base. In fact, all the required libraries used in the OS library are open source. There are different classes of "free software," and there are gray areas between each class. For more information on Open Source and free software and Open Source license, visit the site http://www.opensource.org/.

Objected-oriented Language

Java is a purely objected-oriented programming language. To design a good distributed system framework like the OS framework, the Object-Oriented Programming (OOP) ideology and philosophy should be adopted. We describe OOP in more detail in the next section.

4.1.2 Object-Oriented Programming (OOP)

Object-Oriented Programming is a programming concept developed to make programs more understandable and easier to correct and modify. In the OOP concept, a program is made up of one or more objects, which are small, re-usable chunks of code. Each object is used to perform a specific task and can be shared with other programs. Distributed object-oriented systems require object-based RPC. It is almost a necessity to adopt OOP when designing any good distributed system.

In our Optimization Services system, each component is designed as an *object*. A communication `Agent` object, for example, provides *methods* for generic networking. The mechanism to create the `Agent` objects is the `Agent` class. Users of the `Agent` class are provided with a *specification* of how the class works, but they need no knowledge of how the

Agent class is implemented. The separation of specification from implementation is called *information hiding*. In our example, the Agent class hides all the information of our Optimization Services Protocol based networking. Classes are more formally called "abstract data types" or ADTs in objected-oriented programming terminologies. ADT is the most important characteristic of any OOP language.

A second characteristic of OOP languages is inheritance. In our Optimization Services implementation, for example, there is a library package for basic algebra operations. In the common algebra package, we define a SparseMatrix. A SparseMatrix class is certainly a matrix. Thus the class SparseMatrix can be said to *inherit* from class Matrix. In this context, class Matrix is called a *base class* and class SparseMatrix is called a *derived class*. SparseMatrix can in turn have its own derived classes, for example, DoubleSparseMatrix for a sparse matrix with double precision decimal matrix entries, and BigSparseMatrix, for a sparse matrix with arbitrarily precise matrix entries. An inheritance hierarchy of matrices is shown in Figure 4-1.

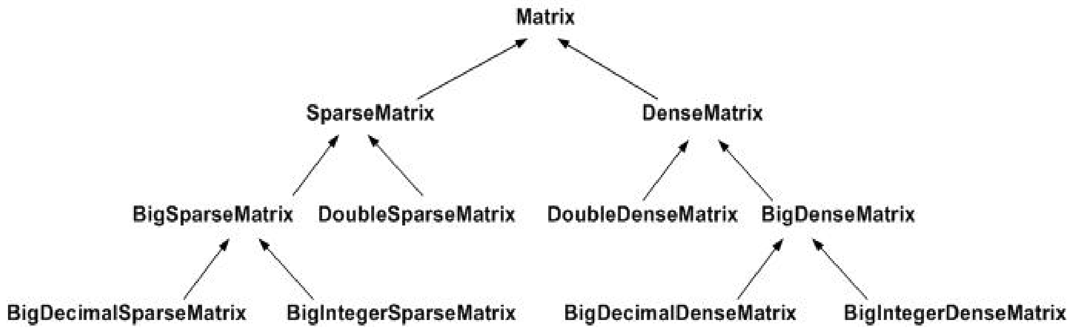


Figure 4-1: Inheritance hierarchy for matrices.

The third characteristic of object-oriented programming language is polymorphism, a dynamic binding of messages to method definitions. This is supported by allowing one to define polymorphic variables of the type of the base class that are also able to reference objects of the derived classes of that class. The base class can define a method that is overridden by its subclasses. The operations defined by these methods are similar, but must be customized for each class in the hierarchy. When such a method is called through the polymorphic variable, that call is bound to the method in the proper class dynamically.

Polymorphism is the key idea in designing nonlinear programming features in Optimization Services. In the case of nonlinear programming, a key aspect of any nonlinear instance parser is some sort of expression tree for the nonlinear part of a model instance. Consider the following Rosenbrock [97] nonlinear function:

$$(1-X_1)^2 + 100*(X_2 - X_1^{\,2})^2 \tag{4-1}$$

An expression tree for the function is illustrated in Figure 4-2.

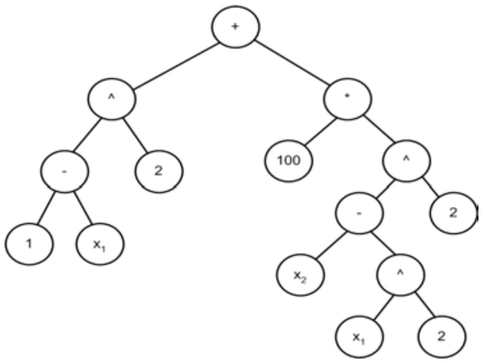


Figure 4-2: An OS expression tree for the Rosenbrock nonlinear function.

One approach is to use a C-structure for each node in the expression tree. The structure can store information as to operator or operand type and pointers to children nodes. A tree-walking method is used to perform operations on the expression tree such as function or derivative evaluations. See Figure 4-3 for an illustration the essential idea. In Figure 4-3, `expr` is a C-structure, `*e` a pointer to the root node of expression tree and `opnum` is an integer value denoting the node type.

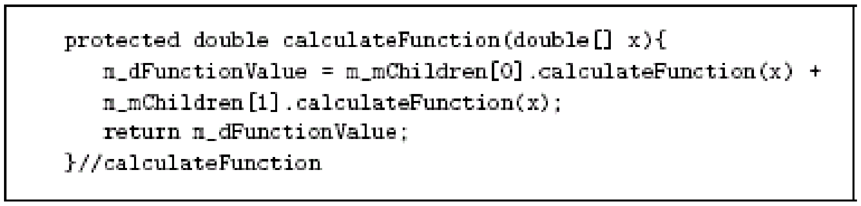
```
double evaluate_function (expr *e, double x[]){
    ...
    opnum = e->op
    switch(opnum){
        case PLUS_opno: ...
        case MINU_opno: ...
        ...
    }
}
```

Figure 4-3: Sample code for parsing a nonlinear instance without polymorphism.

Currently there are more than 200 nonlinear operators supported in the Optimization Services nonlinear Language (OSnL, §6.3). OSnL is included in the Optimization Services instance Language (OSiL, §6.2) for nonlinear extension. A fundamental problem with the above approach is that every method that operates on the expression tree requires a `switch` with a whole series of `case` statements (or a sequence of more than 200 `if` statements) making the code very complex. Updating the code to reflect new operators can be even more time consuming and error prone.

A second approach is to use an object oriented language such as C++ or Java and define a class for each type of node in the expression tree. For example, define a "plus" node class, a "minus" node class, an "exponential function" class, etc. However, for an object-oriented approach to be effective it is necessary to avoid the use switches and complicated logic as much as possible. This is achieved by having each node class extend a single node class and using polymorphism.

In our Optimization Services nonlinear Language (OSnL) library for reading nonlinear expressions, we first define an abstract class `OSnLNode`. All of the operator and operand classes used to define a nonlinear term extend the base class `OSnLNode`. For example, there is a class `OSnLNodeTimes` that extends the base class `OSnLNode`. This is a significant benefit, because we can construct an expression tree of homogenous nodes, i.e. the `OSnLNode`. Methods that operate on the expression tree to calculate function values, derivatives, postfix notation, etc. do not require switches or complicated logic. Since each operator and operand class extends the `OSnLNode`, class polymorphism eliminates the need for switches. For example, the abstract class `OSnLNode` has an abstract method `calculateFunction` that takes a double precision array of variable values and evaluates the expression tree for the give variable values. Every class that extends `OSnLNode` must implement this method. Consider the node class corresponding to the plus operator, `OSnLNodePlus`. The `calculateFunction` method for the `OSnLNodePlus` class is listed in Figure 4-4. Compare the logic in Figure 4-4 with the logic in Figure 4-3. Through the use of polymorphism and recursion the need for switches is eliminated. Because of this design, adding a new operator element is easy. It is simply a matter of adding a new class and implementing the `calculateFunction()` method.



```
protected double calculateFunction(double[] x){
   m_dFunctionValue = m_mChildren[0].calculateFunction(x) +
   m_mChildren[1].calculateFunction(x);
   return m_dFunctionValue;
}//calculateFunction
```

Figure 4-4: Calculating a function value in an `OSnLNodePlus` class.

4.1.3 Networking background and terminologies

Network and network software

We sometimes use the term Optimization Services (OS) network. By that we mean a group of connected computers that allow people to share optimization services. The size of an OS network is therefore the number of computers in this group.

The OS Network software consists of programs that 1) manage the OS network, 2) provide optimization services, and 3) allow computers to communicate and share information on the network.

1). Since Optimization Services relies on widely accepted networking protocols, no software (i.e. network operating system) that manages the OS network needs to be provided. Most of the time, Optimization Services is based on the Internet. Every modern computer has built-in Internet support, so there is no extra installation or configuration in order to run Optimization Services applications. If Optimization Services is used on an Intranet, popular Local Area Network software is also readily available.

2). Optimization software developers (e.g. solver developers) are the ones who provide the optimization services. Most software is built independent of Optimization Services. Therefore some interfacing adaptations need to be made in order to make the software Optimization Services compatible.

3) We as the Optimization Services developers provide the OS server software and library to help OS computers to communicate and share optimization services on the OS network. In this thesis, OS server and library are what we mean by OS network software.

Network protocols and standards

In §1.3.1, we talked about the Open Systems Interconnection (OSI) networking protocol model that specifies seven layers in the OSI model – Application, Presentation, Session, Transport, Network, Data Link and Physical from top to bottom. But OSI is only a reference model and it may be more detailed than necessary. Sometimes two OSI layers are simple enough that they are implemented together. In particular, the best known protocol, the Internet protocol (TCP/IP), aggregates some of layers in the OSI model. TCP/IP is a collection of protocols and it has only 4 distinct layers -- network access (e.g. Ethernet), internet (e.g. IP), transport (e.g. TCP), and application layers (e.g. HTTP).

Many products (including optimization related products) are needed to create and maintain a network. Before a network can function properly, all the products on the network must be able to communicate with each other. Before the protocols are introduced, there was no standardized way to exchange information on a network. Many companies developed their own network hardware or software without considering how different devices would work with other products on a network.

Most companies now follow the standard protocols so that their products will work with products developed by other companies. When companies follow standards, they ensure their devices will communicate with other devices on a network. It seems that the standards at the bottom layers (hardware) tend to be more mature than those at the top. The reason may be due to more diverse types of software at the higher level, which need to allow user flexibility. For example, Optimization Services Protocol (OSP) is a domain-specific application protocol. There can be many domains that need application protocols and benefit from standardization.

A major benefit of OSP is that it is an open standard protocol like TCP/IP. This means that any company or person can design a device or program that uses OSP without having to pay a royalty or licensing fee. Another major advantage of OSP is that due to the layering characteristic of networking protocols, OSP can leverage the many generic networking mechanisms from its underlying protocols and concentrate on domain specific (i.e. optimization) designs. For example OSP can be transmitted (directly or indirectly) over the Secure HyperText Transfer Protocol (HTTPS) instead of the normal HTTP protocol. HTTPS is then used to securely transfer information on the Internet. HTTPS encrypts and decrypts the information exchanged between a server computer and a client computer using a system called Secure Sockets Layer or SSL.

Client/Server networking

A client/server network consists of a central computer that serves resources and services to other computers, called clients. Traditional centralized optimization systems are typically based on such networking.

A client is a computer that requests optimization-related services or access to optimization information stored on a server. People use client computers to enter and display information processed by an optimization server on a network. An optimization server is usually dedicated to providing optimization-related services on a network. As with all centralized networks, administrative tasks such as result backup and security monitoring must be performed on a regular basis to ensure efficiency and reliability. If the optimization server is tampered with or malfunctions, the entire optimization system will be affected. As client/server networks require specialized, dedicated hardware and software, they can be very expensive.

Peer-to-peer networking

Our OS network is essentially a peer-to-peer network, as it allows all the computers on the network to store and share their resources and services, although on each peer-to-peer link, the networking is still set up on a client/server basis. There are no central computers that control the network. Software applications, such as optimization solvers, and the OS networking software/system (the OS library and OS server) run on each computer. Each computer is set up to share and access information and resources on the OS network. Since computers on the OS peer-to-peer network are configured to share and access information, individual developers administer their own computers. There is usually no dedicated system administrator for the OS network.

If a computer on an OS network is not turned on or malfunctions, the other computers on the network will not be able to access the computer's resources and services. However, resources and services on other computers on the network will not be affected. So peer-to-peer network is more fault-tolerant than a centralized client/server setup. It has to be acknowledged that since developers on the OS peer-to-peer network store files and services on their own computers, anyone may be able to access their computers. This makes information on the OS peer-to-peer network less secure than the traditionally more centralized optimization systems. The cost of a peer-to-peer network is, however, generally low.

4.2 XML

Optimization Services is an XML-based framework. The OS framework uses XML to specify both communication and representation standards.

XML stands for Extensible Markup Language. It is a subset of Standard Generalized Markup Language (SGML) constituting a particular text markup language for representation and interchange of structured data. For a quick reference, see [99]. For a complete reference, see [107]. SGML is a standard for how to specify a document markup language or tag set. HTML is another example of SGML.

But unlike HTML, which defines a fixed set of tags describing a fixed number of elements, XML is a meta-markup language in which we can define the tags we need to describe a document's structure and meaning. The tags must be organized according to certain general principles, but they are quite flexible in their meaning. For example, as we are working with optimization and need to describe objectives, variables, constraints, and so forth, we can create tags for each of these elements. The tags that we create can be documented in a *schema*. For now, think of a schema as a vocabulary and syntax for certain kinds of documents. For example, our Optimization Services instance Language (`OSiL.xsd`[1]) is a schema that describes a vocabulary and syntax for an optimization problem instance, and we can use the schema to validate an optimization instance. The validation mechanism ensures the stability of the Optimization Services standards. Formatting of our optimization instance can be added through additional style sheets (e.g. XSL in §4.4), but the instance document itself only contains tags that describe the optimization contents, not the appearance. So XML is a *semantic* markup language.

4.2.1 Why XML

XML became a specification at the World Wide Web Consortium (W3C) in 1998. Since then XML has increasingly been adopted as a standard for information interchange of all kinds.

Domain-specific

This is probably the biggest reason that the Optimization Services is XML-based. XML allows workers in each research area to develop their own XML dialects. Thus XML is ideal for large and complex optimization instance documents. For example, we can create "variable" and "constraint" tags in a way that it is most efficient and effective for storing, transmitting and parsing a mathematical program. XML not only lets us specify a vocabulary for the document, but also lets us specify the relations between elements. For example, we can require that every variable has a lower bound and if the lower bound is missing, it defaults to 0.

Open

Optimization Services, being an open framework, requires the standards that it uses to be completely open and freely available on the Internet. XML is a W3C standard that is nonproprietary, unencumbered by copyright, patent, trade secret, or any other intellectual property restriction.

Interoperable

Optimization Services is intended to solve communication issues between heterogeneous components over a distributed system. XML can be used on a wide variety of platforms and interpreted with a wide variety of tools. XML supports a number of key standards for character encoding, allowing it to be used all over the world in a number of different computing environment. XML complements our programming language Java, another force for interoperability, very well. A considerable amount of early XML development has been in

[1] '.xsd' is the file extension of a schema file.

Java. Because the document structures behave consistently, Optimization Services parsers that interpret them can be built at relatively low cost in many languages. Also by storing an optimization instance in XML format, we are bringing the model closer to the data source and facilitating the integration of optimization-based solutions into IT infrastructures.

Presentable

XML-based Extensible Stylesheet Language (XSL) offers a convenient way to specify translations of XML documents. For example if an optimization solution is formatted in Optimization Services result Language (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.

Simple

XML provides both programmers and document authors with a friendly environment, at least by computing standards. XML documents are built upon a core set of basic nested structures. While the structures themselves can grow complex as layers of detail are added, the mechanisms underlying the structures require very little implementation effort. Furthermore, XML is well-documented. The W3C's XML specification and numerous books and resources tell people how to read and write XML data. At a low level, XML is a simple data format. XML can be written in pure ASCII text as well as a few other well-defined formats. At a higher level, XML is self-describing. Even though most of the Optimization Services representations are intended to be read by computer programs, they are certainly readable by humans. This certainly helps in developing and debugging the Optimization Services components.

4.2.2 XML basics and MathML

An XML representation consists of data delimited by `<element>` tags, much like an html representation of the content of a Web page. Each `<element>` tag can have space-delimited attributes in the form of "name=value" and can contain embedded elements:

```
<element1 attrName1="value1" attrName2="value2">
    <element1 ...>
        ...
    </element1>
    <element2 ...>
        ...
    </element2>
</element1>
```

Elements have to be closed with a start tag and an end tag such as:

```
<element ...>...</element>
```

If an element does not contain embedded elements, the start and end tags can be combined such as:

```
<element .../>
```

to save some space.

New collections of XML tags are defined for any specialized purpose by specifying a schema. One perceived disadvantage of an XML is its verbosity – the considerable file space taken up by tags – but in fact the tags only increase file size by a constant factor, which can be considerably reduced by use of optional alternatives to an ASCII representation [53].

An example of XML is given in Figure 4-5, expressed in MathML [96][109], a dialect of XML that is of some particular interest in this thesis. A dialect is an implementation of domain-specific XML notation governed by a standard schema designed to support languages such as chemical markup (CML), mathematical markup (MathML) and all the representation-related OSxL in Optimization Services. There are two flavors of MathML: Presentation MathML and Content MathML. Figure 4-5 shows the nonlinear expression $(2X_1 + 3X_2)^2$ in Presentation MathML, so-called because it describes math notation without trying to capture meaning.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE math PUBLIC "-//W3C//DTD MathML 2.0//EN"
"http://www.w3.org/TR/MathML2/dtd/mathml2.dtd">
<math xmlns="http://www.w3.org/1998/Math/MathML" xmlns:xlink="http://www.w3.org/1999/xlink">
    <msup>
        <mfenced>
            <mrow>
                <mrow>
                    <mrow>
                        <mn>2</mn>
                        <mo>&InvisibleTimes;</mo>
                        <mi>X1</mi>
                    </mrow>
                    <mo>+</mo>
                    <mrow>
                        <mn>3</mn>
                        <mo>&InvisibleTimes;</mo>
                        <mi>X2</mi>
                    </mrow>
                </mrow>
            </mrow>
        </mfenced>
        <mn>2</mn>
    </msup>
</math>
```

Figure 4-5: Expression $(2X_1 + 3X_2)^2$ in Presentation MathML.

Every XML document begins with an XML declaration, the first line in the above example. Then we define some overhead such as the schema location to validate the XML. But the more important part is that every MathML document starts with the root element `<math>`. Again there is some overhead in the root element such as defining namespaces used to qualify the elements and avoid potential naming conflicts. For simple clarification purposes, in the following XML examples, we skip the overhead parts.

As seen, Presentation MathML is mainly used to describe the layout structure or "rendering" of mathematical notation. Another set of MathML components, Content MathML, attempts to represent meaning. Content MathML is intended to provide an explicit encoding of the underlying mathematical structure without regard to how it is presented visually. Figure 4-6 shows the same expression $(2X_1 + 3X_2)^2$ in Content MathML.

```
<math>
    <apply>
        <power/>
        <apply>
            <plus/>
            <apply>
                <times/>
                <cn>2</cn>
                <ci>X1</ci>
            </apply>
            <apply>
                <times/>
                <cn>3</cn>
                <ci>X2</ci>
            </apply>
        </apply>
        <cn>2</cn>
      </apply>
</math>
```

Figure 4-6: Expression $(2X_1 + 3X_2)^2$ in Content MathML.

Content MathML still has `<math>` as its root element. The fundamental idea of Content MathML is to *apply* (therefore the element `<apply>`) functions and operators to other elements. To do this, Content MathML uses prefix notation. Prefix notation is when the operator comes first and is followed by operands. There are three functions shown in the above example: `<power>`, `<plus>`, and `<times>`. These functions are applied to number tokens (e.g. `<cn>2</cn>`), identifier tokens (e.g. `<ci>X1</ci>`), or expressions that again start with the `<apply>` element.

Content MathML allows information interchange to be more precise to software and systems that are able to manipulate the mathematics. Since optimization is about numerical computing, Content MathML can theoretically be used in optimization problems to represent mathematical expressions, especially nonlinear expressions. But in extending our Optimization Services instance Language (OSiL) from linear to nonlinear optimization, we decided against using Content MathML. Instead we designed our own Optimization Services nonlinear

Language (OSnL) to represent nonlinear expressions in OSiL. We summarize the main reasons below.

- Content MathML is too comprehensive. Content MathML is designed to support the needs of a very diverse set of users. It includes far more than is required in the modeling/optimization community. A significant number of features in Content MathML will never be used by optimization services and modeling systems (e.g. vector calculus, inclusion of Presentation MathML). If an instance unintentionally includes unnecessary MathML features which shouldn't be allowed, the MathML schema will still validate even though none of the solvers would ever recognize such features. We believe simplicity is a virtue and that means including only what is necessary.
- Content MathML is not specifically designed to represent instances or instance components of mathematical programs. OSiL is designed to represent instances of mathematical optimization problems and OSnL is designed to natively complement OSiL for nonlinear extensions. Certain features in OSnL that are critical in optimization such as XPath node, user functions, and variable subscripts are not naturally supported in Content MathML. For example, in Content MathML there is no built-in `<var>` tag to represent variables and variable subscripts. In OSiL a variable is naturally expressed as `<var idx="1"/>` and `<var idx="2"/>`. In the above MathML examples, we used `<ci>X1<ci>` and `<ci>X2<ci>` to artificially make up the variables. The concatenation of a variable name with an index can be confused with other identifiers. Alternatively, we might use `<ci><sub><mi>x</mi><mn>1</mn></sub></ci>` which is a hopelessly verbose and memory consuming way to express a subscript.
- Content MathML is not under control of the optimization community. This is perhaps the single most important reason not to use MathML. We can add optimization-related features to OSiL as needed. Using MathML to support optimization features is awkward at best, and it is unlikely we can get the W3C to adopt optimization-specific features in a timely fashion. Control of a standard for optimization is better left to an organization under the control of the Operations Research community.
- OSiL and OSnL are designed to be easily parsed and used by libraries in the OS API. Content MathML has elements in the following categories: tokens, constructors, operators and functions, qualifiers, constants and symbols, and semantic mapping elements. However, for representing mathematical expressions, OSnL has a very consistent recursive and object-oriented design where every element is an "nl node" that takes zero to an indefinite

number of children as arguments. This design results in extremely convenient parsing. There is a one-to-one mapping between XML DOM (see §4.4) parse tree elements and the corresponding OS Expression Tree. There is also a one-to-one correspondence between each node element in the OS Expression Tree and each node class in the parsing library API. Thus parsing an OSiL document is much easier than parsing a Content MathML document.

However, in order to be as consistent with MathML as possible we adopt the MathML element names whenever possible, for example `<power>` for the power function. Figure 4-7 shows the same expression $(2X_1 + 3X_2)^2$ in OSnL.

```
<nl idx="9">
    <power>
        <plus>
            <var idx="1" coef="2"/>
            <var idx="2" coef="3"/>
        </plus>
        <number value="2"/>
    </power>
</nl>
```

Figure 4-7: Expression $(2X_1 + 3X_2)^2$ in Optimization Services nonlinear Language (OSnL).

There are several things worth noticing. First OSnL is usually embedded in OSiL. The root element in the above example has an attribute `idx="9"` to indicate that it is part of the 9th constraint, whose linear expression part is to be found in the 9th row or constraint, of the OSiL instance. By separating out linear part from nonlinear part of an expression, we can take advantage of sparsity, which is a necessity in large-scale optimization. The second thing to notice is that we avoid the nuisance of unnecessary `<apply>` elements by adopting a recursive design. The result is a cleaner and shorter representation. The third thing to notice is that OSnL has the built-in `<var>` element which can take index ("`idx`") and coefficient ("`coef`") as its attribute. Variables appear so often in optimization that they have to be treated specially to make optimization practical. By designing an XML language natively tailored to optimization, we can achieve both efficiency and effectiveness in representation and communication.

We illustrate one more XML example in the context Optimization Services. Consider the following optimization problem instance which is based on an example of Rosenbrock (1960).

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & 100(x_1 - x_0^2)^2 + (1 - x_0)^2 + 7x_1 \\ \text{subject to} \quad & x_0 + 7x_1 \le 10 \\ & \ln(x_0 x_1) + 7x_0 + 5x_1 \le 10 \\ & x_0, x_1 \ge 0 \end{aligned} \qquad (4\text{-}2)$$

There are two continuous variables, x_0, x_1, in this problem, each with a lower bound of 0, and variable x_1 x_1 with an objective function coefficient of 7. This information is represented in OSiL in Figure 4-8.

```
<variables>
    <var lb="0" name="x0" type="C"/>
    <var lb="0" name="x1" type="C" objCoef="7.0"/>
</columns>
```

Figure 4-8: The OSiL `<variables>` element for the modified Rosenbrock problem in (4-2).

In the example, there are two kinds of elements: a `<variables>` element and a `<var>` element. The `<var>` element has attributes `lb`, `name`, `type`, and `objCoef` that further describe the properties of the variable that `<var>` represents.

Next, we describe the most important technical features of XML schema that are used in the Optimization Services representation design.

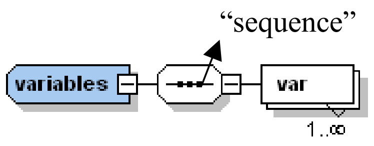
4.3 XML Schema

In order to facilitate communication between solvers and modeling languages, the instance files must conform to an accepted standard. Otherwise, parsing optimization instance files in a meaningful way is impossible. XML Schema is a database-inspired method for specifying constraints and enforcing standards on XML documents. XML Schema is itself an XML-based language.

Given an XML Schema, standard tools are available for parsing files that correspond to it, and for building libraries to display and manipulate the contents of these files [103][119]. For each OS representation language that we introduce for working with instances, we specify representation rules in XML Schema. Schemas are explained in detail in §4.3.

We can think of the schema as a class and an XML instance that conforms to the schema as an object or instance of the class. Just as a class very explicitly describes member and method names and properties, an XML Schema explicitly describes element and attribute names and properties.

For our `<variables>` element in Figure 4-8, Figure 4-9 shows a section of our OSiL Schema that specifies its structure both graphically and in text. Many of the schema examples in the later chapters are shown in this way.



```
<xs:element name="variables">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="var" type="var" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Figure 4-9: The `<variables>` element in OSiL Schema both graphically and in text.

In essence the schema means the `variables` element contains a sequence of 1 or more (1..∞) `var` elements of type also called `var`, which is defined below in Figure 4-10.

```
<xs:complexType name="var">
    <xs:attribute name="name" type="xs:string" use="optional"/>
    <xs:attribute name="init" type="xs:string" use="optional"/>
    <xs:attribute name="type" use="optional" default="C">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="C"/>
                <xs:enumeration value="B"/>
                <xs:enumeration value="I"/>
                <xs:enumeration value="S"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="lb" type="xs:double" use="optional" default="0"/>
    <xs:attribute name="ub" type="xs:double" use="optional" default="INF"/>
    <xs:attribute name="objCoef" type="xs:double" use="optional" default="0.0"/>
    <xs:attribute name="mult" type="xs:positiveInteger" use="optional" default="1"/>
</xs:complexType>
```

Figure 4-10: The `<var>` element in OSiL Schema.

This approach is very object oriented. The `<var>` type defined in Figure 4-10 is analogous to an abstract class in Java. In W3C XML Schema terminology it is called a named type. In order to actually have an instance file with `<var>` elements it is necessary to define in the schema, an element (a class) named `<var>` that is of type `var`. This is done in the part of the OSiL Schema illustrated in Figure 4-9. This allows an instance file to actually instantiate an instance of the `<var>` element.

In defining the `<var>` type element, only the attributes listed in Figure 4-10 are allowed to be present in a `var` element. All of these attributes are optional. Properties of the attributes are explicitly defined. For example, the `lb` attribute (variable lower bound) and the `ub` attribute (variable upper bound) must be double precision numbers and the `type` attribute (variable type) must be a string value that is either `C` (continuous), `B` (binary), `I` (integer), or `S` (string). In Chapter 6, we discuss in further detail the OSiL and OSnL schemas that are used to define an optimization instance representation. We briefly discuss the basic elements in XML Schema next.

Simple Types

We can have *simple types* and *complex types* in an XML Schema. The simple type is a restriction of the text that appears in an attribute or element. For example here is a simple type definition of an element.

```
<element name="source" type="xs:string" />
```

In this case, the defined element `source` cannot have attributes and can only contain text. We can define more complicated simple types such as

```
<xs:element name="maxOrMin" minOccurs="0">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:enumeration value="max"/>
            <xs:enumeration value="min"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>
```

In this example we are defining a simple type called `maxOrMin` (objective sense) that has as its base the type `string.` But we further restrict the text in the attribute to take on values of either `max` or `min`.

Complex Types

Complex types are elements that contain other elements or have attributes. There are two different complex types: *anonymous* and *named*. Here is an example of an anonymous complex type. In the tag `<xs:complexType>` there is no name, hence the term anonymous.

```
<xs:element name="con" minOccurs="0" maxOccurs="unbounded">
  <xs:complexType>
    <xs:attribute name="name" type="xs:string" use="optional"/>
    <xs:attribute name="ub" type="xs:double" use="optional"/>
     <xs:attribute name="lb" type="xs:double" use="optional"/>
     <xs:attribute name="mult" type="xs:int" use="optional"/>
   </xs:complexType>
</xs:element>
```

We could *not* use `con` (constraint) as a type in defining other elements. A named complex type is much like an abstract class in C++ or Java. That is, you cannot actually have an object in the class but you can have objects in classes derived from it. Below is an example of a named type `intVector`. In the `complexType` tag there is now an associated name, in this case `intVector.`

```
<xs:complexType name="intVector">
   <xs:choice>
      <xs:element name="base64BinaryData" type="base64BinaryData"/>
      <xs:element name="el" maxOccurs="unbounded">
         <xs:complexType>
            <xs:simpleContent>
               <xs:extension base="xs:nonNegativeInteger">
                  <xs:attribute name="mult" type="xs:positiveInteger" use="optional" default="1"/>
                  <xs:attribute name="incr" type="xs:int" use="optional"/>
               </xs:extension>
            </xs:simpleContent>
         </xs:complexType>
      </xs:element>
   </xs:choice>
</xs:complexType>
```

Here is another example of a named complexType.

```
<xs:complexType name="sparseVector">
   <xs:sequence>
      <xs:element name="idx" type="intVector" maxOccurs="unbounded"/>
      <xs:element name="nonz" type="doubleVector"/>
   </xs:sequence>
</xs:complexType>
```

Note that in this definition of a named complexType we are using the named type `<intVector>`. For example, the element `idx` is of type `intVector`. We could now use this `<sparseVector>` elsewhere as to define other elements. We can also define an anonymous complexType `sparseVector` that is of named type `sparseVector`. That is,

```
<xs:element name="sparseVector" type="sparseVector" minOccurs="0" maxOccurs="unbounded"/>
```

We can also do a kind of inheritance through extension. First we define a base class called `<baseProgramData>`.

```
<xs:complexType name="baseProgramData" mixed="false">
   <xs:sequence>
      <xs:element name="constraints" type="constraints" minOccurs="0"/>
      <xs:element name="variables" type="variables"/>
      <xs:element name="multiObjectives" type="multiObjectives" minOccurs="0"/>
      <xs:element name="coefMatrix" type="coefMatrix" minOccurs="0"/>
```

```
    </xs:sequence>
</xs:complexType>
```

Note in this definition the `mixed` attribute is set to `false`. This means that the `<baseProgramData>` element can only contain the specified elements. If the `mixed` attribute is set to `true`, the `<baseProgramData>` element can contain text or elements. Now extend this base class to allow more elements, such as `nl` (for nonlinear program extension) and `cones` (for cone programming extension).

```
<xs:complexType name="programData">
    <xs:complexContent>
        <xs:extension base="baseProgramData">
            <xs:sequence minOccurs="0" maxOccurs="unbounded">
                <xs:element name="nl" type="nl" minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="cones" type="cones" minOccurs="0"/>
                <xs:element name="stages" type="stages" minOccurs="0"/>
                <xs:element name="stochastic" type="stochastic" minOccurs="0"/>
                <xs:element name="userFunctions" type="userFunctions" minOccurs="0"/>
                <xs:element name="simulations" type="simulations" minOccurs="0"/>
                <xs:element name="xmlData" type="xmlData" minOccurs="0"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

Similar to the extension, we can also define a restriction on a complex type base. But it is less often used.

Substitution Groups

When deriving a new complex type by extension one can only add new elements or attributes to the base type. When deriving a new complex type by restriction one can only put additional restrictions on existing elements and attributes. Substitution groups allow a new content model. They are somewhat like the concept of polymorphism in object oriented programming in that you can substitute any type in a substitution group for the base type.

For example, to represent a generic tree node (operator or operand) for a nonlinear expression, in our OSnL Schema, we create `OSnLNode`, a complex type that effectively is like a Java abstract class.

```
<xs:complexType name="OSnLNode" mixed="false">
        <xs:annotation>
            <xs:documentation>This is a generic node from which we derive operator
nodes</xs:documentation>
        </xs:annotation>
</xs:complexType>
```

The `annotation` element is just an XML Schema comment that can be ignored. Then we create a substitution group based on the named element `OSnLNode` that is of type OSnLNode. So we can think of `OSnLNode` as a derived class.

```
<xs:element name="OSnLNode" type="OSnLNode" abstract="true">
    <xs:annotation>
        <xs:documentation> Set abstract to true in order to create a substitution group</xs:documentation>
    </xs:annotation>
</xs:element>
```

Note the `abstract` attribute is set to the value of `true` in order to create the abstract class.

Next, we create the actual elements that are in the substitution group for `OSnLNode`. For example, we might have an `OSnLNode` that corresponds to subtraction. First we create the abstract class for this operation.

```
<xs:complexType name="OSnLNodeMinus">
    <xs:complexContent>
        <xs:extension base="OSnLNode">
            <xs:sequence minOccurs="2" maxOccurs="2">
                <xs:element ref="OSnLNode"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

From this abstract class `OSnLNodeMinus` we create the derived element `minus` that is in the substitution group `OSnlNode`.

```
<xs:element name="minus" type="OSnLNodeMinus" substitutionGroup="OSnLNode"/>
```

Note that the `minus` element requires exactly two child elements.

In a similar fashion, we define all other OSnL nodes such as `plus`, `times`, `sin`, `sum`, `PI`, `var`, `geq`, `if`, `PI`, `xPath`, `userF`, `quadratic.` For nodes such as `sin`, the corresponding `OSnLNodeSin` requires exactly one chide element, as the `sin` operator is a unary type. For nodes such as `sum`, the corresponding `OSnLNodeSum` requires one or more child elements, as the `sum` operator is an indefinite type. For nodes such as `PI`, the corresponding `OSnLNodePI` requires zero child elements, as the `PI` operator is a constant.

Essentially every operator or operand that appears in an expression tree is generically regarded as a *node* of type `OSnLNode`. This objected-oriented style treatment provides a significantly simple and powerful way to construct a nonlinear expression So for example, to add nonlinear extensions to our OSiL, we simply define an element `nl` that holds the nonlinear term for a row specified by the attribute `idx`, which indicates a row number of an objective or constraint. As shown below, each `nl` element has exactly one child elements, the expression tree root, which can be anything in the substitution group for `OSnLNode`. Of course, we do not know ahead of time whether the tree root will be a `plus` or a `times` node. But whatever it may become, it has to be of a generic type `OSnLNode`.

```
<xs:element name="nl" type="nl" minOccurs="0" maxOccurs="unbounded"/>
<xs:complexType name="nl">
    <xs:sequence minOccurs="1" maxOccurs="1">
        <xs:element ref="OSnLNode"/>
    </xs:sequence>
    <xs:attribute name="idx" use="required" type="xs:int"/>
</xs:complexType>
```

When a concrete expression tree is finally constructed, it may look like

```
<nl idx="9">
    <power>
        <plus>
            <var idx="1" coef="2"/>
            <var idx="2" coef="3"/>
        </plus>
        <number value="2"/>
    </power>
</nl>
```

for the nonlinear expression $(2X_1 + 3X_2)^2$ that appears in the 9th row (or constraint) of a mathematical program. Every node in this expression tree has to follow the constraints specified by the node's corresponding type, e.g. the number of child nodes it can have.

Namespaces

It is possible for different XML vocabularies to use the same element name, yet the element has a different meaning depending on the vocabulary. For example in one vocabulary the element `<title>` might have a very different meaning than in another vocabulary. Furthermore, when developing vocabulary `a.xsd` one might wish to borrow elements from another vocabulary `b.xsd` or allow elements from vocabulary `b.xsd` to be used instead of elements from `a.xsd`. For example, we have developed our instance representation language `OSiL`. A user might wish to use our other optimization languages and services but use, for example, `MathML` for instance representation. This is easily accomplished through the use of namespaces. The local element together with the name space determines a globally unique name known as a *qualified name*.

Assume a user wishes to represent a math program using `MathML` rather than `OSiL`. They can simply put the nonlinear program inside the `<math>` tag and use the appropriate name space. For example, one approach is:

```
<math xmlns="http://www.w3.org/1998/Math/MathML" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://www.w3.org/1998/Math/MathML
http://www.w3.org/Math/XMLSchema/mathml2/mathml2.xsd">
```

```
<apply>
  <power/>
    <ci>x</ci>
    <cn>2</cn>
</apply>
</math>
```

This syntax declares that the `<math>` element and all of its children are in the MathML vocabulary. That is, all of the elements are qualified and are in the default namespace MathML.

An alternative way to qualify the elements is through the use of a prefix.

```
<ml:math xmlns:ml="http://www.w3.org/1998/Math/MathML" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://www.w3.org/1998/Math/MathML
http://www.w3.org/Math/XMLSchema/mathml2/mathml2.xsd">
  <ml:apply>
    <ml:power/>
      <ml:ci>x</ml:ci>
      <ml:cn>2</ml:cn>
  </ml:apply>
</ml:math>
```

Now, if we had written `<ci>x</ci>` instead of `<ml:ci>x</ml:ci>`, then the `<ci>` would be unqualified and potentially confused with an element defined in other schemas that is named the same.

The namespace that we use to qualify all the OSxL schemas is "`os.optimizationservices.org`" and it should be different from any other namespaces in the world as we have reserved the domain name "optimizationservcies.org."

Import and Include

When working in the same name space it is often convenient to organize a set of schemas in different files. We can then use one schema in another through the `include` element. For example, in the instance language `OSiL.xsd`, we need to define the `nlNode` element that is in the nonlinear language `OSnL.xsd`. To do this, we use the `include` statement as follows.

```
<xs:include schemaLocation="OSnL.xsd"/>
```

When schemas are in different name spaces we need the `import` element. For example, in `OSiL`, we allow the use of `MathML` to describe a nonlinear program. In order to validate a document against the `os.optimizationservices.org` namespace we need to import the `MathML` namespace. This is done as follows. First, in the root element we include the attribute:

```
xmlns:mathML="http://www.w3.org/1998/Math/MathML"
```

Then we include an `<import>` element as follows:

```
<xs:import namespace="http://www.w3.org/1998/Math/MathML"
schemaLocation="http://www.w3.org/Math/XMLSchema/mathml2/mathml2.xsd"/>
```

Then in the schema we declare a `<math>` element.

```
<xs:element ref="mathML:math" minOccurs="0"/>
```

In this the case the `<math>` element is in the MathML namespace. However, we can achieve the same result with

```
<xs:element name="math" type="mathML:math.type" />
```

in which case the `<math>` element is in the `os.optimizationservices.org` namespace.

As a second example in `OSiL` we build up our nonlinear terms recursively through the abstract element `OSnLNode`. We can allow users to include an `OSnLNode` element that is a `MathML` expression. First we import the `MathML` namespace as we illustrated above. Next we define the `<math>` root element in the `MathML` vocabulary as follows:

```
<xs:element name="math" type="mathML:math.type" substitutionGroup="nlNode"/>
```

Then we define an `OSnLNode` which is really `MathML`.

```
<nl idx="2">
  <math xmlns:mathML="http://www.w3.org/1998/Math/MathML">
    <mathML:apply>
      <mathML:power/>
        <mathML:ci>x</mathML:ci>
        <mathML:cn>2</mathML:cn>
    </mathML:apply>
  </math>
</nl>
```

In this example, the element `<math>` is actually in the namespace `os.optimizationservices.org`, but its children are in the `MathML` namespace[1].

4.4 Other XML Technologies

In this section, we briefly describe other XML technologies used in the Optimization Services project and their corresponding references.

<u>SAX and DOM Parsing Models</u>

Our OSiL instance is used to link modeling languages with solvers, typically over a network. In our design, we expect a library/API to sit between the two and translate the XML

[1] In the first release of OSiL, we have taken out all the MathML related elements. It may or may not be added in the later releases.

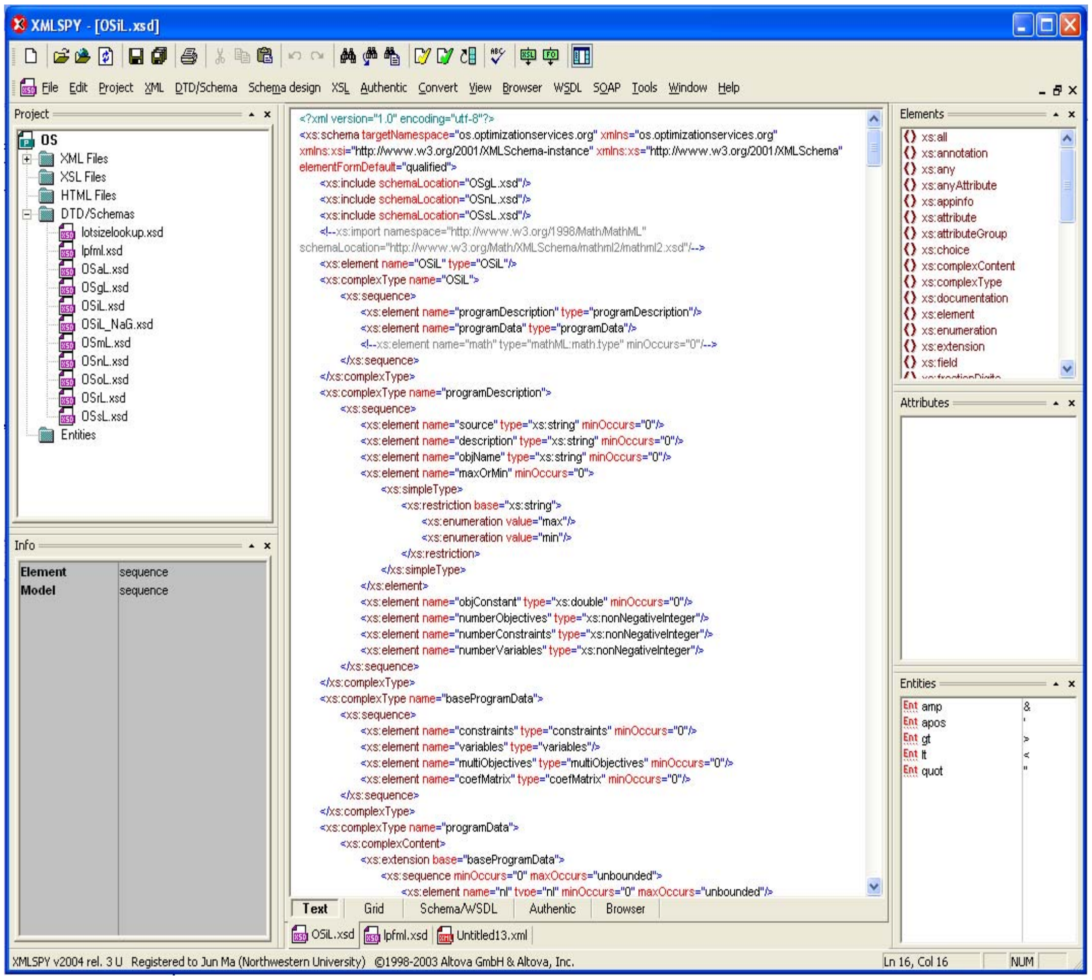
instance into a format that the solver can understand. To this end, the XML file must be parsed. There are two basic approaches to parsing an XML file: Simple API for XML (SAX) [78] and Document Object Model (DOM) [105]. Both are APIs that are used to translate XML documents to some format suitable for use by computer programs. To construct an XML document, DOM is used. To parse an XML document, both DOM and SAX can be used.

SAX is a set of streaming interfaces that decompose the XML documents into a sequence of predefined method calls and fire events when elements and attributes are read. SAX does not store the information in an element or attribute after it is initially read. Because of this, SAX is very efficient and has low memory requirements. But when reading through an XML document, all the previously read sections have to be remembered (stored in memory) for parsing the later sections, so SAX may become less desirable to use. This is the case in reading an optimization instance.

DOM is a set of traversal interfaces that decompose the XML documents into a hierarchal tree of generic nodes. With this approach, the XML document is read into a tree-like data structure and held in memory. In most of our parser library implementations, we use the DOM instead of SAX and then transfer the information from the DOM into our OS Expression Tree. We selected the DOM because it is easier to work with. For example, we have numerous error checking routines to make sure the data is consistent and these routines require keeping information about the problem in memory – information that is lost using SAX.

XML Authoring Tools

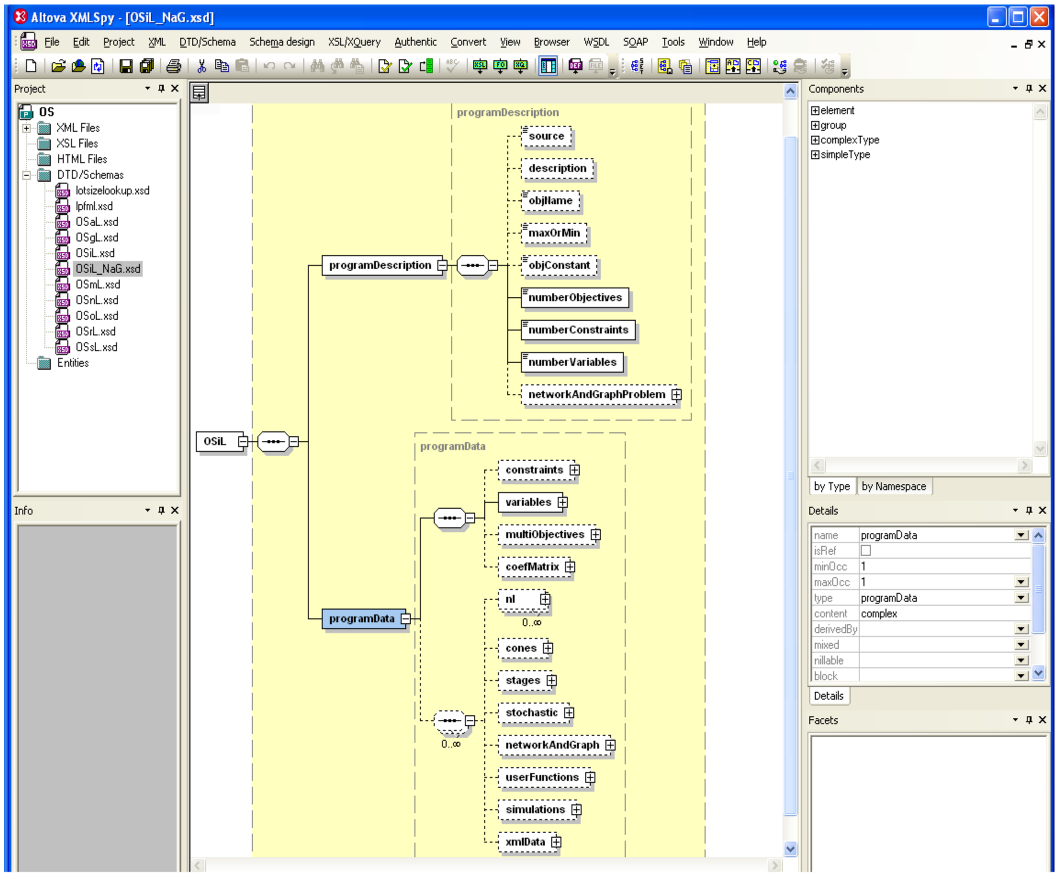
XML Authoring tools assist in editing XML documents or validating XML syntaxes. XML documents can be XML Schemas as well as regular XML instances. The Optimization Services project, for example, uses Altova's XML Spy [1] and Progress Software's Stylus Studio [96]. Both XML Spy and Stylus Studio are comprehensive IDEs for developing XML projects. They provide efficient and flexible environments for creating and editing XML Schemas, XML instances, XQuery and XSLT style sheets. This thesis mostly uses both XML Spy's text view (Figure 4-11) and graphical view (Figure 4-12) for design illustrations.



```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema targetNamespace="os.optimizationservices.org" xmlns="os.optimizationservices.org"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
    <xs:include schemaLocation="OSgL.xsd"/>
    <xs:include schemaLocation="OSnL.xsd"/>
    <xs:include schemaLocation="OSsL.xsd"/>
    <!--xs:import namespace="http://www.w3.org/1998/Math/MathML"
schemaLocation="http://www.w3.org/Math/XMLSchema/mathml2/mathml2.xsd"/-->
    <xs:element name="OSiL" type="OSiL"/>
    <xs:complexType name="OSiL">
        <xs:sequence>
            <xs:element name="programDescription" type="programDescription"/>
            <xs:element name="programData" type="programData"/>
            <!--xs:element name="math" type="mathML:math.type" minOccurs="0"/-->
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="programDescription">
        <xs:sequence>
            <xs:element name="source" type="xs:string" minOccurs="0"/>
            <xs:element name="description" type="xs:string" minOccurs="0"/>
            <xs:element name="objName" type="xs:string" minOccurs="0"/>
            <xs:element name="maxOrMin" minOccurs="0">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="max"/>
                        <xs:enumeration value="min"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:element>
            <xs:element name="objConstant" type="xs:double" minOccurs="0"/>
            <xs:element name="numberObjectives" type="xs:nonNegativeInteger"/>
            <xs:element name="numberConstraints" type="xs:nonNegativeInteger"/>
            <xs:element name="numberVariables" type="xs:nonNegativeInteger"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="baseProgramData">
        <xs:sequence>
            <xs:element name="constraints" type="constraints" minOccurs="0"/>
            <xs:element name="variables" type="variables"/>
            <xs:element name="multiObjectives" type="multiObjectives" minOccurs="0"/>
            <xs:element name="coefMatrix" type="coefMatrix" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="programData">
        <xs:complexContent>
            <xs:extension base="baseProgramData">
                <xs:sequence minOccurs="0" maxOccurs="unbounded">
                    <xs:element name="nl" type="nl" minOccurs="0" maxOccurs="unbounded"/>
```

XMLSPY v2004 rel. 3 U Registered to Jun Ma (Northwestern University) ©1998-2003 Altova GmbH & Altova, Inc.

Figure 4-11: Text view of an XML file (OSiL) in XML Spy.



name	programData
isRef	☐
minOcc	1
maxOcc	1
type	programData
content	complex
derivedBy	
mixed	
nillable	
block	

Figure 4-12: Graphical view of an XML file (OSiL) in XML Spy.

XSL Transformation Tools

XML Transformation tools assist in transforming XML into something that can be displayed in a browser or other rendering device. XSL [114], and its associated language XSLT [115], is the main tool here. XSLT, is an acronym for Extensible Stylesheet Language Transformation, is itself an XML-based declarative programming language to transform XML files into HTML files, or other XML files, or any other plain text files. The following XSL example retrieves all the variable names from an Optimization Services instance Language (OSiL, Chapter 6).

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:os="os.optimizationservices.org" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
    <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>
    <xsl:template match="/">
        <html>
            <body>
                <hr/>
```

```
                    <h1>Optimization Services instance Language Variables</h1>
                    <p/>
                    <xsl:for-each select="os:OSiL/os:programData/os:variables/os:var">
                         <p/>varialbe: <xsl:value-of select="@name"/>
                    </xsl:for-each>
                    <hr/>
               </body>
          </html>
     </xsl:template>
</xsl:stylesheet>
```

The result looks something like:

Optimization Services instance Language Variables

variable: x1

variable: x2

variable: y1

variable: y2

Figure 4-13 shows how the combination of XML and XSLT can serve at least the same purpose as HTML over the Internet. XSLT can be used for example to nicely display optimization results formatted in Optimization Services result Language (OSrL, Chapter 6).

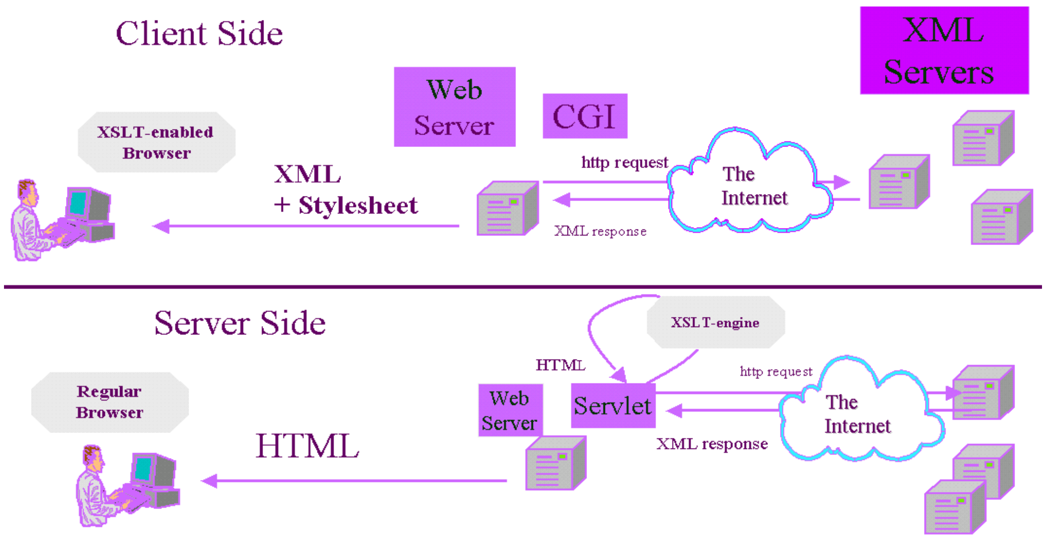


Figure 4-13: An illustration of how the combination of XML and XSL style sheet can serve as the same purpose of HTML.

XLink and XPointer

XLink [108] and XPointer [111] are used to link and reference information within an XML document. XLink is a generalization of the HTML link concept, though it is at a higher abstraction level intended for general XML – not just hypertext. Thus it has more expressive power, such as multiple destinations, special behaviors, and link bases. Linking elements are identified by an `xlink:form` attribute with either the value "simple" or "extended." Furthermore, each linking element contains an `href` attribute whose value is the URI of the linked resource. An XLink example is shown below:

```
<OSiLSchema xlink:form="simple" href="http://www.optimizationservices.org/schemas/OSiL.xsd">Optimization Services instance Language</OSiLSchema>
```

HTML links generally point to a particular document. Additional anchors (pointing to a particular section, chapter, or paragraph of a particular document) are not well-supported. Unlike HTML anchors, XPointers not only allow pointing to a point in a document, but also allow pointing to ranges or spans, such as the root element of an XML document. An XPointer is usually appended to an XLink or URL separated by a "#" sign as in the following example: http://www.optimizationservices.org/schemas/OSoL.xsd#root() .

XPointer is sort of an extension to XPath (described next) to support linking. It specifies connections between XPath expressions and Uniform Resource Identifiers (URIs or more plainly, globally unique addresses). XPath, XLink and XPointer are especially useful when some of the function evaluations in optimization problems can only be obtained from a remote Web service.

XPath

XPath [110] is used to locate data in an XML file. It is a declarative language used to identify subsets (nodes and fragments) of an XML document. XPath is designed standalone, but it can also be used in XSLT (for pattern matching), XPointer (for addressing), XQuery (for selection and iteration) and XML Schema (for uniqueness and scope description).

Unlike many other XML technologies, XPath uses a compact, non-XML format to facilitate use within URIs and attribute values. XPath views an XML document as a tree, containing different kinds of nodes. The XML node types include root, element, text, attribute, namespace etc. XPath imposes a document order (order of occurrence of element start tags) defined on all nodes except attribute and namespace nodes. The root is always the first node. Root and element nodes have an ordered list of children. An element node is the parent of the associated set of attribute/namespace nodes, the attributes/namespaces are not children of the associated element node.

For example, given the following XML file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<stocks>
    <stock name="ge" idx="2" ret=".03" mininv=".1">
        <cov name="msft" idx="0" val="25"/>
        <cov name="pg" idx="1" val="37"/>
        <cov name="ge" idx="2" val="19"/>
    </stock>
    <stock name="msft" idx="0" ret=".07" mininv=".1">
        <cov name="msft" idx="0" val="24"/>
        <cov name="pg" idx="1" val="-10"/>
        <cov name="ge" idx="2" val="25"/>
    </stock>
    <stock name="pg" idx="1" ret=".09" mininv=".1">
        <cov name="msft" idx="0" val="-10"/>
        <cov name="pg" idx="1" val="75"/>
        <cov name="ge" idx="2" val="37"/>
    </stock>
</stocks>
```

the XPath to find the return value of the Microsoft stock (ticker: msft) is:

```
stocks/stock[@name='msft]'/@ret[1]
```

Since XPath 2.0, the XPath language has become a strict syntactic subset of XQuery 1.0, described next.

XQuery

XQuery [112] is a query language for retrieving data items from an XML document. XQuery is designed to meet the needs of the database world and the document processing world. XQuery is to XML what SQL is to relational databases. XQuery is used in our Optimization Services (OS) Registry (Chapter 8) implementation to find registered OS solvers in a native XML database.

From the W3C XQuery [112] introduction page: "The mission of [XQuery] is to provide flexible query facilities to extract data from real and virtual documents on the Web, therefore finally providing the needed interaction between the web world and the database world. Ultimately, collections of XML files will be accessed like databases." With XQuery, we now have a standard syntax by which XML processors can access XML data or non-XML data exposed as virtual XML documents. XQuery expressions can replace procedural code that generates new XML structures from other XML data. Thus XQuery enables robust queries across a large set of XML documents or virtual XML data sources. The OSmL modeling language we present in Chapter 9 is based upon the XQuery standard and is designed to convert raw data in XML format into problem instances that conform to the Optimization Services instance language (OSiL) standard.

Like XPath, XQuery uses a compact, non-XML format. XQuery is essentially an extension to XPath. Where XPath serves simply to address XML document components and return result sets, XQuery adds the ability to combine the result set with locally-defined elements in order to create new XML structures. XQuery includes looping and conditional constructs that XPath 1.0 does not have. XQuery also adds a large number of new functions, as well as built-in support for XML Schema data types.

For example given the following XML file:

```
<bib>

   <book>
      <title>Large Scale Linear and Integer Optimization: A Unified Approach</title>
      <author>Martin</author>
      <publisher>Kluwer Academic Press</publisher>
   </book>
   <book>
      <title>The Essential Guide to Internet Business Technology</title>
      <author>Honda</author>
      <author>Martin</author>
      <publisher>Prentice Hall</publisher>
   </book>
   <book>
      <title>AMPL</title>
      <author>Fourer</author>
      <author>Gay</author>
      <author>Kernighan</author>
      <publisher> Duxbury Press </publisher>
   </book>
</bib>
```

the XQuery to find the titles of all the books written by each distinct author is:

```
for  $a in fn:distinct-values(//author)
return (xs:string($a),
    for $b in //book[author = $a]
    return $b/title)
```

and the XQuery result is:

```
Martin
<title>Large Scale Linear and Integer Optimization: A Unified Approach</title>
<title>The Essential Guide to Internet Business Technology</title>
Honda
<title>The Essential Guide to Internet Business Technology</title>
Fourer
<title>AMPL</title>
Gay
<title>AMPL</title>
Kernighan
<title>AMPL</title>
```

4.5 Web Services and Simple Object Access Protocol (SOAP)

Web services are an evolving, middleware platform that facilitate *program-to-program* interactions. W3C's official definition of Web services [116] is as follows:

"A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

More plainly, Web services are **platform and implementation independent** components that are **described** using a service description language, **published** to a registry of services, **discovered** through a standard mechanism (at runtime or design time), **invoked** through a declared API (usually over a network), and **composed** with other services.

"**Platform and implementation independent**" means a client of the service can not tell what language, operating system, or computer type the service uses. It is achieved through the Simple Object Access Protocol (SOAP, see in this section below).

"**Described**" means that a Web service must describe itself, mainly in terms of what requests are allowed, what the arguments are and which transport it uses. This is achieved through the protocol of Web Services Description Language (WSDL, §4.7).

"**Published**" means that a Web service must tell a registry service where it is located (like "yellow pages"). It is achieved through the protocols of Web Services Inspection Language (WSIL, §4.8), Universal Description, Discovery and Integration (UDDI §4.8), or customized domain specific registry services as in the case of the Optimization Services registry (Chapter 8).

"**Discovered**" means that a Web service's potential clients can find it in a registry service. This is also achieved through the same protocols and registry services as those in the Web service publication.

"**Invoked**" means that the arguments and return types are known. This is achieved through the protocol of SOAP.

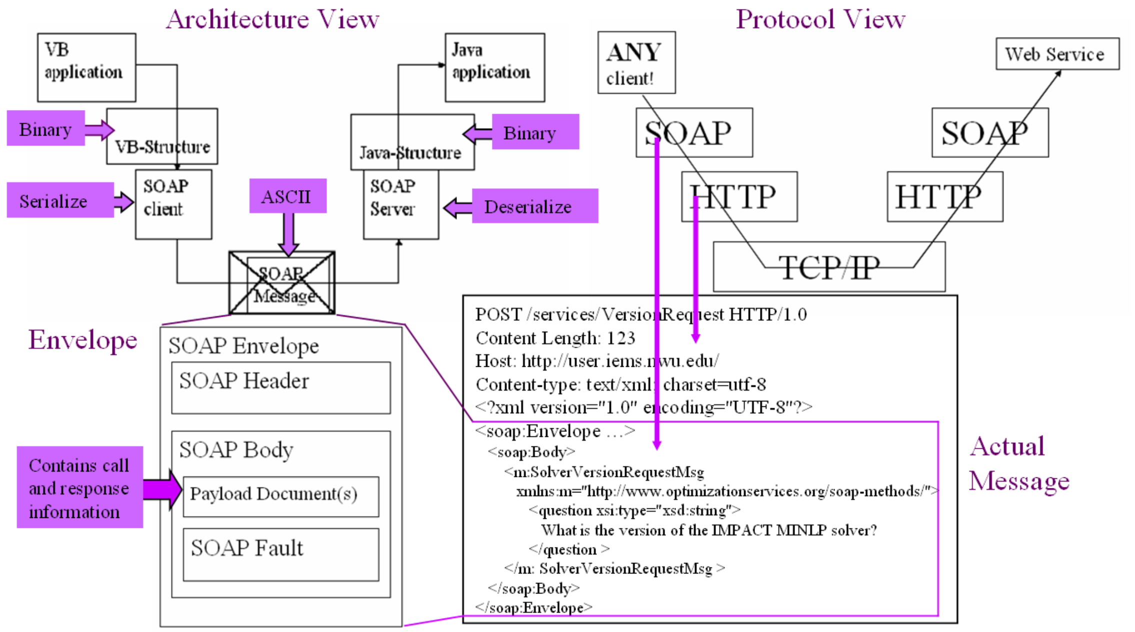
"**Composed**" means that a service can also be a client. It is also achieved through the protocol of SOAP.

In Chapter 5, we illustrate all the above mechanisms in the context of Optimization Services.

The core of Web services is the SOAP protocol for information exchange. The World Wide Web Consortium (W3C) released its first *recommended* version, SOAP 1.2, on June 24, 2003. SOAP Version 1.2 is a relatively simple and powerful XML-based protocol intended for exchanging structured information in a decentralized, distributed environment such as the Web. A W3C Recommendation is the equivalent of a Web standard, indicating that this W3C-developed specification is stable, contributes to Web interoperability, and has been reviewed by the W3C Membership, who favors its adoption by industry.

SOAP allows calls to remote objects' methods and access to remote objects' data using standard Web services, the standard HTTP protocol for those services, and XML to describe the call. SOAP is intended to serve as a more general and flexible successor to DCOM and CORBA mentioned in the beginning of this chapter. Figure 4-14 gives an illustration from the architecture view, the protocol view, the SOAP envelope structure view and the actual HTTP/SOAP message view.

In the architecture view, a user constructs an application in any language (e.g. Visual Basic). The purpose of the application is to call, as a client, a remote application or Web service on the network, again written in any language (e.g. Java). The client's VB structure is serialized (that is transformed from binary to ASCII) through a SOAP client and into a SOAP message. The SOAP message is then transmitted via the network to the remote application service. At the remote end, the SOAP message is deserialized from its ASCII XML form into a binary Java structure, before the application service executes the request call. A response is returned in the same way.



```
POST /services/VersionRequest HTTP/1.0
Content Length: 123
Host: http://user.iems.nwu.edu/
Content-type: text/xml; charset=utf-8
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope ...>
  <soap:Body>
    <m:SolverVersionRequestMsg
      xmlns:m="http://www.optimizationservices.org/soap-methods/">
        <question xsi:type="xsd:string">
          What is the version of the IMPACT MINLP solver?
        </question >
    </m: SolverVersionRequestMsg >
  </soap:Body>
</soap:Envelope>
```

Figure 4-14: SOAP illustration from high to low level.

In the network view protocol, all the information needed for the client call is stored in a SOAP envelope. A SOAP envelope is usually packed inside an HTTP protocol. From that point on, the HTTP packet is transmitted over a TCP/IP transport the same way that an HTTP request for a Web page is transmitted. The only difference is that a request for a Web page usually contains HTTP content such as GET or POST methods for an HTML document, whereas a request for a Web service always contains a SOAP envelope.

Suppose we want to send the problem in (4-3) to the Lindo solver service hosted at http://www.optimizationservices.org/os/LindoSolverService.jws.

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & (1-x_1)^2 + 100(x_0 - {x_1}^2)^2 \\ \text{subject to} \quad & x_0 + x_1 - 100 \le 0 \end{aligned} \tag{4-3}$$

We can call the method `String solve(String OSiL)` in LindoSolverService.jws. The argument `OSiL` is an instance representation of problem (4-3). The returned `String` is an instance representation of the problem solution. There are many libraries (including the OS library in Appendix B) that help parsing and sending XML instances.

First an HTTP header with the POST method is prepared like the following:

```
POST /os/ossolver/LindoSolverService.jws HTTP/1.0

Content-Type: text/xml; charset=utf-8
Accept: application/soap+xml, application/dime, multipart/related, text/*
User-Agent: Axis/1.2beta3
Host: http://www.optimizationservices.org
Cache-Control: no-cache
Pragma: no-cache
Content-Length: 2488

<soapenv:Envelope …>
…
</soapenv:Envelope>
```

Since this is an HTTP POST, we attach the POST data – the SOAP envelope – at the end of the HTTP header with a line separation (i.e. two new line characters).

Inside the SOAP envelope, it is essentially a SOAP encoding of the LindoSolverService method `String solve(String OSiL)` with the actual `OSiL` string argument:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Body>
        <ns1:solve soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:ns1="http://www.optimizationservices.org/os/ossolver/LindoSolverService.jws">
            <OSiL ...>
                ...
            </OSiL>
        </ns1:solve>
    </soapenv:Body>
</soapenv:Envelope>
```

Optimization Services Protocol further specifies that the method signature `solve(String)` should be exactly the same as specified in the Optimization Services hookup Language (OShL Chapter 7) and the actual OSiL string argument should follow the OSiL Schema (Chapter 6):

```
<OSiL xmlns="os.optimizationservices.org" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="os.optimizationservices.org   http://www.optimizationservices.org/schemas/OSiL.xsd">
   <programDescription>
      <!--simulation-->
      <maxOrMin>min</maxOrMin>
      <numberObjectives>1</numberObjectives>
      <numberConstraints>1</numberConstraints>
      <numberVariables>2</numberVariables>
   </programDescription>
   <programData>
      <constraints>
         <con ub="0.0"/>
      </constraints>
      <variables>
         <var lb="0" name="x1" type="C"/>
         <var lb="0" name="x2" type="C"/>
      </variables>
```

```
        <nl idx="-1"><plus><power><minus><number type="real" value="1.0"/><var coef="1.0"
idx="1"/></minus><number type="real" value="2.0"/></power><times><number type="real"
value="100"/><power><minus><var coef="1.0" idx="0"/><power><var coef="1.0" idx="1"/><number type="real"
value="2.0"/></power></minus><number type="real" value="2.0"/></power></times></plus></nl>
        <nl idx="0"><minus><plus><var coef="1.0" idx="0"/><var coef="1.0" idx="1"/></plus><number
type="real" value="100"/></minus></nl>
    </programData>
</OSiL>
```

SOAP, however, has its own set of encoding rules; for example it represents < with `&lt;` and > with `&gt;`. So in an actual SOAP message over the network, the above OSiL string looks like

```
&lt;OSiL xmlns="os.optimizationservices.org" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="os.optimizationservices.org   http://www.optimizationservices.org/schemas/OSiL.xsd"&gt;
    &lt;programDescription&gt;
        &lt;!--simulation--&gt;
        &lt;maxOrMin&gt;min&lt;/maxOrMin&gt;
        &lt;numberObjectives&gt;1&lt;/numberObjectives&gt;
        &lt;numberConstraints&gt;1&lt;/numberConstraints&gt;
        &lt;numberVariables&gt;2&lt;/numberVariables&gt;
    &lt;/programDescription&gt;
    &lt;programData&gt;
        &lt;constraints&gt;
            &lt;con ub="0.0"/&gt;
        &lt;/constraints&gt;
        &lt;variables&gt;
            &lt;var lb="0" name="x2" type="C"/&gt;
            &lt;var lb="0" name="x1" type="C"/&gt;
        &lt;/variables&gt;
        &lt;nl idx="-1"&gt;&lt;plus&gt;    &lt;power&gt;&lt;minus&gt;&lt;number type="real" value="1.0"/&gt;
    &lt;var coef="1.0" idx="1"/&gt;&lt;/minus&gt;&lt;number type="real"
value="2.0"/&gt;&lt;/power&gt;&lt;times&gt;&lt;number type="real" value="100"/&gt;&lt;power&gt;&lt;minus&gt;
    &lt;var coef="1.0" idx="0"/&gt;&lt;power&gt;&lt;var coef="1.0" idx="1"/&gt;&lt;number type="real"
value="2.0"/&gt;&lt;/power&gt;&lt;/minus&gt;&lt;number type="real"
value="2.0"/&gt;&lt;/power&gt;&lt;/times&gt;&lt;/plus&gt;&lt;/nl&gt;
        &lt;nl idx="0"&gt;&lt;minus&gt;&lt;plus&gt;    &lt;var coef="1.0" idx="0"/&gt;&lt;var coef="1.0" idx="1"/&gt;
    &lt;/plus&gt;&lt;number type="real" value="100"/&gt;&lt;/minus&gt;&lt;/nl&gt;
    &lt;/programData&gt;
&lt;/OSiL&gt;
```

Usually a SOAP envelope contains two sections: SOAP header (optional and not shown in the above example) and SOAP body. The SOAP Header mainly has some administrative information to complete a call. The SOAP body contains the major request and response information, for example call methods and arguments. The SOAP body also contains a subsection, SOAP fault, which specifies exception errors returned by the called Web service. For example, if the problem is solved successfully, the Lindo solver service should return an optimal solution of (1.0, 1.0) with an objective value 0.0 for the problem (4-3) in the following SOAP envelope (again with < encoded as `&lt;` and > encoded as `&gt;` for the result string):

```
HTTP/1.1 200 OK
Set-Cookie: JSESSIONID=A8AF406536A271018100F64CFA462FA0; Path=/os
Content-Type: text/xml;charset=utf-8
Date: Sun, 20 Mar 2005 21:28:40 GMT
Server: Apache-Coyote/1.1
```

```
Connection: close

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Body>
        <ns1:solveResponse soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:ns1="http://www.optimizationservices.org/os/ossolver/LindoSolverService.jws">
            <solveReturn xsi:type="xsd:string">
                &lt;OSrL xmlns="os.optimizationservices.org"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="os.optimizationservices.org
           http://www.optimizationservices/schemas/OSrL.xsd"&gt;
                    &lt;result&gt;
                        &lt;status type="optimal"/&gt;
                        &lt;objective&gt;
                            &lt;objectiveValue value="0.000"/&gt;
                        &lt;/objective&gt;
                        &lt;variables&gt;
                            &lt;variableSolution&gt;
                                &lt;description/&gt;
                                &lt;var idx="0" varName="x1" value="1.0"/&gt;
                                &lt;var idx="1" varName="x2" value="1.0"/&gt;
                            &lt;/variableSolution&gt;
                        &lt;/variables&gt;
                    &lt;/result&gt;
                &lt;/OSrL&gt;
            </solveReturn>
        </ns1:solveResponse>
    </soapenv:Body>
</soapenv:Envelope>
```

Notice the HTTP status of 200 OK in the first line. Optimization Services Protocol further specifies that the actual returned string argument should follow the OSrL (Optimization Services result Language Chapter 6) schema.

If the problem is not solved successfully or a networking error occurs, the following message would be returned with a SOAP fault element:

```
HTTP/1.1 500 Internal Server Error

Set-Cookie: JSESSIONID=8AEFE9B91BD586ABFD237F7EEDAAC267; Path=/os
Content-Type: text/xml;charset=utf-8
Date: Sun, 20 Mar 2005 23:07:20 GMT
Server: Apache-Coyote/1
Connection: close

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Body>
        <soapenv:Fault>
            <faultcode>soapenv:Server.userException</faultcode>
            <faultstring>java.lang.NullPointerException</faultstring>
            <detail>
                <ns1:hostname xmlns:ns1="http://xml.apache.org/axis/"> A null pointer
    exception</ns1:hostname>
            </detail>
        </soapenv:Fault>
    </soapenv:Body>
 </soapenv:Envelope>
```

Notice the HTTP status of 500 Internal Server Error in the first line.

As shown in the actual message part of Figure 4-14 and also illustrated in the above examples, the realization of SOAP Envelope, Header, Body and Fault is purely through XML representation. This is one major difference between SOAP and all other major networking protocols. It starts a standard for newly developed network protocols, including Optimization Services Protocol.

4.6 Service Oriented Architecture (SOA)

Early in the Web services history, people noticed a pattern. Each time they applied Web services technologies to an integration solution, the pattern would appear. They generalized the pattern and named it *Service-oriented architecture* (SOA). SOA is a simple concept. Figure 4-15 shows the main components and operations of an SOA.

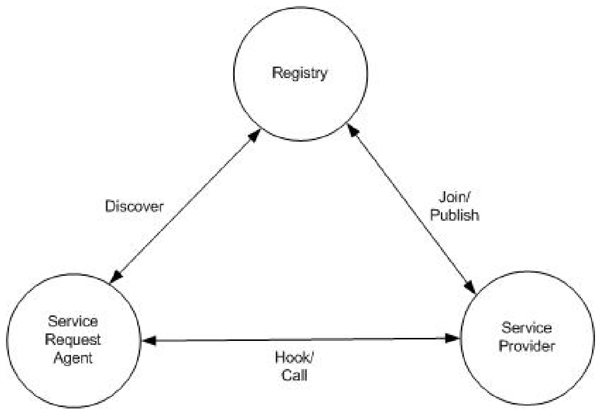


Figure 4-15: Serviced-oriented Architecture.

Any Service-oriented Architecture contains three components (or roles): a (service) registry, a service request agent, and a service provider.

The registry is a match-maker between service request agent and service provider because its address is known to all the service request agents and it contains information about all the service providers. Once the registry makes the match, it is no longer needed as the rest of the interaction is directly between the service request agent and the service provider.

The service request agent first discovers some service descriptions published to the registry. The act of discovery can be thought of as sending a query to a database. The service request agent states some query criteria, such as service types, quality requirements etc. The registry matches the query against its collection of published service descriptions. The result of the discover operation can be a list of service locations with optional descriptions (e.g. WSDL documents, see §4.7) that match the query criteria. The service request agent then uses the location information to hook to or call the service provider. This operation can be quite complex and highly dynamic, such as on-the-fly generation of a client-side proxy based on the service description used to invoke the service provider. Of course if the service description is standardized, as in the case of Optimization Services, the client-side proxy can be pre-built and the process becomes more efficient. Examples in the Optimization Services context are provided in Chapter 7.

The service provider joins the registry by publishing a service description to the registry. The software itself is not published. It then waits for service request agents to make invocations. The act of joining by publication can be thought of as advertising. There is usually some contract between the registry and the service provider. The actual details of the advertised information and the contract depend on how the service registry is implemented. If the service provider is well known, potentially many service request agents can directly invoke the service without first discovering it in the registry.

As briefly described in Chapter 2, Optimization Services also adopts the Service-oriented Architecture. In Figure 4-16 we redraw Figure 2-1: A typical optimization system and component interaction. We highlight the Service-oriented Architecture "triangle." Circle 4 is the service request agent and all the circles to its left can be thought of as the clients of the SOA-based distributed system. Circle 5 is an optimization service registry that keeps information of all the solvers (or analyzers). Circle 7 (or circle 6 or circle 8) is the solver (or analyzer or simulation engine) service provider. This is discussed in Chapter 5.

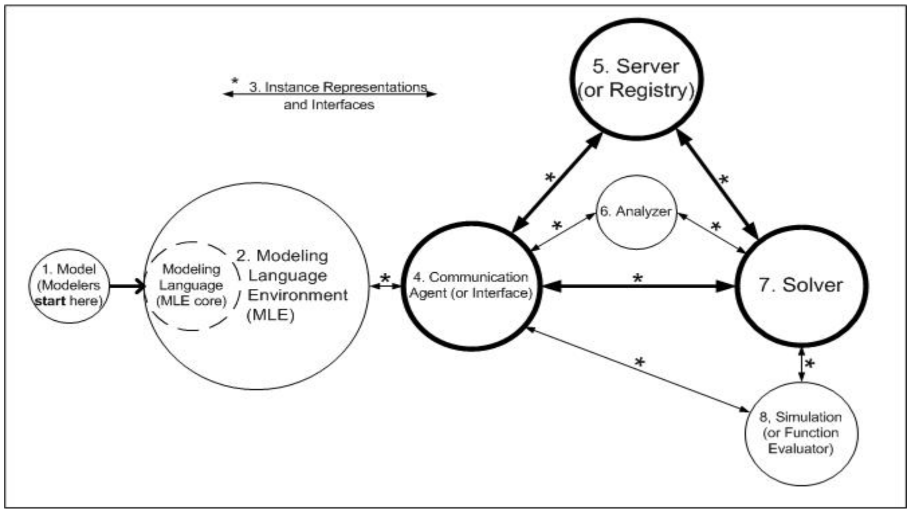

* 3. Instance Representations and Interfaces
5. Server (or Registry)
6. Analyzer
1. Model (Modelers start here)
Modeling Language (MLE core)
2. Modeling Language Environment (MLE)
4. Communication Agent (or Interface)
7. Solver
8, Simulation (or Function Evaluator)
*

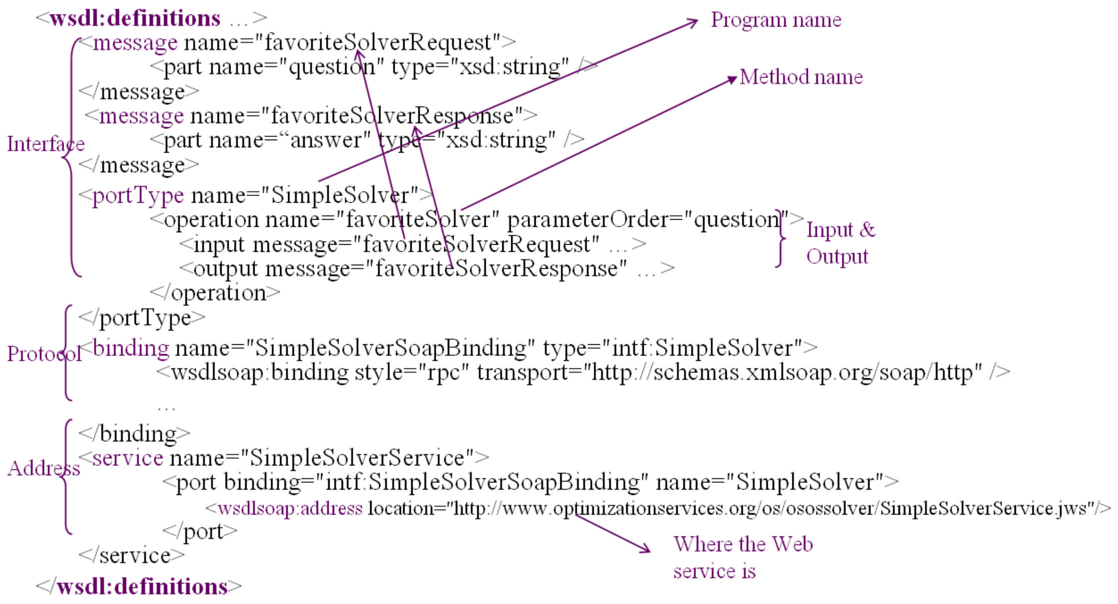
Figure 4-16: A typical optimization system and component interaction and the Service-oriented architecture view by Optimization Services.

4.7 Web Services Description (WSDL)

Web Services Description Language (WSDL) [116] is another XML document type that defines the XML tags used in accessing a Web service. WSDL is optional if a user knows exactly where an Optimization Service is and how the Optimization Service should be invoked. WSDL helps significantly in registering, discovering and automation of heterogeneous Web services. Links to WSDL descriptions can be given through UDDI listings (§4.8). In Optimization Services, we use WSDL mainly as a formal language to describe communication standards.

Two types of information in WSDL are specified. One is about interface semantics and the other is about administrative details of a call to a Web service. Interface semantics includes elements of `portType` (equivalent to a program interface), `operation` (equivalent to a method signature/prototype), `message` (equivalent to input and output) and `types` (equivalent to data types). Administrative details includes elements of `binding` (specifies transport and encoding protocols), `port` (specifies network addresses), `service` (specifies a collection of ports), and `definitions` (root element of WSDL that contains all the above elements). In our communication based Optimization Services Protocols (Chapter 7), we enforce standards on call interface and arguments, fix certain values by default and suggest recommendations that are most suitable for Optimization Services, thus simplifying the invocation processes.

Figure 4-17 shows an abbreviated WSDL definition. Each WSDL document has *definition* as its root element that is usually prefixed with the *wsdl* namespace abbreviation. Illustrated elements about interface, protocol and address are of the most relevance to Optimization Services. The entire program, called `SimpleSolver` in this example contains (in the *portType* element) only one *operation* (or function, method, procedure etc.): `favoriteSolver`, which takes a `favoriteSolverRequest` as an *input* and `favoriteSolverResponse` as an *output*. Both `favoriteSolverRequest` and `favoriteSolverResponse` are defined in their corresponding *message* element. For example `favoriteSolverRequest` has only one *part* (or argument) in it, which has a name `question` and is of type `string`. The protocol related *binding* element specifies that the SOAP call is to be an RPC (Remote Procedure Call) and is to be transported over HTTP. The address related *service* element specifies a location (in the *port* element) that tells where the actual Web service is.



```
<wsdl:definitions ...>
   <message name="favoriteSolverRequest">
         <part name="question" type="xsd:string" />
   </message>
   <message name="favoriteSolverResponse">
         <part name="answer" type="xsd:string" />
   </message>
   <portType name="SimpleSolver">
         <operation name="favoriteSolver" parameterOrder="question">
            <input message="favoriteSolverRequest" ...>
            <output message="favoriteSolverResponse" ...>
         </operation>
   </portType>
   <binding name="SimpleSolverSoapBinding" type="intf:SimpleSolver">
          <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
          ...
   </binding>
   <service name="SimpleSolverService">
          <port binding="intf:SimpleSolverSoapBinding" name="SimpleSolver">
               <wsdlsoap:address location="http://www.optimizationservices.org/os/ossolver/SimpleSolverService.jws"/>
          </port>
   </service>
</wsdl:definitions>
```

Figure 4-17: An abbreviated WSDL document of SimpleSolver, which specifies one operation: `favoriteSolver`.

4.8 Web Services Registration and Discovery (UDDI and WSIL)

After a Web service is deployed, potential users must have a way to discover and use that service. For Web pages/sites, search engines like Google and Yahoo assume this function, though search information is of non-standard form. Web services, unlike Web pages, are to be *invoked* by computers, rather than *viewed* by humans; thus Web services registration and searching require a more rigid set of rules. Universal Description, Discovery, and Inspection (UDDI, [102]) and Web Services Inspection Language (WSIL, [64]) handle the situations for general Web services through standardization. In Optimization Services, we use our own specialized Optimization Services Registry and corresponding OS protocols (Chapter 8) to register and discover Optimization Services, because more domain-specific information needs to be integrated into an Optimization Services registry. However both WSIL and UDDI provide the design inspiration for many of our registry related Optimization Services Protocols. We briefly give an overview of the two standards below.

UDDI is a specification for an online registry of Web services. Service providers can list their services in this registry, and users can seek out services by searching the registry in a standard way. UDDI is heavyweight and is intended to be maintained by centralized registries; it concerns itself not only with service data information (Figure 4-18) but also with service provider's vendor (or business entity, Figure 4-19) information. UDDI usually requires infrastructure to be deployed with substantial overhead and costs. Two main sets of standard functions (or APIs) are provided: vendors *register* services and data via SOAP and users *discover* the services via SOAP query requests (Table 4-1).

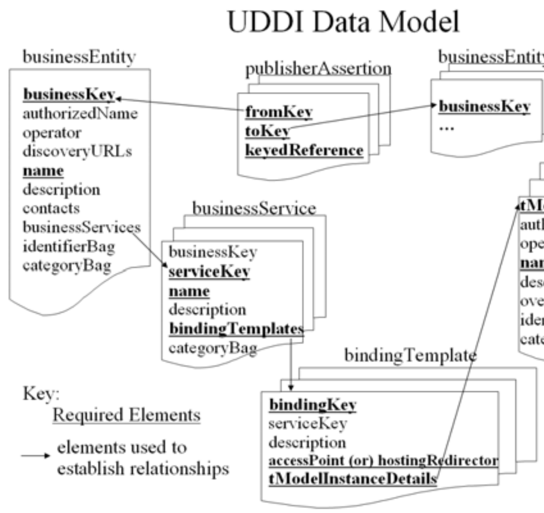


Figure 4-18: Service data information in a UDDI registry.

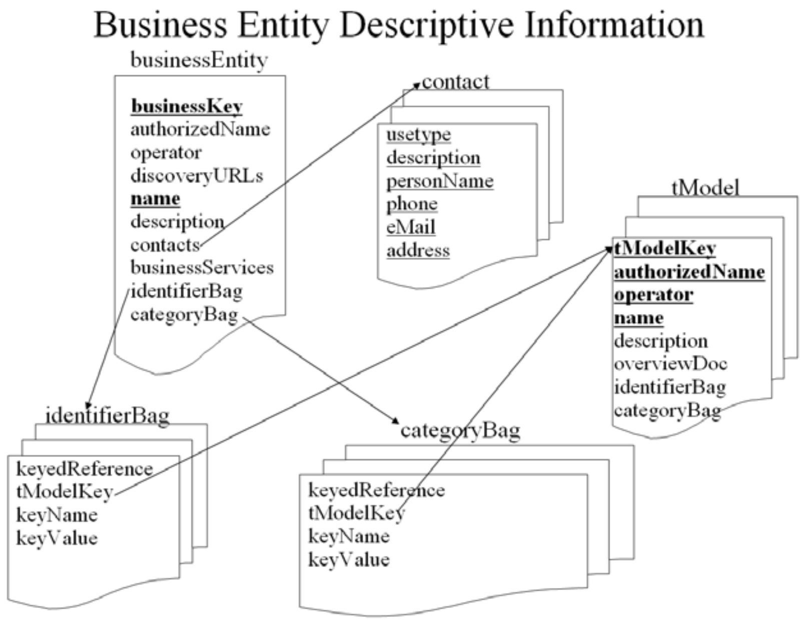


Figure 4-19: Business entity information in a UDDI registry.

SOAP Discover APIs	SOAP Register APIs
• find_binding • find_business • find_relatedBusinesses • find_service • find_tModel • get_bindingDetail • get_businessDetail • get_businessDetailExt • get_serviceDetail • get_tModelDetail	• add_publisherAssertions • delete_binding • delete_business • delete_publisherAssertions • delete_service • delete_tModel • discard_authToken • get_assertionsStatusReport • get_authToken • get_publisherAssertions • get_registeredInfo • save_binding • save_business • save_service • save_tModel • set_publisherAssertions

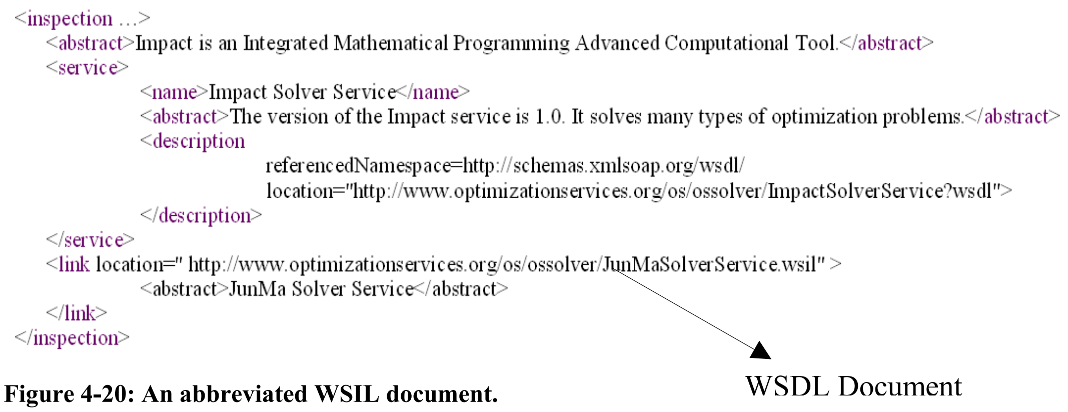
Table 4-1: Major SOAP discover and register operations provided by a UDDI registry.

WSIL is similar in scope to UDDI, but intended to be complementary rather than competitive. If WSIL is comparable to business cards, then UDDI is more like yellow pages, under which multiple "businesses" are grouped, listed along with goods or services offered and business contact information.

WSIL can be used to point to UDDI repositories. Service description information can be distributed to any location using a simple extensible XML document format. Compared with UDDI, WSIL is more decentralized, more lightweight and of lower functionality. WSIL works under the assumption that you are already familiar with the service provider. Both WSIL and UDDI rely on other service description mechanisms such as WSDL and they are located using existing Web infrastructure. On the other hand, in the Optimization Services situation, we no longer need WSDL information in the registry as all the Optimization Services invocations are standardized. Thus all services' WSDL documents will be the same except for location information which can be provided independent of WSDL.

WSIL avoids one of the current difficulties with UDDI: entries in UDDI registries are not moderated and a user can not be sure that a service actually belongs to the service provider who advertises it within the UDDI registry. So Quality of Service and information reliability can be issues with a general UDDI registry.

Figure 4-20 shows an abbreviated example of a WSIL document. Most information is self-explanatory in this example. Each WSIL document has *inspection* as its root element. It contains an *abstract* about the Web service, a *service* section detailing the description of the service, and a *link* to other related Web services. In the *service* section, the WSDL document location is provided in the *description* element.



```
<inspection ...>
    <abstract>Impact is an Integrated Mathematical Programming Advanced Computational Tool.</abstract>
    <service>
            <name>Impact Solver Service</name>
            <abstract>The version of the Impact service is 1.0. It solves many types of optimization problems.</abstract>
            <description
                        referencedNamespace=http://schemas.xmlsoap.org/wsdl/
                        location="http://www.optimizationservices.org/os/ossolver/ImpactSolverService?wsdl">
            </description>
    </service>
    <link location=" http://www.optimizationservices.org/os/ossolver/JunMaSolverService.wsil" >
            <abstract>JunMa Solver Service</abstract>
    </link>
</inspection>
```

Figure 4-20: An abbreviated WSIL document.

4.9 Open Grid Services Architecture (OGSA)

The Globus Alliance [42] is building fundamental grid computing technologies. By its definition, "grids are persistent environments that enable software applications to integrate instruments, displays, computational and information resources that are managed by diverse organizations in widespread locations." A major research effort of Globus Alliance is its Globus Project on developing the Globus Toolkit, which is an open source software toolkit to build grids. A growing number of projects and companies are using the Globus Toolkit which has become a *de facto* standard for major protocols and services, although at the present time its popularity is overshadowed by the recent success of Web services championed by major research institutes and companies.

Globus Alliance's Open Grid Services Architecture (OGSA) [43] represents an evolution towards a Grid system architecture based on Web services concepts, to take advantage of Web services' standard interface definition mechanisms, multiple protocol bindings, multiple implementations, local/remote transparency, etc. All services also have to adhere to specified Grid service interfaces and behaviors. At this point, OGSA is evolving quickly, currently at its fourth version, but far from complete or perfect.

Compared with Web services, OGSA is (potentially) strong in the following areas

- Authentication and authorization
- Global naming and references
- Lifetime management

- Resource registration and discovery
- Resource monitoring, upgradeability, concurrency, and manageability
- Reliable remote service invocation and notification
- High-performance remote data access

OGSA's major disadvantages lie in its protocol deficiencies; it has been implemented on a heterogeneous basis of HTTP, LDAP, FTP, etc. It also lacks (though actively intends to fix) *standard* means of invocation, notification, error propagation, authorization, termination and other functionalities. Little work has been done on total system properties including dependability, end-to-end Quality of Service, and reasoning about system properties.

One major difference between Web services and Grid services is that Web services addresses discovery and invocation of persistent services while Grid Services also supports transient service instances.

Web services combined with Grid is a good idea. It is becoming a topic in the major super computing conferences. It should not be a question of who wins. Both technologies will provide things that are valuable toward our development of Optimization Services. As a matter of fact, some of the design issues in our Optimization Services are based on the fact that components from both technologies can be leveraged upon their maturities. We hope that the two technologies will eventually converge with no distinction.

CHAPTER 5 OPTIMIZATION SERVICES (OS)

Optimization Services is a unified *framework* for the next generation distributed optimization systems, mainly optimization over the Internet. The corresponding Optimization Services Protocol is intended to be a set of industrial *standards*. The phrase "next generation" emphasizes the fact that Optimization Services is a state-of-the-art design and is *not* adapted from any existing system.

In Chapter 4, we provided the necessary background on modern computing and distributed technologies in order to read from this chapter on. In Chapter 1, we gave a general non-technical description of Optimization Services (OS) and the corresponding Optimization Services Protocols (OSP). We describe Optimization Services in a more technical detail here.

From the system design view, Optimization Services is a SOAP protocol based and service-oriented architecture centered framework for optimization over distributed and decentralized systems. Through the corresponding Optimization Services Protocol, Optimization Services specifies behaviors of its standard components on a distributed system. We described in Chapter 2 all the system components that are targeted in the OS framework's standardization process.

5.1 Standardization, OSP and OSxL

The Optimization Services framework is mainly concerned with standardization in three areas:

1 Optimization (instance) representation (Chapter 6);

2 Optimization communication that includes accessing, interfacing and component orchestration (Chapter 7);

3 Optimization service registration, publication, discovery and quality control (Chapter 8).

For the sake of uniformity, we specify Optimization Services Protocols in all these three areas by standard 4-letter acronyms of the form OSxL, standing for Optimization Services x Language, where "x" is aother defined letter. Figure 5-1 shows a tree view of all the current Optimization Services x Languages.

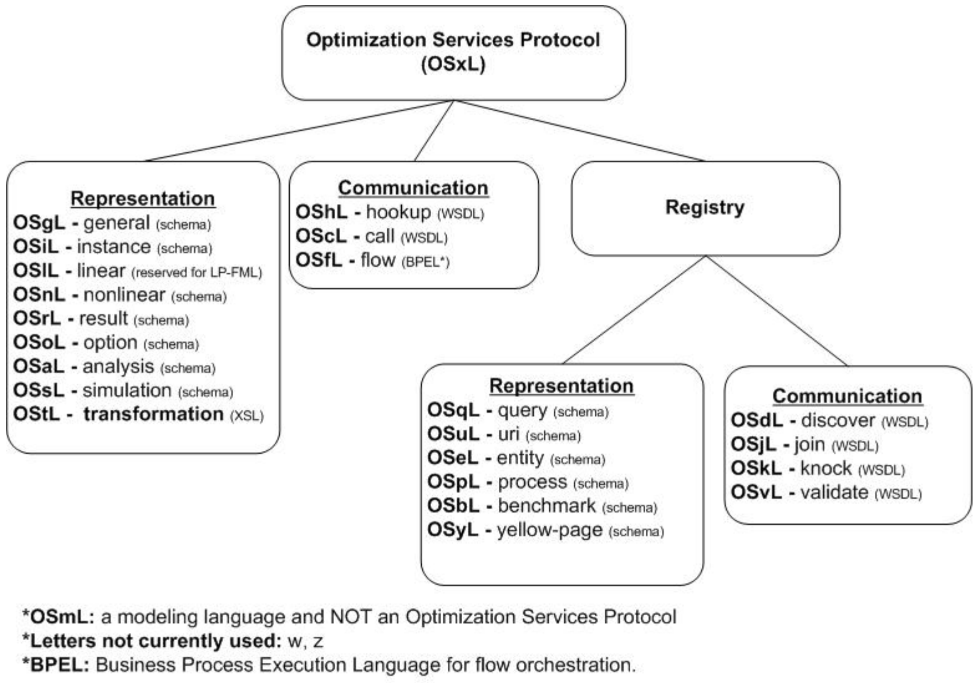


Figure 5-1: A tree view of Optimization Services x Languages (OSxL).

We explain the OSxL languages in each of the three areas below.

1). In §2.1 and §2.3, we discussed the differences between a model and an instance. The Optimization Services framework is *not* intended to standardize high level models. The framework only concerns itself with the low level communication between machine and software components.

All the instance representations are specified in the XML Schema language (§4.3). The most important instance is the representation of an optimization problem. The format of this instance is specified by the Optimization Services instance Language (OSiL). An OSiL instance is usually transmitted from a modeling language environment (MLE) to a solver.

There are other kinds of instances. The Optimization Services result Language (OSrL) specifies the result format of the solver output. It is usually transmitted back from a solver to an MLE. Optimization Services analysis Language (OSaL) specifies the analysis format of the analyzer output. It is usually transmitted from an analyzer to an MLE and helps in discovering solvers in an Optimization Services registry. Optimization Services option Language (OSoL) specifies the option format of solver (or analyzer) algorithm directives. It is usually transmitted along with an OSiL instance. Optimization Services simulation Language (OSsL) specifies the input and output format of a simulation service. It is usually transmitted between a solver and a simulation engine. It facilitates optimization over simulation where simulations are located in places other than the solver.

Many of the generic and common data structures are specified in the Optimization Services general Language (OSgL) and imported by other representation schemas. All the nonlinear functions, operators and operands are specified in the Optimization Services nonlinear Language (OSnL). OSnL is used by the OSiL schema for nonlinear optimization extension.

2). In §2.4, we listed the interface and communication agent as a distinct component in an optimization system. The Optimization Services framework standardizes all the communications between *any* two Optimization Services components on an OS *distributed* system. The framework does *not* standardize *local* interfacing. Related projects such as COIN OSI [23] discussed in §3.1.4 and derived research from Optimization Services (briefly mentioned in the following chapters) such as the Optimization Services instance Interface (OSiI), Optimization Services option Interface (OSoI) and Optimization Services result Interface (OSrI) are intended to do this job.

Invocations of all Optimization Services are specified by WSDL (§4.7) and all the interfaces and transport parts (i.e. except for the location information) in the WSDL documents are standardized. So WSDL documents are not necessarily needed to *dynamically* generate the communication APIs (stubs and skeletons) as we know them ahead of time already, although they can be used for illustrations or as references to construct Optimization Services *beforehand*.

The most common communication is the invocation of solvers. This is specified by the Optimization Services hookup Language (OShL). OShL also applies to hooking up to analyzers, as solvers often analyze an optimization problem and analyzers may potentially solve the problem. The invocation of simulation services is essentially calling a function (§2.8) and it is specified by the Optimization Services call Language (OScL).

Communication is not just about invocations. As we build all the Optimization Services components into a distributed system, the *sequence* of invocations is an issue. For example, if a solver service is known to a client, the client can directly contact the service. Of course the client can still contact a registry and get the location information and then call the solver. But if the client does not know the type of the optimization problem, he may first invoke an analyzer service, and then use the analysis result to query the right solver from the registry. Even more complex, before invoking the analyzer service, the client may need to find the analyzer's location in the registry. There can be many combinations of sequences. Optimization Services flow Language (OSfL), an XML document in BPEL (Business Process Execution Language [91]), predefines certain standard flows.

3). Representations and communications related with the Optimization Services registry are separately grouped in the area of service registration, publication, discovery and quality control. Differences between an optimization registry and an optimization sever are detailed in §2.5.

At the core of our Optimization Services registry implementation is a database and we chose to use a more expressive XML-based native database as versus a relational database. The logic is explained in Chapter 8. The organization of the native XML database is according to the Optimization Services yellow-page Language (OSyL) which is a schema on the syntax of the stored data. To query the database, clients use the Optimization Services query Language (OSqL) which is a schema of the query language format. In the OS registry implementation, an OSqL query is then converted to an XQuery (§4.4) that is executed against the XML database in the registry. The communication of sending the OSqL query to the OS registry is specified in the Optimization Services discover Language (OSdL), a WSDL document. In turn the clients get the location information from the registry that is listed as a sequence of URIs (or URLs). The syntax is specified in the Optimization Services uri Language (OSuL).

On the other side of the *discover* process is the *register* process. The database in the Optimization Services registry is essentially a list of *static* entity information (e.g. solver types, owner information, service location). The entity information is specified by the Optimization Services entity Language (OSeL) items, an XML schema. Optimization Services yellow-page Language (OSyL) can be roughly viewed as a sequence of Optimization Services entity Language (OSeL) items, so we can think of OSyL as a table and OSeL as a row in the table. Besides static entity information, the Optimization Services registry also keeps dynamic process information (e.g. whether the service is running, and number of jobs being solved) using Optimization Services process Language (OSpL). Independent benchmarks are carried out on registered solvers and the benchmark information is kept in Optimization Services benchmark Language (OSbL). OSeL, OSpL, and OSbL are all the information the registry knows about all the registered services.

Service providers join the registry with OSeL information. The WSDL document Optimization Services join Language (OSjL) specifies how this is done. During runtime, the Optimization Services registry periodically “knocks” on the registered services to make sure they are live and running and to get the OSpL information. The WSDL document Optimization Services knock Language (OSkL) specifies how this is done. Service providers can also publish the OSeL, OSpL and OSbL information on their own Web site. To facilitate standardization, the standard XSL transformation style sheet (§4.4) OStL (Optimization Services transformation Language) is provided so individual Web publications have the same look-and-feel. The OStL style sheet, as a matter of fact, can be used with any Optimization Services XML representations for publication and presentation.

The decentralized Optimization Services system leaves open the question of how optimization "jobs" will be scheduled to run on available solver services. Centralized schemes, such as that used by the NEOS server, usually maintain one queue for each solver/format combination, along with a list of the workstations on which each solver can run.

In Optimization Services, we want to maintain this scheduling control, while at the same time making the scheduling decisions more distributed. Optimization Services process Language can play an important role in dynamic optimization scheduling in a decentralized environment.

5.2 Architecture Design

In Chapter 2, we showed a general architecture of optimization systems and discussed the major system components of an optimization system. Most of the current centralized optimization systems, such as the two examples illustrated in Chapter 3, serve as the initial motivation for Optimization Services. The Optimization Services simplified view of any centralized optimization system is shown in Figure 5-2.

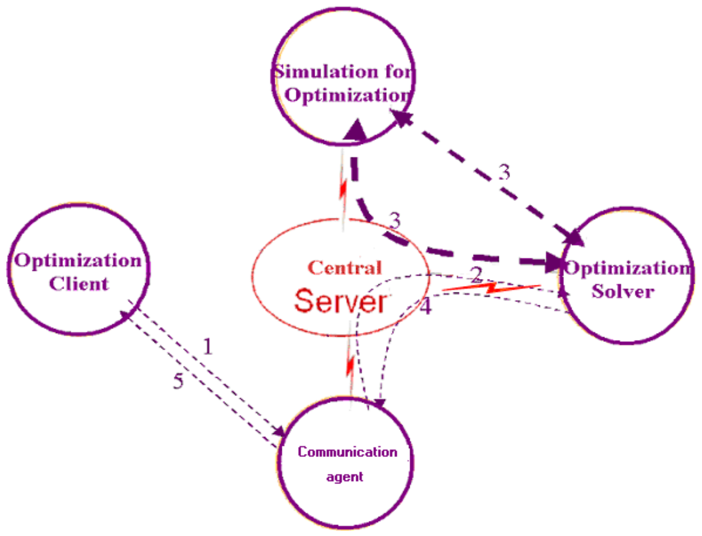


Figure 5-2: Optimization Services' simplified architecture view of a centralized optimization system.

All the components shown in Figure 5-2 were discussed in detail in Chapter 2. The *optimization client* is often a modeling language environment (MLE) or some customized graphical user interface (GUI) with prewritten optimization models behind it. Dotted arrows indicate data flow and corresponding numbers show a typical flow sequence. The data are usually some *instance representations*. Arrows that do not go through the *central server* mean a direct local invocation, so a *communication agent* is usually bundled together with the optimization client. The communication agent can actually be bundled with any component that needs to make a remote connection. The *simulation* can be called by the *optimization solver* either remotely or locally. If locally, the simulation is usually a simple function or expression tree(§2.8). The arrows (3) between solver and simulation are in bold because the data flow between the two can be highly iterative. The *model* component mentioned in Chapter 2 is not part of the Optimization Services framework. It belongs to the user end and is isolated from the software system by the optimization client. The *analyzer* component is usually not separated out in a centralized optimization system.

Figure 5-3 matches the system components in Figure 3-9 with those in Figure 5-2 and shows how the Motorola Labs Intelligent Optimization System discussed in Chapter 3 fits in the Optimization Services view of a centralized optimization architecture. A similar analysis can be done on the AMPL-NEOS system also discussed in Chapter 3.

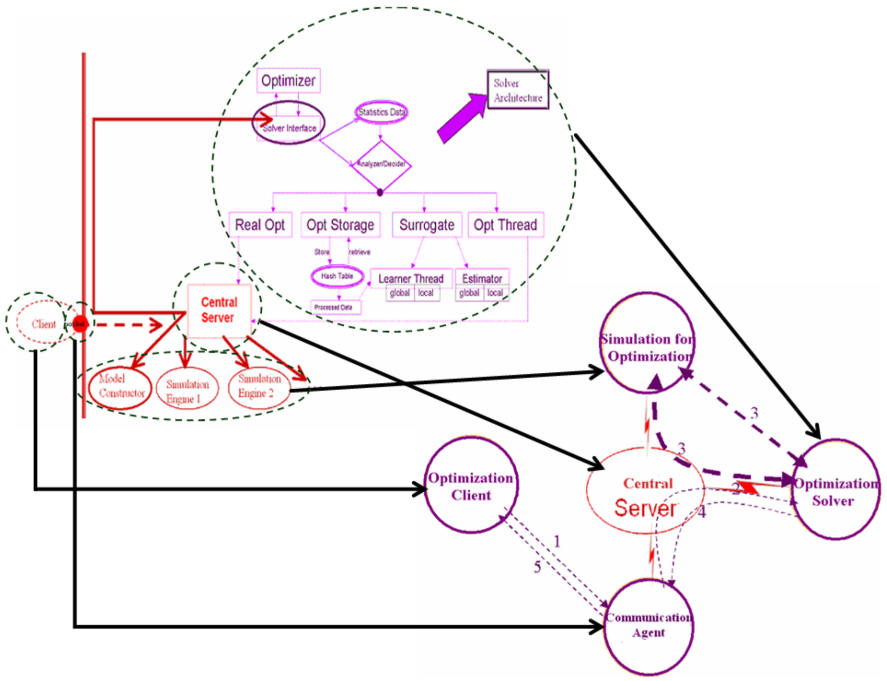


Figure 5-3: Optimization Services' simplified architecture view of Motorola Lab's Optimization System (Chapter 3).

Optimization Services' own approach to the next generation architecture design is an approach of decentralization shown in Figure 5-4. The advantage of the decentralized scheme over the centralized scheme was mentioned in Chapter 1 and described in detail in §2.5.

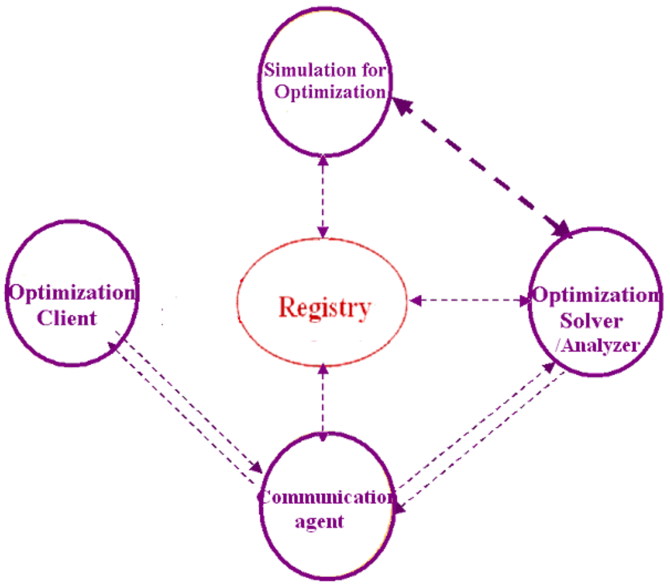


Figure 5-4: Optimization Services' simplified architecture approach of a decentralized optimization system; compare with Figure 5-3.

The *optimization client* in Figure 5-4 still invokes the *communication agent*, but the agent no longer connects to the *optimization solver* through a *server*. The *registry* replaces the server in a centralized scheme. All the components in the distributed system talk in a peer to peer mode. After the communication agent discovers a solver from the registry, it contacts the solver directly. In a decentralized system, the *analyzer* plays an important role as argued in §2.6. But from the architecture view, the optimization solver and analyzer are of no difference as they are both services provided over the distributed system and both can be discovered in the registry. As in the centralized scheme, the *simulation service* is usually iteratively invoked by a solver either locally or remotely, except that the invocation is no longer routed through the server. Also notice that there is a link between the registry and all the services as the registry can periodically check these services to get their latest process information. Dotted arrows that indicate *data flow* no longer have corresponding numbers showing a typical flow sequence. There can be many process flows as explained in the next section. From the Optimization

Services standardization perspective, the most important parts of the system components are *instances* (data flow on the dashed arrows) and communication agents.

Figure 5-3 matches the system components in Figure 3-2 with those in Figure 5-4 and shows how the AMPL-NEOS system discussed in Chapter 3 can be adapted to the "next-generation NEOS" that is built on the decentralized Optimization Services architecture. The exact effects of Optimization Services on NEOS can be multifaceted and are discussed in §3.1.4.

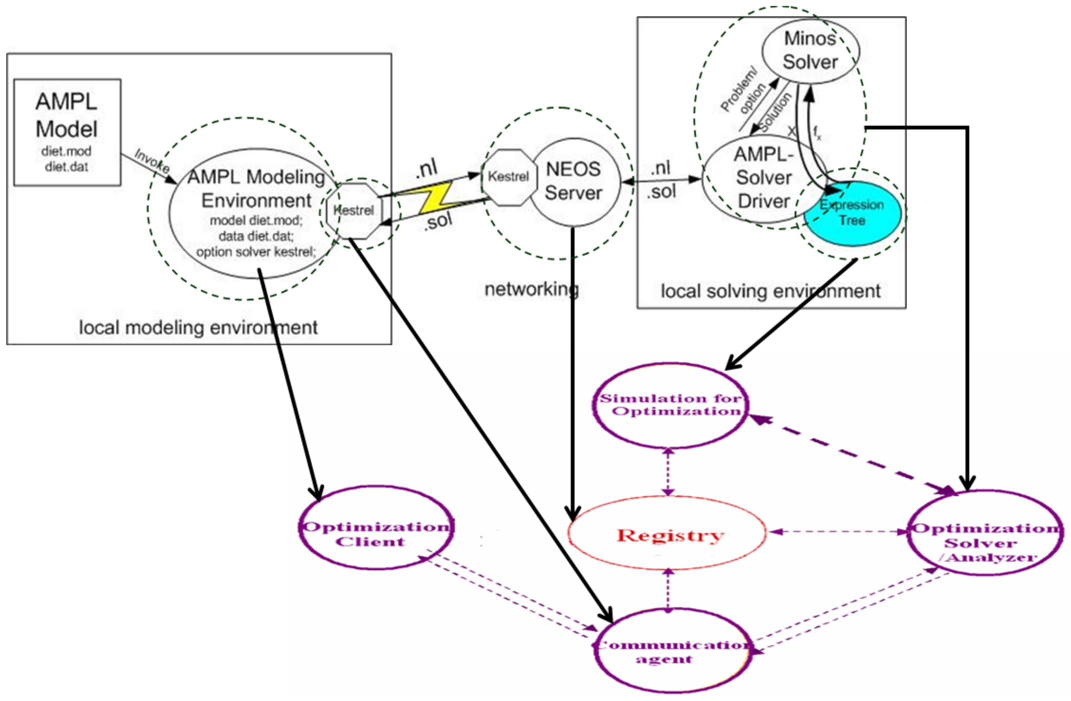


Figure 5-5: Optimization Services' simplified next-generation architecture approach of AMPL-NEOS system (Chapter 3).

5.3 Optimization Services Process

Optimization Services can have various process starting points. For better illustration from a user perspective, we start the process with a modeler. Suppose the modeler has an optimization model shown in (5-1).

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & 100(x_2 - x_1^{\,2})^2 + (1 - x_1)^2 \\ \text{subject to} \quad & x_1 + x_2 \le 100 \end{aligned} \tag{5-1}$$

All that the modeler cares is to have the model solved by an appropriate solver and get the optimized result (Figure 5-6).

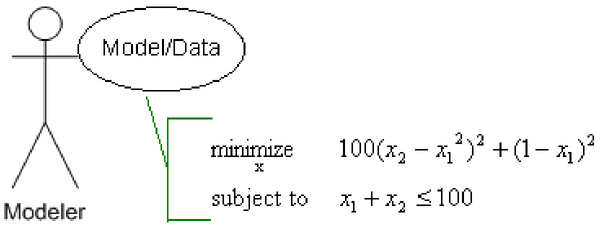

Model/Data
Modeler
minimize x $100(x_2 - x_1^2)^2 + (1 - x_1)^2$
subject to $x_1 + x_2 \leq 100$

Figure 5-6: A modeler starts with a model and some data and wants the model solved.

But in practice there is no direct connection between the model/data and the solver (Figure 5-7). No solver understands the optimization model (5-1).

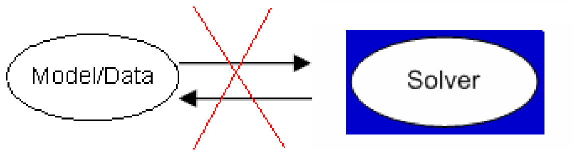


Figure 5-7: There is no direct connection between the model and the solver.

The user has to formulate his model in a formal modeling language such as AMPL (Chapter 3) or OSmL (Chapter 9). Alternatively the user can construct an application as a graphical user interface with prewritten optimization models underneath, or in a spreadsheet (Figure 5-8). Optimization Services is *not* intended to standardize these user environments. What is natural for one modeler may not be for another. But Optimization Services does require all these user environments translate the user's model into the standard Optimization Services instance Language (OSiL, Chapter 6). From this point on the modeler is "isolated" from the computing system world and no longer knows what is happening inside. In fact, by standardizing the underlying system communications, the Optimization Services framework promotes the flexibility for users to express models differently with various modeling languages and tools, as they will no longer be limited by the choices of software due to interface compatibility issues.

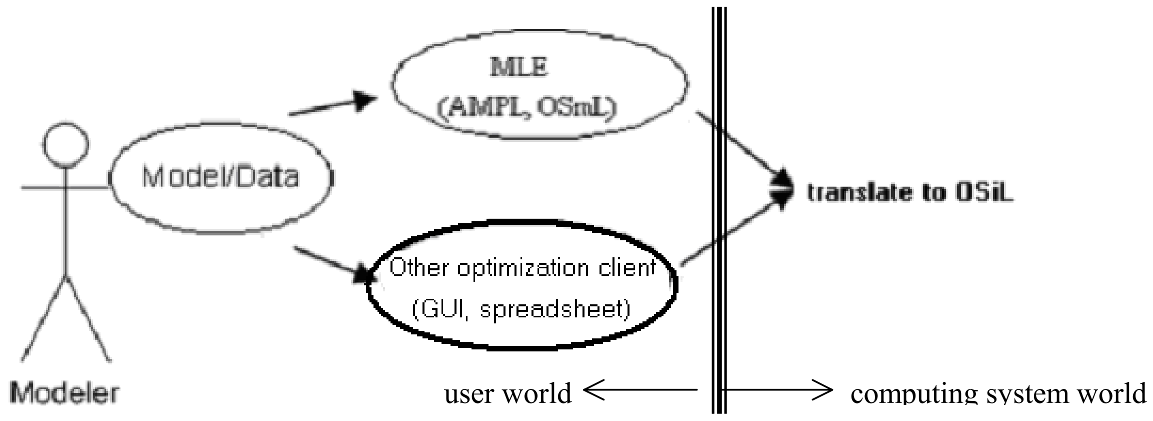


Figure 5-8: The modeler has to formulate his model in an MLE (or GUI, spreadsheet etc.) and the model gets translated into an OSiL instance.

Suppose the modeler chooses to formulate the optimization model (5-1) in the OSmL modeling language (Figure 5-9). The OSmL engine will compile the model into an OSiL instance and delegate a communication agent to send the OSiL instance to the appropriate solver on the OS network (Figure 5-10).

```
return
<mathProgram>
<obj maxOrMin="min" name="Rosenbrock">
100*(x2 - x1^2)^2 + (1 - x1)^2
</obj>
<constraints>
<con>
x1 + x2 <= 100
</con>
</constraints>
</mathProgram>
```

Figure 5-9: The model can be formulated in the OSmL modeling language.

The communication agent (in this case a solver agent) knows everything about hooking up with any OS solver. It takes OSiL as an input and contacts an OS solver using the OShL communication protocol in OSP (Chapter 7). OShL is explained in Chapter 7. It roughly corresponds to invoking an operation (or method) in a local environment, so OSiL can be thought of as an input argument of this operation; this corresponds to the part "`OShL`

`(OSiL)`” in Figure 5-10. In all the following figures we use this convention by putting input and output instances in parentheses.

A solver can be developed independently from Optimization Services and its API (Application Program Interface) may be different from that specified by OShL. For example, OShL specified an operation name called “`solve(String osil)`” but the solver may be using the name “`optimize`” and its own data representation So in order to be OS-compatible, the solver has to expose a standard OS API. The solver can do that by hiding the original solver in a wrapper class and make the wrapper class implement the standard interface with all the methods specified in the OShL WSDL document. For example the solver can implement the following wrapper class:

```
String solve(String osil){
    convert osil into solver’s own data representation;
    solver_own_result = optimize(solver_own_representation);
    convert solver’s own result to the standard result and return;
}
```

All the OS solvers are hosted in an OS server just as all the web pages are hosted in a Web server. We provide the OS Server *software* to host the OS solver. The solver developer can, however, implement their own Optimization Services solver server, as long as the exposed service API follows the OSP protocols. After the solver solves the problem represented by OSiL, it returns the result in Optimization Services result Language (OSrL, Chapter 6).

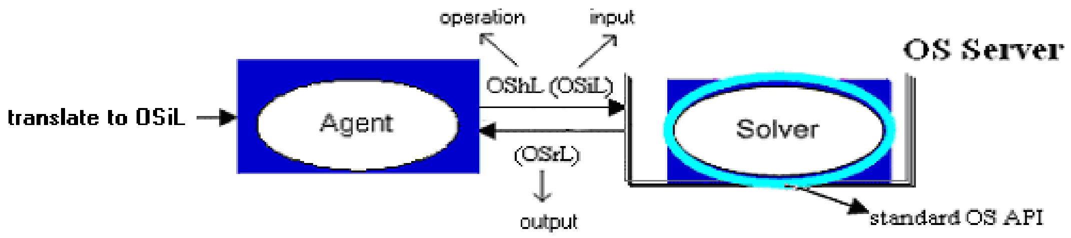


Figure 5-10: After the model is translated into the OSiL instance, an agent is delegated to send the instance to a solver. The agent hooks up the solver using the OShL communication protocol. All OS solvers expose themselves with a standard OS API and return the output in OSrL. An OS server is needed to host the solver and all other Optimization Services. We provide the OS Server software.

After the agent gets the OSrL result back, it returns the result to the calling environment. None of the OSxL instances are meant for humans to read. A standard OStL (Optimization Services transformation Language, Chapter 6) style sheet is provided to present the OSxL instances. OSrL is among the instances whose contents need to be understood by humans most frequently.

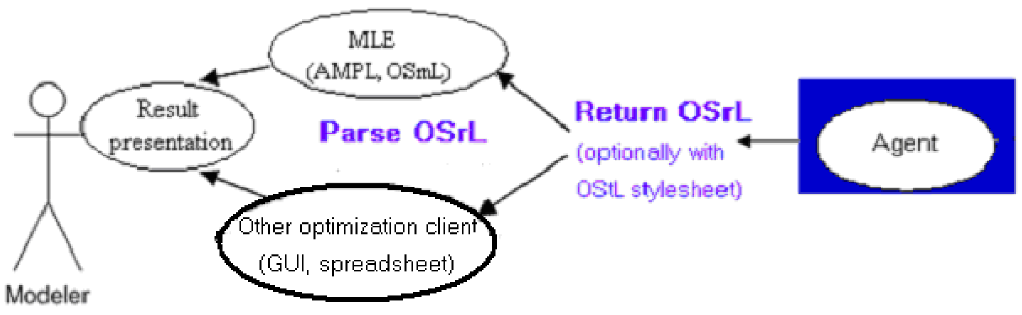
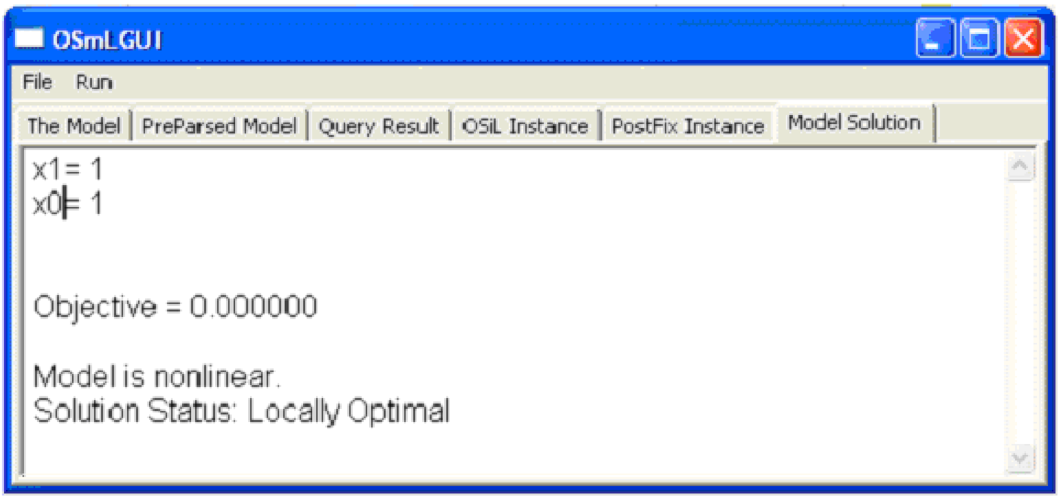


Figure 5-11: The agent returns the OSrL and possibly with the standard OStL style sheet to the MLE (or GUI, spreadsheet, etc.) and the result gets nicely presented to the modeler.

Since different users have different tastes, modeling environments can choose not to use the provided OStL style sheet (Chapter 6), and instead present the OSrL in a different way. In situations where post-processing of OSrL is necessary, names and indexes of the original model may be different from those in the instance, so OStL may not be appropriate to use. For example the OSmL modeling environment presents the optimized result without using the OStL as shown in Figure 5-12.



```
x1= 1
x0= 1


Objective = 0.000000

Model is nonlinear.
Solution Status: Locally Optimal
```

Figure 5-12: The OSmL modeling environment presents the result (without the OStL style sheet).

The agent can talk to any solver service on the Optimization Services network (Figure 5-13). This is possible because all the solver services expose the same standard OS API. All the solvers can be invoked with an operation specified by OShL, they all take OSiL as an input, and they all return OSrL as an output. But all these are based on the assumption that the agent knows where the solvers are. Each solver is at a unique address (URI) but either it may not be known to the agent or it may change frequently.

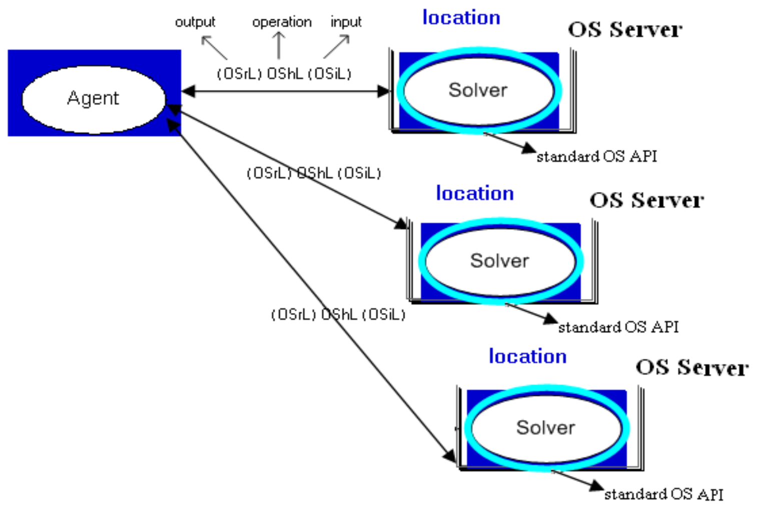


Figure 5-13: The agent can talk to any solver on the Optimization Services network. This is possible because all the solver services are standardized; they can be invoked with an operation specified by OShL, they all take OSiL as input, and they all return OSrL as output.

So the agent needs to first contact the OS registry (Figure 5-14). The location of the OS registry is well known or is easily found. For example our test OS registry is currently at the address http://gsbkip.chicagogsb.edu/os/osregistry/NEOSRegistryService.jws.

The agent discovers the right solvers in the OS registry with an OSdL (Optimization Services discover Language, Chapter 8) operation which passes the OSqL (Optimization Services query Language, Chapter 8) query as an input. The OS registry returns the locations of the solvers that match the OSqL query in OSuL (Optimization Services uri Language, Chapter 8).

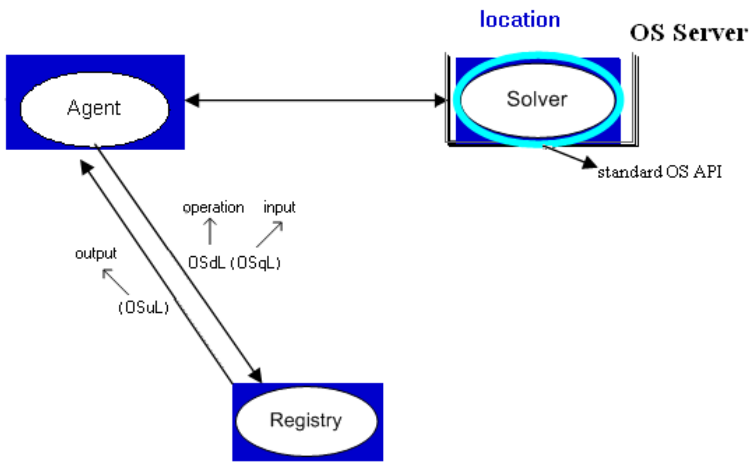


Figure 5-14: The agent knows how to hook up any solver, but first it needs to know where the solvers are. So the agent discovers the solver in the OS registry with an OSdL operation, which passes OSqL as an input. The OS registry returns the matched locations in OSuL.

The OS registry has all the solver information because all the OS solvers have to join the registry by publishing their OSeL (Optimization Services entity Language, Chapter 8) information to the registry with an OSjL (Optimization Services join Language, Chapter 8) operation. OSeL describes the static information about all the solvers. Usually the information can be filled in on a Web form and when the service provider submits the Web form, an OSjL operation is used to register the solver service. The OS registry in return sends back the OStL style sheet with which the solver providers publish their solver information (in OSeL) on their individual Web sites. Besides the OSeL information, the registry also separately benchmarks all the registered solvers and holds the benchmark information in OSbL (Optimization Services bench Language, Chapter 8). The OS registry's own Web site also publishes the OSeL and OSbL information. To facilitate a uniform look-and-feel of publications over the OS network, all the service providers have to use the provided OStL style sheet. The "triangle" between the agent, the solver and the registry is called a Service-oriented Architecture (SOA, §4.6).

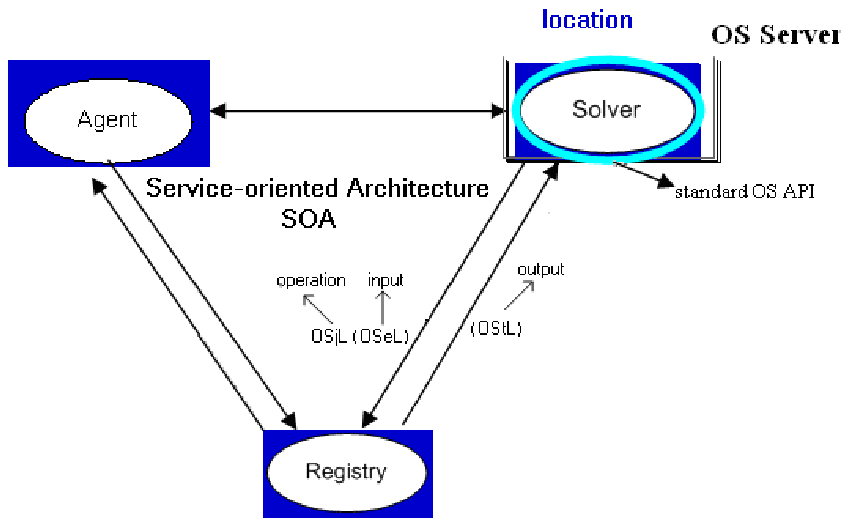


Figure 5-15: The OS registry has all the solver information because all the OS solvers have to join the registry by publishing their OSeL information with an OSjL operation beforehand. The OS registry in return sends back the OStL style sheet with which the solver providers publish their OSeL information on their own Web site. The “triangle” between the agent, the solver and the registry is called a Service-oriented Architecture (SOA).

In reality, the Optimization Services process can be more complex. Before sending a query to the OS Registry, the solver agent may not know what query to send, as it can be hard to determine the optimization type from an optimization instance. This was discussed in §2.6. So the solver agent may first send the OSiL instance to an analyzer for analysis (Figure 5-16). All the OS analyzers on an OS network are invoked in the same way as OS solvers, i.e. using OShL. The OS analyzer takes OSiL as an input, but unlike the OS solvers, sends back OSaL (Optimization Services analysis Language, Chapter 6) as an output. Of course if the agent does not know the location of the OS analyzer, it again needs to first discover the analyzer in the OS registry, just like it discovers OS solvers.

Figure 5-16 also shows some other process complications. For example, the solver may need to call a remote simulation service to get function values. The solver calls using an OScL (Optimization Services call Language, Chapter 7) operation. Both input and output of calling the simulation are specified in OSsL (Optimization Services simulation Language, Chapter 6) as their formats are simple and similar.

Many of these processes are so common that they are predefined in OSfL (Optimization Services flow Language, Chapter 7). OSfL, unlike most other communication related OSxL’s, is an XML document in BPEL (Business Process Execution Language, [91]) that descriptively lists all the flows.

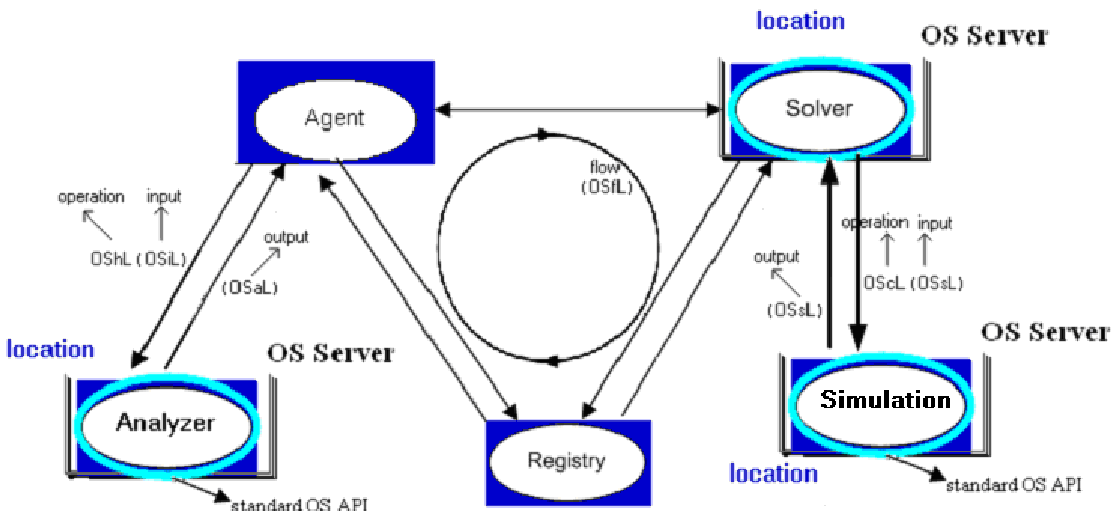


Figure 5-16: Before sending a query to the OS Registry, the agent may first send the OSiL problem instance to an analyzer using OShL. The analyzer sends back OSaL as an output. On the other hand, the solver may need to call a simulation service to get function values. The solver calls using an OScL operation. Both the input and output of calling the simulation are specified in OSsL. Some of the standard process flows are predefined in OSfL.

In all the figures above, the OS registry is not drawn inside an OS server. But in fact, the OS registry is itself also an Optimization Service hosted in our own OS server and has a standard OS API exposed (Figure 5-17). Besides the discovery and registration services that the OS registry provides, the OS registry also provides a validation service. For example any component on the OS network can send an OSxL instance representation to the registry for validation using the OSvL (Optimization Services validate Language, Chapter 8) operation. The OS registry will return an error message if there is any warning or error in the OSxL instance submitted. Otherwise it returns a null or empty string.

On the other hand, the OS registry, as a client, can "knock" on all the services with an OSkL (Optimization Services knock Language, Chapter 8) operation and all the services are required to send the current process information in OSpL (Optimization Services process Language, Chapter 8). This is possible because all the services are required to implement the standard interface with all the methods specified in the OSkL WSDL document. So a solver now has to implement operations specified both in OShL and OSkL.

OSeL (entity), OSbL (benchmark) and OSpL (process) information is all that the OS registry knows about any registered service. All the three types of information play important roles for the registry to find the most appropriate service against a submitted query.

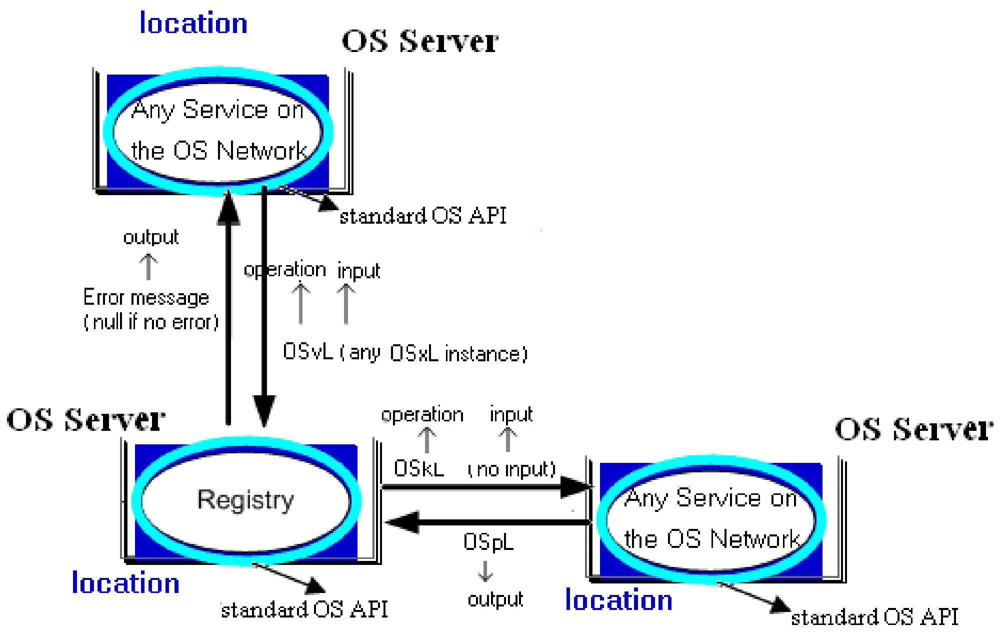


Figure 5-17: The OS registry is in fact also an Optimization Service hosted in an OS server and has a standard OS API exposed. For example any service on the OS network can send an OSxL instance representation to the registry for validation (OSvL) and the OS registry will return an error message if there is any. Otherwise it returns a null or empty string. On the other hand, the OS registry can "knock" on all the services with an OSkL operation and all the services are required to send the current process information in OSpL.

As described in Chapter 1, Optimization Services and the Internet are closely related because of the decentralized architecture. In Figure 5-18, we show that most of the components in the Optimization Services system have a corresponding similar part in the Internet architecture. The similarity is not the initial intention of the OS project; rather it is the result that both are good designs based on a decentralized architecture.

Writing the *model/data* in a *modeling language environment (MLE)* more or less corresponds with an Internet user filling in a *Web form* in a *browser*, only that the model/data construction can be more complex. The MLE converts the model/data into an *OSiL* instance and delegates the *communication agent* to send the instance, whereas the browser converts the Web form into an *html* file and delegates a *socket* to send the html. The agent uses the *OSP* communication protocol (in this case *OShL*) to contact the remote service (in this case a *solver*), whereas in the Internet architecture, the socket uses the *HTTP* protocol to contact the remote *Web page*. The solver is hosted in an *OS server*, whereas the Web page is hosted in a *Web server*. As a matter of fact our OS server implementation is based on an existing Web server and adds in extra plug-ins for all the Optimization Services operations. The *location* of an OS service is in URI format, whereas a *Web address* is in plain URL format , a subset of URI but practically the same. A Web page is usually a static html page whereas a solver is more dynamic and mostly about computation. But many Web pages are also generated on the run. If the codes behind a Web page dynamically compute the Web contents, it may use *CGI* (Common Gateway Interface) or other dynamic Web technologies to wrap the codes behind the static html page. This more or less corresponds the *standard OS API* that wraps the solver codes. Like a dynamic Web page, which can get extra data from a remote *database*, the solver can get function values from a *simulation.* The OS *registry* naturally plays the role of any Internet search engine, only that the OS registry is intended for machine or software and is highly standardized and automated while the search engine is more for humans. Services like the *analyzer* are more or less like many of the HTML checker Web sites (e.g. W3C) that look into an instance and report an analysis result.

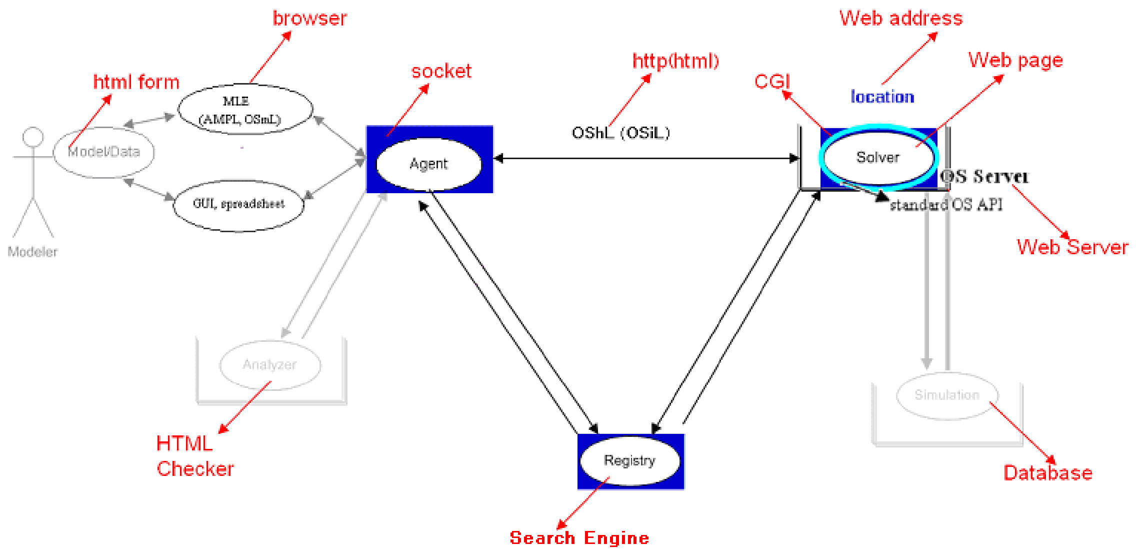


Figure 5-18: A close analogy between Optimization Services and Internet.

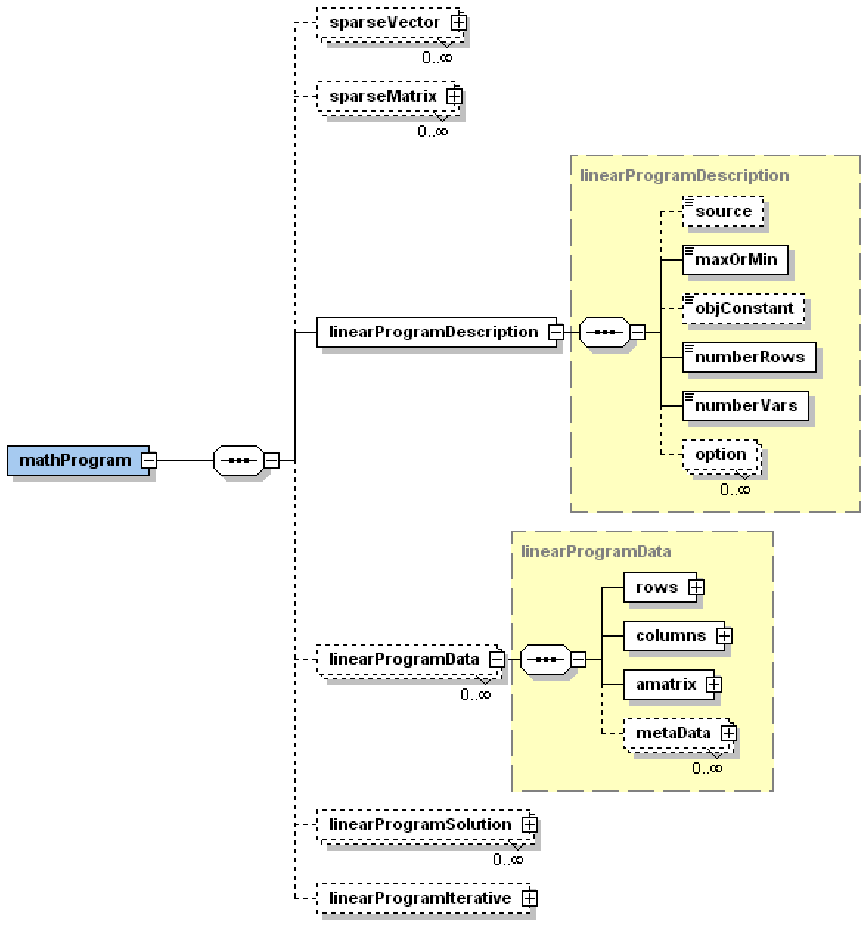
CHAPTER 6 OPTIMIZATION SERVICES REPRESENTATION

In this chapter, we present the instance representation part of Optimization Services Protocol (OSP). The instance representations are a set of *low-level* formats for data communication between different Optimization Services components. The difference between low-level instances and high level models is explained in Chapter 2. All the registry related OSxL representations are covered in Chapter 8. We provide open-source libraries (Appendix B) for reading and writing all the instances to facilitate parsing and simplify exchange of information. All the representation schemas and libraries are available at www.optimizationservices.org [92] and www.optimizationservices.net [93].

Standards for instance representation are not new. In Chapter 2, we list all the major instance formats. But they are all limited to optimization problem *input* and highly fragmented in representing different input types. The scope of Optimization Services representations is much more general and comprehensive. Currently all the major optimization problem types are supported.

We are also not the first to incorporate XML into optimization representations. Fourer, Lopes, and Martin proposed the LPFML Schema [53] for representing instances of mixed integer linear programs. Chang [19] and Kristjánsson [69] also proposed XML representations for linear-program instances. Bradley [15] proposed an XML markup grammar for networks and graphs. But all these XML representations deal mainly with one or two optimization types and none support the general nonlinear optimization problems.

The Optimization Services representation project started with the Optimization Services instance Language (OSiL, §6.2) for representing general optimization input instances. OSiL has its roots in LPFML for representing linear program instances (Figure 6-1). For linear programming, an instance can be represented as a list of nonzero coefficients of variables in the objective and constraint functions, along with bounds on the variables and constraint functions. LPFML also has slight support for solver options and optimization outputs. OSiL extends and improves LPFML's idea for linear program design and adds other optimization types. There is no separate *linear* instance representation in Optimization Services. We reserve the acronym OS*l*L (Optimization Services *linear* Language) in honor of LPFML for providing us the base and insight in linear program representations and for its early adoption of XML technologies in optimization. Optimization Services, however, has its own separate supports for optimization options and results through OSoL (Optimization Services option Language, §6.5) and OSrL (Optimization Services result Language, §6.4).



```
<xs:element name="mathProgram">
      <xs:complexType>
            <xs:sequence>
                  <xs:element name="sparseVector" type="sparseVector" minOccurs="0" maxOccurs="unbounded"/>
                  <xs:element name="sparseMatrix" type="sparseMatrix" minOccurs="0" maxOccurs="unbounded"/>
                  <xs:element name="linearProgramDescription" type="linearProgramDescription"/>
                  <xs:element name="linearProgramData" type="linearProgramData" minOccurs="0" maxOccurs="unbounded"/>
                  <xs:element name="linearProgramSolution" type="linearProgramSolution" minOccurs="0"
maxOccurs="unbounded"/>
                  <xs:element name="linearProgramIterative" type="linearProgramIterative" minOccurs="0"/>
            </xs:sequence>
      </xs:complexType>
</xs:element>
```

Figure 6-1: LPFML Schema at the root level.

The key benefit of defining the OSxL Schemas is that they impose standards for representing optimization instances. This is critical for parsers that read an instance that can be validated against OSxL Schemas. However, as useful as the validation concept is, validation is about syntax, not semantics. For example, a problem instance that validates against the OSiL

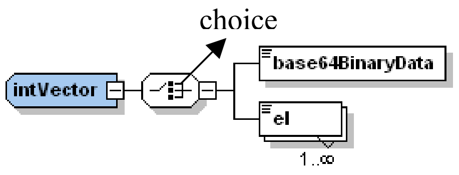
Schema may list a value for the `<numberVariables>` element that is not consistent with the actual number of `<var>` elements in the `<variables>` elements. These problems are not detected by XML validation software and require additional checking on the part of a parser.

6.1 Optimization Services general Language (OSgL)

The OSgL schema is located at http://www.optimizationservices.org/schemas/OSgL.xsd. OSgL defines general elements and data types used by many other OSxL schemas. Thus OSgL is usually included in the beginning of another OSxL schema by

```
<xs:include schemaLocation="OSgL.xsd"/>
```

In the subsequent sections, we will frequently refer to many of the elements and types defined in OSgL. Figure 6-2, for example, shows the `<intVector>` data type in OSgL that is used in OSiL for defining a vector of row or column indexes.



```
<xs:complexType name="intVector">
    <xs:choice>
        <xs:element name="base64BinaryData" type="base64BinaryData"/>
        <xs:element name="el" maxOccurs="unbounded">
            <xs:complexType>
                <xs:simpleContent>
                    <xs:extension base="xs: int ">
                        <xs:attribute name="mult" type="xs:positiveInteger" use="optional" default="1"/>
                        <xs:attribute name="incr" type="xs:int" use="optional" default="0"/>
                    </xs:extension>
                </xs:simpleContent>
            </xs:complexType>
        </xs:element>
    </xs:choice>
</xs:complexType>
```

Figure 6-2: `<intVector>` data type in OSgL.

An `<intVector>` may have one or more `<el>` children or a `<base64BinaryData>` child if the data in the `<el>` elements are compressed. The compression is explained in detail in the LPFML paper [53]. Each `<el>` element has a `mult` attribute (for multiplicity) with a default value 1 and an `incr` attribute (for increment) with a default value 0. For example

`<el>0</el><el>0</el><el>0</el><el>0</el><el>0</el>` and

`<el>0</el><el>1</el><el>2</el><el>3</el><el>4</el>`

is more concisely expressed as

`<el mult="5">0</el>` and `<el mult="5" incr="1">0</el>`

There is also a similar `<doubleVector>` data type defined in OSgL.

`<listMatrix>` (Figure 6-3) is another commonly used data type. It stores the nonzero elements of a sparse matrix. It has three child elements. The first child is `<start>` of type `intVector`. The *ith* `<el>` element in the `intVector` points to the start of the nonzero elements for column (row) i (≥ 0). The number of `<el>` elements should be the number of columns (rows) plus 1. The first `<el>` element should always be 0 and the last `<el>` element should always be the number of nonzero elements. The second child of `<listMatrix>` is `<rowIdx>` (or `<colIdx>)` again of type `intVector` for storing row (or column) indices if the matrix is stored by column (or row). The third element is `<value>` of type `doubleVector` for storing all nonzero values in the matrix.

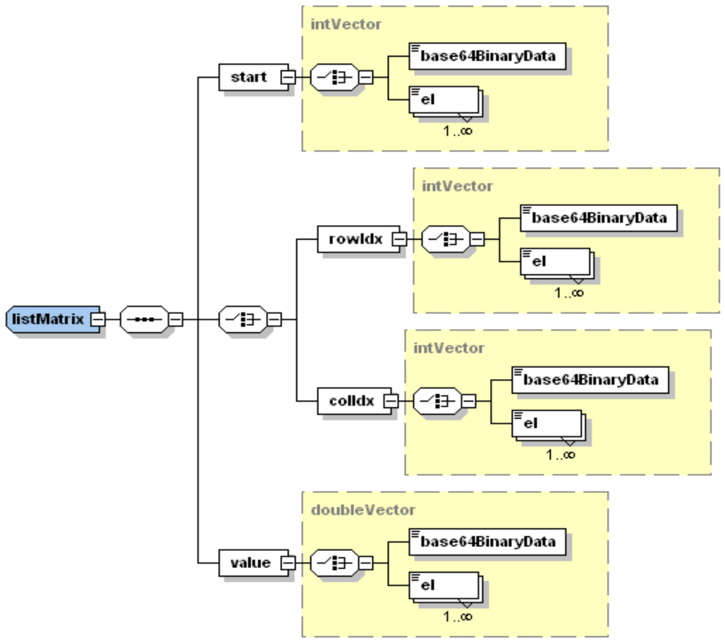


Figure 6-3: `<listMatrix>` data type in OSgL.

In Table 6-1, we list some common data types defined in OSgL.

Type Name	Brief Description
intVector	vector of integers
doubleVector	vector of doubles
elType	`el` element with a `name` and a `value` attribute and a `description` text
mapType	a sequence of `el` elements (name-value pairs)
base64Binary	compression of a sequence of data usually expressed in el elements; see the LPFML paper [53]
sparseVector	sparse vector with an `idx` array and a `nonz` value array

listMatrix	typical sparse matrix storage type with a `start` array, a choice of `rowIdx` or `colIdx` array, and a nonzero `value` array
mpsMatrix	MPS style sparse matrix storage with a sequence of `col` (or `row`) elements; each `col` (or `row`) element is in turn a sequence of `row` (or `col`) elements
matrixMarket	common sparse matrix storage used in linear algebra with a sequence of `el` elements; each element is a `double` value (for a matrix entry) with a `row` and a `col` attribute (for matrix indexes)
xmlData (see § 6.2.2)	a sequence of any data
networkAndGraph (see Appendix A)	comprehensive description of a network and graph topology through a set of `nodes` and `arcs` elements and definitions of `nodeProperties` and `arcProperties`; Reserved for future use.

Table 6-1: Common data types defined in OSgL.

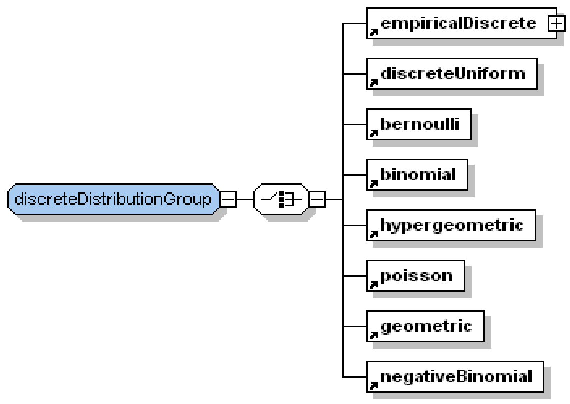
In Table 6-2, we list typical function-related elements. Many of these functions are distribution functions. All the distribution functions can have an optional `cdf` boolean attribute which is false by default. If true, the distribution function is a cumulative distribution function. If false, the distribution is a probability distribution function (pdf, for continuous distributions) or probability mass function (pmf, for discrete distributions). Many of these functions have parameters which are expressed as element attributes. The distribution functions are widely used in the OSiL extension to stochastic programming.

Function Name	Brief Description
userFunctions	a sequence of `userFunction` elements; each `userFunction` element contains one `OSnLNode` as an expression tree root for expressing a function (see § 6.2.2)
userVariables	a sequence of `userVariable` elements; each `userVariable` element contains one `OSnLNode` as an expression tree root for expressing a user defined variable (see §6.2.2)
discreteUniform	Discrete Uniform function with a parameter `N`
bernoulli	Bernoulli function with a parameter `p`
binomial	Binomial function with parameters `N`, `p`
hypergeometric	Hypergeometric function with parameters `N`, `M`, `n`
poisson	Poisson function with a parameter `lamda`
geometric	Geometric function with a parameter `p`
negativeBinomial	Negative Binomial function with a parameter `p`, `r`
empiricalDiscrete	Empirical Discrete function with a sequence of `el` elements; each `el` element is a double value and has a `prob` attribute
empiricalContinuous	Empirical Continuous function with a sequence of `el` elements; each `el` element contains one `OSnLNode` as an expression tree root for expressing a function and a `from` and a `to` attribute for the function domain
uniform	Uniform function with parameters `a`, `b`
normal	Normal function with parameters `mu`, `sigma`
stdNormal	Standard normal function
exponential	Exponential function with a parameter `lamda`
weibull	Weibull function with parameters `location`, `scale`, `shape`
erlang	Erlang function with parameters `lamda`, `n`
gamma	Gamma function with parameters `location`, `scale`, `shape`
beta	Beta function with parameters `degree1`, `degree2`
betaGeneral	General Beta function with parameters `degree1`, `degree2`, `min`, `max`
lognormal	Lognormal function with parameters `mu`, `sigma`
cauchy	Cauchy function with parameters `location`, `scale`

t	Student T function with a parameter `degree`
chiSquare	Chi Square function with a parameter `degree`
f	F function with parameters `degree1`, `degree2`
logistic	Logistic function with parameters `mu` and `beta`
logLogistic	Log Logistic function with parameters `mu` and `beta`
logarithmic	Logarithmic function with parameters `a`, `b`
pareto	Pareto function with parameters `shape` and `scale`
rayleigh	Rayleigh function with a parameter `beta`
pert	Pert function with parameters `a`, `c`, `b`
triangular	Triangular function with parameters `a`, `c`, `b`
multivariateDiscrete	Multivariate Discrete function with a sequence of 2 or more `scenario` elements; each scenario is a sequence of 2 or more `el` elements of double values
multinomial	Multinomial function with a parameter `N` and a sequence of `el` elements of probability values
bivariateNormal	Bivariate Normal function with parameters `mu1`, `sigma1`, `mu2`, `sigma2`, `rho`
multivariateNormal	Multivariate Normal function with a sequence of 3 or more `mu` elements and a `covariance` matrix of `matrixMarket` type
linearTransformation	Linear transformation function with a `numberRows` and a `numberColumns` attribute; it contains one `constants` element of type `doubleVector`, one `matrix` element of type `matrixMarket`, and a `randomVariables` element to indicate a multivariate distribution

Table 6-2: Common function related types defined in OSgL.

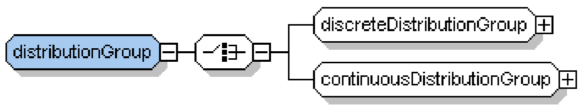
Elements of similar types can be grouped and referenced together. For example the `discreteDistributionGroup` group shown in Figure 6-4 is used to group all the discrete distribution functions shown in Table 6-2.



```
<xs:group name="discreteDistributionGroup">
    <xs:choice>
        <xs:element ref="empiricalDiscrete"/>
        <xs:element ref="discreteUniform"/>
        <xs:element ref="bernoulli"/>
        <xs:element ref="binomial"/>
        <xs:element ref="hypergeometric"/>
        <xs:element ref="poisson"/>
        <xs:element ref="geometric"/>
        <xs:element ref="negativeBinomial"/>
    </xs:choice>
</xs:group>
```

Figure 6-4: `<discreteDistributionGroup>` group in OSgL.

A `continuousDistributionGroup` is similarly defined. The more general `distributionGroup` is a group of `discreteDistributionGroup` and `continuousDistributionGroup` as shown in Figure 6-5.



```
<xs:group name="distributionGroup">
    <xs:choice>
        <xs:group ref="discreteDistributionGroup"/>
        <xs:group ref="continuousDistributionGroup"/>
    </xs:choice>
</xs:group>
```

Figure 6-5: `<distributionGroup>` group in OSgL.

6.2 Optimization Services instance Language (OSiL)

The OSiL schema is located at http://www.optimizationservices.org/schemas/OSiL.xsd. OSiL is definitely the most critical instance representation. OSiL should be interpreted as Optimization Services *input instance* Language. The contents of many other OSxL representations such as the Optimization Services result Language and Optimization Services analysis Language are based on and driven by the OSiL design. In explaining the Optimization Process in §5.3, we see that OSiL is transmitted from and to nearly all the major components on the OS network.

As explained in Chapter 2, there are many modeling languages and even more solvers for computing solutions to mathematical programs. If there are M modeling languages and N solvers, then $M \times N$ drivers are required for complete interoperability. One way to encourage modeler-solver compatibility is to use a standard problem instance representation, so that all modeling languages and all solvers deal with problem instances in the same form. With a standard representation, only $M + N$ software drivers are needed for complete interoperability: each modeling language environment supplies its own driver to output the standard instance and each solver supplies its own driver to read the standard instance.

There are derived research projects in Optimization Services such as Optimization Services instance Interface (OSiI), Optimization Services result Interface (OSrI) and Optimization Services option Interface (OSoI) to standardize local interfaces. An instance is parsed into a standard set of data structures in OSiI. If all the solvers adopt the standard local interfaces, there is potentially only *one* driver at the solver side instead of N drivers. In reality, solvers are implemented in different languages. Suppose there are L (a number $<< N$) programming languages used; then ideally only L driver copies of the same OSiI local interface specification are implemented. Our ultimate goal is thus to have a very small number of drivers. The same logic applies to the adoption of Optimization Services result Language and the corresponding OSrI local interface. More is explained in the examples illustrated in §7.1.

Figure 6-6 shows the root element `<OSiL>` of the OSiL schema. This is the convention of all the OSxL schemas: their root elements are the same as their schema names.

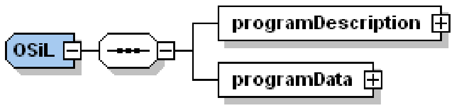


Figure 6-6: OSiL Schema at the root level `<OSiL>`.

The `<OSiL>` element has two children `<programDescription>` and `<programData>`. The `<programDescription>` element is used to convey the basic properties of an optimization instance. All its children are shown in Figure 6-7 and are self-explanatory. Elements in dashed rectangles are optional.

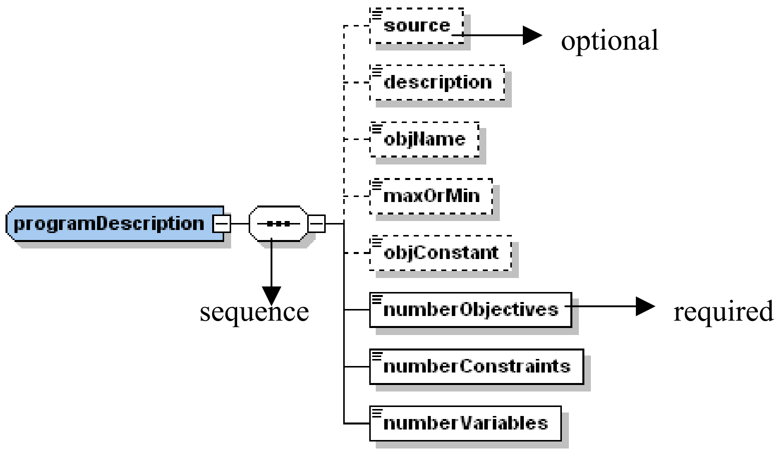


Figure 6-7: `<programDescription>` element in OSiL.

Consider the following optimization problem:

$$\begin{aligned}\underset{x}{\text{minimize}}\quad & 100(x_1 - x_0^2)^2 + (1-x_0)^2 + 7x_1 \\ \text{subject to}\quad & x_0 + 7x_1 \le 10 \\ & \ln(x_0 x_1) + 7x_0 + 5x_1 \le 10 \\ & x_0, x_1 \ge 0\end{aligned} \qquad (6\text{-}1)$$

There are two continuous variables, x_0 and x_1, each with a lower bound of 0. There is one nonlinear objective function. There are two constraints, each with a lower bound (or left-hand side) of $-\infty$ and an upper bound (or right-hand side) of 10. The first constraint is linear and the second constraint is nonlinear.

The `<programDescription>` element for the math program instance is:

```
<OSiL xmlns="os.optimizationservices.org" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="os.optimizationservices.org
http://www.optimizationservices.org/schemas/OSiL.xsd">
    <programDescription>
        <source>Optimization Services, Jun Ma's Thesis</source>
        <description>Adapted from an example of Rosenbock (1960)</description>
        <objName>adaptedRosenbrock</objName>
        <maxOrMin>min</maxOrMin>
        <objConstant>0.0</objConstant>
        <numberObjectives>1</numberObjectives>
        <numberConstraints>2</numberConstraints>
        <numberVariables>2</numberVariables>
    </programDescription>
    <programData>
    . . .
    </programData>
</OSiL>
```

The actual math program data are contained in `<programData>` (Figure 6-8).

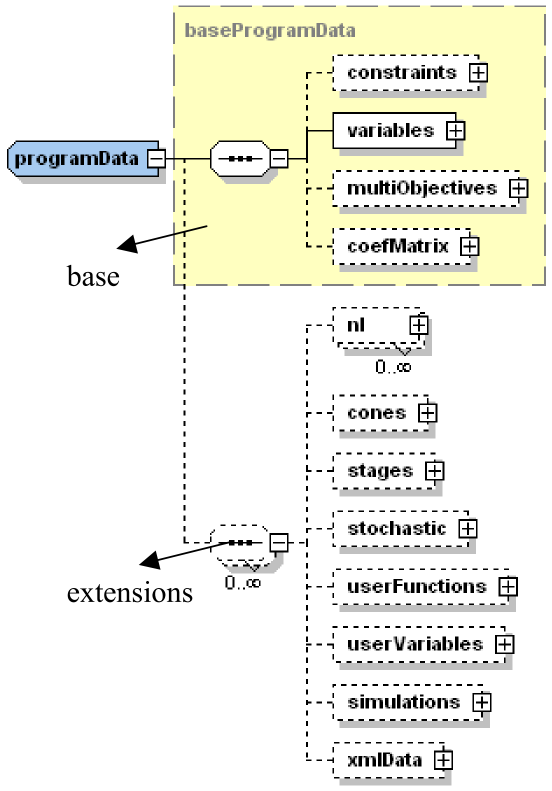


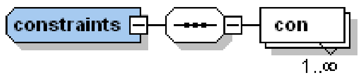
Figure 6-8: `<programData>` element in OSiL.

6.2.1 Base program data

As mentioned in the beginning of this chapter, OSiL has its roots in LPFML for representing linear programs. Our approach for a general nonlinear optimization problem is to write the problem as a linear program (the `baseProgramData` part in Figure 6-8) plus a set of nonlinear expressions for each objective or constraint function (the `<nl>` elements). This allows us to take advantage of the sparsity of most of the linear structures and save space for general nonlinear programs.

In the base program data part, there are four parts, `<constraints>`, `<variables>`, `<multiObjectives>`, and `<coefMatrix>`, which we next describe in sequence.

1. The `<constraints>` element is shown in Figure 6-9.

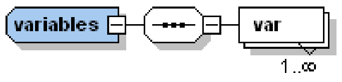


```
<xs:complexType name="constraints">
        <xs:sequence>
            <xs:element name="con" type="con" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
<xs:complexType name="con">
    <xs:attribute name="name" type="xs:string" use="optional"/>
    <xs:attribute name="ub" type="xs:double" use="optional" default="INF"/>
    <xs:attribute name="lb" type="xs:double" use="optional" default="-INF"/>
    <xs:attribute name="mult" type="xs:positiveInteger" use="optional" default="1"/>
</xs:complexType>
```

Figure 6-9: `<constraints>` element in OSiL.

The `<constraints>` element contains a sequence of 1 or more `<con>` elements. Each `<con>` element has an optional `name` attribute. The constraint name is optional because each constraint is referenced by its index (starting from 0) according to the order the `<con>` element is listed in the `<constraints>` element. The `<con>` element also has an optional `ub` attribute which by default is `INF` (positive infinity), and an `lb` attribute which by default is –`INF` (negative infinity). The optional attribute `mult` (for multiplicity) is similar to the `mult` attribute of the `intVector` element explained in the OSgL section (§6.1).

2. The `<variables>` element is shown in Figure 6-10.



```
<xs:complexType name="variables">
    <xs:sequence>
        <xs:element name="var" type="var" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="var">
    <xs:attribute name="name" type="xs:string" use="optional"/>
    <xs:attribute name="init" type="xs:string" use="optional"/>
    <xs:attribute name="type" use="optional" default="C">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="C"/>
                <xs:enumeration value="B"/>
                <xs:enumeration value="I"/>
                <xs:enumeration value="S"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="lb" type="xs:double" use="optional" default="0"/>
    <xs:attribute name="ub" type="xs:double" use="optional" default="INF"/>
    <xs:attribute name="objCoef" type="xs:double" use="optional" default="0.0"/>
    <xs:attribute name="mult" type="xs:positiveInteger" use="optional" default="1"/>
</xs:complexType>
```

Figure 6-10: `<variables>` element in OSiL.

The `<variables>` element contains a sequence of 1 or more `<var>` elements. Each `<var>` element has an optional `name` element. Like constraints, a variable is referenced by its index (starting from 0) according to the order the `<var>` element is listed in the `<variables>` element. The `<var>` element also has an optional `init` attribute of `string` (*not* `double`) type. This is because in certain optimization problems (such as those typically solved by constraint programming), variables may assume non-numeric values. The optional `type` attribute has four possible values: `C` for continuous (default), `B` for binary, I for integer, and `S` for string. The optional `ub` attribute is by default `INF`. The optional `lb` attribute is by default 0. The optional `objCoef` attribute is by default 0. The optional attribute `mult` (for multiplicity) is similar to the `mult` attribute of the `intVector` element explained in the OSgL section (§6.1).

3. The `<coefMatrix>` element is shown in Figure 6-11.

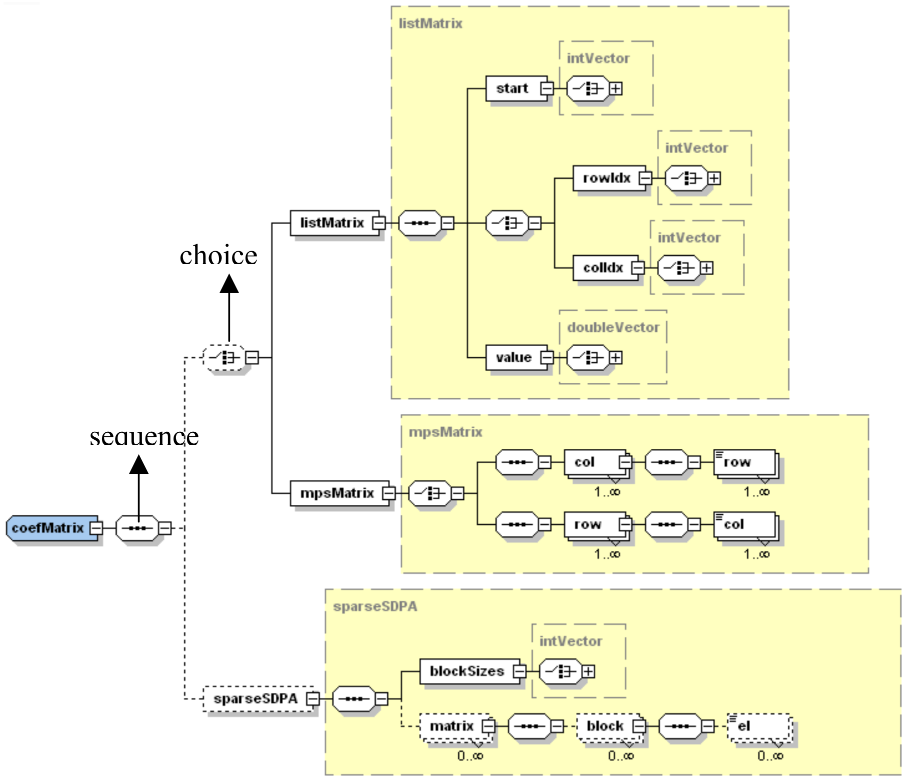


Figure 6-11: `<coefMatrix>` element in OSiL.

The coefficient matrix contains the linear part of the constraints. The `<coefMatrix>` can contain a choice of `listMatrix` and `mpsMatrix`. Both types of matrices can be combined with a `sparseSDPA` matrix, if a sparseSDPA matrix is mixed with a regular linear matrix. The `listMatrix` and `mpsMatrix` elements are briefly explained in the OSgL section (§6.1). The `sparseSDPA` matrix is for semidefinite programming and is mainly an XML version of the sparse SDPA format [89]. There can be other common semidefinite formats, so the semidefinite programming representation may adapt to a better one in the future.

The linear part of the objective function in (6-1) is $7x_1$. The first constraint ($7x_0 + 5x_1 \leq 10$) is linear. The linear part of the second constraint is $7x_0 + 5x_1$. By using the `con` elements to store upper and lower bounds on the constraints, the `var` elements to store the upper and lower bounds on the variables and the objective function coefficients, and the `coefMatrix` element to store the linear part of the constraint matrix, all of the information necessary to represent a linear programming instance, or the linear part of a nonlinear program is represented. For example we can represent the linear part of (6-1) as:

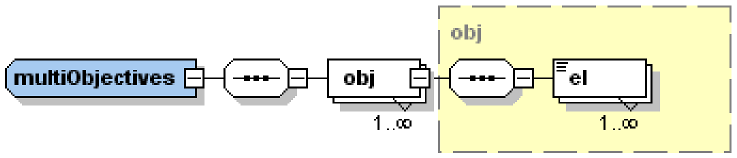
```
<programData>
    <constraints>
        <con ub="10.0"/>
        <con ub="10.0"/>
    </constraints>
    <variables>
        <var name="x0" objCoef="0"/>
        <var name="x1" init="1" lb="0" ub="INF" type="C" objCoef="7"/>
    </variables>
    <coefMatrix>
        <listMatrix>
            <start>
                <el>0</el>
                <el>2</el>
                <el>4</el>
            </start>
            <rowIdx>
                <el>0</el>
                <el>1</el>
                <el>0</el>
                <el>1</el>
            </rowIdx>
            <value>
                <el>1</el>
                <el>7</el>
                <el>7</el>
                <el>5</el>
            </value>
        </listMatrix>
    </coefMatrix>
. . .
<programData>
```

Some of the optional attributes in the above example are explicitly shown for the purpose of illustration. The `listMatrix` element is listed in column major form using the `rowIdx` element. It can be listed in row major form using the `colIdx` element in a similar fashion. Alternatively the `coefMatrix` can be represented using the `mpsMatrix` element:

```
< coefMatrix >
```

```
    <mpsMatrix>
        <col idx="0">
            <row idx="0">1</row>
            <row idx="1">7</row>
        </col>
        <col idx="1">
            <row idx="0">7</row>
            <row idx="1">5</row>
        </col>
    </mpsMatrix>
</ coefMatrix >
```

4. The `<mutlObjectives>` element is shown in Figure 6-12. This element is for optimization with respect to more than one objective.



```
<xs:complexType name="multiObjectives">
    <xs:sequence>
        <xs:element name="obj" maxOccurs="unbounded">
            <xs:complexType>
                <xs:complexContent>
                    <xs:extension base="obj"/>
                </xs:complexContent>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="obj">
    <xs:sequence>
        <xs:element name="el" maxOccurs="unbounded">
            <xs:complexType>
                <xs:simpleContent>
                    <xs:extension base="xs:double">
                        <xs:attribute name="varIdx" type="xs:nonNegativeInteger" use="required"/>
                    </xs:extension>
                </xs:simpleContent>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="optional"/>
    <xs:attribute name="maxOrMin" use="required">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="min"/>
                <xs:enumeration value="max"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="weight" type="xs:double" use="optional" default="1.0"/>
    <xs:attribute name="constant" type="xs:double" use="optional" default="0.0"/>
</xs:complexType>
```

Figure 6-12: `<multiObjectives>` element in OSiL.

The `<mutlObjectives>` element has a sequence of 1 or more `<obj>` elements. Each `<obj>` child element has an optional name attribute. Again like constraints and variables, each objective is referenced by its index; the first objective starts with an index -1, the second with -2, and downwards. Since constraint indexes start from 0 upwards, the objective and constraint indexes do not conflict with each other and together all potential the row indexes span the entire integer domain. If the optimization has only one objective (`objNumber` = 1 in the `programDescription` element), it is recommended that the single objective be specified using the regular mechanism described above and not go inside the `<multiObjectives>` element. Each `<obj>` has a required `maxOrMin` attribute which can take on a value of either `max` or `min`. Each `<obj>` can also have an optional `weight` attribute (1 by default), and an optional `constant` attribute (0 by default). The `<obj>` element contains a sequence of 1 or more double-valued `<el>` elements to specify objective coefficients. Each `<el>` element has a required `varIdx` attribute to indicate to which variable the coefficient belongs.

6.2.2 Extension elements

There are currently eight extension elements in the `programData` (Figure 6-8) of OSiL. They are `<nl>` for nonlinear programming, `<cones>` for cone programming, `<stages>` for any math programming that uses stage information (e.g. dynamic programming, stochastic programming), `<stochastic>` for stochastic programming, `<userFunctions>` for user-defined functions, `<userVariables>` for user-defined variables, `<simulations>` for definitions of simulations, and `<xmlData>` for data representation in XML form. The extension to semidefinite programming is already included in the `baseProgramData` part through the representation of constraint matrix in `sparseSDPA`. The extensions to quadratic programs, constraint programs, and complementarity problems are included in the nonlinear programming extension through incorporation of special nonlinear nodes described in §6.3. We go through each of the eight extensions below.

1. `<nl>` for nonlinear programming (including quadratic programming, constraint programming, complementarity programming)

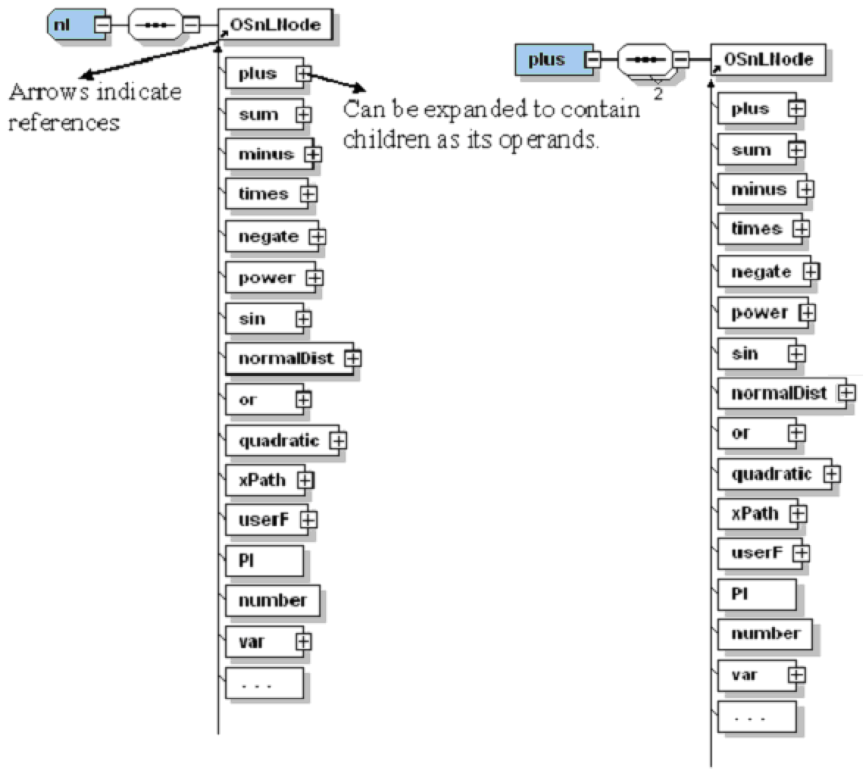
In keeping with the philosophy of separating out the linear and nonlinear parts of an optimization instance, the nonlinear *terms* in an instance are defined using the OSnL schema.

The OSiL schema then imports the OSnL schema. The OSnL schema represents a wide variety of general nonlinear operators, functions and operands and is made simple to parse by adoption of a recursive design. OSnL is detailed in §6.3.

For nonlinear extensions, an alternative to the OSiL approach is Content MathML as described in §4.2.2. In that same section, we listed reasons why we decided against using

Content MathML to represent general nonlinear optimization problems. However, in order to be as consistent with MathML as possible, we adopt the MathML element names whenever possible, for example `<ln>` for natural logarithm.

The way OSiL uses OSnL is through the use of a sequence of 0 or more `<nl>` elements immediately after the last element in the base program data as shown in Figure 6-8. Figure 6-13 shows the definition of the `<nl>` element in OSiL.



```
<xs:complexType name="nl">
    <xs:sequence>
        <xs:element ref="OSnLNode"/>
    </xs:sequence>
    <xs:attribute name="idx" type="xs:int" use="required"/>
</xs:complexType>
```

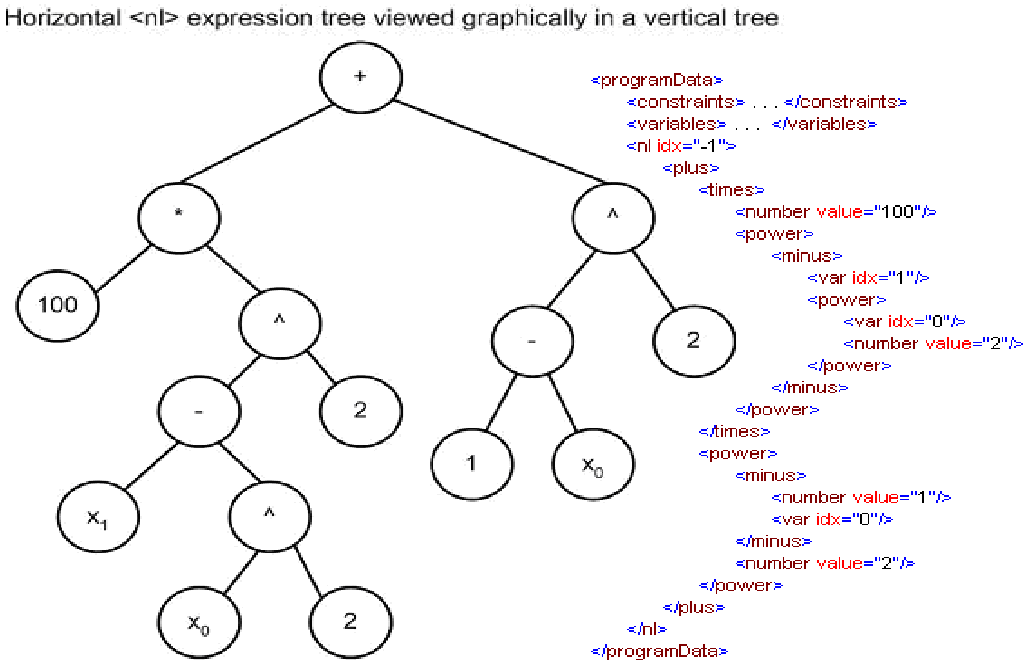
Figure 6-13: `<nl>` element in OSiL.

The `<nl>` element has a required attribute `idx`, to indicate that it is *part* of an objective or constraint function whose index is equal to `idx`. Objectives are indexed by negative integers, with the first one being -1, the second one -2, and so on. Constraints are indexed by nonnegative integers, with the first one being 0, the second one 1, and so on. So `<nl`

`idx="0">` indicates the nonlinear expression belongs to the first constraint; and `<nl idx="-1">` indicates the nonlinear expression belongs to the first objective function.

The `<nl>` element contains one and only one `OSnLNode` element. This single child element is the root element of the nonlinear expression. `OSnLNode` is an *abstract* element defined in OSnL. In a real instance, it is represented by a *concrete* element, such as `plus`, `times`, `sum`, `PI`, `number`, `var`. That is why in the `<nl>` schema, the child element is written as `<xs:element ref="OSnLNode"/>`, using a `ref` attribute rather than a `type` attribute. The concrete elements are all defined in OSnL. There are more than 200 concrete elements that represent various operators, functions, and operands. All inherit from the abstract `OSnLNode` element through the idea of *substitution groups*. Inheritance through substitution groups was described in §4.3 and is illustrated in detail in §6.3.

Each concrete `OSnLNode` element can have 0 or more concrete child elements that also inherit from `OSnLNode`. For example, if the concrete element is `plus`, it has exactly two child concrete elements that inherit from OSnLnodes. If the concrete element is an operand (e.g. a constant `PI`), it has no children. This recursive design allows us to build an entire expression tree in a clean, effective and scalable way.


Horizontal <nl> expression tree viewed graphically in a vertical tree
+
*
^
100
^
-
2
-
2
x1
^
1
x0
x0
2

```
<programData>
    <constraints> . . . </constraints>
    <variables> . . . </variables>
    <nl idx="-1">
        <plus>
            <times>
                <number value="100"/>
                <power>
                    <minus>
                        <var idx="1"/>
                        <power>
                            <var idx="0"/>
                            <number value="2"/>
                        </power>
                    </minus>
                </power>
            </times>
            <power>
                <minus>
                    <number value="1"/>
                    <var idx="0"/>
                </minus>
                <number value="2"/>
            </power>
        </plus>
    </nl>
</programData>
```

Figure 6-14: Objective function nonlinear part $100(x_1 - x_0^2)^2 + (1 - x_0)^2$ represented in `<nl>` and the corresponding vertical tree view of the expression.

The nonlinear part of the objective function (`idx` = -1) in (6-1) is $100(x_1 - x_0^2)^2 + (1 - x_0)^2$.
Its XML representation (horizontal tree) and the corresponding tree visualization (vertical tree) is shown in Figure 6-14. The first constraint in (6-1) is linear; it does not have a corresponding `<nl>` element. The linear part of the second constraint (`idx` = 1) is $\ln(x_0 x_1)$. It is represented as

```
<nl idx="1">
    <ln>
        <times>
            <var idx="0"/>
            <var idx="1"/>
        </times>
    </ln>
</nl>
```

Thus the entire optimization problem (6-1) has been characterized in XML.

2. `<cones>` for cone programming

The cone programming extension mainly addresses second-order cone programming (SOCP). SOCP is usually solved with some kind of primal-dual interior point method. The objective function is usually linear, while the constraints are an intersection of an affine set and the direct product of quadratic cones. See [75] for more details. OSiL extension to cone programming using the `<cones>` element is explained in detail in Appendix A.

3. `<stages>` for math programs using stage information

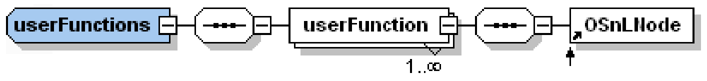
Information of stages is used in several optimization types, such as dynamic programming, and stochastic programming. OSiL extension to these problems types using the `<stages>` element is explained in detail in Appendix A.

4. `<stochastic>` for stochastic programming

For a complete review of stochastic programming, refer to [11]. The OSiL stochastic programming extension is designed to make it convenient and powerful to transform existing deterministic linear or nonlinear programs into stochastic programs by adding dynamic and stochastic structure information. It was first designed totally independent of the SMPS format[10] and later, through working with Horand Gassmann, one of the coauthors of the original SMPS format, added many new ideas. OSiL extension to these problems types using the `<stages>` element is explained in detail in Appendix A.

5. `<userFunctions>` for user-defined functions in terms of OSnLNode expression trees

Figure 6-15 shows the `<userFunctions> element`.



```
<xs:element name="userFunction" maxOccurs="unbounded">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="OSnLNode"/>
        </xs:sequence>
        <xs:attribute name="name" type="xs:ID" use="required"/>
        <xs:attribute name="numArg" type="xs:nonNegativeInteger" use="required"/>
    </xs:complexType>
</xs:element>
```

Figure 6-15: `<userFunctions>` element in OSiL.

The `<userFunctions>` element contains a sequence of one or more `<userFunction>` elements. Each `<userFunction>` has a required `name` and `numArg` (number of arguments) attribute. `<userFunction>` has one and only one `OSnLNode` element. This single child element is the root element of the nonlinear expression that represents a user-defined function. The use of `OSnLNode` is just like defining any nonlinear expression in the `<nl>` element, except that it uses some `<arg>` elements. The definition of the user function should be independent of the mathematical program represented by the OSiL instance. For example, it is required that the user function definition can not use math program variables (`<var>` elements) because the variable indexes change between instances of the same optimization problem; instead, the `<arg>` element from OSnL is provided to define the user function. If a modeler wants to define “user” variables using the math program variables, he should use the `<userVariables>` element described below. For example, suppose the nonlinear term $100(x_1 - x_0^2)^2$ in the objective function of (6-1) is defined by a user function called `myFunction` as $myFunction(\text{arg}_0, \text{arg}_1) = 100(\text{arg}_1 - \text{arg}_0{}^2)^2$. It is represented using the `<userFunctions>` element as

```
<userFunctions>
    <userFunction name="myFunction" numArg="2">
        <times>
            <number value="100"/>
            <power>
                <minus>
                    <arg idx="1"/>
                    <power>
                        <arg idx="0"/>
                        <number value="2"/>
                    </power>
                </minus>
                <number value="2"/>
            </power>
        </times>
    </userFunction>
```

```
</userFunctions>
```

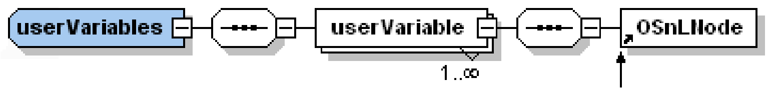
In the above example, we should *not* use `<var idx="0"/>` instead of `<arg idx ="0"/>`. With the definition of the user function, we can represent the nonlinear part of the objective function in OSiL as:

```
<nl idx="-1">
    <plus>
        <userF name="myFunction">
            <var idx="0"/>
            <var idx="1"/>
        </userF>
        <power>
            <minus>
                <number value="1"/>
                <var idx="0"/>
            </minus>
            <number value="2"/>
        </power>
    </plus>
</nl>
```

The `<userF>` nonlinear node is explained in §6.3.

6. `<userVariables>` for user-defined variables in terms of OSnLNode expression trees

Figure 6-16 shows the `<userVariables>` element, which contains a sequence of `<userVariable>` child elements.



```
<xs:complexType name="userVariables">
    <xs:sequence>
        <xs:element name="userVariable" maxOccurs="unbounded">
            <xs:complexType>
                <xs:sequence>
                    <xs:element ref="OSnLNode"/>
                </xs:sequence>
                <xs:attribute name="name" type="xs:ID" use="required"/>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
</xs:complexType>
```

Figure 6-16: `<userVariables>` element in OSiL.

The use of the `<userVariables>` element is very similar to that of `<userFunctrions>`, only that the user-defined variables are defined over the math program variables declared in the `<variables>` element and therefore there are no arguments to pass to each `userVariable`. So, unlike the user functions, user variables highly depend on the math program instance.

Like `<userFunction>`, each `<userVaraible>` has one and only one `OSnLNode` element which defines the root element of the nonlinear expression that represents a user-defined variable.

Suppose the nonlinear term $100(x_1 - x_0^2)^2$ in the objective function of (6-1) is defined by a user-defined variable called `myVariable`. It is represented using the `<userVariables>` element as

```
<userVariables>
    <userVariable name="myVariable" numArg="2">
        <times>
            <number value="100"/>
            <power>
                <minus>
                    <var idx="1"/>
                    <power>
                        <var idx="0"/>
                        <number value="2"/>
                    </power>
                </minus>
                <number value="2"/>
            </power>
        </times>
    </userFunction>
</userFunctions>
```

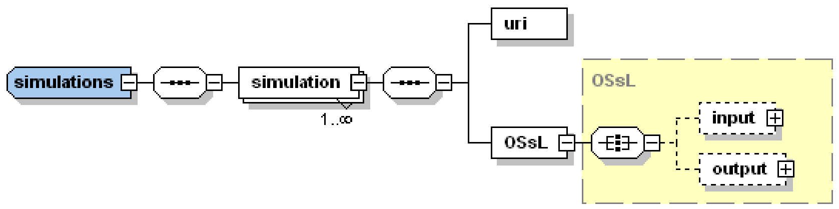
In the above example, there are no `<arg>` elements as there are not going to be arguments passed in. We directly use `<var idx="0"/>` and `<var idx="1"/>` from the math program instance. In a sense, every `<userVariable>` element has all the `<var>` elements as their predefined arguments. With the user variable definition, the nonlinear part of the objective function is represented in OSiL as:

```
<nl idx="-1">
    <plus>
        <userVar name="myVariable"/>
        <power>
            <minus>
                <number value="1"/>
                <var idx="0"/>
            </minus>
            <number value="2"/>
        </power>
    </plus>
</nl>
```

The `<userVar>` nonlinear node is explained in §6.3.

7. `<simulations>` for definition of black-box calculations of any type

Figure 6-17 shows the `<simulations> element`.



```
<xs:complexType name="simulations">
    <xs:sequence>
        <xs:element name="simulation" maxOccurs="unbounded">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="uri">
                        <xs:complexType>
                            <xs:attribute name="value" type="xs:anyURI" use="required"/>
                        </xs:complexType>
                    </xs:element>
                    <xs:element name="OSsL" type="OSsL"/>
                </xs:sequence>
                <xs:attribute name="name" type="xs:ID" use="required"/>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
</xs:complexType>
```

Figure 6-17: `<simulations>` element in OSiL.

The `<simulations>` element contains a sequence of one or more `<simulation>` elements. Each `<simulation>` has a required `name` attribute. A simulation is similar to a user function as described in §2.8, except that there is no longer a closed form expression for the function. Instead, three things have to be specified for the simulation: input, output, and the simulation's address. Thus each `<simulation>` has a required `<uri>` child to specify an address in URI format and a required OSsL element to specify the input and output of the simulation. OSsL is specified in the Optimization Services simulation Language of §6.7. Suppose the nonlinear term $100(x_1 - x_0^2)^2$ in the objective function of (6-1) is one of the multiple outputs from a simulation called `simpleSimulation` shown in Figure 6-18. Of course, in reality the simulation calculation can be extremely complex.

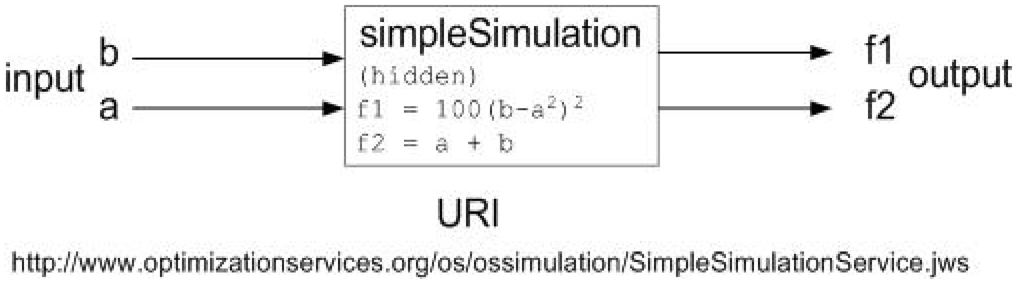

input
b
a
simpleSimulation
(hidden)
f1 = 100(b-a²)²
f2 = a + b
f1
f2
output
URI
http://www.optimizationservices.org/os/ossimulation/SimpleSimulationService.jws

Figure 6-18: simpleSimulation with two inputs (a, b), two outputs (f1, f2) and an address at http://www.optimizationservices.org/os/ossimulation/SimpleSimulationService.jws.

The `simpleSimulation` element is represented using the `<simulations>` element as

```
<simulations>
    <simulation name="simpleSimulation">
        <uri value="http://www.optimizationservices.org/os/ossimulation/SimpleSimulationService.jws"/>
        <OSsL>
            <input>
            <el name="a"/> <el name="b"/>
            </input>
            <output>
            <el name="f1"/><el name="f2"/>
            </output>
        </OSsL>
    </simulation>
</simulations>
```

Notice `simpleSimulation` has two outputs and we only need the first output `f1`. Now the objective function is written as $mySimulation(x_0, x_1) \rightarrow f_1 + (1 - x_0)^2 + 7x_1$ and the nonlinear part of the user function is represented as

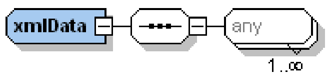
```
<nl idx="-1">
    <plus>
        <sim name="simpleSimulation">
            <simInput inputName="a"> <var idx="0"/> </simInput>
            <simInput inputName="b"> <var idx="1"/> </simInput>
            <simOutput outputName="f1"/>
        </sim>
        <power>
            <minus>
                <number value="1"/>
                <var idx="0"/>
            </minus>
            <number value="2"/>
        </power>
    </plus>
</nl>
```

The `<sim>` nonlinear node is explained in more detail in §6.3. The process of invocation of simulation services in simulation optimization is explained in more detail in §7.2.

8. `<xmlData>` for data definition in XML form

Figure 6-19: `<xmlData>` element in OSiL.

Figure 6-19 shows the `<xmlData> element`.



```
<xs:complexType name="xmlData">
    <xs:sequence>
        <xs:any processContents="skip" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
```

Figure 6-19: `<xmlData>` element in OSiL.

In an optimization instance, there is usually no need to separately keep a data list, as all the data are substituted into the instance. But occasionally there is the need to keep separate data. For example, in constraint programming, a parameter can be indexed over variables such as p[x[0], x[1]]. Parameter values are not known at compile time. It is not until run time, when the variable values (x[0], x[1]) are known, can parameter values be retrieved. A simple example of keeping the parameters ($p[i, j]$) in `xmlData` is shown below:

```
<xmlData>
    <p>
        <i>
            <j>1.2</j><j>1.3</j>
        </i>
        <i>
            <j>0.4</j><j>0.5</j>
        </i>
        <i>
            <j>3.1</j><j>4.5</j>
        </i>
    </p>
</xmlData>
```

In the example $p[0,0] = 1.2$, $p[0,1] = 1.3$, $p[1,0] = 0.4$, $p[1,1] = 0.5$, $p[2,0] = 3.1$ and $p[2,1] = 4.5$. Notice almost all the databases and spreadsheets are xml-enabled, meaning that they can at least export the data in XML formats, which can then be retrieved with the standard XPath language (§4.4). So this simple example has its universal appeal in practice. For example to get the value of $p[2,0]$ (the 1st <j> element in the 3rd <i> element; xPath starts element index with 1), we construct the following XPath:

```
xmlData/p/i[position()="3"]/j[position()="1"]
```

Of course, in an optimization process, we pass in x[0] and x[1] instead of the numbers "`3`" and "`1`". We will explain more on the `<xPath>` nonlinear node in §6.3; the nonlinear node uses the XPath syntax to retrieve data values from any XML data.

6.3 Optimization Services nonlinear Language (OSnL)

The OSnL schema is located at http://www.optimizationservices.org/schemas/OSnL.xsd. In keeping with the philosophy of separating out the linear and nonlinear parts of an optimization instance, the nonlinear *expressions* in an instance are defined using the OSnL schema. OSnL itself is *not* a nonlinear program instance representation. As described in the OSiL section (§6.2), all types of optimization instances are described using OSiL. More appropriately OSnL should be interpreted as Optimization Services *nonlinear node* Language. OSnL defines nonlinear nodes and nodes *only*. The nodes can be operators, functions or terminal operands. Operators always have child nodes. Function may or may not have child nodes. Terminal operands do not have children. Examples of terminal nodes are the `number` node and constant nodes such as `PI` and `E`.

OSnL is then included in the OSiL schema to support nonlinear instance representation *in OSiL*. The way OSiL uses OSnL is through the use of a sequence of 0 or more `<nl>`; each `<nl>` element has an only child OSnLNode as an expression tree root to define a nonlinear function. This is described in detail in the OSiL section (§6.2).

In §4.3, we described schema type inheritance through the idea of substitution groups. For a nonlinear expression, we use an expression tree and view every node in the expression tree as a generic node, which we call "OSnLNode." Each OSnLNode can have 0 or more OSnLNode children. A terminal node is just an OSnLNode without children. To represent a generic node, at the beginning of the OSnL schema, we create a complex type `OSnLNode`:

```
<xs:complexType name="OSnLNode" mixed="false">
    <xs:annotation>
        <xs:documentation>This is a generic node from which we derive operator nodes</xs:documentation>
    </xs:annotation>
</xs:complexType>
```

The `annotation` element is just an XML schema comment. Next we create a substitution group based on the named *element* `OSnLNode`, which is of the above *type* `OSnLNode`.

```
<xs:element name="OSnLNode" type="OSnLNode" abstract="true">
    <xs:annotation>
        <xs:documentation> Set abstract to true in order to create a substitution group</xs:documentation>
    </xs:annotation>
</xs:element>
```

So we can think of `OSnLNode` as a derived class. Note the `abstract` attribute is set to the value of `true` in order to create the abstract class. Now with the substitution group defined, throughout the rest of the OSnL schema, we create concrete OSnLNode elements that are in the substitution group for `OSnLNode`. For example, the first concrete element we define is an `OSnLNode` for addition:

```
<xs:complexType name="OSnLNodePlus">
    <xs:complexContent>
        <xs:extension base="OSnLNode">
            <xs:sequence minOccurs="2" maxOccurs="2">
                <xs:element ref="OSnLNode"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name="plus" type="OSnLNodePlus" substitutionGroup="OSnLNode"/>
```

We first define the complex type `OSnLNodePlus` and we then create the derived element `plus` that is in the substitution group `OSnLNode`. Note that the `plus` element requires exactly two child elements (`<xs:sequence minOccurs="2" maxOccurs="2">`), both of which should be in the `OSnLNode` substitution group too (`<xs:element ref="OSnLNode"/>`). In a similar fashion, we define all other OSnL nodes such as `minus`, `divide`, `arcsin`, `sum`, `E`, `var`, `leq`, `if`, `complements`, `xPath`, `userF`, `quadratic`. For nodes such as `sum`, as the `sum` operator is an indefinite type, the corresponding `OSnLNodeSum` requires one or more child elements:

```
<xs:complexType name="OSnLNodeSum">
    <xs:complexContent>
        <xs:extension base="OSnLNode">
            <xs:sequence minOccurs="1" maxOccurs="unbounded">
                <xs:element ref="OSnLNode"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name="sum" type="OSnLNodeSum" substitutionGroup="OSnLNode"/>
```

For nodes such as `E`, as `E` is a constant, the corresponding `OSnLNodeE` has no children:

```
<xs:complexType name="OSnLNodeE">
    <xs:complexContent>
        <xs:extension base="OSnLNode"/>
    </xs:complexContent>
</xs:complexType>
<xs:element name="E" type="OSnLNodeE" substitutionGroup="OSnLNode"/>
```

he recursive design provides a significantly simple and powerful way to construct a nonlinear expression. As shown in Figure 6-13 in the OSiL section (§6.2), the definition of an `<nl>` element for the nonlinear extension is only six short lines.

When a concrete expression tree is finally constructed, it may look like:

```
<sum>
    <times>
        <var idx="0" coef="3"/>
        <var idx="1"/>
    </times>
    <power>
        <var idx="0" coef="4"/>
        <number value="2"/>
    </power>
    <divide>
        </PI>
        <var idx="0"/>
    </divide>
</sum>
```

for the nonlinear expression $(3x_0x_1) + 4x_0^2 + \pi / x_0$.

The OSnL schema is very compressive; over 200 elements are supported. They fall broadly into the following 8 categories:

1. Arithmetic operators

2. Elementary functions

3. Trigonometric functions
4. Statistical and probability functions
5. Terminals and constants
6. Optimization related elements
7. Logic and relational operators
8. Special elements

The input and output of all the currently defined operators/functions are scalars ($R^n \to R^1$). Set or vector valued function elements may be added in the future. Next we describe elements in each category.

1. Arithmetic operators

In Table 6-3, we list the arithmetic operator elements.

Name	Child #	Note		Name	Child #	Note
plus	2	+		divide	2	÷
sum	1 or more	Σ		quotient	2	\; e.g. 11 quotient 4 = 2
minus	2	-		rem	2	remainder, e.g. 11 rem 4 = 3
negate	1	(-)		power	2	^ ; base is the 1st child; exponent is the 2nd child
times	2	×		product	1 or more	Π

Table 6-3: Arithmetic operators in OSnL.

"Child #" indicates the number of OSnLNode children that an element can take as operands. Elements with 1 operand are of *unary* type. Elements with 2 operands are of *binary* type. Elments with *n* (*n*>2) operands are of *n*-nary type. Some of the subsequent element types have n-nary functions. Elements such as `sum` and `product` are of indefinite types. As useful as the validation concept is, validation is about syntax not semantics. For example, the OSnL schema can make sure there are exactly 2 child elements for the `<divide>` element, but the schema cannot make sure the second child evaluates to a non-zero. Similarly the schema does not check whether the child elements of the integer-based `quotient` and `rem` operators evaluate to integers. These require additional checking and interpretation on the part of a parser.

2. Elementary functions

In Table 6-4, we list the elementary function elements.

Name	Child #	Note		Name	Child #	Note
abs	1	$\|a\|$		ln	1	natural log of a
squareRoot	1	$\sqrt{a}$		log	2	log (a, b) = $\log_a b$
square	1	a^2		log10	1	log10(a) = $\log_{10} b$
floor	1	$\lfloor a \rfloor$		round*	2	see blow
ceiling	1	$\lceil a \rceil$		roundToInt	1	round to integer
factorial	1	$n!$		gcd	2	greatest common divisor

exp	1	e^a		lcm	2	least common multiple
combination	2	${}_nC_k$		truncate*	2	see below
permutation	2	${}_nP_k$		rand*	1	see below
percent	1	$a\%$		gammaFn	1	gamma function
sign	1	1 or -1		gammaLn	1	natural log of the gamma function

Table 6-4: elementary functions in OSnL.

Most of the elementary functions are self-explanatory.

The `round` function takes 2 children. The first child is the number to be rounded. The second child is the number of digits to round; a negative number rounds to the left of the decimal point; zero to the nearest integer.

The `truncate` function truncates a number to an integer by removing the fractional part of the number. It takes 2 children. The first child is the number to be truncated. The second child indicates truncation precision; negative number truncates to the left of the decimal point; zero to the nearest integer.

The `rand` function takes 1 child as a seed. It returns a random number from a continuous uniform distribution ≥ 0 and < 1.

3. Trigonometric functions

All the 24 standard trigonometric functions are defined in OSnL (Table 6-5).

Name		**Name**		**Name**		**Name**		**Name**		**Name**
sin		cos		tan		cot		sec		csc
sinh		cosh		tanh		coth		sech		csch
arcsin		arccos		arctan		arccot		arcsec		arccsc
arcsinh		arccosh		arctanh		arccoth		arcsech		arccsch

Table 6-5: Trigonometric functions in OSnL.

A trigonometric function takes one and only one child.

4. Statistical and probability functions

In Table 6-6, we list the statistical function elements that take one list of data, and hence are of indefinite types.

Name (*no notes*)		**Name**	**Note**
mean		Absdev	average of absolution deviations from the mean
geometricMean		Stddev	standard deviation
harmonicMean		Cv	coefficient of variance (standard deviation / mean)
count		Large	the nth largest number in a data list; n (>0) is the 1st child
median		Small	the nth smallest number in a data list; n (>0) is the 1st child
mode		Percentile	the nth percentile in a data list; n $(\geq 0, \leq 1)$ is the 1st child
min		interQuantileRange	thirdQuartile - firstQuartile
max		Range	max - min

skewness		trimMean*	see below; fractional value is the 1st child
kurtosis		Npv	net present value; discount rate r is the 1st child
firstQuartile		Irr	internal rate of return
thirdQuartile		autocorrelation1	regular autocorrelation with lag = 1
variance		autocorrelation	general correlation with lag = n (≥ 1); n is the 1st child

Table 6-6: Statistical functions that take a list of data in OSnL (indefinite types).

Each entry in the data list corresponds to a child node. The parameters (if any) of a statistical function go before the data list children. For example the `large` function takes a number `n` as its first child to indicate the nth largest number in the rest of the children. Most of the statistical functions are self-explanatory.

The `trimMean` function takes 2 or more children. The first child is a number ($\in [0,1]$) indicating the fraction of data points to exclude from the top and bottom of the data list. The rest of the children (from the second on) are the data list.

In Table 6-7, we list the statistical function elements that take two data lists as operands.

Name	Note
covariance	covariance of two data lists; 1st data list is the 1st half of the children
correlation	correlation of two data lists; 1st data list is the 1st half of the children
pearsonCorrelation	Pearson product moment correlation coefficient; 1st data list is the 1st half of the children
rankCorrelation	rank correlation of two data lists; 1st data list is the 1st half of the children

Table 6-7: Statistical functions that take two lists of data in OSnL (indefinite types).

Each entry in the 2 data lists corresponds to a child node. The parameters (if any) of a statistical function go before the data list children. After the parameter children, there should be even number of the rest of the children; the first half of these children corresponds to the first data list; the second half of these children corresponds to the second data list.

In Table 6-8, we list the probability function elements. Almost all probability functions can have three versions: density, cumulative, and inverse; child arguments for the three versions are exactly the same. An OSnL element is created for each version (if there is one). Density type elements are suffixed with "`Dist`", cumulative type elements are suffixed with "`Cum`", and inverse type elements are suffixed with "`Inv`."

Name (Density)	Name (Cumulative)	Name (Inverse)	Child #	Sequence of Children `(param1, ..., param2, x)`
discreteUniformDist	discreteUniformCum	discreteUniformInv	2	`(N, x)`
bernoulliDist	bernoulliCum	bernoulliInv	2	`(p, x)`
binomialDist	binomialCum	binomialInv	3	`(N, p, x)`
multinomialDist	multinomialCum	multinomialInv	3 or more	(N, p_1, p_2, ..., p_n, x)

hypergeometricDist	hypergeometricCum	hypergeometricInv	4	`(N, M, n, x)`
poissonDist	poissonCum	poissonInv	2	`(lamda, x)`
geometricDist	geometricCum	geometricInv	2	`(p, x)`
negativeBinomialDist	negativeBinomialCum	negativeBinomialInv	3	`(p, r, x)`
uniformDist	uniformCum	uniformInv	3	`(a, b, x)`
normalDist	normalCum	normalInv	3	`(mu, sigma, x)`
stdNormalDist	stdNormalCum	stdNormalInv	1	`(x)`
bivariateNormalDist	bivariateNormalCum	/	7	`(mu1, sigma1, mu2, sigma2, pho, x1, x2)`
exponentialDist	exponentialCum	exponentialInv	2	`(lamda, x)`
weibullDist	weibullCum	weibullInv	4	`(location, scale, shape, x)`
erlangDist	erlangCum	erlangInv	3	`(lamda, n, x)`
gammaDist	gammaCum	gammaInv	4	`(location, scale, shape, x)`
betaDist	betaCum	betaInv	3	`(degree1, degree2,x)`
betaGeneralDist	betaGeneralCum	betaGeneralInv	5	`(degree1, degree2, min, max, x)`
lognormalDist	lognormalCum	lognormalInv	3	`(mu, sigma, x)`
cauchyDist	cauchyCum	cauchyInv	3	`(location, scale, x)`
tDist	tCum	tInv	2	`(degree, x)`
chiSquareDist	chiSquareCum	chiSquareInv	2	`(degree, x)`
fDist	fCum	fInv	3	`(degree1, degree2, x)`
logisticDist	logisticCum	logisticInv	3	`(mu,beta, x)`
logLogisticDist	logLogisticCum	logLogisticInv	3	`(mu,beta, x)`
logarithmicDist	logarithmicCum	logarithmicInv	3	`(a, b, x)`
paretoDist	paretoCum	paretoDist	3	`(shape, scale, x)`
rayleighDist	rayleighCum	rayleighInv	2	`(beta, x)`
pertDist	pertCum	pertInv	4	`(a, c, b, x)`
triangularDist	triangularCum	triangularInv	4	`(a, c, b, x)`

Table 6-8: Probability functions (density, cumulative, inverse) in OSnL.

All the probability functions and related parameters are quite standard; they are named to be indicative of what are used in common practice. The last child (or last two in bivariate cases) always evaluates to a number that corresponds to the distribution function variable (or variables). The parameters of a probability function (if any) go before the variable child (or children).

5. Terminals and constants

In Table 6-9, we list the terminal elements, which do not have children.

Name	Attributes
number	`value`, `type`, `id`
identifier	`Name`

Table 6-9: Terminals in OSnL.

The `number` schema is shown below:

```
<xs:complexType name="OSnLNodeNumber">
```

```
    <xs:complexContent>
        <xs:extension base="OSnLNode">
            <xs:attribute name="value" type="xs:string" use="required"/>
            <xs:attribute name="type" use="optional" default="real">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="real"/>
                        <xs:enumeration value="string"/>
                        <xs:enumeration value="random"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="id" type="xs:ID" use="optional"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name="number" type="OSnLNodeNumber" substitutionGroup="OSnLNode"/>
```

The `<number>` element has three optional attributes `value`, `type`, and `id`. The `value` attribute is required and is of `string` type. The `type` attribute is optional and can take on a value of either `real` (default), `string`, or `random`. A "string-valued" `number` is often used in constraint programming. A "random" `number` is often used in stochastic programming and in this case the `value` attribute of `number` can either be treated as an initial value or ignored. The `id` attribute is optional. But if there is one, it has to be unique as it of type `ID`. A number with an `id` can be located. For example, in stochastic programming, we may need to change the number to different values in different scenarios. For example, all the following are valid `number` elements:

```
<number value="100"/>
<number value="100" type="real"/>
<number value="Chicago" type="string"/>
<number value="3.2" type="random" id="n4"/>
```

The `identifier` schema is shown below:

```
<xs:complexType name="OSnLNodeIdentifier">
    <xs:complexContent>
        <xs:extension base="OSnLNode">
            <xs:attribute name="name" type="xs:string" use="required"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name="identifier" type="OSnLNodeIdentifier" substitutionGroup="OSnLNode"/>
```

The `<identifier>` element has one required `name` attribute. It is seldom used in numerical optimization. It can potentially be used for symbolic optimization. The following is an example of an `identifier` element:

```
<identifer name="a"/>
```

A variable is not always a terminal node, as it may take a child operand as its index. This is explained later.

In Table 6-10, we list the constant elements, which do not have children.

Constants	PI, E, TRUE, FALSE, EULERGAMMA, INF (infinity), EPS (epsilon), NAN (Not a Number)

Table 6-10: Constants in OSnL.

Most of these constants are well supported in various programming languages. So parser implementation can leverage on the support from the programming languages. `TRUE` and `FALSE` are not double values, but parsers may for example choose to use a positive number to represent `TRUE` and a negative number to represent `FALSE`.

6. Optimization related elements

In Table 6-11, we list the three optimization related elements.

Name	Child #	Attributes
var	0 or 1	`idx`, `coef`
objective	0 or 1	`idx`
constraint	0 or 1	`idx, valueType`

Table 6-11: Optimization related elements in OSnL.

The `var` element schema is shown below:

```
<xs:complexType name="OSnLNodeVar">
    <xs:complexContent>
        <xs:extension base="OSnLNode">
            <xs:sequence minOccurs="0">
                <xs:element ref="OSnLNode"/>
            </xs:sequence>
            <xs:attribute name="idx" type="xs:nonNegativeInteger" use="optional"/>
            <xs:attribute name="coef" type="xs:double" use="optional" default="1"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name="var" type="OSnLNodeVar" substitutionGroup="OSnLNode"/>
```

`<var>`[1] has two optional attributes, `idx` and `coef`. The `idx` attribute is nonnegative and if it's not there, an optional child can be used to evaluate to the variable index. This can be useful in, for example, constraint programming in which a variable's index can sometimes be an integer-valued variable or expression. The `coef` attribute is designed as a shorthand to avoid explicitly expressing a constant times a variable, which appears frequently in optimization. By default, `coef` is 1. For example, all the following are valid `variable` elements:

$x[0]$: `<var idx="0"/>`

$3x[10]$: `<var idx="10" coef="3"/>`

$5x[1+x[2]]$: `<var coef="5.0">   <plus><number value="1"/><var idx="2"/></plus>   </var>`

[1] We did not choose to use `<variable>` because variables appear too often in an optimization instance.

The `objective` element schema is shown below:

```
<xs:complexType name="OSnLNodeObjective">
    <xs:complexContent>
        <xs:extension base="OSnLNode">
            <xs:sequence minOccurs="0">
                <xs:element ref="OSnLNode"/>
            </xs:sequence>
            <xs:attribute name="idx" type="xs:int" use="optional" default="-1"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name="objective" type="OSnLNodeObjective" substitutionGroup="OSnLNode"/>
```

`<objective>` has one optional attribute `idx`; `idx` is negative and by default -1, which corresponds to the first objective. The element evaluates to the objective value corresponding to the index. Like the `var` element, an optional child can be used to evaluate to the objective index. The following is an example of an `objective` element:

```
<objective idx="-1"/>
```

The `Constraint` element schema is shown below:

```
<xs:complexType name="OSnLNodeConstraint">
    <xs:complexContent>
        <xs:extension base="OSnLNode">
            <xs:sequence minOccurs="0">
                <xs:element ref="OSnLNode"/>
            </xs:sequence>
            <xs:attribute name="idx" type="xs:int" use="required"/>
            <xs:attribute name="valueType" use="optional" default="value">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="value"/>
                        <xs:enumeration value="status"/>
                        <xs:enumeration value="surplus"/>
                        <xs:enumeration value="shortage"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name="constraint" type="OSnLNodeConstraint" substitutionGroup="OSnLNode"/>
```

The `<constraint>` element has two optional attributes `idx`, and `valueType`. The `idx` attribute is nonnegative. Like the `var` and `objective` elements, an optional child can be used to evaluate to the constraint index. The value of the `valueType` attribute can be either of "`value`" (default), "`status`" (whether the constraint is satisfied, a boolean), "`surplus`" ($(value - ub)^+$), or "`shortage`" ($(lb - value)^+$). The following are valid examples of a `constraint` element:

```
<constraint idx="2"/>
<constraint idx="3" valueType="status"/>
<constraint idx="0" valueType="surplus"/>
```

```
<constraint idx="11" valueType="shortage"/>
```

7. Logic and relational operators

In Table 6-12, we list the standard logic and relational operator elements.

Name	Child #	Note		Name	Child #	Note
Lt	2	$<$		and	2	&&
Leq	2	$\leq$		or	2	||
Gt	2	$>$		not	1	!
geq	2	$\geq$		xor	2	exclusive or
Eq	2	$=$		implies	2	$\rightarrow$: true if both children are true or false
neq	2	!=		if	3	If(a, b, c): if a is true, then b, else c

Table 6-12: Standard logic and relational operators in OSnL.

In Table 6-13, we list the extended logic and relational operator elements.

Name	Child #	Note	Example

forAll	1 or more	true if all the child nodes evaluate to true	`<forAll>`
			`<constraint idx="0" valueType="status"/>`
			`<constraint idx="1" valueType="status"/>`
			`<or>`
			`<constraint idx="3" valueType="status"/>`
			`<constraint idx="4" valueType="status"/>`
			`<or/>`
			`</forAll>`

exists	1 or more	true if any of the child nodes evaluate to true	`<exists>`
			`<gt><var idx="0"/><number value="1.2"/></gt>`
			`<geq><constraint idx="2"/><number value="1.2"/></geq>`
			`<implies>`
			`<constraint idx="3" valueType="status"/>`
			`<constraint idx="4" valueType="status"/>`
			`<implies/>`
			`</exists>`

logicCount	1 or more	number of child nodes that evaluate to true	`<logicCount>`
			`<neq><var idx="0"/><number value="3"/></neq>`
			`<and>`
			`<constraint idx="0" valueType="status"/>`
			`<constraint idx="1" valueType="status"/>`
			`<and/>`
			`</logicCount>`

allDiff	1 or more	true if all the child nodes evaluate to different values	`<allDiff>`
			`<constraint idx="0" valueType="value"/>`
			`<plus><var idx="0"/><var idx="1"/></plus>`
			`<objective idx="-1"/>`
			`</allDiff>`

atMost	2 or more	1st child evaluates to an integer `n`; true if at most n of the rest of the child nodes are true	`<atMost>`
			`<number value="2"/>`
			`<if>`
			`<eq><var idx="1"/><PI/></eq>`
			`<FALSE/>`
			`<TRUE/>`
			`</if>`
			`<constraint idx="1" valueType="status"/>`
			`<neq><var idx="0"/><number value="1.2"/></neq>`
			`</atMost>`

atLeast	2 or more	1st child evaluates to an integer `n`; true if at least	`<atLeast>`
			`<number value="1"/>`
			`<xor>`
			`<constraint idx="1" valueType="status"/>`

		n of the rest of the child nodes are true	<constraint idx="2" valueType="status"/> <xor/> <lt><var idx="0"/><number value="1.2"/></lt> </atLeast>
exactly	2 or more	1st child evaluates to an integer `n`; true if exactly n of the rest of the child nodes are true	<exactly> <number value="2"/> <not><constraint idx="1" valueType="status"/></not> <constraint idx="2" valueType="status"/> <leq><var idx="0"/><number value="1.2"/></leq> </exactly>
inSet	2 or more	true if 1st child's value is equal to one of the rest of the child nodes	<inSet> <number value="2"/> <constraint idx="0" valueType="value"/> <plus><var idx="0"/><var idx="1"/></plus> </inSet>
inRealSet	1	true if the child is a real number	<inRealSet> <var idx="2"/> </inRealSet>
inPositiveRealSet	1	true if the child is a positive real number	<inPositiveRealSet> <constraint idx="6" valueType="surplus"/> </ inPositiveRealSet>
inNonnegativeRealSet	1	true if the child is nonnegative real number	<inNonnegativeRealSet> <constraint idx="4" valueType="shortage"/> </ inNonnegativeRealSet >
inIntegerSet	1	true if the child is an integer number	<inIntegerSet> <divide><var idx="4"/><number value="2"/><divide/> </inIntegerSet>
inPositiveIntegerSet	1	true if the child is a positive integer number	<inPositiveIntegerSet> <minus><var idx="4"/><number value="2"/><minus/> </inPositiveIntegerSet>
inNonnegativeIntegerSet	1	true if the child is a nonnegative integer	<inNonnegativeIntegerSet> <ceiling><objective idx="-2"/><ceiling/> </inNonnegativeIntegerSet >

Table 6-13: Extended logic and relational operators in OSnL.

For instance, the first example (`forAll`) in the table:

```
<forAll>
    <constraint idx="0" valueType="status"/>
    <constraint idx="1" valueType="status"/>
    <or>
        <constraint idx="3" valueType="status"/>
        <constraint idx="4" valueType="status"/>
    <or/>
</forAll>
```

means that the `forAll` operator is true if constanint 0 and constraint 1 are both true, and one of the constraints, constraint 3 or constraint 4, is true.

All of the extended logic and relational operators are explained with an example in the above table. Most of these are used in combinatorial and discrete optimization such as constraint programming. Potentially more logic and relational operators will be added, especially the set-valued operators. For more details, refer to [47].

8. Special elements

In Table 6-14, we list the special elements.

Name	Children	Attributes
quadratic	1 or more `qpTerm` elements (only for quadratic programs)	none
qpTerm	0 or 1 child; the optional child evaluates to the coefficient value of the quadratic term which must evaluate to a constant term (only under the `<quadratic>` element)	`idxOne` (required) `idxTwo` (required) `coef` (optional, default = 1)
userF	0 or more children as the userF arguments	`name` (required)
arg	no children	`idx` (required, nonnegative)
userVar	no children	`name` (required)
sim	0 or more simInput elements, the last child is simOutput	`name` (required)
simInput	0 or 1 child; the optional evaluates to the simInput value	`simName` (optional) `inputName` (required)
simOutput	0 or 1 child; the optional evaluates to the simOutput value	`simName` (optional) `outputName` (optional)
xPath	0 or more `xPathIndex` elements	`uri` (optional, default= ".") `path` (required)
xPathIndex	0 or 1 child; the optional child evaluates to the index value	`indexName` (required) `indexValue` (optional)
complements	2 children	none
nodeRef	0 or 1 child; the optional child evaluates to the node property value	`nodeID` (optional, nonnegative) `propName` (required)
arcRef	0 or 1 child; the optional child evaluates to the arc property value	`arcID` (optional, nonnegative) `propName` (required)

Table 6-14: Special elements in OSnL.

Unlike most of the previous elements, many of these special elements have complex attributes and indefinite number of children. The special elements are described below. Several elements are explained using the Markowitz [76] optimization problem in (6-2) using a three stock instance where x_i represents the percentage of the portfolio invested in stock i. Assume the portfolio is re-balanced when returns and covariances are updated.

$$\begin{array}{lll}
\underset{x}{\text{minimize}} & 24x[msft]^2 + 75x[pg]^2 + 19x[ge]^2 - & \\
 & \quad 2*10x[msft]x[pg] + 2*25x[msft]x[ge] + 2*37x[pg]x[ge] & \\
\text{subject to} & & \\
 & .07x[msft] + .09x[pg] + .03x[ge] \geq= 0.5 \quad (1) & \\
 & x[msft] + x[pg] + x[ge] = 1 \quad (2) & (6\text{-}2)\\
 & if\ \ x[msft] > 0\ then\ x[msft] - .1\ else\ 0 \geq 0 \quad (3) & \\
 & if\ \ x[pg] > 0\ then\ x[pg] - .1\ else\ 0 \geq 0 \quad (4) & \\
 & if\ \ x[ge] > 0\ then\ x[ge] - .1\ else\ 0 \geq 0 \quad (5) &
\end{array}$$

<u>quadratic, qpTerm</u>

The `<quadratic>` and `<qpTerm>` schemas are shown below:

```
<xs:complexType name="OSnLNodeQuadratic">
    <xs:complexContent>
        <xs:extension base="OSnLNode">
            <xs:sequence maxOccurs="unbounded">
                <xs:element ref="qpTerm"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name="quadratic" type="OSnLNodeQuadratic" substitutionGroup="OSnLNode"/>

<xs:complexType name="OSnLNodeQpTerm">
    <xs:complexContent>
        <xs:extension base="OSnLNode">
            <xs:sequence minOccurs="0">
                <xs:element ref="OSnLNode"/>
            </xs:sequence>
            <xs:attribute name="idxOne" type="xs:nonNegativeInteger" use="required"/>
            <xs:attribute name="idxTwo" type="xs:nonNegativeInteger" use="required"/>
            <xs:attribute name="coef" type="xs:double" use="optional" default="1"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name="qpTerm" type="OSnLNodeQpTerm" substitutionGroup="OSnLNode"/>
```

Although the instance of any quadratic program is easily represented as a general nonlinear program using OSnLNode elements, a more compact representation is provided for quadratic terms. A `<qpTerm>` element is used to represent each quadratic term. The `<quadratic>` element sums up all its `<qpTerm>` child elements. The `<qpTerm>` element has two required integer attributes (`idxOne`, `indxTwo`) that specify the two variable indices in the quadratic term. The coefficient of the quadratic term is specified using either a third optional double attribute `coef` or by a single child element. One advantage of using the `<qpTerm>` elements is that quadratic programming solvers typically take coefficient lists rather than nonlinear expressions. An added advantage is that if an analyzer applied to the problem instance discovers that the only nonlinear terms are `<quadratic>` and `<qpTerm>` terms, it can classify the problem as a quadratic program. Suppose in the instance representation of (6-2), the variable index for $x[msft]$ is 0, for $x[pg]$ is 1, and for $x[ge]$ is 2. The objective function is represented as

```
<quadratic>
    <qpTerm idxOne="0" idxTwo="0" coef="24"/>
    <qpTerm idxOne="1" idxTwo="1" coef="75"/>
    <qpTerm idxOne="2" idxTwo="2" coef="19"/>
    <qpTerm idxOne="0" idxTwo="1" coef="20"/>
    <qpTerm idxOne="0" idxTwo="2" coef="50"/>
    <qpTerm idxOne="1" idxTwo="2" coef="74"/>
</quadratic>
```

userF, arg

Often a problem instance has an expression that is repeated numerous times. As in programming, where a method (subroutine) simplifies repeated logic, the `<userF>` element is used to simplify instance representation by calling a pre-defined user function. Consider the constraint set (3)-(5) of (6-2). These constraints require that if a nonzero investment is made in stock i, then at least 10% of the portfolio must be invested in stock i. Rather than repeat the same logic for each stock, it is much cleaner to first write the logic only once in a user defined function: $minInv(\arg_0) = \mathit{if}\ \arg_0 > 0\ \mathit{then}\ \arg_0 - .1\ \mathit{else}\ 0$, where $\arg_0$ is to be passed a value of x_i. User functions are defined in OSiL though the `<userFunction>` element (discussed in the OSiL section §6.2). The representation for the `minInv` function looks like:

```
<userFunction name="minInv" numArg="1">
    <if>
        <gt>
            <arg idx="0"/>
            <number value="0"/>
        </gt>
        <minus>
            <arg idx="0"/>
            <number value="0.1"/>
        </minus>
        <number value="0"/>
    </if>
</userFunction>
```

The `<userF>` and `<arg>` schemas are shown below:

```
<xs:complexType name="OSnLNodeUserF">
    <xs:complexContent>
        <xs:extension base="OSnLNode">
            <xs:sequence minOccurs="0" maxOccurs="unbounded">
                <xs:element ref="OSnLNode"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:IDREF" use="required"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name="userF" type="OSnLNodeUserF" substitutionGroup="OSnLNode"/>

<xs:complexType name="OSnLNodeArg">
    <xs:complexContent>
        <xs:extension base="OSnLNode">
            <xs:attribute name="idx" type="xs:nonNegativeInteger" use="required"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name="arg" type="OSnLNodeArg" substitutionGroup="OSnLNode"/>
```

As mentioned in §6.2.2, the definition of the user function should be independent of the optimization problem instance represented by the OSiL instance; thus it is required that the user function definition use the `<arg>` elements instead of the math program variables `<var>` elements. The `<arg>` element has one required index attribute (`idx`) which is a nonnegative number. The `numArg` attribute of `userFunction` is used to check that all the argument indexes are ≥ 0 and $\leq numArg - 1$.

Now with the `minInv` user function definition, we can write constraint set (3)-(5) of (6-2) using the `<userF>` element as:

```
<nl idx="2">
    <userF name="minInv">
        <var idx="0"/>
    </userF>
</nl>
<nl idx="3">
    <userF name="minInv">
        <var idx="1"/>
    </userF>
</nl>
<nl idx="4">
    <userF name="minInv">
        <var idx="2"/>
    </userF>
</nl>
```

In this example, the `<userF>` element's required attribute `name` is `minInv`. `<userF>` can take 0 or more children as function arguments to pass. Here, we only have one argument which is `<var idx="…"/>`.

userVar

Sometimes a problem has some "new" variables defined over other math program decision variables and these user-defined variables are used repeatedly in the objective or constraint functions. The `<userVar>` element is used to simplify instance representation by calling a pre-defined user variable. Notice user variables are not math program variables and thus not counted in the total number of math program variables. `<userVar>` is just another special nonlinear node and is very similar to the use of `<userF>`, only that `<userVar>` no longer carries any child elements as its arguments, as all its arguments are from the already defined math program variables. Consider constraint (2) of (6-2), which is the unity constraint that requires the percentages of stock investments add up to one. We can define a new variable called `total` such that `total` $= x_0 + x_1 + x_2$. User variables are defined in OSiL though the `<userVariable>` element (discussed in the OSiL section §6.2). The representation for the `total` variable looks like:

```
<userVariable name="total">
    <sum>
        <var idx="0"/>
        <var idx="1"/>
        <var idx="2"/>
    </sum>
</userVariable>
```

The `<userVar>` schema is shown below:

```
<xs:complexType name="OSnLNodeUserVar">
    <xs:complexContent>
        <xs:extension base="OSnLNode">
            <xs:attribute name="name" type="xs:IDREF" use="required"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name="userVar" type="OSnLNodeUserVar" substitutionGroup="OSnLNode"/>
```

As mentioned in §6.2.2, the definition of the user variable is entirely dependent on the variables already defined in the mathematical program represented by the OSiL instance, that is, it becomes meaningless outside of the OSiL context. In practice, a `userVariable` definition can be a more complex nonlinear expression than just the variable summation. Now with the `total` user variable definition, we can write constraint (2) of (6-2) using the `<userVar>` element simply as:

```
<nl idx="1">
    <userVar name="total"/>
</nl>
```

`<userVar>` is a terminal node and does not take any children. In this example, the `<userVar>` element's required attribute `name` is `total`. Of course in more complex examples, `<userVar>` is used in more than one constraint or objective function and can be inside a bigger expression.

sim, simInput, simOutput

In some optimization problems there may not be a closed form expression for all functions – they may be black boxes. This case is handled by the `<sim>` element. As explained in §2.8, a simulation is similar to a user function, only that there is no longer a closed-form that can be expressed; three things have to be specified for the simulation: input, output, and the simulation's address. The simulation definition, like the user function definition is specified in OSiL. This was discussed in the OSiL section (§6.2).

Suppose the above `minInv` user function is now calculated by a simulation called `stockSimulation` shown in Figure 6-20.

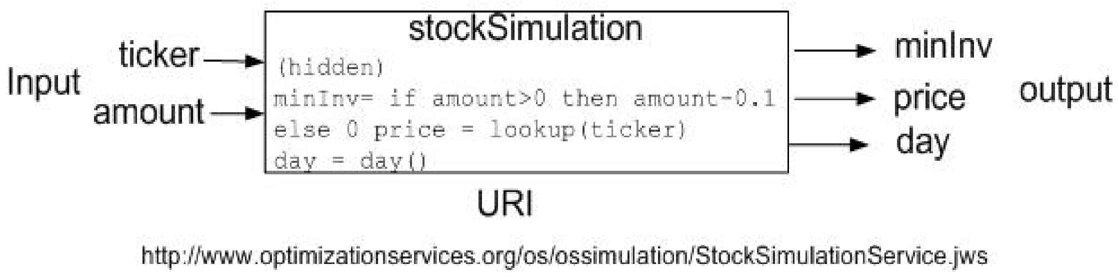

stockSimulation
Input
ticker
amount
(hidden)
minInv= if amount>0 then amount-0.1
else 0 price = lookup(ticker)
day = day()
minInv
price
day
output
URI
http://www.optimizationservices.org/os/ossimulation/StockSimulationService.jws

Figure 6-20: stockSimulation with two inputs (ticker, amount), three outputs (minInv, price, day) and an address (http://www.optimizationservices.org/os/ossimulation/StockSimulationService.jws).

There are two inputs of the simulation service: ticker for the stock symbol and amount for the percentage of the stock in the portfolio. Notice the stockSimulation engine provides more “services” than just calculating the minimum investment. It can look up a stock price according to a stock ticker (a string). It also outputs the day of the week (no input needed for this function). So it has three outputs: `mininv`, `price` and `day`.

The `stockSimulation` element is then be represented using the `<simulation>` element of OSiL (§6.2.2) as

```
<simulation name="stockSimulation">
    <uri value="http://www.optimizationservices.org/os/ossimulation/StockSimulationService.jws"/>
    <OSsL>
        <input>
            <el name="ticker"/>
            <el name="amount"/>
        </input>
        <output>
            <el name="minInv"/>
            <el name="price"/>
            <el name="day"/>
        </output>
    </OSsL>
</simulation>
```

The format of the `OSsL` child is described in detail in the OSsL section (§6.7). It contains information about inputs and outputs. Note that simulations generally refer to inputs and outputs by name rather than by order.

The `<sim>`, `<simInput>`, and `<simOutput>` schemas are shown below:

```
<xs:complexType name="OSnLNodeSim">
    <xs:complexContent>
        <xs:extension base="OSnLNode">
            <xs:sequence>
                <xs:element ref="simInput" minOccurs="0" maxOccurs="unbounded"/>
                <xs:element ref="simOutput"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:IDREF" use="required"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name="sim" type="OSnLNodeSim" substitutionGroup="OSnLNode"/>

<xs:complexType name="OSnLNodeSimInput">
    <xs:complexContent>
        <xs:extension base="OSnLNode">
            <xs:sequence minOccurs="0">
                <xs:element ref="OSnLNode"/>
            </xs:sequence>
            <xs:attribute name="simName" type="xs:IDREF" use="optional"/>
            <xs:attribute name="inputName" type="xs:IDREF" use="required"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

```
<xs:element name="simInput" type="OSnLNodeSimInput" substitutionGroup="OSnLNode"/>

<xs:complexType name="OSnLNodeSimOutput">
   <xs:complexContent>
      <xs:extension base="OSnLNode">
         <xs:sequence minOccurs="0">
            <xs:element ref="OSnLNode"/>
         </xs:sequence>
         <xs:attribute name="simName" type="xs:IDREF" use="optional"/>
         <xs:attribute name="outputName" type="xs:string" use="optional"/>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>
    <xs:element name="simOutput" type="OSnLNodeSimOutput" substitutionGroup="OSnLNode"/>
```

Now with the `stockSimulation` definition, we can write constraint set (3)-(5) of (6-2) using the `<sim>` element as:

```
<nl idx="2">
   <sim name="stockSimulation">
      <simInput inputName="amount">
         <var idx="0"/>
      </simInput>
      <simOutput outputName="minInv"/>
   </sim>
</nl>
<nl idx="3">
   <sim name="stockSimulation">
      <simInput inputName="amount">
         <var idx="1"/>
      </simInput>
      <simOutput outputName="minInv"/>
   </sim>
</nl>
<nl idx="4">
   <sim name="stockSimulation">
      <simInput inputName="amount">
         <var idx="2"/>
      </simInput>
      <simOutput outputName="minInv"/>
   </sim>
</nl>
```

In this example, the `<sim>` element's required attribute `name` is `stockSimulation`. `<sim>` can take 0 or more `<simInput>` child elements, followed by one required `<simOutput>` child element because we must have *one* output value to further calculate an objective or constraint function value. So <sim> always has at least one child. Here, we only have one `<simInput>` element which is `<var idx="…"/>`. Each `<simInput>` element has a required `inputName` attribute, which refers to an input defined in the corresponding `<simulation>` definition. Each `<simInput>` element also has an optional `simName` attribute. If the attribute is not there, as in the above example, it defaults to the name of the parent <sim> element. So in the above example we can also write `<simInput simName="stockSimulation" inputName="amount">` with an explicit `simName` attribute. The same rule applies to the `<simOutput>` element. `<simInput>` can have an optional child that evaluates to an input value and `<simOutput>` can have an optional child that evaluates to an output value. If the child is not there, `<simInput>` or `<simOutput>` *takes* the value from the OSsL element (`<el>`) with the same input or output name. In our example `<simOutput>` is a *taker*. If there is a child of `<simInput>`, it *supplies* the value to the OSsL element with the same input name. In our example `<simInput>` is a *supplier*. If there is a child of `<simOutput>`, it *constructs* a new value from the OSsL elements.

Of course the example is somewhat simplified. The child element of `<simInput>` can be more complex than just one single `<var>` node. In reality, the child can be a more complex expression tree with many nodes. Also `<simOutput>` may not just directly *take* the `minInv` output value. For example we can say if the `day` output from stockSimulation is 1 (Monday), we want to add a fixed amount (say 0.05) to the minimum investment requirement for the stock `ge`. This corresponds to constraint (4). So constraint (4) now looks like:

```
<nl idx="4">
    <sim name="stockSimulation">
        <simInput inputName="amount">
            <var idx="2"/>
        </simInput>
        <simOutput>
            <if>
                <eq>
                    <simOutput outputName="day"/>
                    <number value="1"/>
                </eq>
                <plus>
                    <simOutput outputName="minInv"/>
                    <number value="0.05"/>
                </plus>
                <simOutput outputName="minInv"/>
            </if>
        </simOutput>
    </sim>
</nl>
```

Here `<simOutput>` *constructs a new output*. That is why, unlike `<simInput>` whose `inputName` attribute is required, the `outputName` attribute of `<simOutput>` is optional. Such a construction can be commonly used in optimization via *stochastic* simulation, where the simulation usually outputs a variance value as well as a mean value, and the optimization uses some combination of both the mean and the variance.

xPath, xPathIndex

In practice, problem parameters are often dynamic over time. If the value of a parameter changes, a new instance must be created using the modeling language. These problems are eliminated using `<xPath>` and `<xPathIndex>` elements. By allowing xPath nodes in an OSiL instance representation it is possible to reference data in an external XML data file. Thus a modeler, in a distributed environment, can generate a model, send it to the server, and the server can operate with current data without the necessity of the modeling language creating a new instance file.

The `<xPath>` and `<xPathIndex>` schemas are shown below:

```
<xs:complexType name="OSnLNodeXPath">
    <xs:complexContent>
        <xs:extension base="OSnLNode">
            <xs:sequence>
                <xs:element ref="xPathIndex" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="uri" type="xs:anyURI" use="optional" default="."/>
            <xs:attribute name="path" type="xs:string" use="required"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name="xPath" type="OSnLNodeXPath" substitutionGroup="OSnLNode"/>'

<xs:complexType name="OSnLNodeXPathIndex">
    <xs:complexContent>
        <xs:extension base="OSnLNode">
            <xs:sequence minOccurs="0">
                <xs:element ref="OSnLNode"/>
            </xs:sequence>
            <xs:attribute name="indexName" type="xs:string" use="required"/>
            <xs:attribute name="indexValue" type="xs:string" use="optional"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name="xPathIndex" type="OSnLNodeXPathIndex" substitutionGroup="OSnLNode"/>
```

The `<xPath>` element has an optional `uri` attribute which specifies where the XML data file is. It is by default "." which is the current OSiL instance file, that is, the data are included in the `<xmlData>` element as explained in the OSiL section (§6.2). The `<xPath>` element also has a required `path` attribute which is a string of XPath syntax (§4.4), used to locate values *within* the XML data. An XPath string may contain one or more "XPath variables" indicated by an initial "$" sign.

The `<xPath>` element can have zero or more `<xPathIndex>` child elements. Each `<xPathIndex>` has a required `indexName` attribute and an optional `indexValue` attribute. An optional child of `xPathIndex` can be used that evaluates to the index value if `indexValue` is missing. The `indexName` attribute is used to match the xPathIndex with a `$variable` in the `path` attribute of `<xPath>` and `indexValue` is used to supply the value for the variable. So the number of `xPathIndex` child elements has to be exactly the same as the number of variables in the `path` attribute of `<xPath>`.

As an example, consider the Markowitz optimization problem (6-2). Assume the data on returns and covariances are located within the file stockdata.xml in same directory as the OSiL instance (`uri` = "./stockdata.xml"). The xml data file is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<stocks>
    <stock name="msft" idx="0" ret=".07" minInv=".1">
        <cov name="msft" idx="0" val="24"/>
        <cov name="pg" idx="1" val="-10"/>
        <cov name="ge" idx="2" val="25"/>
    </stock>
    <stock name="pg" idx="1" ret=".09" minInv=".1">
        <cov name="msft" idx="0" val="-10"/>
        <cov name="pg" idx="1" val="75"/>
        <cov name="ge" idx="2" val="37"/>
    </stock>
    <stock name="ge" idx="2" ret=".03" minInv=".1">
        <cov name="msft" idx="0" val="25"/>
        <cov name="pg" idx="1" val="37"/>
        <cov name="ge" idx="2" val="19"/>
    </stock>
</stocks>
```

There are three stocks each corresponding to a <stock> element. Each <stock> contains information about its ticker (`name`), index (`idx`), return (`ret`), minimum investment requirement (`minInv`) and covariances with all the stocks (`<cov>`). The data within the XML file at the indicated uri are located using the `path` attribute of XPath syntax. So if we use the `<xPath>` elements to locate the coefficients (stock return values) for each variable in constraint (1) of (6-2) instead of directly specifying the values inside the instance, we come up with the following representation for constraint (1):

```
<nl idx="0">
    <sum>
        <times>
            <var idx="0"/>
            <xPath uri="./stockdata.xml" path="/stocks/stock[@name='msft'/@return"/>
        </times>
        <times>
            <var idx="1"/>
            <xPath uri="./stockdata.xml" path="/stocks/stock[@name='pg'/@return"/>
        </times>
        <times>
            <var idx="2"/>
            <xPath uri="./stockdata.xml" path="/stocks/stock[@name='ge'/@return"/>
        </times>
    </sum>
</nl>
```

Alternatively we can use `xPathIndex` if the name of the stocks are variables:

```
<nl idx="0">
    <sum>
        <times>
            <var idx="0"/>
            <xPath uri="./stockdata.xml" path="/stocks/stock[@name=$stockName/@return">
                <xPathIndex indexName="stockName" indexValue="msft"/>
            </xPath>
        </times>
        <times>
            <var idx="1"/>
            <xPath uri="./stockdata.xml" path="/stocks/stock[@name=$stockName/@return">
                <xPathIndex indexName="stockName" indexValue="pg"/>
            </xPath>
        </times>
        <times>
            <var idx="2"/>
            <xPath uri="./stockdata.xml" path="/stocks/stock[@name=$stockName/@return">
                <xPathIndex indexName="stockName" indexValue="ge"/>
            </xPath>
        </times>
    </sum>
</nl>
```

The above two examples are equivalent, but by using the variable `$stockName` and `xPathIndex` to supply the values ("msft", "pg", "ge"), the 3 `xPath` elements become the same and we can potentially simplify the syntax by designing a user function using an argument to pass the stock names.

The library that reads the OSiL instance can use the `xPath` element to locate the stock return data before sending the instance to the solver. It is also possible to carry the XML data with the instance file. If this is desired, it is done by putting the data in the `xmlData` element.

complements

The `<complements>` element allows complementarity problems to be constructed for solvers to search for a feasible solution. The `<complements>` element is explained in detail in Appendix A.

nodeRef, arcRef

As the first release of OSiL does not include network and graph extension, the `<nodeRef>` and `<arcRef>` elements, which are used to reference node and arc property values in a network, are reserved for future use. See Appendix A for more descriptions.

6.4 Optimization Services result Language (OSrL)

The OSrL schema is located at http://www.optimizationservices.org/schemas/OSrL.xsd. OSrL is a general optimization result format specification, mainly outputted by solvers.

OSrL is among the instances whose contents need to be understood by humans most frequently. The optional OStL transformation style sheet (§6.8) allows OSrL to be presented in a clear and nice form. Of course as OSrL is well structured, it can also be analyzed and reused in the middle of a large computation where sub-problems are constantly solved and results are resubmitted for subsequent calculations.

OSrL, an output format, can be thought of as the counterpart to the input format OSiL. The structure and contents of OSrL is based on and driven by the OSiL design. But compared with OSiL, OSrL is more straightforward. The separation of OSrL from OSiL helps in reducing network traffics and enhancing flexibility.

Figure 6-21 shows the root element `<OSrL>` of the OSrL schema.

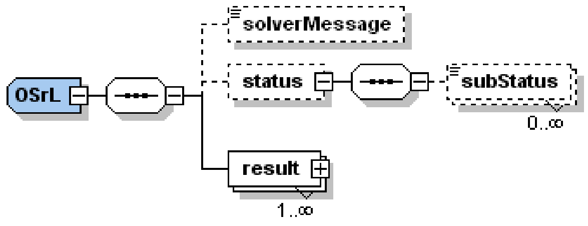


Figure 6-21: OSrL Schema at the root level `<OSrL>`.

The `<OSrL>` element has an optional `<solverMessage>` child, an optional `<status>` child, and 1 or more `<result>` children for all the solutions. In `<solverMessage>`, a solver can put a general message on the whole optimization process (*not* on each solution). The `<status>` element has a required `type` attribute used to indicate various predefined *standard* status types on the general optimization process, e.g. "success", warning, and "error." The `<status>` element can have 0 or more `<subStatus>` elements. As sub-statuses are not standardized, each `<subStatus>` element has a required `name` attribute for sub-status name and a `value` for sub-status value, and inside the `<subStatus>` element, a description can be put. The following is an example of the `<status>` element:

```
<status type="error">
    <subStatus name="inputError" value="array out of bound">
            variableNumber inconsistent with the number of var elements
        </subStatus>
    <subStatus name="internalError" value="out of memory">
            data too large to handle
        </subStatus>
</status>
```

There has to be at least one `<result>` child element under the root `<OSrL>`. Each `<result>` element corresponds to one optimization solution. In most cases, there is only one `<result>` child in the root `<OSrL>` element. But in situations such as nonlinear optimization where several locally optimal points are found, or multi-objective optimization where a set of pareto-optimal solutions are returned, we may have more than 1 `<result>` element. The `<result>` element is shown in Figure 6-22.

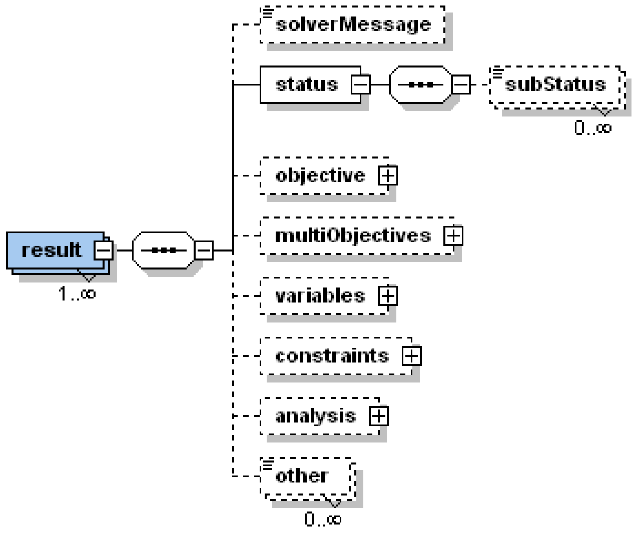


Figure 6-22: `<result>` element in OSrL.

The `<message>` and `<status>` are similar to the `<solverMessage>` and `<status>` elements directly defined under the root `<OSrL>` element, only that now they are message and status on each result. Status types are standardized and can be “optimal”, “infeasible”, “unbounded”, “error” etc. Sub-statuses are again not standardized.

Objective-related results should be put in the <objective> element. The `<multiObjectives>`, `<variables>`, `<constraints>` elements are provided for similar purposes. These are explained below.

When solving an optimization problem, a solver may gather additional analysis of the problem. The `<analysis>` element is provided for this purpose. The `<analysis>` element is of the Optimization Services analysis Language (OSaL) format explained in §6.6. Results that do not belong to the above categories should go into the `<other>` elements. The `<other>` element has a required `resultName` attribute and a required `value` attribute. A description can be put in the `<otherResult>` element.

1. The `<objective>` element is shown in Figure 6-23.

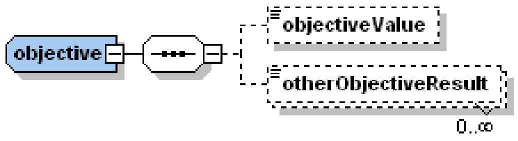


Figure 6-23: `<objective>` element in OSrL.

The `<objective>` element currently contains one predefined standard element `<objectiveValue>`. More standard objective related elements may be added in the future. Like most of the elements in OSrL, the `<objectiveValue>` element is optional, as a solver may not be able to find any solution. Of course, even if there is a solution, the objective value can be constructed from the variable solution. So it is not absolutely necessary for solvers to explicit output the value. But if a solver does output the value, it has to be put inside the `<objectiveValue>` element. A sequence of 0 or more `<otherObjective>` elements follows `<objectiveValue>`. If a solver provides objective results other than the objective value, they should go inside these elements. As these non-standard results vary between solvers, each `<otherObjectiveResult>` element has a required `resultName` and `value` attribute. A description can go inside the `<otherObjectiveResult>` element for further clarification.

2. The `<multiObjectives>` element is shown in Figure 6-24.

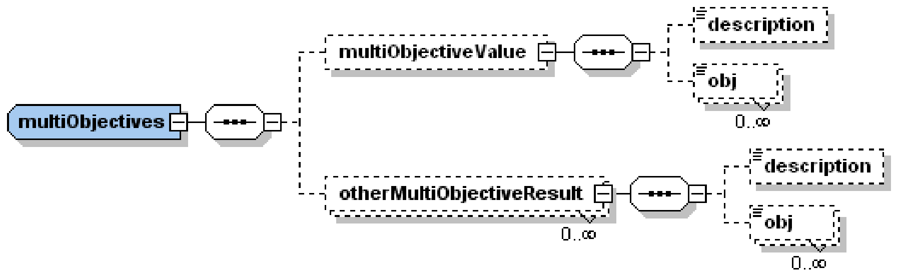


Figure 6-24: `<multiObjectives>` element in OSrL.

The `<multiObjectives>` element currently contains one predefined standard element `<multiObjectiveValue>`. More standard multi-objective related elements may be added in the future. Like the `<objectiveValue>` element, `<multiObjectiveValue>` is optional. The `<multiObjectiveValue>` element has an optional `value` attribute if there is a multi-objective function value. It can have an optional `<description>` element for further elaboration. There are 0 or more `<obj>` elements after `<description>`, each corresponding to one objective component of the multi-objective function. Each `<obj>` has a required `value` attribute to specify the objective value. An `<obj>` element also has an optional `idx` and `objName` attribute. The `idx` attribute is optional only if the `<obj>` elements are listed in the same order as those in the OSiL input instance. Again objectives are indexed from -1 downward; thus the `idx` attribute is a negative number. If a solver provides other multi-objective results, they should go in the `<otherMultiObjectiveResult>`

elements. As these non-standard results vary between solvers, each `<otherMultiObjectiveResult>` element has a required `resultName` attribute and an optional `<description>` child for further elaboration. `<otherMultiObjectiveResult>` also has an optional `value` for the specified result on the entire multi-objective. The individual result for each objective component should go inside the `<obj>` children, each having a required `value` attribute. `<obj>` also has an optional `idx` and `objName` attribute, just like the `<obj>` elements in `<multiObjectiveValue>`.

3. The `<variables>` element is shown in Figure 6-25.

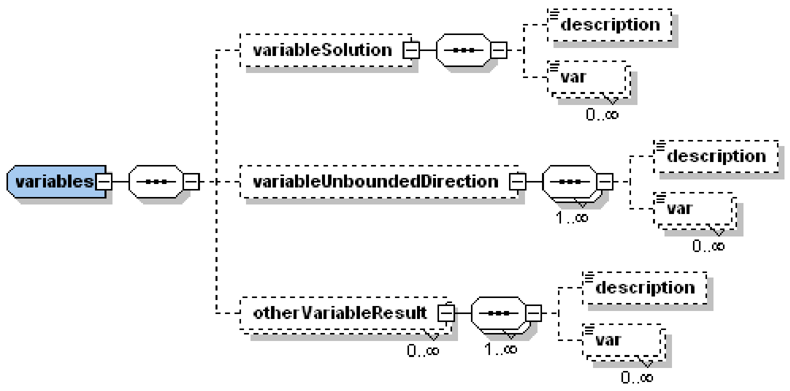


Figure 6-25: `<variables>` element in OSrL.

The `<variables>` element currently contains two predefined standard elements `<variableSolution>` and `<variableUnboundedDirection>`. Each has a `<description>` element for further elaboration, following which are 0 or more `<var>` elements to specify variable solutions, and/or unbounded directions. More standard variable related elements may be added in the future. The corresponding variable values should go inside the required `value` attributes of `<var>`. A `<var>` element also has an optional `idx` and `varName` attribute. The `idx` attribute is optional only if the `<var>` elements are listed in the same order as those in the OSiL input instance. Again variables are indexed from 0 on; thus the `idx` attribute is a nonnegative number. If a solver provides other variable results, they should go in the `<otherVariableResult>` elements. As these non-standard results vary between solvers, each `<otherVariableResult>` element has a required `resultName` attribute and an optional `<description>` child for further elaboration. The individual result for each variable should go inside the <var> children, each having a required `value` attribute. The `<var>` element also has an optional `idx` and `varName` attribute, just like the <var> elements in the standard variable result elements.

4. The `<constraints>` element is shown in Figure 6-26.

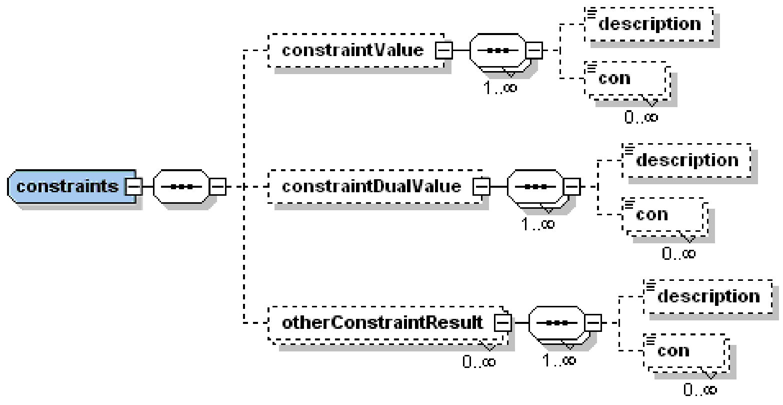


Figure 6-26: `<constraints>` element in OSrL.

The `<constraints>` element currently contains two predefined standard elements `<constraintValue>` and `<constraintDualValue>`. Each has a `<description>` element for further elaboration, following which are 0 or more `<con>` elements to specify constraint values or dual values. More standard constraint related elements may be added in the future. The individual constraint values should go inside the required `value` attributes of `<con>`. A `<con>` element also has an optional `idx` and `conName` attribute. `idx` is optional only if the `<con>` elements are listed in the same order as those in the OSiL input instance. Again constraints are indexed from 0 on; thus the `idx` attribute is a nonnegative number. If a solver provides other constraint results, they should go in the `<otherConstraintResult>` elements. As these non-standard results vary between solvers, each `<otherConstraintResult>` element has a required `resultName` attribute and an optional `<description>` child for further elaboration. The individual result for each constraint should go inside the <con> children, each having a required `value` attribute. The `<con>` element also has an optional `idx` and `conName` attribute, just like the <con> elements in the standard constraint result elements.

6.5 Optimization Services option Language (OSoL)

The OSoL schema is located at http://www.optimizationservices.org/schemas/OSoL.xsd. OSoL is a general optimization option format specification mainly for solver algorithm directives. OSoL is probably the instance least able to be standardized as different solvers have different options and even if the optional names are the same, they are used differently. An OSoL instance is usually sent to a solver along with an OSiL instance. If the OSoL instance is missing, default options are assumed by the solvers. OSoL can potentially be used to discover a solver in the OS registry if a user requires a solver to support a specified option.

Figure 6-27 shows the root element `<OSoL>` of the OSoL schema.

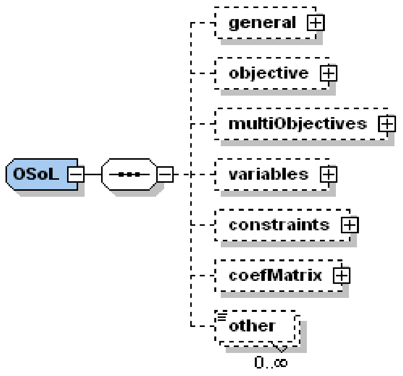


Figure 6-27: OSoL Schema at the root level `<OSoL>`.

Options can be specified in an appropriate child of the 7 types of children of `<OSoL>`: `<general>`, `<objective>`, `<multiObjectives>`, `<variables>`, `<constraints>`, `<coefMatrix>`, and 0 or more `<other>` elements.

1. The `<general>` element is shown in Figure 6-28.

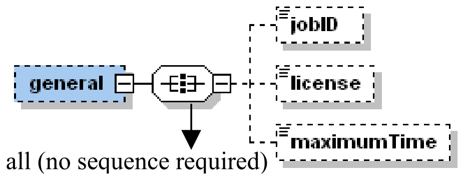


Figure 6-28: `<standard>` element in OSoL.

The `<general>` element currently has three predefined options, all optional. No sequence of the child options is required. The `<general>` element has an optional `serviceName` and an optional service`Address` attribute. The service name and address should be the same as those published in the OS registry. The `<jobID>` element contains a job ID that has previously been assigned by the service. Solver client can, for example, use the job ID to retrieve intermediate (if supported by the solver) or final optimization results. The `<license>` element contains a license key that may be required by commercial services. The `<maximumTime>` element is the maximum amount of time in minutes for an optimization job.

2. The `<objective>` element is shown in Figure 6-29.

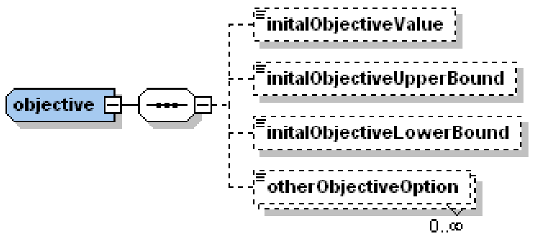


Figure 6-29: `<objective>` element in OSoL.

The `<objective>` element currently contains three predefined standard options: `<initialObjectiveValue>`, `<initialObjectiveUpperBound>`, and `<initialObjectiveLowerBound>`. Option values should be specified in the required `value` attribute of each element. An option description can be put in the elements. More standard objective options may be added in the future. Nonstandard options can be specified in the subsequent of 0 ore more `<otherObjectiveOption>` element. Each `<otherObjectiveOption>` has a required `optionName` attribute besides the `value` attribute.

3. The `<multiObjectives>` element is shown in Figure 6-30.

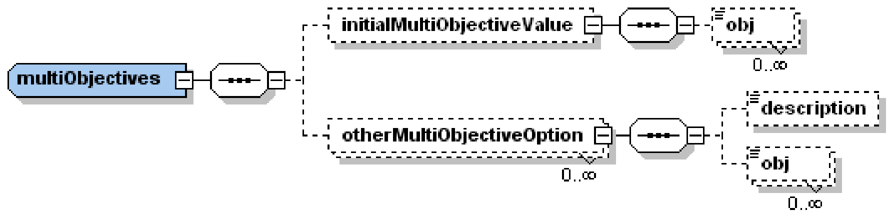


Figure 6-30: `<multiObjectives>` element in OSoL.

The `<multiObjecitves>` element currently contains one predefined standard option `<initialMultiObjectiveValue>`. The `<initialMultiObjectiveValue>` element has an optional `value` attribute if there is an initial value for the entire multi-objective function. `<initialMultiObjectiveValue>` can have 0 or more `<obj>` elements, each corresponding to one objective component of the multi-objectives. The individual initial function value for each objective component is specified in the required `value` attribute of `<obj>`. An `<obj>` element also has an optional `idx` and `objName` attribute. `idx` is optional only if the `<obj>` elements are listed in the same order as those in the OSiL input instance. If there are other multi-objective options, they should go in the `<otherMultiobjectiveOption>` elements. As these non-standard results vary between solvers, each `<otherMultiobjectiveOption>` element has a required `optionName` attribute and an optional `<description>` child for further elaboration. `<otherMultiobjectiveOption>` also has an optional `value` for the specified option on the entire multi-objective function. The individual option for each objective component should go inside the <obj> children, each having a required `value` attribute. `<obj>` also has an optional `idx` and `objName` attribute, just like the <obj> elements in `<initialMultiObjectiveValue>`.

4. The `<variables>` element is shown in Figure 6-31.

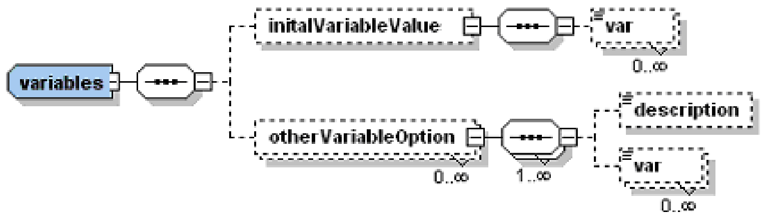


Figure 6-31: `<variables>` element in OSoL.

The `<variables>` element currently contains one predefined standard option `<initialVariableValues>`. The `<initialVariableValue>` element can have 0 or more `<var>` elements, each having a required `value` attribute for an initial variable value. A `<var>` element also has an optional `idx` and `varName` attribute. `idx` is optional only if the `<var>` elements are listed in the same order as those in the OSiL input instance. If there are other variables options, they should go in the `<otherVariableOption>` elements. As these non-standard results vary between solvers, each `<otherVariableOption>` element has a required `optionName` attribute and an optional `<description>` child for further elaboration. The individual option for each variable should go inside the `<var>` children, each having a required `value` attribute. `<var>` also has an optional `idx` and `varName` attribute, just like the <var> elements in `<initialVariableValues>`.

5. The `<constraints>` element is shown in Figure 6-32.

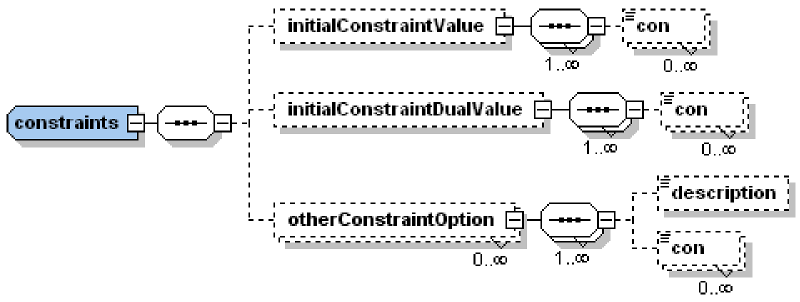


Figure 6-32: `<constraints>` element in OSoL.

The `<constraints>` element currently contains two predefined standard options `<initialConstraintValue>` and `<initialConstraintDualValue>`. Each has 0 or more `<con>` elements to specify initial constraint values or dual values. More standard constraint related options may be added in the future. The individual constraint options should go inside the required `value` attributes of `<con>`. A `<con>` element also has an optional `idx` and `conName` attribute. The `idx` attribute is optional only if the `<con>` elements are listed in the same order as those in the OSiL input instance. If there are other constraint options, they should go in the `<otherConstraintOption>` elements. As these non-standard options vary between solvers, each `<otherConstraintOption>` element has a required `optionName` attribute and an optional `<description>` child for further elaboration. The individual option for each constraint should go inside the <con> children, each having a required `value` attribute. The `<con>` element also has an optional `idx` and `conName` attribute, just like the <con> elements in the standard constraint option elements.

6. The `<coefMatrix>` element is reserved for future use. Currently there are no coefficient matrix related options.

7. The `<other>` elements are for options that do not belong to the above categories. The `<other>` element has a required `optionName` attribute and a required `value` attribute. A description can be put in the `<other>` element.

6.6 Optimization Services analysis Language (OSaL)

The OSaL schema is located at http://www.optimizationservices.org/schemas/OSaL.xsd. OSaL is a general optimization analysis format specification, mainly outputted by analyzers. The role of analyzer and its output standardization is discussed in detail in §2.6.

As discussed in §6.4, when solving an optimization problem, a solver may gather additional analysis on the problem. Thus an `<analysis>` element can be embedded in the Optimization Services result Language (OSrL). The `<analysis>` element is exactly of the OSaL format. On the other hand, if an optimization model is easy enough, it can potentially be solved by an analyzer without sending to a solver. In this situation, the analyzer should return an OSrL instance with an embedded `<analysis>` element of OSaL.

OSaL, an analysis output format, can be thought of as another counterpart (besides OSrL) to the input format OSiL. The structure and contents of OSaL is based on and driven by the

OSiL design. But compared with OSiL, OSaL is more straightforward. Figure 6-33 shows the root element `<OSaL>` of the OSaL schema.

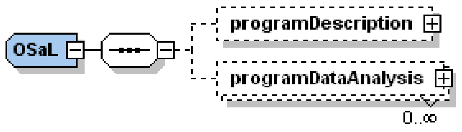


Figure 6-33: OSaL Schema at the root level `<OSaL>`.

The `<OSaL>` element has two children, `<programDescription>` and `<programDataAnalysis>`. The `<programDescription>` element conveys the basic analyses of an optimization instance. All its children are shown in Figure 6-34 and are self-explanatory. The last element `<specific>` is of `mapType` which is briefly explained in the OSgL section (§6.1). Basically it is an array of `<el>` elements each with a `name` and a `value` attribute. The `<specific>` element is for analyzers to output nonstandard analyses and is used at many places in the `<programDataAnalysis>` element too.

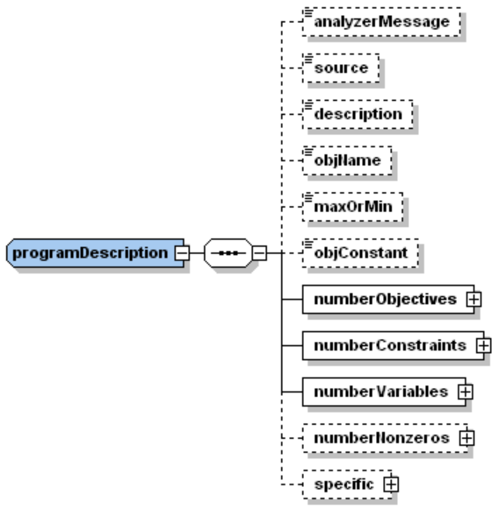


Figure 6-34: `<programDescription>` element in OSaL.

The `<numberObjectives>`, `<numberConstraints>`, and `<numberVariables>` child elements of `<programDescription>` are required. Each of the three elements has a nonnegative integer `num` attribute and each has a break down of children as shown in Figure 6-35. For example we can have sub-counts of linear constraints, quadratic constraints and (general) nonlinear constraints under `<numberConstraints>`. For linear constraints, we can have a further break down of equality constraints, inequality constraints (one-sided), and range constraints (two-sided). All these count numbers play important roles for the OS registry to find an appropriate solver.

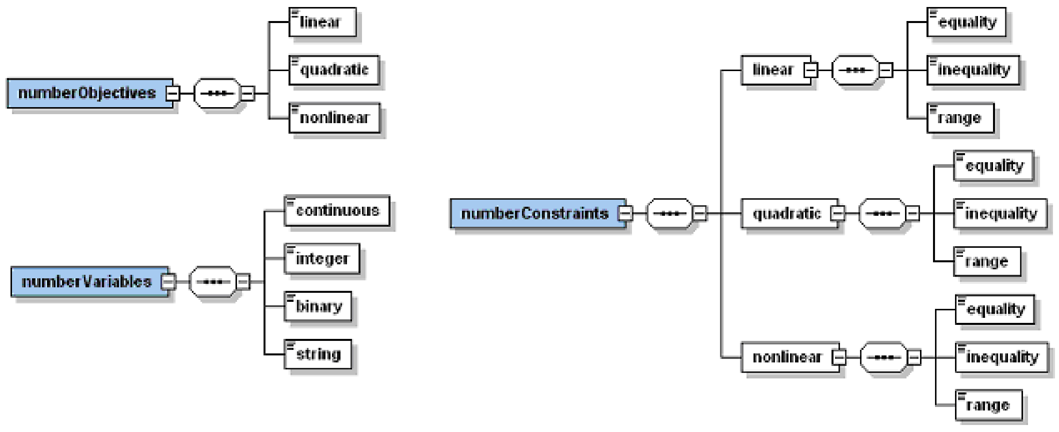


Figure 6-35: `<numberObjectives>`, `<numberVariables>`, and `<numberConstraints>` elements in OSaL.

The actual program data analysis is in `<programDataAnalysis>` (Figure 6-36).

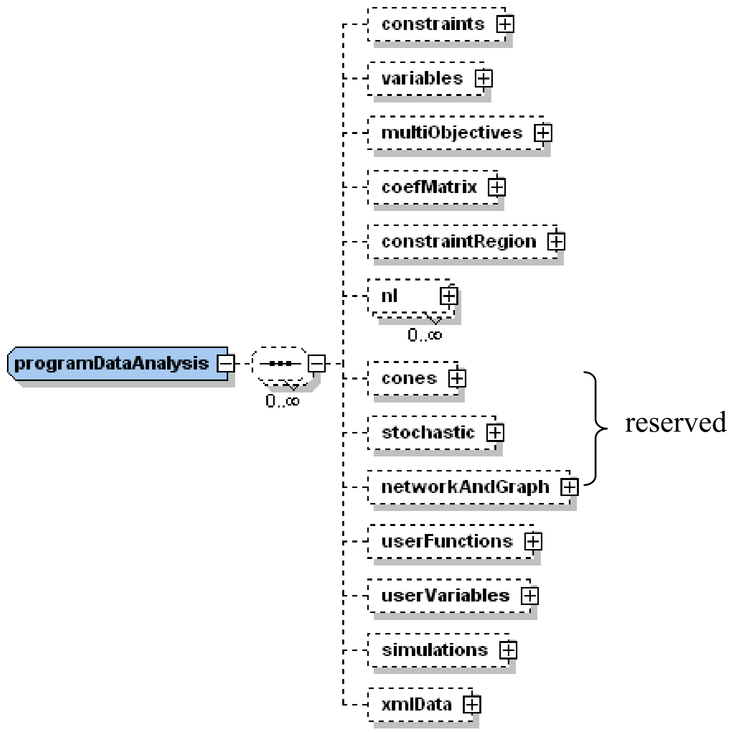


Figure 6-36: `<programDataAnalysis>` element in OSaL.

The `<programDataAnalysis>` element is very similar to the <programData> element in OSiL (Figure 6-8). The only additional child element is the `<constraintRegion>`

element for such analyses as constraint region convexity. The similarity is because all the analyses are done based on the OSiL input, so analysis results can be viewed as the metadata from the OSiL data in different sections. For example, in the first child `<constraints>` of `<programDataAnalysis>` (Figure 6-37), we can have analyses on each constraint (`<con>`). Each `<con>` element has attributes such as `type`, `priority`, `linearity` and `convexity`. If there are other analyses not specified by an attribute, they can be put in a sequence of `<specific>` elements as children of the `<con>` element.

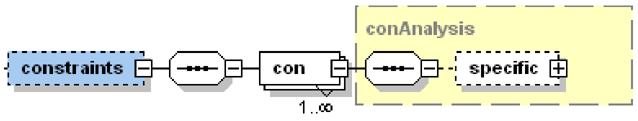


Figure 6-37: `<constraints>` element in OSaL.

Here is an analysis example of the constraints:

```
<constraints>
  <con idx="0" type="geq" linearity="linear" convexity="linear" regionEffect="linear"/>
  <con idx="1" type="leq" linearity="quadratic" convexity="concave" regionEffect="convex"/>
  <con idx="2" type="leq" linearity="nonlinear" convexity="nonconvex" regionEffect="nonconvex"/>
  . . .
</constraints>
```

Analyses on other parts more or less follow the same pattern. In Table 6-15 we list the common analyses that can be put in OSaL. There can be endless analyzes, but in Optimization Services, we emphasize on those that can facilitate matching between instances and appropriate solvers.

Data Part	Common Analyses and Descriptions
constraints	type: geq, leq, eq, geqLeq (constrained on both sides) etc. linearity: linear, quadratic, (general) nonlinear, closeToLinearity etc. convexity: linear, convex, concave, almostConvex, etc. regionEffect: whether the constraint makes the constrained region linear, convex, etc. effectiveness: fraction of the variable space that each constraint eliminates redundant: whether the constraint is redundant and should be eliminated
variables	type: C (continuous), I (integer), B (binary), S (string) priority: for pivoting in integer programming init: suggested initial variable values fixed: whether the variable should be fixed at the initial value (or lb = ub)
objectives	lb and ub : lower and upper bound shape: linear, convex, concave, etc. steepness: objective slope at the current point objectiveEffect: whether the objective is likely to be a global optimum or local optimum
coefMatrix	Density: or sparsity of the coefficient matrix type: listMatrix, coefMatrix, sparseSDPA or mixture
constraintRegion	convexity: linear, convex, almostConvex, etc.
nl	numberQuardratic: number of quadratic terms in each nonlinear function numberLogic: number of logic operators numberRelational: number of relational operators in each nonlinear function numberSimulations: number of simulations in each nonlinear function numberComplementarity: number of complementarity (0 or 1) in each nonlinear function

	numberXPath: number of XPath nodes in each nonlinear function
cones	(reserved)
stochastic	(reserved)
networkAndGraph	(reserved)
userFunctions	rowIn: which rows (constraints or objectives) the user functions are in
userVariables	rowIn: which rows (constraints or objectives) the user variables are in
simulations	rowsIn: which rows the simulations are in time: an estimated time a simulation may take
xmlData	numberData: number of data in the xml data numberLevel: height of the xml tree

Table 6-15: Typical data analyses on different optimization parts in OSaL.

6.7 Optimization Services simulation Language (OSsL)

The OSsL schema is located at http://www.optimizationservices.org/schemas/OSsL.xsd. Simulations are explained in 2.8 and OSsL facilitates enables objective or constraint functions to incorporate simulations, which may be located in places other than the solver. An OSsL instance is usually transmitted between a solver and a simulation engine. From the Optimization Services framework point of view, if a simulation is to be invoked by an OS-compatible solver, its input and output have to be put in the standard OSsL format.

As explained in the OSiL section (§6.2), the definition of a simulation is specified in the `<simulations>` element of OSiL. Each `simulation` consists of the simulation's address using the `<uri>` child and its input and output using the `<OSsL>` child. Figure 6-18 is a good illustration, which we show below again.

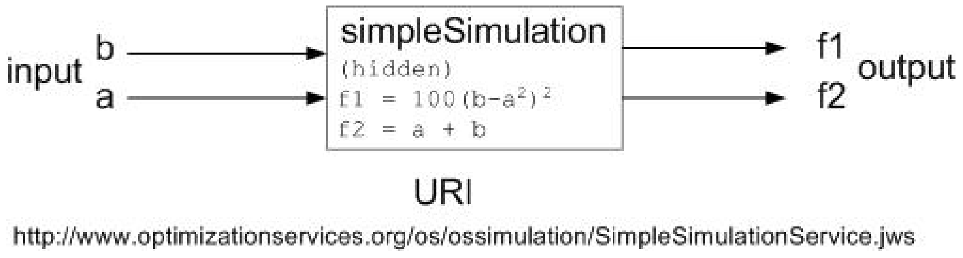

input
b
a
simpleSimulation
(hidden)
f1 = 100(b-a²)²
f2 = a + b
f1
f2
output
URI
http://www.optimizationservices.org/os/ossimulation/SimpleSimulationService.jws

Figure 6-38: simpleSimulation with two inputs (a, b), two outputs (f1, f2) and an address at http://www.optimizationservices.org/os/ossimulation/SimpleSimulationService.jws.

The definition of `simpleSimulation` looks like:

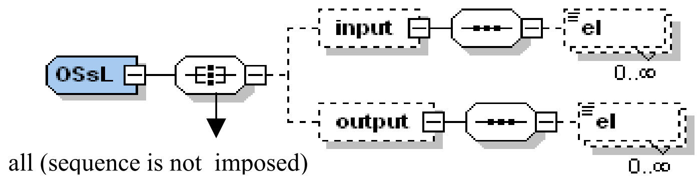
```
<simulation name="simpleSimulation">
   <uri value="http://www.optimizationservices.org/os/ossimulation/SimpleSimulationService.jws"/>
   <OSsL>
      <input>
         <el name="a">1</el>
         <el name="b">MSFT</el>
      </input>
      <output>
         <el name="f1"/>
         <el name="f2"/>
```

```
        </output>
    </OSsL>
</simulation>
```

As explained in the OSnL section (§6.3), to construct a nonlinear expression that contains `simpleSimulation`, we use the `<simInput>` and `<simOutput>` nodes:

```
<nl idx="-1">
    <plus>
        <sim name="simpleSimulation">
            <simInput inputName="a"> <var idx="0"/> </simInput>
            <simInput inputName="b"> <var idx="1"/> </simInput>
            <simOutput outputName="f1"/>
        </sim>
        <number value="2"/>
    </plus>
</nl>
```

In Figure 6-39 below, we list the entire OSsL schema.



```
<xs:complexType name="OSsL">
    <xs:all>
        <xs:element name="input" minOccurs="0">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="el" type="ioType" minOccurs="0" maxOccurs="unbounded"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="output" minOccurs="0">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="el" type="ioType" minOccurs="0" maxOccurs="unbounded"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
    </xs:all>
</xs:complexType>

<xs:complexType name="ioType">
    <xs:simpleContent>
        <xs:extension base="xs:string">
            <xs:attribute name="name" type="xs:ID" use="required"/>
            <xs:attribute name="type" type="type" use="optional" default="string"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:simpleType name="type">
    <xs:restriction base="xs:string">
        <xs:enumeration value="string"/>
        <xs:enumeration value="link"/>
    </xs:restriction>
</xs:simpleType>
```

Figure 6-39: `<OSsL>` root element.

As illustrated in the above figure, the OSsL schema is very simple, but still general enough to accommodate to any existing simulations. The `<OSsL>` element has two optional children `<input>` and `<output>` and it does not matter which comes first. The two child elements are very similar, which is why we don't have a separate schema for each. Both elements have 0 or more `<el>` elements. This corresponds to the notion that simulations in general can take any number of inputs and generate any number of outputs. An `<el>` element is of `ioType`; it takes a required `name` attribute for the input or output name and an optional `type` attribute. The input or output values go inside the elements. By default the `type` attribute is "`string`" which is the most general an input or output value can be. The other `type` is "`link`" which indicates that the value inside the <el> element is a pointer and the actual data is to be obtained from the specified link address. For instance, in the following example:

```
<simulation name="simpleSimulation">
    <uri value="http://www.optimizationservices.org/os/ossimulation/SimpleSimulationService.jws"/>
    <OSsL>
        <input>
            <el name="a" type="string">1</el>
            <el name="b" type="link">http://www.optimizationservices.org/data/stock.html</el>
        </input>
        <output>
            <el name="f1">
            <el name="f2"/>
        </output>
    </OSsL>
</simulation>
```

input "`a`" is a string (= "`1`") and input "`b`" is a link. The value of `b` (e.g. "`MSFT`") should be obtained from the address http://www.optimizationservices.org/data/stock.html.

6.8 Optimization Services transformation Language (OStL)

The OStL schema is located at http://www.optimizationservices.org/schemas/OStL.xsl. OStL is an XML-based Extensible Stylesheet Language (XSL). XSL is covered in §4.4. XSL offers a convenient way to specify translations of XML documents. For example if an optimization solution is formatted in Optimization Services result Language (OSrL), XSL can be applied to the solution instance to easily produce an HTML document that transforms the raw result data into a user-friendly form. Other OSxL representations that can use the OStL style sheet are OSaL (for displaying analysis results), OSeL (for publishing solver entity descriptions) and OSiL (for presenting optimization instances).

Since different users have different tastes of what looks the nicest, OStL is mostly provided as an *optional* alternative for data transformation. A modeling language environment

(MLE), for example, is not recommended to use OStL to display the OSrL results, because the input instance may have been pre-processed and the OSrL result instance needs to be post-processed before it is presented to the user. The MLE can display the data in what the user thinks is the best way, with much more flexibility than a style sheet; for example MLE can allow analyzing the result interactively or displaying values of any expression in the result. In situations where post-processing of OSrL is necessary, names and indexes of the original model may be different from those in the instance, so OStL may not be appropriate to use.

On the other hand, if a service is registered in the OS registry and the service provider wants to publish the standard service information (OSeL) on his own Web site, it is required that he publishes the information using the OStL, that is at the beginning of his OSeL document specify the following OStL style sheet location: `http://www.optimizationservices.org/schemas/OStL.xsd.`

Another purpose for such a requirement is that the Optimization Services registry can advertise the latest news and information by changing the OStL at the above link, so that the revised information is automatically shown on the individual Web sites of those who registered. Unlike most other style sheets such as CSS (cascading style sheet), the XSL based OStL can not only control font weight, style, size and color but can also rearrange the structure of a document, add new contents, tags and attributes.

As an example, the following section of `OStL.xsl` is used to present the objective value, variable solutions and constraint values in OSrL:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:os="os.optimizationservices.org" exclude-result-prefixes="os">
    <xsl:output method="html" version="1.0" encoding="UTF-8" indent="yes"/>
    <xsl:template match="/">
        <html>
            <body>
                <h1>Result</h1>
                <b>objective: </b><xsl:value-of
select="/os:OSrL/os:result/os:objective/os:objectiveValue/@value"/>
                <p/>
                <table>
                    <td colspan="2" align="center">
                        <b>Variables</b>
                        <table border="2" width="10">
                            <tr><td><b>variable</b>    </td><td><b>solution</b></td></tr>
                            <xsl:for-each select="/os:OSrL/os:result/os:variables/os:variableSolution/os:var">
                                <tr>
                                    <td><xsl:value-of select="@varName"/></td>
                                    <td><xsl:value-of select="@value"/></td>
                                </tr>
                            </xsl:for-each>
                        </table>
                    </td>
                </table>
                <table>
```

```
                    <td colspan="2" align="center">
                        <b>Constraints</b>
                        <table border="2" width="10">
                            <tr><td><b>constraint</b></td><td><b>value</b></td></tr>
                            <xsl:for-each select="/os:OSrL/os:result/os:constraints/os:constraintValue/os:con">
                                <tr>
                                    <td><xsl:value-of select="@conName"/></td>
                                    <td><xsl:value-of select="@value"/></td>
                                </tr>
                            </xsl:for-each>
                        </table>
                    </td>
                </table>
            </body>
        </html>
    </xsl:template>
</xsl:stylesheet>
```

Suppose the OSrL looks like:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="http://www.optimizationservices.org/schemas/OStL.xsl"?>
<OSrL xmlns="os.optimizationservices.org" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="os.optimizationservices.org
http://www.optimizationservices.org/schemas/OSrL.xsd">
    <result>
        <status type="optimal"/>
        <objective>
            <objectiveValue value="4.5"/>
        </objective>
        <variables>
            <variableSolution>
                <var varName="x1" value="2.1"/>
                <var varName="x2" value="3.5"/>
                <var varName="x3" value="3.5"/>
            </variableSolution>
        </variables>
        <constraints>
            <constraintValue>
                <con conName="con1" value="-3.4"/>
                <con conName="con2" value="0.0"/>
            </constraintValue>
        </constraints>
    </result>
</OSrL>
```

Notice the second line includes a link to where the `OStL.xsl` file is. The OSrL example is presented as:

Result

objective: 4.5

Variables

variable	solution
x1	2.1
x2	3.5
x3	3.5

Constraints

constraint	value

con1	-3.4
con2	0.0

Note if in the above OStL example, we (designers of OS protocols) change the heading part (`<body><h1>Result</body></h1>`) in `<xsl:template match="/">` to:

```
<xsl:template match="/">
   <html>
      <body>
         <h1>Result found by Optimization Services</h1>
         <b>objective: …
      . . .
      </body>
   </html>
</xsl:template>
```

The display will automatically change to:

Result found by Optimization Services

objective: 4.5

Variables

variable	**solution**
x1	2.1
x2	3.5
x3	3.5

Constraints

constraint	**value**
con1	-3.4
con2	0.0

In this way by changing `OStL.xsl` at ://www.optimizationservices.org/schemas/OStL.xsl, we can have a control over various publications (at least those required to include the `OStL.xsl` link) over the entire decentralized OS network.

CHAPTER 7 OPTIMIZATION SERVICES COMMUNICATION

In this chapter, we present the instance communication part of Optimization Services Protocol (OSP). OS communication is about the exchange of a set of *low-level* data instances between different Optimization Services components. (The difference between low-level instances and high level models is explained in Chapter 2).

Communication sub-protocols deal with the general areas of optimization access, operations and flows. All but one communication sub-protocol deals with optimization access and operations which are specified using the WSDL documents (§4.7). The only exception is the Optimization Services flow Language (OSfL, §7.3), which defines flow orchestration in the XML-based BPEL (Business Process Execution Language [91]) format.

All the OS communication WSDL documents have three main parts: interface, protocol (binding & encoding), and service address. The Interface part varies between different WSDL documents as different *types* of services have different functions and methods. The protocol part is exactly the same for all the WSDL documents as we currently require all the services on an OS network use exactly the same communication binding and message encoding mechanisms. So we will only illustrate the protocol part in the first OS communication protocol that we introduce, namely OShL in the next section. The service address part of all the generic OSxL WSDL documents is empty (not specified), as addresses of the individual services are different. So technically speaking, OS communication protocols standardize the interface (operations, messages, parts) and protocol (binding, encoding) parts of the WSDL documents of all the OS services. All the OS services have their own addresses listed in their individual WSDL documents and the rest of the WSDL documents should be exactly the same as specified by the OS communication protocols.

No mechanisms such as encoding and security are addressed in OSP. OSP leverages the mechanisms provided by its underlying protocols such as SOAP and HTTP. All the registry related OSxL communications are covered in Chapter 8. We provide open-source libraries (Appendix B) for sending and receiving all the instances to simplify exchange of information. Some of the examples illustrated in this chapter demonstrate the use of communication agents in the OS libraries. All the communication documents (WSDL, BPEL) and libraries are available at www.optimizationservices.org [92] and www.optimizationservices.net [93].

Standards for optimization instance communication over *distributed* systems are new. But the standardization is technologically timely. Distributed technologies such as Web services (Chapter 4) are growing rapidly in importance in today's computing environment and are already widely accepted as industrial standards. It is our vision that by combining Operations Research and modern distributed technologies, Optimization Services will make a wider audience able to easily access and benefit from the increasing number of OR software packages.

Through standardization of communication, the OS framework provides an open infrastructure for all optimization system components to communicate with each other as shown in §5.3. The goal is that all the algorithmic codes will be implemented as services under this framework and customers will use these computational services like utility services. Special knowledge of optimization algorithms, problem types, and solver options required of users should be minimized. Everything that involves finding the right solver, invoking the software, providing the computing resources and presenting the solution is automatically taken care of by Optimization Services infrastructure.

The Optimization Services framework does not standardize local interfacing. As mentioned in the previous chapters, related projects such as COIN [23] and derived research from Optimization Services such as the Optimization Services instance Interface (OSiI), Optimization Services option Interface (OSoI) and Optimization Services result Interface (OSrI) are intended to do this job. The COIN project includes the OSI (Open Solver Interface) library which is an API for linear programming solvers, and NLPAPI, a subroutine library with routines for building nonlinear programming problems. Another proposed nonlinear interface by Halldórsson, Thorsteinsson, and Kristjánsson is MOI (Modeler-
Optimizer Interface [60]) that specifies the format for a callable library. This library is based on representing the nonlinear part of each constraint and the objective function in post-fix (reverse Polish) notation [2] and then assigning integers to operators, characters to operands, integer indices to variables and finally defining the corresponding set of arrays. The MOI data structure then corresponds to the implementation of a stack machine. A similar interface is described in the LINDO API manual [74].

The Optimization Services framework is complementary to the standardization of local interfaces. The connection between Optimization Services and local interfacing is illustrated in Figure 7-1.

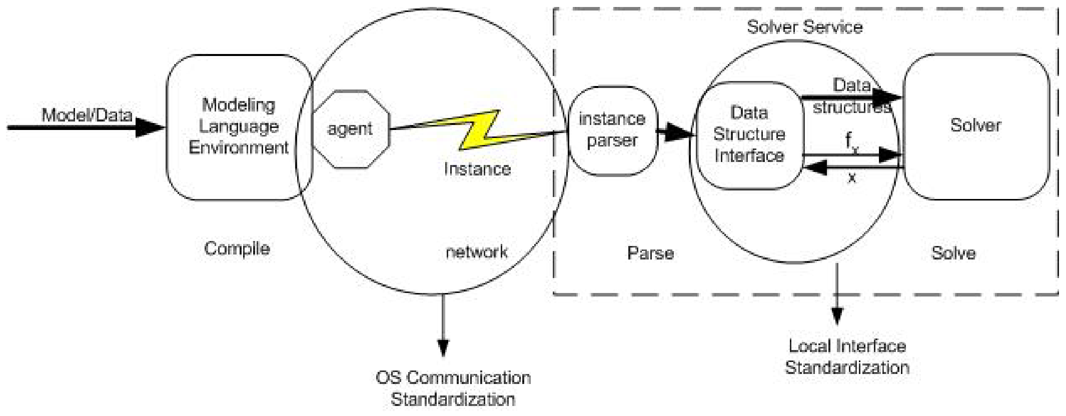


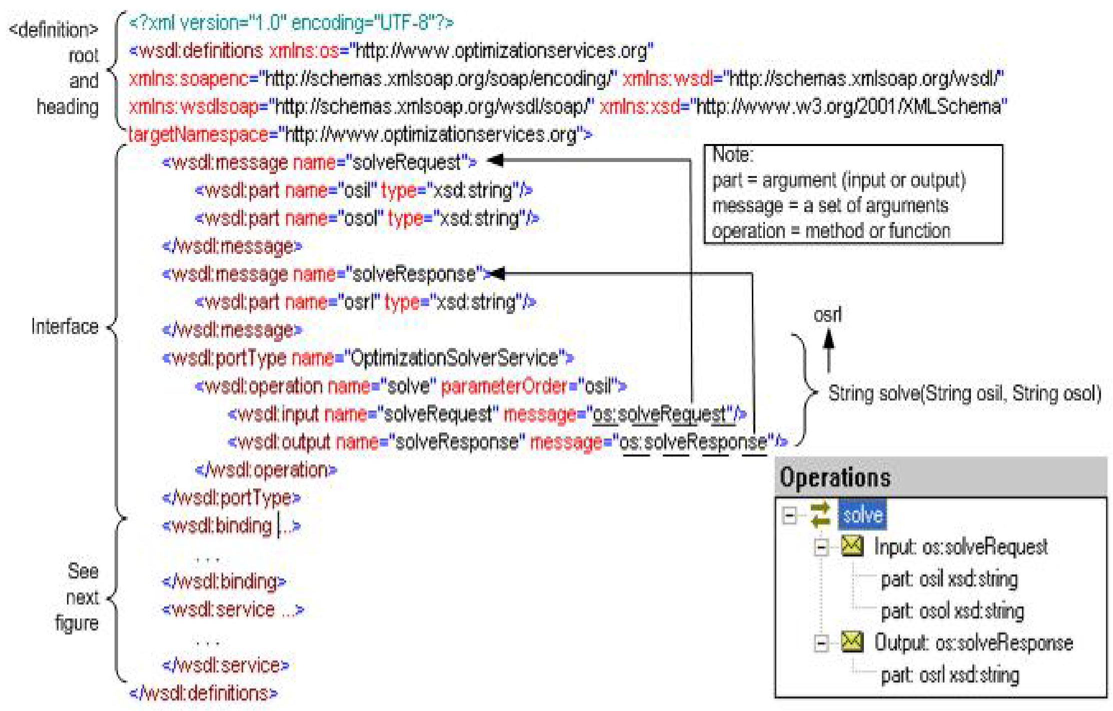
Figure 7-1: Relationship between OS Communication and local interface standardization.

In the figure, the Modeling Language Environment generates an instance (OSiL) and delegates a communication agent (solver agent) to send the instance to the remote solver service. OS communication standardizes this distributed process. After the solver service receives the instance from the network, the local solver uses an instance parser to parse the instance into a set of standard objects/data structures that are held in the data structure interface (e.g. COIN-OSI). As the instance is a standard instance, only one parser needs to be written to read the instance and as the local interface is also a standard interface, both can be provided in *one* library. All that a solver developer needs to do is to include this library to resolve all interface or format issues.

The success of Optimization Services will promote the work of local interface standardization and in turn the wide acceptance of the standard local interfaces will allow more solvers to be easily hooked into the Optimization Services system.

7.1 Optimization Services hookup Language (OShL)

The OShL document is at http://www.optimizationservices.org/schemas/OShL.wsdl. In the above Figure 7-1, a (solver) agent is delegated to contact the (solver) service. Communication is always between two components; therefore both the agent and the service have to follow certain rules. The rules are specified in the `OShL.wsdl` document. Figure 7-2 shows the first half (interface part) of a simplified version of the WSDL document. This part varies between different *types* of services. All the solver services and analyzer services are required to follow the interface specification of OShL.



```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:os="http://www.optimizationservices.org"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.optimizationservices.org">
    <wsdl:message name="solveRequest">
        <wsdl:part name="osil" type="xsd:string"/>
        <wsdl:part name="osol" type="xsd:string"/>
    </wsdl:message>
    <wsdl:message name="solveResponse">
        <wsdl:part name="osrl" type="xsd:string"/>
    </wsdl:message>
    <wsdl:portType name="OptimizationSolverService">
        <wsdl:operation name="solve" parameterOrder="osil">
            <wsdl:input name="solveRequest" message="os:solveRequest"/>
            <wsdl:output name="solveResponse" message="os:solveResponse"/>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding |...>
        . . .
    </wsdl:binding>
    <wsdl:service ...>
        . . .
    </wsdl:service>
</wsdl:definitions>
```

Note:
part = argument (input or output)
message = a set of arguments
operation = method or function

```
Operations
solve
  Input: os:solveRequest
    part: osil xsd:string
    part: osol xsd:string
  Output: os:solveResponse
    part: osrl xsd:string
```

<definition> root and heading
Interface
See next figure
osrl
String solve(String osil, String osol)

Figure 7-2: Illustration of a simplified OShL (interface part).

The most important part of Figure 7-2 is the `<wsdl:portType>` element. The `portType` element can have one or more `<operation>` elements. In this simplified example, we only list one operation whose name is `solve`. Each `operation` corresponds to a method or function in a programming language. So there are usually two parts to an operation: the `input` element and the `output` element. The format of both elements is controlled by the `message` attribute. In the `solve` operation, we require its input to be of message type “`solverRequest`” and its output to be of message type “`solverResponse`.” The `solverRequest` message has two `part` elements, `osil` and `osol`, both of string types. A part corresponds to an argument of a function or method. So we can regard a message as a sequence of arguments to be passed to the function or method.

Simply put, the WSDL document in Figure 7-2 specifies the following operation for each solver:

```
String solve(String osil, String osol);
```

that is, every solver service is required to have a method called “`solve`” that takes two input strings and returns one string. The first input string should be an OSiL optimization instance, the second input string should be an OSoL option instance, and the returned string should be an

OSrL result instance. OShL, as well as other OS communication protocols, does not specify how the strings should look inside. This is the responsibility of OS representation protocols discussed in Chapter 6. So without the OS representation protocols, a client can still transmit any junk strings to a solver service successfully. Of course, all the OS-compatible components are required to validate input and output instances, so no invalid instances will be ever transmitted onto the network. WSDL documents and XML schemas are two key technologies to ensure the high quality of an entire OS network. In Table 7-1, we list the operations currently specified in the OShL WSDL document.

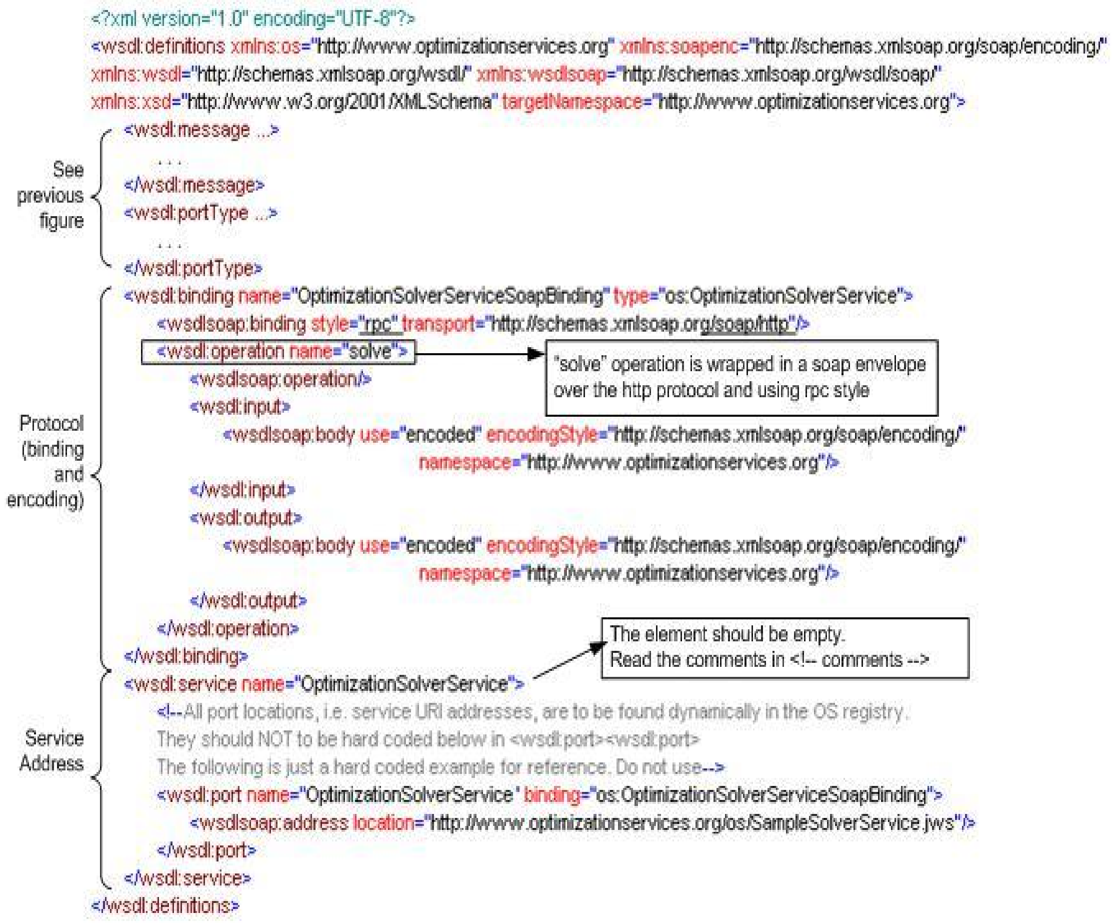
Operation	Description
String getJobID( )	No input. Output string is a unique job id.
String solve (String, String)	1st input is an OSiL instance for optimization problem. 2nd input is an OSoL instance for solver option. Output is an OSrL instance for an optimization.
String solve (String)	1st input is an OSiL instance for optimization problem. Solver options are assumed to be default. Output is an OSrL instance for an optimization.
String retrieveResult (String)	1st input is a job id. Output is an OSrL instance for an optimization
String analyze(String)	1st input is an OSiL instance for optimization problem. Output is an OSaL instance for analysis.

Table 7-1: Operations in OShL.

The `getJobID` and `retrieveResult` operations will be explained in the OSfL section (§7.3, Figure 7-8). For the one-argument `solve` operation that does not take options, the solver should use its default options. Many solvers may not do analysis. In this case solver services can just implement a dummy `analyze` operation, which returns an empty analysis result (e.g.) <OSaL/>. Conversely, analyzers may do dummy implementations for the solver-related operations.

Figure 7-3 shows the other half (protocol & address part) of the OShL WSDL document. The hard-coded service address part should be empty and is only shown for the purpose of a complete illustration. The *generic* OShL WSDL document does not specify where the service is. In reality, the service location is dynamically discovered in the OS registry (Chapter 8). Each individual solver or analyzer service has exactly the same OShL WSDL document following the OShL protocol except that it has an extra location specified in the `<wsdlsoap:address>` element under `<wsdl:service>`. In Figure 7-3, this is illustrated with an example address as:

http://www.optimizationservices.org/os/SampleSolverService.jws.


See previous figure
Protocol (binding and encoding)
Service Address
"solve" operation is wrapped in a soap envelope over the http protocol and using rpc style
The element should be empty. Read the comments in <!-- comments -->

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:os="http://www.optimizationservices.org" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www.optimizationservices.org">
  <wsdl:message ...>
    . . .
  </wsdl:message>
  <wsdl:portType ...>
    . . .
  </wsdl:portType>
  <wsdl:binding name="OptimizationSolverServiceSoapBinding" type="os:OptimizationSolverService">
    <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="solve">
      <wsdlsoap:operation/>
      <wsdl:input>
        <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                       namespace="http://www.optimizationservices.org"/>
      </wsdl:input>
      <wsdl:output>
        <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                       namespace="http://www.optimizationservices.org"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="OptimizationSolverService">
    <!--All port locations, i.e. service URI addresses, are to be found dynamically in the OS registry.
    They should NOT to be hard coded below in <wsdl:port><wsdl:port>
    The following is just a hard coded example for reference. Do not use-->
    <wsdl:port name="OptimizationSolverService" binding="os:OptimizationSolverServiceSoapBinding">
      <wsdlsoap:address location="http://www.optimizationservices.org/os/SampleSolverService.jws"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Figure 7-3: Illustration of a simplified OShL (protocol and address part).

The most important part of Figure 7-3 is the protocol part represented by the `<wsd:binding>` element. The protocol part of any other OSxL WSDL document is exactly the same as the OShL WSDL document here. Currently we require all the services on an OS network use exactly the same communication binding and message encoding mechanisms. So in this thesis this is the only section that we illustrate the protocol part of all OSxL WSDL documents.

The `binding` element contains one or more operation elements for each operation specified in Figure 7-2. Each operation can potentially be called using a different protocol binding. All the operations in Optimization Services, however, use the same protocol binding. In this simplified example, we only have the `String solve(String, String)` operation. As for any other operation, the `solve` operation is required to be of `rpc` style (remote procedure call, Chapter 4), which is the most typical request and response calling style, or a blocking call. So the request client that invokes the `solve` operation from a solver service has to wait for the response. Of course, the client application can launch a separate process or thread to issue this `solve` operation and let the thread wait there for the response, so that the user of the application can go on with other tasks. Depending on the application settings, the `solve` operation and its argument instances can either be directly sent to the solver service or first submitted to a queue server. On the solver service side, when the `solve` operation is received, it can either solve it directly, or launch a separate process or thread to solve the instances, or it can put the instances in its own queue. Thus the request and response `rpc` style specified in all the OSxL WSDL documents is general enough for all the current needs, while developers can have their own innovative implementations that fit their users the best.

As for any other operation, the `solve` operation and its arguments are required to be packaged in a SOAP envelope which is transported over the HTTP protocol. This is specified by the `transport=http://schemas.xmlsoap.org/soap/http` attribute of the `<wsdl:binding>` element. Like the "RPC" style, the "SOAP over HTTP" transport binding is general enough for all the current needs. As Optimization Services evolves, more transport binding (SOAP over other protocols) may be added.

Each `<operation>` element has a `<wsdl:input>` element and a `<wsdl:output>` element in which we specify the encoding styles. In Optimization Services, we use leverage on the standard soap encoding: "http://schemas.xmlsoap.org/soap/encoding." This makes the input and output arguments platform and programming language independent.

The protocol part of an OSxL WSDL document is technically complex to implement. These are all taken care of by the libraries provided. On the modeler (client) side, the agent hides all the networking details, such as encoding the operation arguments, packing the operations in a SOAP envelope, transmitting the SOAP envelope over the HTTP protocol and handling the HTTP response. On the service (server) side, the OS Server that hosts the service takes care of reading the HTTP request, extracting the SOAP envelope, decoding the operation arguments invoking the service interface, and sending back the result. So as long as the service interface follows the OS communication standards (e.g. Figure 7-2), the entire call should be completed successfully.

We illustrate the process using the OS library below.

Imagine we formulate the problem (7-1) in AMPL (`7_1.mod`) as it would be under the Optimization Services framework and AMPL uses the Knitro solver service hosted at http://www.ziena.com/os/KnitroSolverService.jws to solve the problem. Assume that we already found this address in the OS Registry (Chapter 8).

$$\begin{aligned} &\underset{x}{\text{minimize}} \quad (1-x_1)^2 + 100(x_0 - x_1{}^2)^2 \\ &\text{subject to} \quad x_0 + x_1 - 100 \le 0 \end{aligned} \tag{7-1}$$

At the modeler side, to solve the problem, the user would type the following commands at the AMPL prompt:

```
ampl: model 7_1.mod;
ampl: option OptimizationServices on;
ampl: solve;
```

Underneath, AMPL first translates the model (7_1.mod) to an OSiL instance:

```
<OSiL xmlns="os.optimizationservices.org" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="os.optimizationservices.org
   http://www.optimizationservices.org/schemas/OSiL.xsd">
   <programDescription>
      <maxOrMin>min</maxOrMin><numberObjectives>1</numberObjectives>
      <numberConstraints>1</numberConstraints><numberVariables>2</numberVariables>
   </programDescription>
   <programData>
      <constraints><con ub="0.0"/></constraints>
      <variables><var lb="0" name="x1" type="C"/><var lb="0" name="x2" type="C"/></variables>
      <nl idx="-1"><plus><power><minus><number type="real" value="1.0"/><var coef="1.0"
      idx="1"/></minus><number type="real" value="2.0"/></power><times><number type="real"
      value="100"/><power><minus><var coef="1.0" idx="0"/><power><var coef="1.0" idx="1"/><number
      type="real" value="2.0"/></power></minus><number type="real"
      value="2.0"/></power></times></plus></nl>
      <nl idx="0"><minus><plus><var coef="1.0" idx="0"/><var coef="1.0" idx="1"/></plus><number
      type="real" value="100"/></minus></nl>
   </programData>
</OSiL>
```

It is AMPL's job to make sure this string validates against the OSiL schema. Suppose this instance is stored in the `String` variable:

`sOSiL`.

Next AMPL instantiates a communication agent provided in the OS library (Appendix B):

`OSSolverAgent osSolverAgent = new OSSolverAgent();`

Suppose we already found the solver address and the address is stored in a `String` variable:

`sSolverAddress` (=http://www.ziena.com/os/KnitroSolverService.jws)

AMPL then tells the address to `osSolverAgent` and delegates `osSolverAgent` to send and solve the problem. For simplicity, we will not pass the solver options and from Table 7-1 we choose to use the simpler operation: `String solve(String)`, which takes as input an OSiL instance and return as output an OSrL instance:

```
osSolverAgent.solverAddress = sSolverAddress;
String sOSrL = osSolverAgent.solve(sOSiL);
```

In the `solve` operation, the agent contacts the solver service at the given `solverAddress` and gets back an OSrL result instance stored in the String variable:

`sOSrL`

which looks like:

```
<OSrL xmlns="os.optimizationservices.org" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="os.optimizationservices.org http://www.optimizationservices/schemas/OSrL.xsd">
    <result>
        <status type="optimal"/>
        <objective> <objectiveValue value="0.000"/></objective>
        <variables><variableSolution><description/>
                <var idx="0" varName="x1" value="1.0"/><var idx="1" varName="x2" value="1.0"/>
                </variableSolution>
        </variables>
    </result>
</OSrL>
```

On receiving the OSrL result, AMPL parses the result and presents it to its user. Notice that on the client side, we do not worry what language the solver service is implemented in and what platform the solver service is installed on.

The major step in the entire process is the code below:

```
String sOSrL = osSolverAgent.solve(sOSiL);
```

AMPL itself does not need to know how the `solve` operation is implemented; the `OSSolverAgent` class from the OS library hides all the networking complexities from the modeling language environment. In `OSSolverAgent`'s `solve` operation, four steps are taken:

Solver agent step 1: Encoding

`OSSolverAgent` encodes the `sOSiL` input string into the following encoded string, according to the soap encoding style specified in the OShL WSDL document; most distinctly all the "<" and ">" signs are repectively encoded as "`&lt;`" and "`&gt;`".

```
&lt;OSiL xmlns="os.optimizationservices.org" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="os.optimizationservices.org
    http://www.optimizationservices.org/schemas/OSiL.xsd"&gt;
    &lt;programDescription&gt;
        &lt;maxOrMin&gt;min&lt;/maxOrMin&gt;&lt;numberObjectives&gt;1&lt;/numberObjectives&gt;

    &lt;numberConstraints&gt;1&lt;/numberConstraints&gt;&lt;numberVariables&gt;2&lt;/numberVariables&gt;
    &lt;/programDescription&gt;
    &lt;programData&gt;
        &lt;constraints&gt;&lt;con ub="0.0"/&gt;&lt;/constraints&gt;
        &lt;variables&gt; &lt;var lb="0" name="x1" type="C"/&gt;&lt;var lb="0" name="x2"
      type="C"/&gt;&lt;/variables&gt;
        &lt;nl idx="-1"&gt;&lt;plus&gt;    &lt;power&gt;&lt;minus&gt;&lt;number type="real" value="1.0"/&gt;
            &lt;var coef="1.0" idx="1"/&gt;&lt;/minus&gt;&lt;number type="real"
        value="2.0"/&gt;&lt;/power&gt;&lt;times&gt;&lt;number type="real"
        value="100"/&gt;&lt;power&gt;&lt;minus&gt;&lt;var coef="1.0" idx="0"/&gt;&lt;power&gt;&lt;var coef="1.0"
        idx="1"/&gt;&lt;number type="real" value="2.0"/&gt;&lt;/power&gt;&lt;/minus&gt;&lt;number type="real"
        value="2.0"/&gt;&lt;/power&gt;&lt;/times&gt;&lt;/plus&gt;&lt;/nl&gt;
        &lt;nl idx="0"&gt;&lt;minus&gt;&lt;plus&gt;    &lt;var coef="1.0" idx="0"/&gt;&lt;var coef="1.0" idx="1"/&gt;
            &lt;/plus&gt;&lt;number type="real" value="100"/&gt;&lt;/minus&gt;&lt;/nl&gt;
    &lt;/programData&gt;
&lt;/OSiL&gt;
```

Solver agent step 2: Constructing SOAP envelope

According to the OShL WSDL document, we should use the "SOAP over "HTTP" transport, so `OSSolverAgent` packs the operation `String solve(String)` and the above encoded `sOSiL` input string argument into a SOAP envelope, which looks like this:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
     <soapenv:Body>
          <ns1:solve soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          xmlns:ns1="http://www.optimizationservices.org/os/ossolver/KnitroSolverService.jws">
               lt;OSiL xmlns="os.optimizationservices.org" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"                 ...
                    ...
                    ...
               &lt;/OSiL&gt;
          </ns1:solve>
     </soapenv:Body>
</soapenv:Envelope>
```

Solver agent step 3: Sending and receiving

Again according to the OShL WSDL document, we should send the above constructed SOAP envelope over the HTTP networking protocol using RPC style, so `OSSolverAgent` constructs the following HTTP POST header. Since this is an HTTP POST, we attach the POST data – the SOAP envelope – at the end of the HTTP header with a line separation (i.e. two new line characters):

```
POST /os/ossolver/KnitroSolverService.jws HTTP/1.0
Content-Type: text/xml; charset=utf-8
Accept: application/soap+xml, application/dime, multipart/related, text/*
User-Agent: Axis/1.2beta3
Host: http://www.ziena.com
Cache-Control: no-cache
Pragma: no-cache
Content-Length: 2488

<soapenv:Envelope …>
…
…
…
</soapenv:Envelope>
```

The HTTP POST method is itself of RPC style and waits until it receives the result from the remote server. When the Knitro solver service sends back the result, it should be a string and the string should be of the OSrL format. The returned result is a SOAP envelope encoded under an HTTP response header:

```
HTTP/1.1 200 OK
Set-Cookie: JSESSIONID=A8AF406536A271018100F64CFA462FA0; Path=/os
Content-Type: text/xml;charset=utf-8
Date: Sun, 20 Mar 2005 21:28:40 GMT
Server: Apache-Coyote/1.1
Connection: close

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Body>
        <ns1:solveResponse soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:ns1="http://www.optimizationservices.org/os/ossolver/LindoSolverService.jws">
            <solveReturn xsi:type="xsd:string">
                &lt;OSrL xmlns="os.optimizationservices.org"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="os.optimizationservices.org
                                    http://www.optimizationservices/schemas/OSrL.xsd"&gt;
                    &lt;result&gt;
                        &lt;status type="optimal"/&gt;
                        &lt;objective&gt;&lt;objectiveValue value="0.000"/&gt;&lt;/objective&gt;
                        &lt;variables&gt;&lt;variableSolution&gt;
                                &lt;description/&gt;
                                &lt;var idx="0" varName="x1" value="1.0"/&gt;
                                &lt;var idx="1" varName="x2" value="1.0"/&gt;
                        &lt;/variableSolution&gt;&lt;/variables&gt;
                    &lt;/result&gt;
                &lt;/OSrL&gt;
            </solveReturn>
        </ns1:solveResponse>
    </soapenv:Body>
</soapenv:Envelope>
```

Solver agent step 4: Decoding

On getting the encoded result, `OSSolverAgent` then extracts out the SOAP envelope attached at the end of the HTTP response header and decodes the result into:

```
<OSrL xmlns="os.optimizationservices.org" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="os.optimizationservices.org http://www.optimizationservices/schemas/OSrL.xsd">
    <result>
        <status type="optimal"/>
        <objective><objectiveValue value="0.000"/></objective>
        <variables><variableSolution><description/>
                <var idx="0" varName="x1" value="1.0"/><var idx="1" varName="x2" value="1.0"/>
                </variableSolution>
        </variables>
    </result>
</OSrL>
```

The AMPL modeling language only sees the above decoded result in the `String` variable `sOSrL`. AMPL then post -processes the `sOSrL` string and waits for further user instructions (e.g. `display`).

At the solver side, to solve the problem, the solver service also has to follow the OShL WSDL document as communication always involves two parties. The simplified Knitro solver service code looks like:

```
public class KnitroSolverService {
    public String solve(String osil){
        //read OSiL
        OSiLReader osilReader = new OSiLReader();
        OSiI inputInterface = osilReader.getStandardInputInterface(osil);

        //solve
        KnitroSolver knitro = new KnitroSolver();
        OSrI outputInterface = knitro.solve(inputInterface);

        //write and return OSrL
        OSrLWriter osrlWriter = new OSiLWriter();
        String osrl = osrlWriter.getOutputInterface (outputInterface);
        return osrl;
    }//solve

    public String solve(String osil, String osrl){
        …
        return osrl;
    }

    public String getJobID( ){
        …
        return jobID;
    }

    public String retrieve(String jobID){
        …
        return osrl;
    }

    public String analyze(String osil){
        …
        return osal;
    }
}//class KnitroSolverService
```

As we can see, the Knitro solver service implements all the operations required in the OShL WSDL document (listed in Table 7-1). We only briefly show the implementation of the `String solve(String)`. There are mainly 3 steps in this operation:

1. Reading

```
OSiLReader osilReader = new OSiLReader();
OSiI inputInterface = osilReader.getStandardInputInterface(osil);
```

The Knitro solver service gets the OSiL string and use the `OSiLReader` class provided in the OS library to parse the OSiL instance into a set of in-memory data structures that are held in a standard input interface (`OSiI`).

2. Solving

```
KnitroSolver knitro = new KnitroSolver();
OSrI outputInterface = knitro.solve(inputInterface);
```

The Knitro solver engine (`KnitroSolver`) is instantiated. The solver takes the optimization problem input interface (`OSiI`), solves the problem and outputs the result into a standard optimization output interface (`OSrI`).

3. Writing and returning

```
OSrLWriter osrlWriter = new OSiLWriter();
String osrl = osrlWriter.getOutputInterface (outputInterface);
return osrl;
```

The Knitro solver service uses the `OSrLWriter` class provided in the OS library to write the OSrL instance from the in-memory result data structures held in the `OSrI` output interface. The Knitro solver service then simply returns the OSrL result instance. Of course the Knitro solver service has to make sure the OSrL instance is valid. By using the OS library to construct the OSrL instance, the result should be automatically validated.

In the above 3 steps, we see that the Knitro solver service does not need to worry about how the input OSiL instance received from the internet (using SOAP over HTTP) should be decoded. Neither does it need to worry about how to encode and send back the OSrL output instance (again using SOAP over HTTP) to the client. This is because the Knitro solver service is hosted by instance our OS Server software. The OS Server hides all the networking complexities from the solver service. Our OS Server can be downloaded from the Optimization Services Web site at http://www.optimizationservices.org or http://www.optimizationservices.net. The Knitro solver developers simply put the above Knitro solver service code in a file called **`KnitroSolverService.jws`** and put the file in the **`os`** sub-directory relative to the OS Server's public root directory. Since in our example, we host the Knitro solver service at http://www.ziena.com, thus the service address is at http://www.ziena.com/**os/KnitroSolverService.jws** (i.e. http://domain name + directory + service, just like a regular Web page address). What the OS Server does is more or less the opposite of what the client's communication agent does, again in 4 steps:

Solver server step 1: Decoding

On getting the “SOAP over HTTP” request (sending part of **Solver agent step 3)** from the client, the OS Server extracts out the SOAP envelope attached at the end of the HTTP request header, gets the operation name and decodes the operation input into a regular OSiL instance. This is similar to any Web server in handling a regular HTTP request with POST message (e.g. HTML form data).

Solver server step 2: Invoking

The OS Server loads the `KnitroSolverService.jws` file and compiles the class only if it is the first time the service is called and loaded. The OS Server then invokes the operation with the decoded input according to what is specified in the SOAP envelope; in our example it is `String solve(String)`. Upon invocation, the Kniro solver service starts the optimization process. This is similar to any Web server in locating and loading an html page from its file system.

Solver server step 3: Encoding

The OS Server then waits for the Knitro solver service to return the optimization result, which is in plain OSrL format. Upon getting the OSrL instance, the OS Server encodes the OSrL instance and packs the encoded result instance into a SOAP envelope. This step has no equivalent in a usual Web server, as a Web server does not need to encode an html file.

Solver server step 4: Returning

The OS Server prepares an HTTP response header, attaches the constructed SOAP envelope at the end with a line separation (i.e. two new line characters), and sends the HTTP response header with SOAP attachment back to the client. This is what the client agent sees in Solver agent step 3 when the agent receives the response from the OS Server. This step is similar to any Web server preparing an HTTP response and return a plain HTML page back to the calling browser.

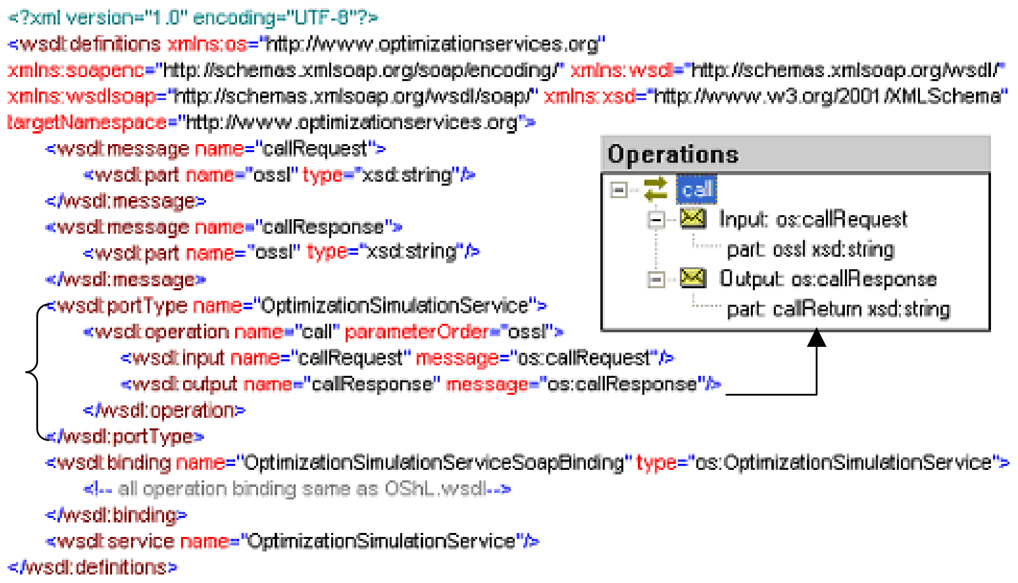
This completes the entire solver “hook up” process according the Optimization Services hookup Language. All the networking complexities are hidden and taken care by the OS library. The OS library also provides parsers to read and write standard instances. All that a modeling language environment does is to use the OS parser library to write standard OSiL instances and read standard OSrL instances after delegating an OS communication agent to contact the solver service. All that a solver service does is to expose the standard interface, implement all the required operations in the interface listed in OShL, let the OS Server take care of the underneath networking, and use the OS parser library to read OSiL instances and write OSrL instances.

7.2 Optimization Services call Language (OScL)

The OScL document is at http://www.optimizationservices.org/wsdl/OScL.wsdl. OScL is mainly used to call simulation services. Theoretically, any client or user on the network can call the simulation service just like a regular Web Service. But under the Optimization Services framework, the main purpose to standardize simulation services is to provide extension to simulation optimization. Therefore in the OS practice, the client that calls the simulation is usually a solver service.

Simulations are fully explained in 2.8. From the Optimization Services framework point of view, if a simulation is to be invoked by an OS-compatible solver, its input and output first have to follow the standard OSsL schema (§6.7).

To make the communication, both the solver agent and the simulation service have to follow the rules specified in the `OScL.wsdl` document, just as the communication between a modeler and a solver follows the OShL WSDL document. Figure 7-2 shows the interface part of the OScL WSDL document. The other part (protocol & address part) of the WSDL document, like all other OSxL WSDL documents, uses the same specifications as in OShL shown in Figure 7-3; the reason is explained in the beginning of this chapter and also in the OShL section (§7.1). Briefly the networking protocol has to be "SOAP over HTTP" with an RPC style call; the input and output encoding has to be the standard SOAP encoding; the address is empty and to be dynamically found in the OS Registry if unknown.



```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:os="http://www.optimizationservices.org"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.optimizationservices.org">
    <wsdl:message name="callRequest">
        <wsdl:part name="ossl" type="xsd:string"/>
    </wsdl:message>
    <wsdl:message name="callResponse">
        <wsdl:part name="ossl" type="xsd:string"/>
    </wsdl:message>
    <wsdl:portType name="OptimizationSimulationService">
        <wsdl:operation name="call" parameterOrder="ossl">
            <wsdl:input name="callRequest" message="os:callRequest"/>
            <wsdl:output name="callResponse" message="os:callResponse"/>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="OptimizationSimulationServiceSoapBinding" type="os:OptimizationSimulationService">
        <!-- all operation binding same as OShL.wsdl-->
    </wsdl:binding>
    <wsdl:service name="OptimizationSimulationService"/>
</wsdl:definitions>
```

```
Operations
call
    Input: os:callRequest
        part: ossl xsd:string
    Output: os:callResponse
        part: callReturn xsd:string
```

Figure 7-4: Illustration of OScL (interface part); other parts are the same as OShL in Figure 7-3.

The `<wsdl:portType>` element in Figure 7-4 has only one operation whose name is `call`. The `call` operation's input is required to be of message type "`callRequest`" and its output is required to be of message type "`callResponse`." The `callRequest` message has one `part` element (i.e. one argument), `ossl`, which is of `string` types. Simply put, the WSDL document in Figure 7-4 specifies the following single operation for each simulation:

```
String call(String ossl);
```

that is, every simulation service is required to have a method called "`call`" that takes one input string and returns one string. Both the input and the output strings have to follow the OSsL schema (§6.7). No invalid OSsL instances should be transmitted by the solver onto the network. In Table 7-2, we list the operations specified in the OShL WSDL document (currently only one).

Operation	**Description**
String call (String)	1st input is an OSsL instance for simulation input. Output is an OSsL instance for simulation output.

Table 7-2: Operations in OScL (currently only one).

We continue the example in the above OShL section, to illustrate the process of calling a simulation using the OS library below.

Imagine the entire objective function in problem (7-1) function is now calculated by a simulation called `sampleSimulation` shown in Figure 7-5. Note that it is also possible to have part of an objective or constraint function calculated by a simulation; thus the simulation becomes one internal tree node in an entire expression tree rather than the tree root. As explained in §2.8, three things have to be specified for the simulation: input, output, and the simulation's address. The simulation definition is specified in OSiL. This was discussed in detail in the OSiL section (§6.2) and the OSnL section (§6.3).

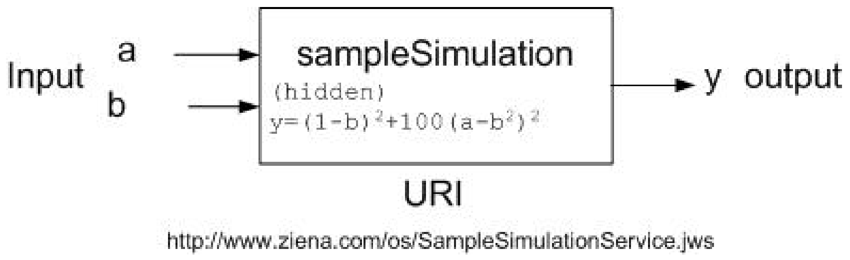


Figure 7-5: sampleSimulation with two inputs (a, b), one output (y) and an address (http://www.ziena.com/os/SampleSimulationService.jws).

Notice the sampleSimulation engine can provide more "services" than just calculating the output y from the two inputs a and b.

`sampleSimulation` can be represented using the OSiL `<simulation>` element:

```
<simulation name="sampleSimulation">
    <uri value="http://www.ziena.com/os/SampleSimulationService.jws"/>
    <OSsL>
        <input>
            <el name="a"/>
            <el name="b"/>
        </input>
        <output>
            <el name="y"/>
        </output>
    </OSsL>
</simulation>
```

With the `sampleSimulation` definition, we can write down the objective (idx = -1) and the constraint (idx = 0) of (7-1) as described in the OSiL instance (§6.2, §6.3). In the example below we pass x_0 to the simulation input a and x_1 to the simulation input b.

```
<OSiL…>
…
    <nl idx="-1">
        <sim name="sampleSimulation">
            <simInput inputName="a"><var idx="0"/></simInput>
            <simInput inputName="b"><var idx="1"/></simInput>
            <simOutput outputName="y"/>
        </sim>
    </nl idx="0">
    <minus>
        <plus><var idx="0"/><var idx="1"/></plus>
        <number value="100"/>
    </minus>
    <simulation name="sampleSimulation">
        <!--definition--> …
    </simulation>
    …
</OSiL>
```

When the Knitro solver service gets the OSiL instance, it uses the OS library to parse the input and output instances with no difference and solve the optimization problem as usual:

```
public class KnitroSolverService {
    public String solve(String osil){
        //read OSiL
        OSiLReader osilReader = new OSiLReader();
        OSiI inputInterface = osilReader.getStandardInputInterface(osil);

        //solve
        KnitroSolver knitro = new KnitroSolver();
        OSrI outputInterface = knitro.solve(inputInterface);

        //write and return OSrL
        OSrLWriter osrlWriter = new OSiLWriter();
        String osrl = osrlWriter.getOutputInterface (outputInterface);
    }//solve
                …
}//class KnitroSolverService
```

As the Knitro solver is a nonlinear solver, at each iteration, it uses the OS library to calculate the function value for the objective (idx = -1) and constraint (idx = 0) functions. Knitro does not care how the function values are obtained; at each iteration, it passes the variable values $[x_0, x_1]$ to the OS library to get the function values of indexes -1 and 0. So nothing changes for the solver either. What has changed are the function calculations inside the OS library. Below are some snippet examples of how the OS library does the function calculations. The OS library adopts the Objective-oriented philosophy using information hiding, inheritance, and most importantly polymorphism as described in detail in §4.1.2.

First Knitro calls the following method in the `OSiLReader` class:

```
public double calculateFunction(int rowIdx, double x[]){
    double dResult = calculateLinearFunction(rowIdx, x) + calculateNonlinearFunction(rowIdx, x);
    return dResult;
}//calculateFunction in OSiLReader
```

Knitro passes in the variable array $x[]$ and the `rowIdx`, -1 for the objective and 0 for the constraint. In the `calculateFunction` method, there are two parts: `calculateLinearFunction` and `calculateNonlinearFunction`.
In the above OSiL example, we list both functions using only nonlinear expressions (`<nl>`), so `calculateLinearFunction` returns 0. `calculateNonlinearFunction` then uses the a expression tree structure to further calculate the nonlinear function:

```
public double calculateNonlinearFunction(int rowIdx, double x[]){
    getNonlinearExpressions();
    OSExpressionTree exTree = (OSExpressionTree)(m_expressionTrees.get(rowIdx+""));
    return exTree.calculateFunction(x);
}//calculateNonlinearFunction in OSiLReader
```

The OSExpressionTree is a tree of operation nodes all of abstract type OSnLNode, including the root node: `m_treeRoot`. All concrete nodes that extend the abstract OSnLNode, implement the method calculationFunction(double x[]). The line `exTree.calculateFunction(x)` invokes the calculateFunction method on the `m_treeRoot`, which is from the class `OSExpressionTree`:

```
public double calculateFunction(double x[]){
    return m_treeRoot.calculateFunction(x);
}//calculateFunction in OSExpressionTree
```

In the constraint function the root is `minus`. The concrete `OSnLNode` that corresponds to `minus` is `OSnLNodeMinus`. The `m_treeRoot.calculateFunction(x)` function calls the `calculationFunction` in `OSnLNodeMinus`:

```
protected double calculateFunction(double[] x){
        m_dFunctionValue = m_mChildren[0].calculateFunction(x) -
m_mChildren[1].calculateFunction(x);
        return m_dFunctionValue;
}//calculateFunction in OSnLNodeMinus
```

`OSnLNodeMinus` subtracts the value of its second child (of type `OSnLNode`) from the value of its first child (again of type `OSnLNode`). What happens next is basically a recursive tree invocation using the polymorphism idea from Object-oriented Programming (OOP). A similar example on polymorphism is also explained in detail in the OOP section (§4.1.2).

In the objective function the root is `sim`. The concrete `OSnLNode` that corresponds to `sim` is `OSnLNodeSim`. The `m_treeRoot.calculateFunction(x)` function calls the `calculationFunction` in `OSnLNodeSim`, which is more complex than `OSnLNodeMinus`:

```
protected double calculateFunction(double[] x){
        //1. get simulation inputs from each <simInput> child of <sim>
        for(int i = 0; i < m_mChildren.length - 1; i++){
                OSnLNodeSimInput simInput = (OSnLNodeSimInput)(m_mChildren[i]);
                String sInputName = simInput.getInputName();
                . . .
        }
        //2. construct the OSsL simulation service input
        String sOSsLInput = XMLUtil.writeXMLElementToString(m_osslReader.getRootElement());
        . . .
        //3. instantiate an OS simulation agent to contact the remote simulation at the URI address
        //and get the result from the simulation service output (in OSsL)
        OSSimulationAgent osSimulationAgent = new OSSimulationAgent();
        osSimulationAgent.simulationAddress = m_sURI;
        String sOSsLOutput = osSimulationAgent.call(sOSsLInput);
        // 4. construct the result according to <simOutput> (last child) of <sim> from the returned
OSsL.
        //We know the result has to be in OSsL format as we called an OS simulation service.
        OSnLNodeSimOutput simOutput = (OSnLNodeSimOutput)(m_mChildren[m_mChildren.length -
1]);
        OSsLReader osslReader = new OSsLReader();
        osslReader.readString(sOSsLOutput);
        . . .
        m_dFunctionValue = simOutput.calculateFunction(x);
        . . .
        //5. return function value from the constructed simulation result.
        return m_dFunctionValue;
}//calculateFunction in OSnLNodeSim
```

Five steps are involved in `OSnLNodeSim` to get the value from the simulation services:

1. `OSnLNodeSim` gets simulation inputs from each <simInput> child of <sim>. The value of the `simInput` a is x_0 and the value of the `simInput` b is x_1:

```
<simInput inputName="a">
```

```
    <var idx="0"/>
</simInput>
<simInput inputName="b">
    <var idx="1"/>
</simInput>
```

2. `OSnLNodeSim` constructs an OSsL input for the sample simulation service using the `OSsLWriter` provided in the OS library. Suppose Knitro Solver passes in $x_0 = 1.2$ and $x_1 = 3.4$, the OSsL would look like:

```
<OSsL>
    <input>
        <el name="a">1.2</el>
        <el name="b">3.4</el>
    </input>
</OSsL>
```

3. `OSnLNodeSim` instantiates an OS simulation agent to contact the remote sample simulation service at the URI address. The simulation service output is in OSsL.

This step is the only step that involves communication. The communication should be carried out according to the OScL WSDL documents. `OSnLNodeSim` delegates an OS simulation agent to make the contact to the remote simulation service. The agent hides all the networking complexities. When `OSnLNodeSim` calls the method:

```
String sOSsLOutput = osSimulationAgent.call(sOSsLInput);
```

four communication steps are involved, which is similar to the 4 solver agent steps described in the above OShL section (§7.1). Briefly **simulation agent step 1** is encoding of the above constructed OSsL input (in 2.); **simulation agent step 2** is packing the encoded input and the `call` operation specified in OShL into a SOAP envelope; **simulation agent step 3** is sending the SOAP envelope over HTTP to the remote simulation service and wait for a response; and **simulation agent step 4** is decoding the result from the simulation service into a plain OSsL format.

4. `OSnLNodeSim` retrieves the result according to <simOutput> (last child) of <sim> from the decoded OSsL using the `OSsLReader` provided in the OS library:

```
<simOutput outputName="y"/>
```

5. `OSnLNodeSim` returns function value from the constructed simulation result.

At the simulation side, the simulation service also has to follow the OScL WSDL document. The sample simulation service code looks like:

```
public class SampleSimulationService {
    public String call(String ossl){
        //read OSsL
        OSsLReader osslReader = new OSsLReader();
        double a = Double.parseDouble(osslReader.getInputByName("a"))
```

```
            double b = Double.parseDouble(osslReader.getInputByName("b"))

            //run simulation
            double y = Math.pow((1-b), 2) + 100 * Math.pow(a – b * b);

            //write and return OSsL
            OSsLWriter osslWriter = new OSsLWriter();
            String[ ] outputNames = {"y"};
            String[ ] outputValues = {y+""};
            String ossl = osslWriter.setOutput({outputNames, outputValues);
            return ossl;
      }//call
}//class SampleSimulationService
```

As we can see, the sample simulation service implements all the operations required in the OShL WSDL document (listed in Table 7-2). There are mainly 3 steps in this operation:

1. Reading

```
OSsLReader osslReader = new OSsLReader();

double a = Double.parseDouble(osslReader.getInputByName("a"))
double b = Double.parseDouble(osslReader.getInputByName("b"))
```

The sample simulation service gets the OSsL string and use the `OSsLReader` class provided in the OS library to parse the OSsL instance into a set of input values (`double a, b`).

2. running simulation

```
double y = Math.pow((1-b), 2) + 100 * Math.pow(a – b * b);
```

The sample simulation basically calculates the function $(1-b)^2 + 100(a-b^2)^2$.

3. Writing and returning

```
OSsLWriter osslWriter = new OSsLWriter();
String[ ] outputNames = {"y"};
String[ ] outputValues = {y+""};
String ossl = osslWriter.setOutput({outputNames, outputValues);
return ossl;
```

The sample simulation service uses the `OSsLWriter` class provided in the OS library to write the OSsL result instance from an array of output names and output values. In our example the array sizes are 1, as there is only one output name `"y"` and one output value `y+""`. The sample simulation service then returns the OSsL result instance. Of course the sample simulation service has to make sure the OSsL result instance is valid. By using the OS library to construct the OSsL instance, the result should be atomically validated.

In the above 3 steps, we see that the sample simulation service does not need to worry about how the input OSsL instance received from the internet should be decoded. Neither does it need to worry about how to encode and send back the OSsL output to the client. This is because the sample simulation service is hosted by an OS Server in the same way that all other

OS services are hosted. The OS Server hides all the networking complexities from the solver service. We simply put the above sample simulation service code in a file called **`SampleSimulationService.jws`** and put the file in the **`os`** sub-directory relative to the OS Server's public root directory. Since in our example, we host the sample simulation service at http://www.ziena.com, thus the service address is http://www.ziena.com/**os/SampleSimulationService.jws**). What the OS Server does is similar to the four solver server steps described in the above OShL section (§7.1). Briefly **Simulation server step 1** is decoding the "SOAP over HTTP request" from the solver client that contains the `call` operation and the OSsL input; **Simulation server step 2** is invoking the sample simulation service on the `call` operation with the decoded OSsL input; **Simulation server step 3** is encoding the OSsL output returned by the sample simulation service into a SOAP envelope; and **Simulation server step 4** is returning the SOAP envelope over the HTTP transport to the client solver.

This completes the entire simulation "call" process according the Optimization Services call Language. All the networking complexities are hidden and taken care by the OS library. The OS library also provides parsers to read and write standard instances. All that a solver does is to use the OS parser library for reading and writing OSsL instances, delegate the simulation agent to call the simulation and get the function value. All that a simulation service does is to expose the standard interface, implement all the required operations in the interface listed in OScL, let the OS Server take care of the underneath networking, and use the OS parser library to read and write OSsL instances.

7.3 Optimization Services flow Language (OSfL)

The OSfL document is at http://www.optimizationservices.org/wsdl/OSfL.bpel. In §5.3, we described various Optimization Services processes. OSfL is used to predefine certain standard process flows. Unlike most other communication related OSxL's, OSfL is an XML document written in BPEL (Business Process Execution Language [91]). OSfL is an optional *non-binding* specification that is provided as a reference and guidance to facilitate *implementation* of OS components by the developers. Developers do not have to use the predefined flows in the OSfL BPEL document. Either they can define their own flow logic or it is not even necessary for them to use any flow languages, as the logic of invoking various Optimization Services can just be hard coded. It is, however, highly recommended that developers look into the BPEL standard when building a state-of-the-art Optimization Services component (e.g. modeling language environment, solver services). Implementing an industry standard for orchestrating Optimization Services will not only speed up the development and deployment of new optimization processes but will also make the current processes easily maintainable. This section gives an overall description with some simple examples. As BPEL itself is still a new standard, OSfL will be evolving along with the development of BPEL.

The business motivation behind a standard Web service flow orchestration is the same motivation behind the use of any proprietary EAI (Enterprise Application Integration) [19][76] solution: to increase productivity, to reduce costs, and to improve service levels through automation. Traditionally the process integration is achieved by hard coding extra embedded logic inside of heterogeneous applications such as CRM (Customer Relationship Management), ERP (Enterprise Resource Planning) or SCM (Supply Chain Management) and modifying the interfaces to make the applications work with each other (Figure 7-6). The development, testing, and deployment efforts required for the changes make the entire integration process very complex and expensive.

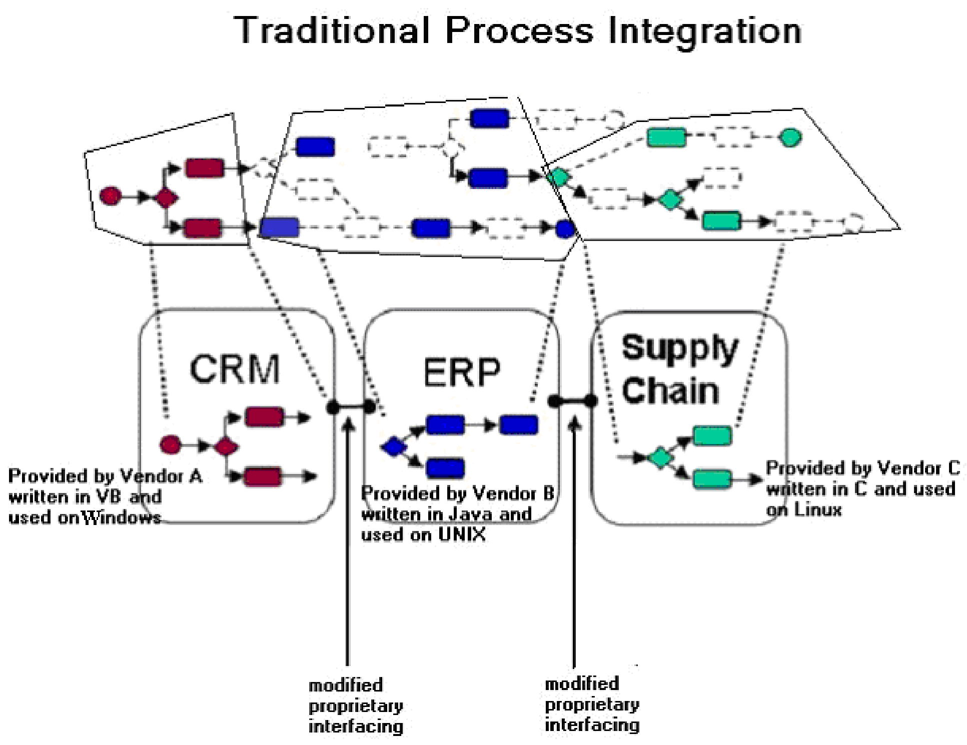

Traditional Process Integration
CRM
ERP
Supply Chain
Provided by Vendor A written in VB and used onWindows
Provided by Vendor B written in Java and used on UNIX
Provided by Vendor C written in C and used on Linux
modified proprietary interfacing
modified proprietary interfacing

Figure 7-6: Traditional process integration.

The BPEL specification [90] recently released by OASIS is positioned to be the Web services standard for process flow composition. It is the result of a cross-company initiative that includes IBM [61], Microsoft [80] and Oracle [94]. In terms of language features, BPEL is a convergence of IBM's Web Service Flow Language (WSFL [63]) and Microsoft's XLANGE [83], which is the orchestration language for Microsoft's BizTalk server [81]. Both WSFL and XLANG are now superseded by BPEL. Many major companies are starting to provide BPEL process engine software that handles flow logics specified in any standard BPEL document. Figure 7-7 shows an example of Oracle's BPEL process engine [94].

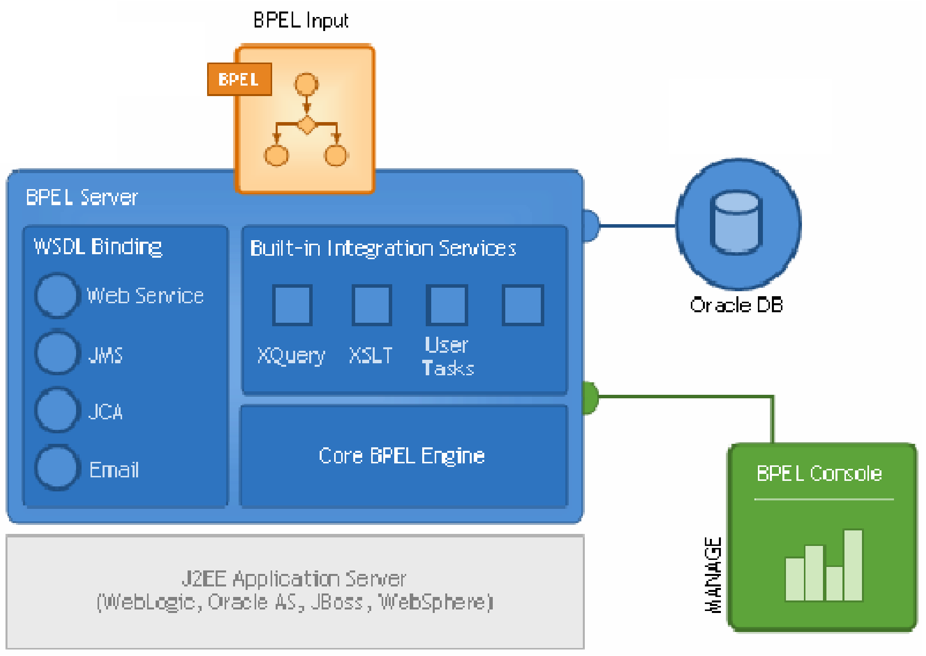


Figure 7-7: Oracle's BPEL process engine.

Optimization Services process orchestration is not as complex as the enterprise business process orchestration. Therefore it is much easier for Optimization Service developers to hard code and maintain various process logics in their software. But still some, especially the commercial developers, can benefit significantly from the standardized integration interface and standardized language for integration and process automation. Optimization processes exported to BPEL will be able to execute in a variety of standards-compliant process engines, offering customers more choices and the ability to mix and match tools.

A typical Optimization Services process flow chart that corresponds to the BPEL Input in Figure 7-7 is shown in Figure 7-8. The flow chart in the figure corresponds to a flow of solving an optimization problem that starts and ends at the modeler's application side.

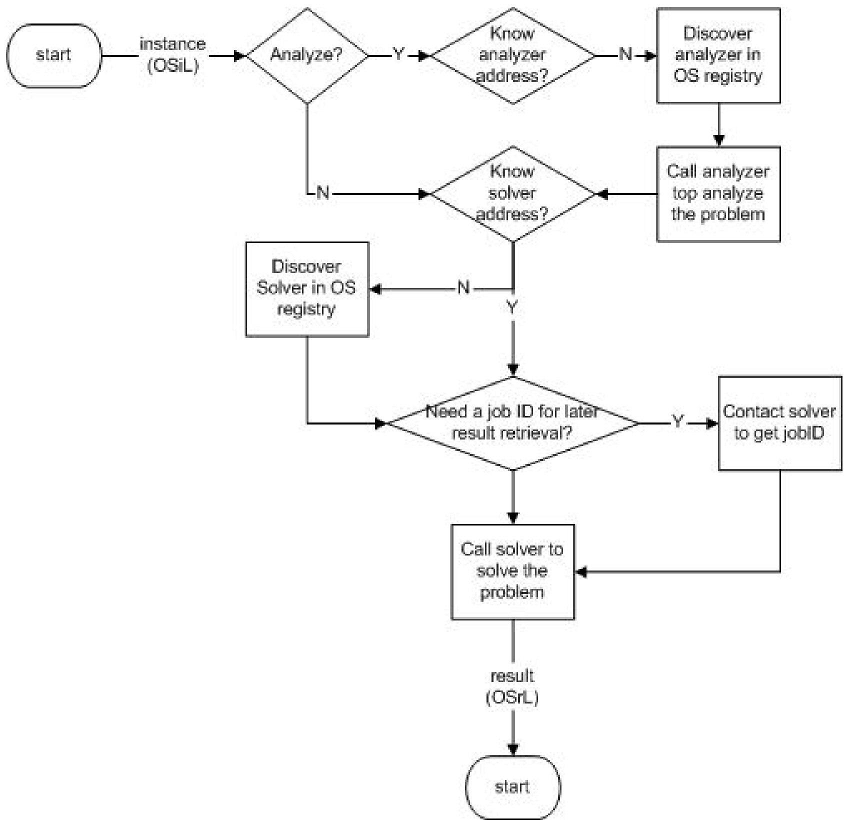

start
instance (OSiL)
Analyze?
Y
N
Know analyzer address?
Discover analyzer in OS registry
Call analyzer top analyze the problem
Know solver address?
Discover Solver in OS registry
Need a job ID for later result retrieval?
Contact solver to get jobID
Call solver to solve the problem
result (OSrL)
start

Figure 7-8: A typical Optimization Services process flow chart.

There can be other kinds of flows. For example, a flow can be as simple as one process. If all that a solver service provides involves one single optimization process that receives an OSiL input and returns an OSrL output, there is still an advantage in writing the process in a BPEL document and letting the BPEL process engine execute the optimization job. The biggest advantage is probably that the solver service can use the queuing service provided by the BPEL engine. Almost all the BPEL engines provide queuing management service that is independent of the queue server implementation the process engines use. The queuing logics as well everything else specified in BPEL is language and platform neutral.

The flow chart of Figure 7-8 would be written in the Optimization Services flow Language BPEL document. BPEL provides an open programming abstraction for Optimization Services developers to create complex optimization processes, such as service discovery, instance analysis, solver hookup, and simulation invocation into an end-to-end process flow. The programming abstraction is platform and language independent and fully supports features such as Web services invocation, data manipulation, exception handling, activity nesting and sequencing, process parallelization and job termination.

Another noticeable mechanism in BPEL is its built-in support for asynchronous interactions. Early in this section we specified that all the OS communications should use the RPC style invocation style. Although RPC is a blocking call based on the request and response model, we described how the actual implementation can launch a separate process or thread to issue the RPC call so that the user of the application can go on with other tasks. But as a much better alternative, the application can build on a BPEL process engine to do the entire job more effectively and efficiently. All that the application does is to pass the BPEL process engine the standard OSfL BPEL document. As the BPEL language is a universally accepted standard, any BPEL process engine, whether from Microsoft or IBM or anywhere else, can take and handle exactly the same OSfL input. Different companies may use their own proprietary message queuing software to manage the underlying asynchronous interactions, for example MSMQ [82] from Microsoft and MQSeries [65] from IBM, but this is all hidden away from the standard BPEL input that is passed in.

Technologically, BPEL leverages on other Web Services standards such as SOAP and WSDL for communication interface description. BPEL describes the process interfaces in WSDL so that they can be easily integrated into other processes or applications. From a user's point of view, BPEL is just a meta-process that is no different from other single processes and it can be invoked like any Web service as shown in Figure 7-9.

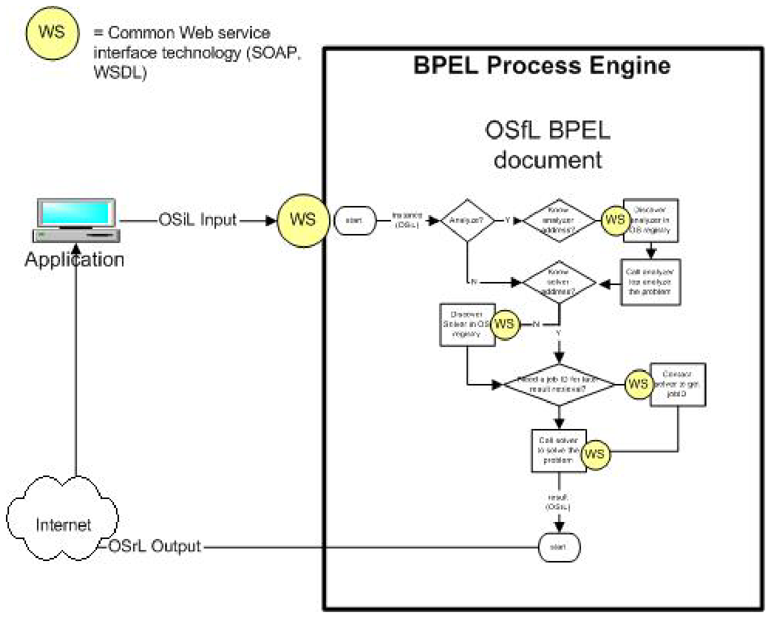
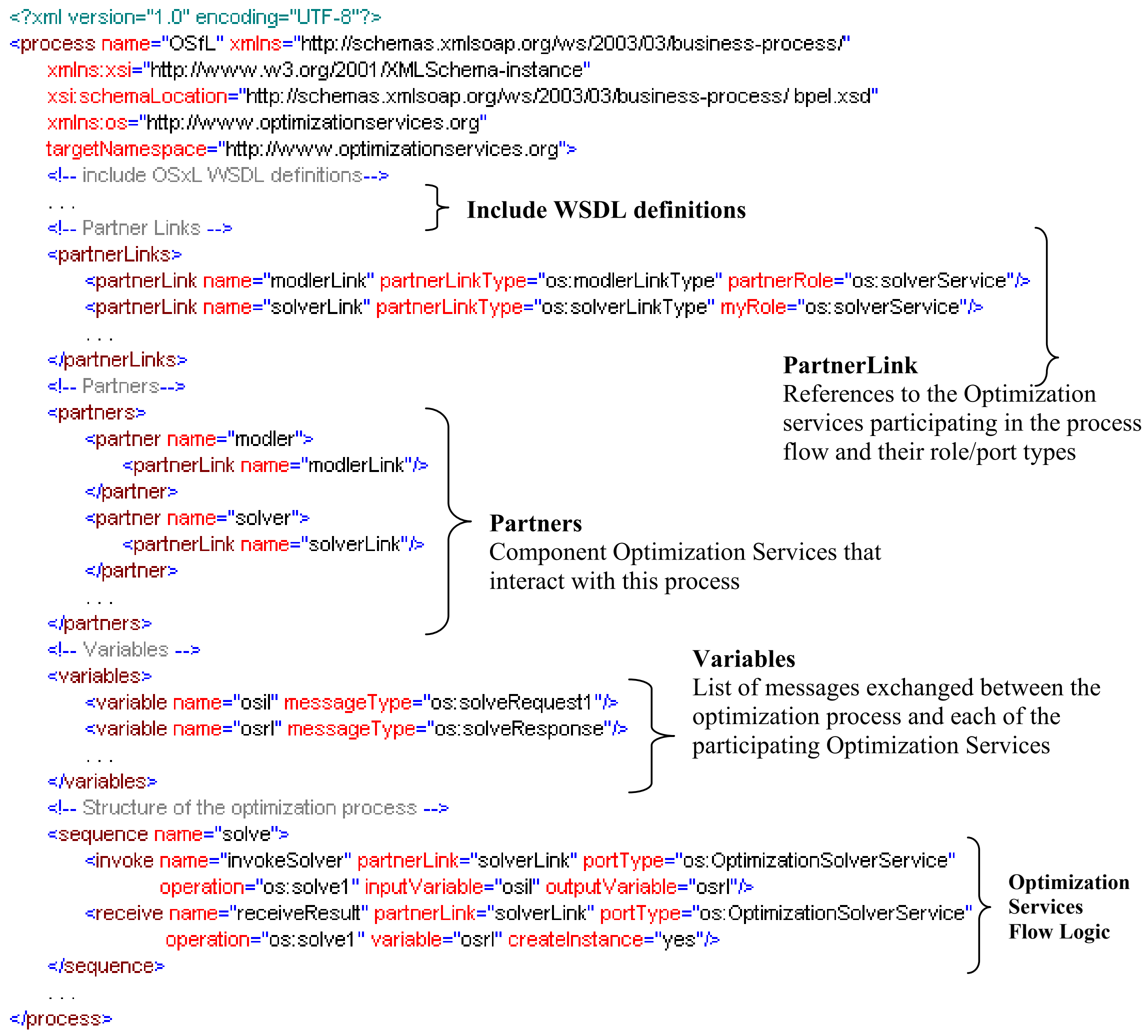

WS
= Common Web service interface technology (SOAP, WSDL)
BPEL Process Engine
OSfL BPEL document
Application
OSiL Input
WS
start
instance (OSiL)
Analyze?
Y
N
Know analyzer address?
Discover analyzer in OS registry
Call analyzer to analyze the problem
Know solver address?
Discover Solver in OS registry
N
Y
Need a job ID for later result retrieval?
Contact solver to get jobID
Call solver to solve the problem
result (OSrL)
start
Internet
OSrL Output

Figure 7-9: Calling BPEL process engine as a Web service, which in turn calls various Optimization services according to the OSfL BPEL document in Figure 7-8.

Figure 7-10 is a simple anatomy of OSfL shown in BPEL. In reality BPEL documents are constructed graphically by BPEL designers such as Microsoft's Office charting tool Visio.



```
<?xml version="1.0" encoding="UTF-8"?>
<process name="OSfL" xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://schemas.xmlsoap.org/ws/2003/03/business-process/ bpel.xsd"
    xmlns:os="http://www.optimizationservices.org"
    targetNamespace="http://www.optimizationservices.org">
    <!-- include OSxL WSDL definitions-->
    . . .
    <!-- Partner Links -->
    <partnerLinks>
        <partnerLink name="modlerLink" partnerLinkType="os:modlerLinkType" partnerRole="os:solverService"/>
        <partnerLink name="solverLink" partnerLinkType="os:solverLinkType" myRole="os:solverService"/>
        . . .
    </partnerLinks>
    <!-- Partners-->
    <partners>
        <partner name="modler">
            <partnerLink name="modlerLink"/>
        </partner>
        <partner name="solver">
            <partnerLink name="solverLink"/>
        </partner>
        . . .
    </partners>
    <!-- Variables -->
    <variables>
        <variable name="osil" messageType="os:solveRequest1"/>
        <variable name="osrl" messageType="os:solveResponse"/>
        . . .
    </variables>
    <!-- Structure of the optimization process -->
    <sequence name="solve">
        <invoke name="invokeSolver" partnerLink="solverLink" portType="os:OptimizationSolverService"
                operation="os:solve1" inputVariable="osil" outputVariable="osrl"/>
        <receive name="receiveResult" partnerLink="solverLink" portType="os:OptimizationSolverService"
                 operation="os:solve1" variable="osrl" createInstance="yes"/>
    </sequence>
    . . .
</process>
```

Include WSDL definitions

PartnerLink
References to the Optimization services participating in the process flow and their role/port types

Partners
Component Optimization Services that interact with this process

Variables
List of messages exchanged between the optimization process and each of the participating Optimization Services

Optimization Services Flow Logic

Figure 7-10: Anatomy of the OSfL BPEL document.

The BPEL language defines a process by composing a set of existing services that are described as a collection of WSDL definitions. The composition (`<process>` root element) indicates how each service interface fits into the overall flow. The `<partner>` elements are basically the component services with which the process interacts. They can either be invoking (or client) partners and/or invoked (or server) partners. After defining the `<partner>` elements, we need to logically tie a BPEL process to an existing Web service via the `<partnerLink>` element. Every Web service used in a BPEL process requires a `<partnerLink>` describing which set of Web service operations (`<portType>` elements) is used in the BPEL process. After all the logic links are specified, `<partner>` elements are then used in the activities (e.g. `<invoke>`, `<receive>`). the `<variable>` elements are used to specify the list of messages that are exchanged between the BPEL processes and any participating.

Each step in the process is called an *activity*. Activities can include invoking a service (`<invoke>`, Figure 7-10), receiving a message from a client (`<receive>`, Figure 7-10), generating a reply to the client (`<reply>`), waiting for some time (`<wait>`), copying data (`<assign>`), throwing an exception (`<throw>`), catching and handling exceptions (`<scope>`, `<faultHandlers>`, `<catch>`), handling events (`<eventHandlers>`), compensating actions for certain irreversible actions (`<compensation>`), terminating a process (`<terminate>`) or doing nothing (`<empty>`). These are called *primitive* activities as they can be combined into more *complex* activities through some *structure* elements such as `<sequence>` for ordering, `<switch>` for branching, `<while>` for looping, `<pick>` for selection, `<flow>` parallelization, and `<link>` for parallelization order constraints.

CHAPTER 8 OPTIMIZATION SERVICES REGISTRY

The address of the OS registry service is:

http://www.optimizationservices.org/os/OSRegistryService.jws.

To locate services in a decentralized serviced-oriented distributed system, software agents coordinate with each other and with registries. Some registries are general ones that keep information of all kinds of Web services, such as Universal Description, Discovery and Integration (UDDI, §4.8). Others are specialized ones like the Optimization Services registry that only serves registration and discovery of Optimization Services.

The OS registry knows all the registered services (solvers, analyzers, simulations) on the OS network by keeping their metadata information. "Metadata" means that the registry contains information *about* the software, but not the software itself. The OS registry can be viewed as a "light" weight server in that no registered services are actually executed by this registry; instead clients directly contact the services in a peer-to-peer mode (Figure 8-1).

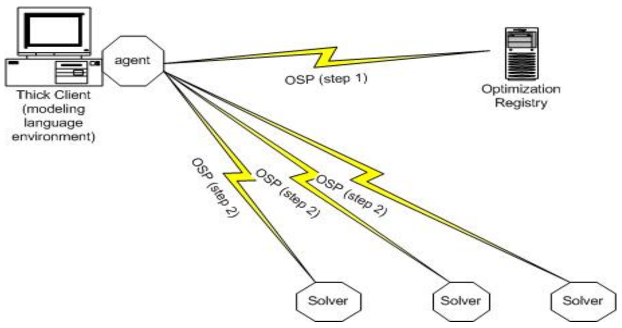


Figure 8-1: The optimization registry architecture.

The fundamental differences between an optimization server and the OS registry have been explained in §2.5. The advantages of a decentralized Service-oriented architecture (§4.6) have been elaborated throughout the thesis. It is our vision that a decentralized architecture can better promote research and development in optimization.

The Optimization Services registry serves the function of a search engine. But unlike the Internet search engines, there has to be a unique registry on the entire OS network to ensure Quality of Service (see §1.2.4). The OS registry operators make sure (e.g. through advertisement) that all communication agents know or can easily find out where the OS registry is. When a certain query is sent to the OS registry from a client, the OS registry returns the locations of the matched OS services and the client contacts each service directly at the provided location. On the opposite side of the "discovery" process is the "join" process. It is the

OR software developer's responsibility to submit the required information to, and get approved by, the OS registry, possibly through a mixture of automatic and manual procedures. The fundamental differences between Internet search engines and the OS registry have been explained in §1.2 and §2.5.

In terms of standardization, OS-registry related protocols do not face as much imminent pressure of universal acceptance as the non-registry related protocols. There is only one public registry on the entire Internet and there are much fewer registry developers compared with the other OS developers. In the following sections, we more descriptively illustrate the registry-related protocols using an example of the "Impact" solver service in order to give a general idea of how we designed our OS registry. There are mainly two categories of registry-related OSP protocols; one deals with representation (OSeL, OSpL, OSbL, OSyL, OSqL, OSuL) and the other deals with communication (OSjL, OSkL, OSdL, OSvL); all the 10 sub-protocols are explained in the following sections of this chapter.

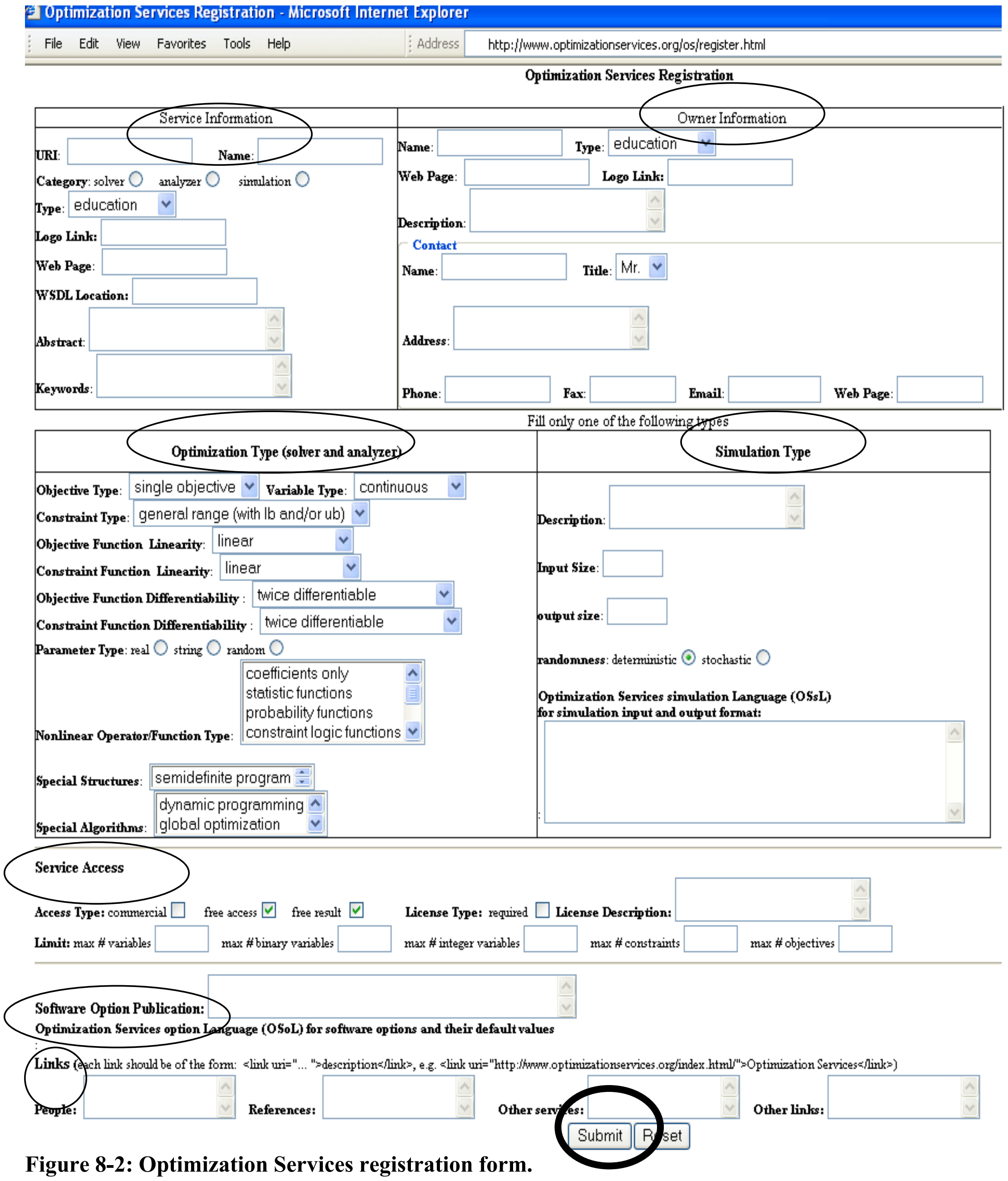
To ensure that the OS registry only sends addresses of the services (especially solvers) that are of reasonably high quality, regulations are imposed when an OS-compatible service is to be registered in the OS registry. The following three OSP protocols are designed to make sure a solver is and continues to be well-described, live, reliable, and robust. Information about registered services in the OS registry includes three main categories:

1. Entity information that is reported by service developers at registration, e.g. service and owner information, solver or simulation types and service locations. We call this category of information "entity" information to emphasize the information is relatively static. This is addressed by the Optimization Services entity Language (§8.1).
2. Real-time process information that is either reported by the registered service ("push") or detected by the OS registry ("pull"), e.g. how many optimization jobs are at the service server. We call the information "process" information to emphasize the information is dynamic. This is addressed by the Optimization Services process Language (§8.2).
3. Benchmark information that is gathered separately by auxiliary benchmarker tools designated by the OS registry, e.g. general solver ratings and performance profiles. This is addressed by the Optimization Services benchmark Language (§8.3).

All the three types of information are kept in an XML database of the OS registry. As the OS registry is an open registry, to facilitate communication (especially discovery) with the registry, the structure and contents of the OS database are made public just like a yellow pages directory. The structure and contents in the OS database are addressed by Optimization Services yellow-page Language (§8.4). The other sections deal with various interactions with the OS registry.

8.1 Optimization Services entity Language (OSeL, representation)

The OSeL schema is located at http://www.optimizationservices.org/schemas/OSeL.xsd. OSeL is an XML specification of entity information used to describe the *static* information of an optimization service. However, to register a service, the registrant usually goes to the Optimization Services registry Web site to fill out the form shown in Figure 8-2. Of course the registrant can also submit the OSeL description directly.


Optimization Services Registration - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://www.optimizationservices.org/os/register.html

Optimization Services Registration

Service Information

URI: Name:
Category: solver ○ analyzer ○ simulation ○
Type: education
Logo Link:
Web Page:
WSDL Location:
Abstract:
Keywords:

Owner Information

Name: Type: education
Web Page: Logo Link:
Description:
Contact
Name: Title: Mr.
Address:
Phone: Fax: Email: Web Page:

Fill only one of the following types

Optimization Type (solver and analyzer)

Objective Type: single objective Variable Type: continuous
Constraint Type: general range (with lb and/or ub)
Objective Function Linearity: linear
Constraint Function Linearity: linear
Objective Function Differentiability: twice differentiable
Constraint Function Differentiability: twice differentiable
Parameter Type: real ○ string ○ random ○
Nonlinear Operator/Function Type: coefficients only, statistic functions, probability functions, constraint logic functions
Special Structures: semidefinite program
Special Algorithms: dynamic programming, global optimization

Simulation Type

Description:
Input Size:
output size:
randomness: deterministic [✓ ●] stochastic ○
Optimization Services simulation Language (OSsL) for simulation input and output format:

Service Access

Access Type: commercial ☐ free access ☑ free result ☑ License Type: required ☐ License Description:
Limit: max # variables max # binary variables max # integer variables max # constraints max # objectives

Software Option Publication:
Optimization Services option Language (OSoL) for software options and their default values

Links (each link should be of the form: <link uri="... ">description</link>, e.g. <link uri="http://www.optimizationservices.org/index.html/">Optimization Services</link>)
People: References: Other services: Other links:

Figure 8-2: Optimization Services registration form.

Notice the 6 main categories are circled in the figure: service information, owner information, optimization (or simulation) type, service access, service options (e.g. algorithm directives) and links to other places. When the registrant clicks the **submit** button, the entered data is organized into an OSeL file and sent to the OS registry using the communication specified by the OSjL (Optimization Services join Language, §8.5) WSDL document.

An OSeL XML example of a hypothetical "Impact" GMIP solver service looks like the following, with the 6 major categories highlighted:

```
<?xml version="1.0" encoding="UTF-8"?>
<OSeL xmlns="os.optimizationservices.org" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="os.optimizationservices.org http://www.optimizationservices.org/schemas/OSeL.xsd">
    <service>
        <uri>http://www.impactservice.net/impactGMIP.jws</uri>
        <name>Impact Generalized Mixed Integer Solver</name>
        <category>solver</category>
        <type>education</type>
        <publicationDate>2005-04-06</publicationDate>
        <abstract>Generalized mixed integer nonlinear convex solver</abstract>
        <description>
            <general>ImpactGMIP is a  generalized mixed integer nonlinear convex solver</general>
            <software>Impact </software>
            <hardware>Pentium Intel 4, Dell, Linux Enterprise </hardware>
            <algorithm>natural heuristic way of geenerating braching hyperplanes</algorithm>
            <other>developer: Wayne Sheng</other></description>
        <webPage>http://www.impactservice.net/impactGMIP.html</webPage>
        <wsdlLocation>http://www.impactservice.net/impactGMIP.jws?wsdl</wsdlLocation>
        <logoImageLink uri="http://www.impactservice.net/images/impactGMIP.jpeg">impact
GMIP</logoImageLink>
        <keyWords><key>mixed integer nonlinear programming</key><key>Interior Point
Methods</key></keyWords>
    </service>
    <owner>
        <name>Impact</name>
        <primaryType>education</primaryType>
        <mainWebPage uri="http://www.impactservice.net">Impact</mainWebPage>
        <description>Impact is for Inegrated Math Programming Advanced Computational Tool</description>
        <logoImageLink uri="http://www.impactservice.net/images/impact.jpeg">Impact</logoImageLink>
        <contact>
            <name>Sanjay Mehrotra</name><title>Professor</title>
            <address>IEMS, Northwestern University, 2145 North Sheridan Road, Evanston, IL 60208-3119
</address>
            <phone>8474913155</phone><fax>8474918005</fax><email>mehrotra@iems.northwestern.edu
</email>
            <webPage uri="http://users.iems.nwu.edu/~mehrotra/">Mehrotra's Web page</webPage>
        </contact>
    </owner>
    <optimizationType>
        <objectiveType>singleObjective</objectiveType>
        <variableType>mixedInteger</variableType>
        <constraintType>generalRange</constraintType>
        <linearity><objective>convexNonlinear</objective>
<constraints>convexNonlinear</constraints></linearity>
        <differentiability>
            <objective>twiceDifferentiable</objective>
            <constraints>twiceDifferentiable </constraints></differentiability>
        <parameterType><real/></parameterType>
```

```
            <functionType><general/></functionType>
            <specialStructure/>
            <specialAlgorithm/>
        </optimizationType>
        <serviceAccess commercial="false" freeAccess="true" freeResult="true">
            <licenseType licenseIDRequired="false"/>
            <limit><maxVariables>1000000</maxVariables>
                  <maxBinaryVariables>1000</maxBinaryVariables>
                  <maxIntegerVariables>500</maxIntegerVariables>
                  <maxConstraints>1000000</maxConstraints><maxObjectives>1</maxObjectives>
            </limit>
        </serviceAccess>
        <serviceOptionsAndDefaultValues>
            <general serviceName="Impact Generalized Mixed Integer Solver"
                        serviceAddress="http://www.impactservice.net/impactGMIP.jws">
                <maximumTime value="6000"/>
            </general>
            <other optionName="preprocess" value="true">preprocessing before solving it</other>
            <other optionName="arbitraryPreciseness" value="false">Algorithm based on arbitrary precise
numbers</other>
        </serviceOptionsAndDefaultValues>
        <links>
            <people>
                <link uri="http://users.iems.nwu.edu/~maj/">Jun Ma</link>
                <link uri="mailto://h-sheng@northwestern.edu">Wayne Sheng</link>
            </people>
            <references><link uri="http://www.optimizationservices.org">Optimization Services</link></references>
            <otherServices><link uri="http://www.impactservice.net/impactLP.jws">Impact
LP</link></otherServices>
        </links>
</OSeL>
```

The OSeL schema of the example and its 6 children are shown in Figure 8-3.

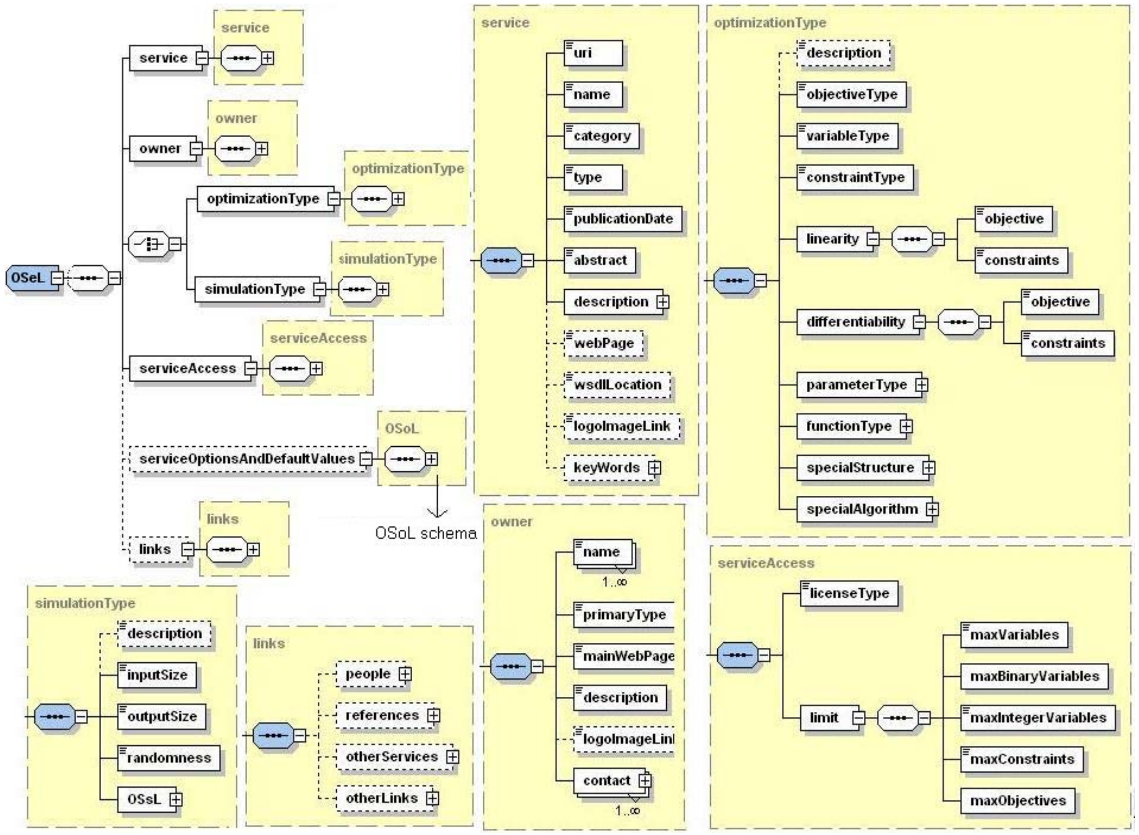


Figure 8-3: `<OSeL>` root element in OSeL and it 6 main category elements.

The 6 major "category" elements are listed around the <OSeL> element schema in the figure. The `<service>` element gives the information about the registered Optimization Service. Each OSeL document is identified by a unique `uri`, the first child element of `<service>`; this is where the service should be invoked. `<owner>` provides information about the people or companies who register the service. The `<optimizationType>` (or `<simulationType>`) element provides optimization (or simulation) related descriptions. Notice OSeL does not categorize an optimization solver by a specific type; rather it breaks down the type into several subtypes that include `objectiveType`, `variableType`, `constraintType`, `linearity`, `parameterType`, `functionType`, `specialStructure`, and `specialAlgorithm`. OSeL does not intend to combine the various sub-types into a specific type (e.g. mixed integer, linear, deterministic solver). As each subtype contains many values, the number of combinations can be extremely large and leads to poor scalability in practice. The `<simulationType>` element is relatively simple, and contains information about a simulation's input and output size and format (in OSsL, §6.7) and whether it is deterministic or stochastic. The `<serviceAccess>` element contains information about whether the service is commercial, whether it is free to access, and whether the result is free to retrieve. It also contains the limits (if any) of using the service in terms of the problem size (e.g. maximum variable number). The `<serviceOptionsAndDefaultValues>` element lets the service registrant list the software options and default values in Optimization Services option Language (OSoL, §6.5). The `<links>` element allows linking to relevant people, references and other services.

8.2 Optimization Services process Language (OSpL, representation)

The OSpL schema is located at http://www.optimizationservices.org/schemas/OSpL.xsd. Besides *static* OSeL entity information, the Optimization Services registry also keeps *dynamic* process information (such as number of jobs being solved) using Optimization Services process Language. Unlike the OSeL information that is submitted at registration, OSpL information is collected at run time. During runtime, the Optimization Services registry periodically "knocks" on the registered services (Optimization Services knock Language, §8.6) to make sure they are live and running and to collect the OSpL information. It is also possible that the services themselves push the OSpL information to the OS registry.

The decentralized Optimization Services system leaves open the question of how optimization "jobs" are scheduled to run on available solver services. Centralized schemes, such as that used by the NEOS server, may maintain one queue for each solver/format combination, along with a list of the workstations on which each solver can run. In Optimization Services, we want to maintain this scheduling control, while at the same time making the scheduling decisions more distributed. Optimization Services process Language can play an important role in dynamic optimization scheduling in a decentralized environment.

An OSpL example of the "Impact" GMIP solver service looks like the following:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<OSpL xmlns="os.optimizationservices.org" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="os.optimizationservices.org http://www.optimizationservices.org/schemas/OSpL.xsd">
    <serviceURI>http://www.impactservice.net/impactGMIP.jws</serviceURI>
    <serviceName>Impact Generalized Mixed Integer Solver</serviceName>
    <time>2004-04-17T15:50:04Z</time>
    <status busy="true" accepting="true">The server is currently busy but can still accept new jobs</status>
    <statistics>
        <totalJobs>3</totalJobs>
        <timeLastJobSolved>2004-04-17T15:32:12Z</timeLastJobSolved>
        <timeLastJobTaken>13.5</timeLastJobTaken>
    </statistics>
</OSpL>
```

The OSpL schema of the example and its 6 children are shown in Figure 8-4.

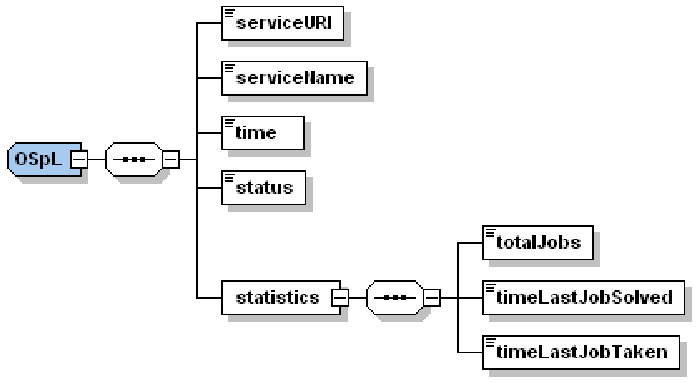


Figure 8-4: `<OSpL>` root element in OSpL.

In an `<OSpL>` element, `<serviceURI>` and `<serviceName>` should be the same as those listed in the registry. As OSpL is about dynamic information, the `<time>` element shows how recent the process information is. The time should be of the XML schema `xs:date` format. We require using the standard UTC time (Coordinated Universal Time, or sometimes called "Greenwich Mean Time") as services can be distributed all over the world. In the above "Impact" example (2004-04-17T15:50:04Z), we append the letter "Z" at the end according to the `xs:date` format to show that the time is UTC time. The `<status>` element has two boolean attributes, `busy` and `accepting`. In the above example, although the Impact solver service is busy, the service still accepts new optimization jobs. Currently the `<statistics>` element contains three children, `<totalJobs>`, `<timeLastJobSolved>` and `<timeLastJobTaken>`. The `<totalJobs>` element is the total number of jobs currently at the service that are being solved or waiting to be solved. `timeLastJobSolved` is the time the last job is solved, again using UTC time in the `xs:date` format. `timeLastJobTaken` is the time in minutes the last job took. The OS registry collects the status and statistics information for various purposes such as service benchmarking, better scheduling and future research.

8.3 Optimization Services benchmark Language (OSbL, representation)

The OSbL schema is located at http://www.optimizationservices.org/schemas/OSbL.xsd. OSbL is a specification of benchmark information on each optimization service. OSbL is the third and last piece of information (along with OSeL and OSpL in the previous two sections) that the OS registry keeps. The OS registry publishes all the three pieces of information on its corresponding Web site.

The availability of more than one service (especially solvers) for many classes of problems makes the OS registry an obvious choice as a benchmarking facility. It can potentially be useful both in choosing a solver for a particular application and in comparing solvers generally. One certain thing is that the benchmarking should be independent of any claims or statistics made by individual service providers.

There are significant barriers to achieving these potentials, however, which motivate some derived research. Someone who has developed a new model, but who is not sure which of the several applicable solver services to apply, is often advised that the only way to be sure is to carry out some test runs on typical problem instances. The straightforward way to do this is to send several test instances to each candidate solver service. But benchmarking on only a few related instances can be misleading. Furthermore, if different services are not on comparable machines under comparable conditions, the results may say little about the relative efficiency of the solver algorithms. The results may say more about the reliability of the solver services, but even so they may be distorted by differences in the memory available on the workstations devoted to different solver services, or by differences in limits imposed by the providers of services. There is not necessarily any obvious way to compensate for the differences between runs, moreover, because in general each solver service may be selected by the OS registry according to the load at the time a job is submitted. Deciding on an appropriate benchmarking methodology is related to other concurrent researches at NEOS; see [30] [31] and the NEOS benchmark solver at:

http://www-neos.mcs.anl.gov/neos/solvers/MULTI:BENCHMARK-AMPL/

The Performance World Forum from GAMS World at http://www.gamsworld.org/performance/ is also good site for discussion and dissemination of information and tools about all aspects of performance testing of solvers for mathematical programming problems. Possible collaboration on Optimization Services benchmarking can be established with these concurrent project.

For all the aforementioned issues, OSbL uses a relatively safe rating system based on a set of performance scores on the base of 100 as shown in the OSbL schema in Figure 8-5.

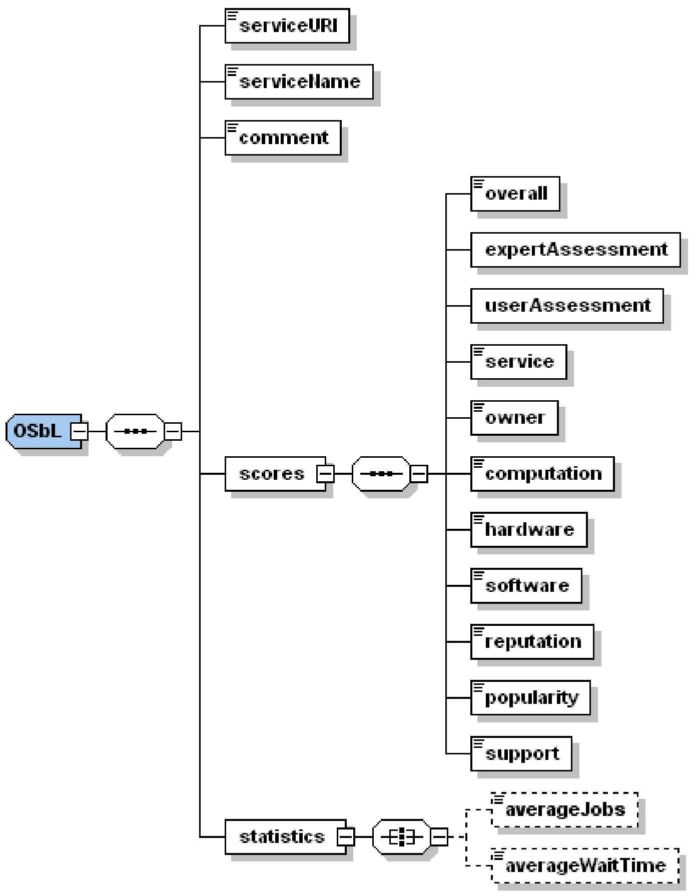


Figure 8-5: `<OSbL>` root element in OSbL.

In an `<OSbL>` element, `<serviceURI>` and `<serviceName>` should be the same as those listed in the registry. There is a general `<comment>` given by the OS registry for extra explanation. Besides the first `<overall>` child score element, the `<scores>` element also contains scores on a set of sub-criteria, such as software, hardware, and support. Besides the scores, the `<statistics>` element is for extra references; the data on `averageJobs` and `averageWaitTime` are based on the `totalJobs`, `timeLastJobSolved` and `timeLastJobTaken` data in the OSpL information (Figure 8-4). An example of the "Impact" solver service is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<OSbL xmlns="os.optimizationservices.org" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="os.optimizationservices.org http://www.optimizationservices.org/schemas/OSbL.xsd">
    <serviceURI>http://www.impactservice.net/impactGMIP.jws</serviceURI>
    <serviceName>Impact Generalized Mixed Integer Solver</serviceName>
    <comment>An outstanding solver service!</comment>
    <scores>
        <overall>90</overall>
        <expertAssessment>100</expertAssessment>
        <userAssessment>95</userAssessment>
        <service>85</service>
        <owner>100</owner>
        <computation>100</computation>
        <hardware>50</hardware>
        <software>100</software>
        <reputation>95</reputation>
        <popularity>75</popularity>
        <support>83</support>
    </scores>
    <statistics>
        <averageJobs>1.34</averageJobs>
        <averageWaitTime>1.5</averageWaitTime>
    </statistics>
</OSbL>
```

8.4 Optimization Services yellow-page Language (OSyL, representation)

The OSyL schema is located at http://www.optimizationservices.org/schemas/OSyL.xsd. At the core of our Optimization Services registry is a database, and we use a more expressive XML-based native database as versus a relational database. The organization of the native XML database is specified by Optimization Services yellow-page Language (OSyL).

One immediate benefit of using a native XML database is that we do not have to worry about mapping XML to some other data structure. We just insert the data (e.g. OSeL) as XML and retrieve it as XML. We also gain a lot of flexibility through the semi-structured nature of XML and the schema independent model used by all of the XML-enabled database engines. Most of the native XML databases are also open source and of production quality. The OS registry, by its nature as storage for registered Optimization Services, is much smaller compared with large enterprise databases used in daily production and operations, therefore a native XML database should fit the OS registry both in terms effectiveness and efficiency.

Another main advantage is that the database information of the OS registry, i.e. the OSyL file, can easily be made open and published on the OS registry's Web site for public reference. Since the OSyL file is in XML, the client can use the standard XQuery language discussed in §4.4 to retrieve any information, specifically the location of a desired service. Using XQuery to query the database is a built-in function in the Optimization Services query Language (OSqL, §8.7). For more overview of XML databases, see [6][36].

With the information of OSeL, OSpL, and OSbL discussed in the previous three sections, the schema of OSyL looks extremely simple as shown in Figure 8-6. After the first <description> element for a general database description and the second <news> element for latest OS registry database news, OSyL is then a sequence of [`OSeL`, `OSpL`, `OSbL`] triplets, with each triplet corresponding to a `service`. OSeL (entity, required) is submitted at registration time; OSpL (process, optional) is collected at run time, and (OSbL, optional) is inserted after benchmarking by the OS registry or its designated benchmarker.

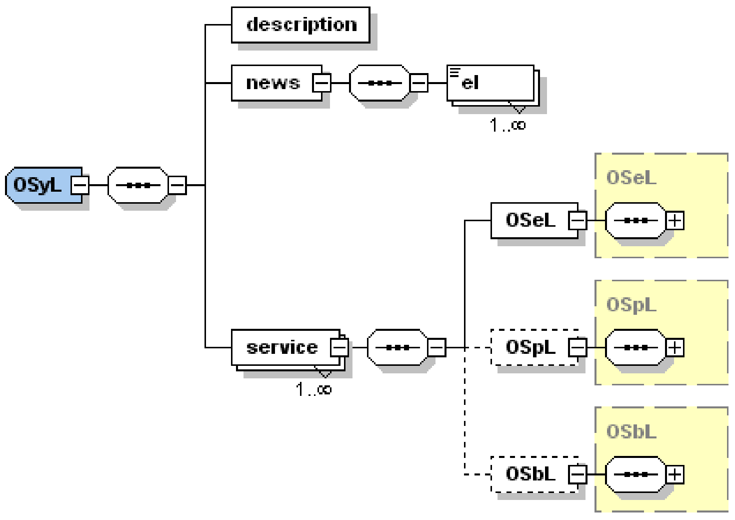


Figure 8-6: <`OSyL`> root element in OSyL.

A rough sketch of the OSyL is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Sample XML file generated by XMLSPY v2004 rel. 3 U (http://www.xmlspy.com)-->
<OSyL xmlns="os.optimizationservices.org" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="os.optimizationservices.org http://www.optimizationservices.org/schemas/OSyL.xsd">
    <description>
        OS registry is a native XML data base.
        It contains a sequence of service, each consisting of a triplet (OSeL, OSpL, OSbL).
     </description>
    <news>
        <el date="2005-04-06">Impact Generalized Mixed Integer Solver joins the OS registry</el>
        <el date="2005-03-29">Ziena Knitro Service joins the OS registry</el>
        <el date="2005-02-27">Lindo MINLP Service joins the OS registry</el>
    </news>
    <service>
        <OSeL>
            <service>
                <uri>http://www.impactservice.net/impactGMIP.jws</uri>
                <name>Impact Generalized Mixed Integer Solver</name>
            </service>
            ...
        </OSeL>
```

beginning

services

```
        <OSpL> . . .  </OSpL>
        <OSbL>  . . . </OSbL>
    </service>
    <service>
        <OSeL>  . . . </OSeL>
        <OSpL> . . .  </OSpL>
        <OSbL> . . .  </OSbL>
    </service>
    <service>
        <OSeL>  . . . </OSeL>
        <OSpL> . . .  </OSpL>
        <OSbL> . . .  </OSbL>
    </service>
    . . .
    . . .
</OSyL>
```

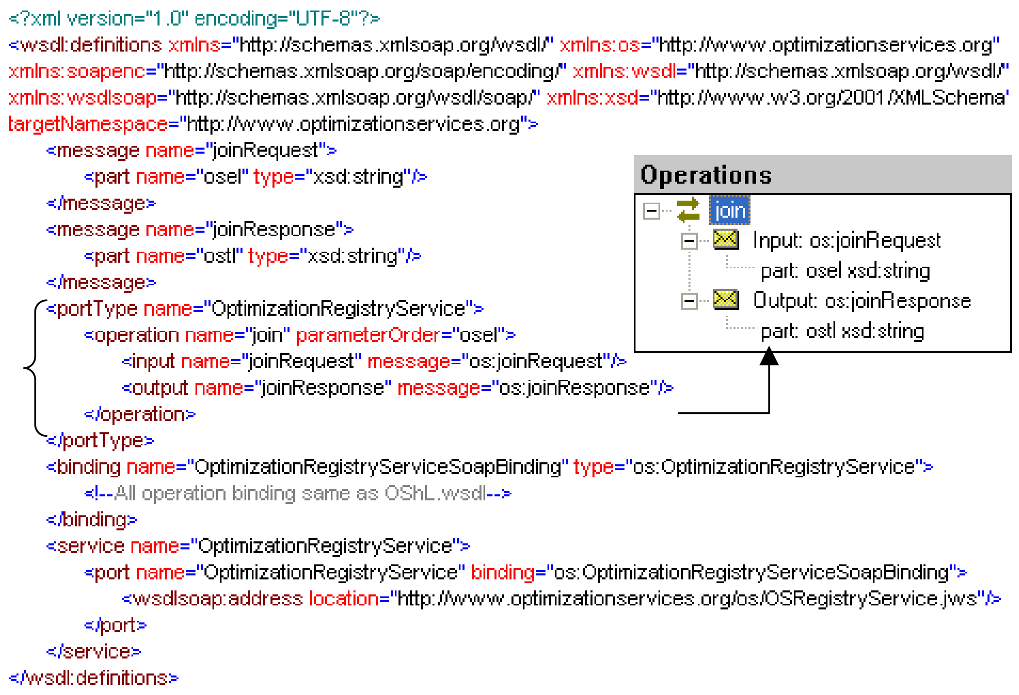
8.5 Optimization Services join Language (OSjL, communication)

The OSjL document is located at http://www.optimizationservices.org/wsdl/OSjL.wsdl. OSjL is a WSDL description of how an optimization service can join the OS registry. As described from §8.1 to §8.3, there are three pieces of information the OS registry keeps. But service providers only need the OSeL (entity) information to join the OS registry. The OSpL (process) information and the OSbL (benchmark) information are collected later.

But as discussed in §8.1, to register a service, the registrant usually goes to the Optimization Services registry Web site to fill out the form as shown in Figure 8-2. When the registrant clicks the **submit** button, the entered information is automatically organized into an OSeL file and sent to the OS registry using the communication specified by the OSjL WSDL document. The registrant can also directly submit the OSeL using the OSjL communication.

To make the *join* communication, both the client registrant and the OS registry have to follow the rules (operations, protocols, etc.) specified in the `OSjL.wsdl` document. The communication is just like the communication between a modeler and a solver using the OShL WSDL document discussed in §7.1. The OS registry is just another Optimization Service based on Web services and the SOAP protocol. It is hosted by an OS Server in the same way that all other OS services are hosted by their individual OS servers. So the underlying networking process includes the similar four client steps (encoding, SOAP envelope construction, sending/receiving, and decoding) and the similar 4 server side steps (decoding, invoking, encoding, and returning), as detailed in the OShL and OSsL sections in Chapter 7.
Figure 8-7 shows the interface part of the OSjL WSDL document. The other part (protocol) of the WSDL document, like all other OSxL WSDL documents, uses the same specifications as OShL shown in Figure 7-3; the reason is explained in the beginning of Chapter 7 and also in

the OShL section (§7.1). Briefly the networking protocol has to be "SOAP over HTTP" with an RPC style call; the input and output encoding has to be the standard SOAP encoding. The service address of the OS registry is shown at the beginning of this chapter.



```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:os="http://www.optimizationservices.org"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema'
targetNamespace="http://www.optimizationservices.org">
    <message name="joinRequest">
        <part name="osel" type="xsd:string"/>
    </message>
    <message name="joinResponse">
        <part name="ostl" type="xsd:string"/>
    </message>
    <portType name="OptimizationRegistryService">
        <operation name="join" parameterOrder="osel">
            <input name="joinRequest" message="os:joinRequest"/>
            <output name="joinResponse" message="os:joinResponse"/>
        </operation>
    </portType>
    <binding name="OptimizationRegistryServiceSoapBinding" type="os:OptimizationRegistryService">
        <!--All operation binding same as OShL.wsdl-->
    </binding>
    <service name="OptimizationRegistryService">
        <port name="OptimizationRegistryService" binding="os:OptimizationRegistryServiceSoapBinding">
            <wsdlsoap:address location="http://www.optimizationservices.org/os/OSRegistryService.jws"/>
        </port>
    </service>
</wsdl:definitions>
```

Figure 8-7: Illustration of OSjL (interface part); other parts are the same as OShL in Figure 7-3.

The `<wsdl:portType>` element in Figure 8-7 has only one operation whose name is `join`. The `join` operation's input is required to be of message type "`joinRequest`" and its output is required to be of message type "`joinResponse`." The `joinRequest` message has one `part` element (i.e. one argument), `osel`, which is of `string` types. Simply put, the WSDL document in Figure 8-7 specifies the following single operation:

```
String join(String osel);
```

that is, the OS registry service has a method called "`join`" that takes one input string and returns one string. The input string has to follow the OSeL schema (§8.1). The output string has to follow the OStL schema (§6.8), which is a transformation style sheet returned by the OS registry. If a service is registered in the OS registry and the service provider also wants to publish the standard service entity information (OSeL) on his own Web site, it is required that he publishes the information using the OStL style sheet for uniform look and feel on the OS network. In Table 8-1, we list the operations specified in the OSjL WSDL document (currently only one).

Operation	Description
String join (String)	1st input is an OSeL instance for service entity information. Output is an OStL style sheet for individual publication.

Table 8-1: Operations in OSjL (currently only one).

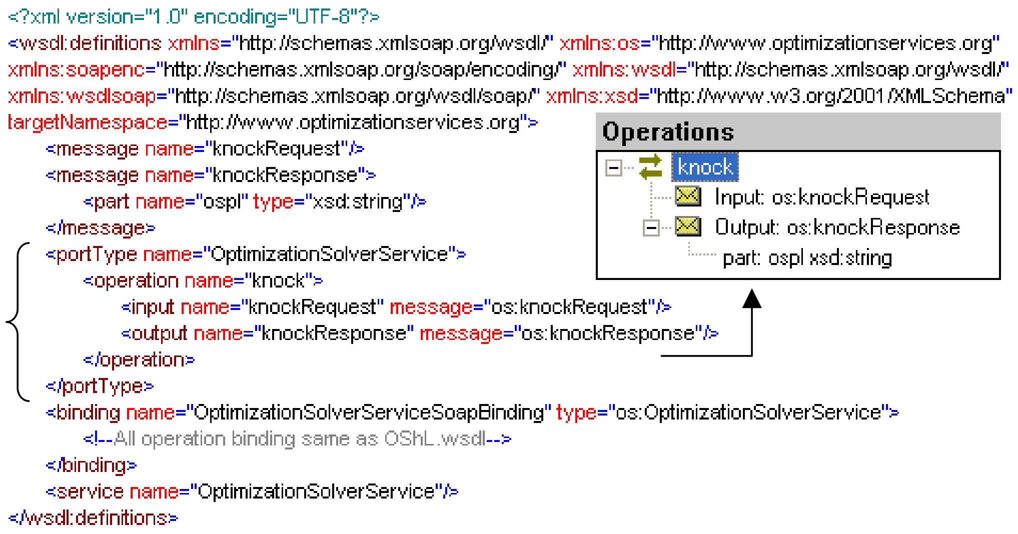
The OSjL process is exactly what happens when the registrant clicks the **submit** button of Figure 8-2. The OSeL information, however, may not be immediately published by the OS registry, as the joining process may involve manual processes, such as human review and approval for quality assurance.

8.6 Optimization Services knock Language (OSkL, communication)

The OSkL document is located at http://www.optimizationservices.org/wsdl/OSkL.wsdl. OSkL is a WSDL description of how the OSpL (process) information (§8.2) is collected at run time by the OS registry. When the OS registry "knocks" on an Optimization Service, the Optimization Service is required to respond with the current run time process information.

To make the *knock* communication, both the client (the OS registry) and the service (usually a solver service) have to follow the rules specified in the `OSkL.wsdl` document. The communication is just like the communication between a modeler and a solver using the OShL WSDL document discussed in §7.1 and any other OSxL client-service style communication on an OS network, with the same underlying networking process described in the previous sections and chapters.

Figure 8-8 shows the interface part of the OSkL WSDL document. The other part (protocol) of the WSDL document, like all other OSxL WSDL documents, uses the same specifications as OShL shown in Figure 7-3; the service address of the OS registry should be empty.



```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:os="http://www.optimizationservices.org"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.optimizationservices.org">
    <message name="knockRequest"/>
    <message name="knockResponse">
        <part name="ospl" type="xsd:string"/>
    </message>
    <portType name="OptimizationSolverService">
        <operation name="knock">
            <input name="knockRequest" message="os:knockRequest"/>
            <output name="knockResponse" message="os:knockResponse"/>
        </operation>
    </portType>
    <binding name="OptimizationSolverServiceSoapBinding" type="os:OptimizationSolverService">
        <!--All operation binding same as OShL.wsdl-->
    </binding>
    <service name="OptimizationSolverService"/>
</wsdl:definitions>
```

```
Operations
knock
    Input: os:knockRequest
    Output: os:knockResponse
        part: ospl xsd:string
```

Figure 8-8: Illustration of OSkL (interface part); other parts are the same as OShL in Figure 7-3.

The `<wsdl:portType>` element in Figure 8-8 has only one operation whose name is `knock`. The `knock` operation's input is required to be of message type "`knockRequest`" and its output is required to be of message type "`knockResponse`." The `knockRequest` message does not have any part (no arguments). Simply put, the WSDL document in Figure 8-8 specifies the following single operation:

```
String knock();
```

that is, all the services are required to implement a method called "`knock`" that takes no arguments and returns one string. The output string has to follow the OSpL schema (§8.2). When a client agent "knocks" on a service, the service is required to return the current process information. In Table 8-2, we list the operations specified in the OSkL WSDL document (currently only one).

Operation	Description
String knock ()	No input. Output is an OSpL string for process information.

Table 8-2: Operations in OSkL (currently only one).

So a solver or analyzer service should not only implement all the methods required by OShL (§7.1), such as `String solve(String osil)` and `retrieveResult(String jobID)`, but also add the extra method `String knock()` required by OSkL.

8.7 Optimization Services query Language (OSqL, representation)

The OSqL schema is located at http://www.optimizationservices.org/schemas/OSqL.xsd. OSqL is a specification of the query language format used to discover the optimization services in the OS registry. The OS registry returns the locations of the solvers that match the OSqL query in OSuL (Optimization Services uri Language, §8.8).

In the OS registry implementation, an OSqL query is converted to an XQuery (§4.4) that is executed against the XML database (OSyL, §8.4) in the registry. The OSyL based XML database is open and published on the OS registry's Web site for public references. Since the OSyL file is in XML, the client can directly use the XQuery language to retrieve any information. Using XQuery language to query the database is a built-in feature of OSqL. The second feature OSqL provides is support for Optimization Services analysis Language (OSaL, §6.6). If the query client has already had an OS analyzer analyze the problem instance and obtained the analysis result in OSaL, the client can embed the OSaL in the OSqL instance and the OS registry will try its best in trying to find the most appropriate solver services. The third feature OSqL provides is some predefined standard information structured according to the entity information (OSeL, §8.1), process information §8.2) and benchmark information (§8.3). The OSqL schema is shown in Figure 8-9.

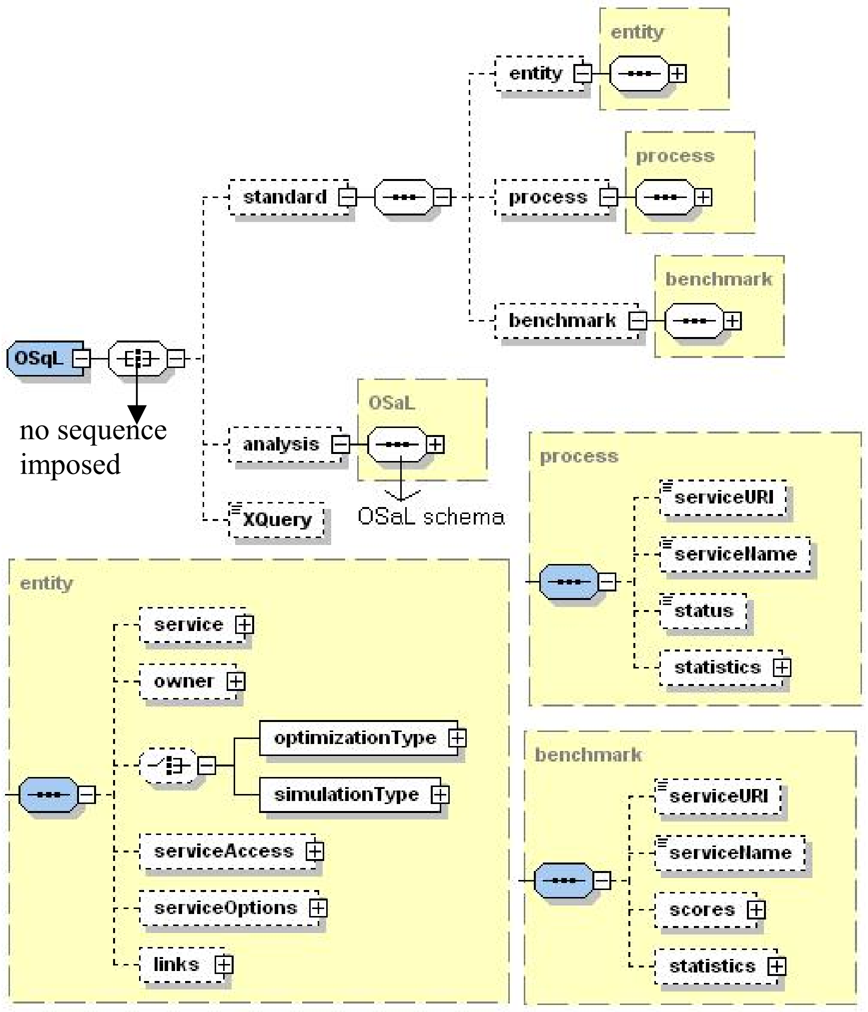


Figure 8-9: `<OSqL>` root element in OSqL and descriptions of its immediate children.

The three children (`<standard>`, `<analysis>`, and `<XQuery>`) of `<OSqL>` correspond to the three ways of query discussed above and they can be mixed with each other. The following is an example using XQuery to discover the URIs of solver services that solve optimization problems with convex nonlinear objective functions:

```
 <?xml version="1.0" encoding="UTF-8"?>
<OSqL xmlns="os.optimizationservices.org" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="os.optimizationservices.org http://www.optimizationservices.org/schemas/OSpL.xsd">
    <XQuery>
        for  $a in OSeL where $a/optimizationType/linearity/objective = 'convexNonlinear' return $a/service/uri
    </XQuery>
</OSqL>
```

The next example mixes the above XQuery with an OSaL analysis element:

```
<OSqL xmlns="os.optimizationservices.org" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="os.optimizationservices.org http://www.optimizationservices.org/schemas/OSpL.xsd">
    <XQuery>
        for  $a in OSeL where $a/optimizationType/linearity/objective = 'convexNonlinear' return $a/service/uri
    </XQuery>
    <OSaL>
        <programDescription>
            <numberObjectives num="1">
                <linear num="0"/>
                <quadratic num="0"/>
                <nonlinear num="1"/>
                <networkAndGraphProblem num="0"/>
            </numberObjectives>
            <numberConstraints num="12">
                <linear num="4">
                    <equality num="1"/>
                    <inequality num="1"/>
                    <range num="2"/>
                </linear>
                <quadratic num="8">
                    <equality num="0"/>
                    <inequality num="0"/>
                    <range num="0"/>
                </quadratic>
                <nonlinear num="8">
                    <equality num="3"/>
                    <inequality num="4"/>
                    <range num="1"/>
                </nonlinear>
            </numberConstraints>
            <numberVariables num="12">
                <continuous num="3"/>
                <integer num="9"/>
                <binary num="0"/>
                <string num="0"/>
            </numberVariables>
        </programDescription>
        <programDataAnalysis> . . . </programDataAnalysis>
    </OSaL>
</OSqL>
```

Since there are some nonlinear constraints and some integer variables in `<analysis>`, the OS registry will likely find a mixed integer nonlinear solver's location for the client.

The third feature OSqL provides is some predefined query structures under the `<standard>` child. As can be seen in Figure 8-9, the `<standard>` element contains three elements, `<entity>`, `<process>`, and `<benchmark>`. Each element schema is listed in the same figure as the `<OSqL>` element. Compare them with the OSeL (Figure 8-3), OSpL (Figure 8-4) and OSbL (Figure 8-4) schemas. They are very similar; basically the predefined elements under `<entity>`, `<process>`, and `<benchmark>` are a subset of those in OSeL, OSpL, and OSbL with some modifications.

For example the following example mixes the above XQuery with the `<standard>` element to find a service that contains the keyword "interior point methods" and "convex programming" and whose variable type is "mixedInteger."

```
<OSqL xmlns="os.optimizationservices.org" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="os.optimizationservices.org http://www.optimizationservices.org/schemas/OSpL.xsd">
   <standard>
      <entity>
         <service>
            <keyWords><key>interior point method</key><key>convex programming</key></keyWords>
         </service>
         <optimizationType>
            <variableType>mixedInteger</variableType>
         </optimizationType>
      </entity>
   </standard>
   <XQuery>
      for  $a in OSeL where $a/optimizationType/linearity/objective = 'convexNonlinear' return $a/service/uri
   </XQuery>
</OSqL>
```

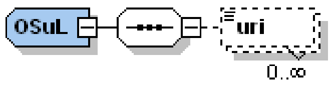
However the elements under `<entity>`, `<process>`, and `<benchmark>` are not without modifications from those in OSeL, OSpL, and OSbL. For example, the "`relation`" attribute is added on several elements to define matching types. For numeric data, the relation can be "geq", "leq", or "eq." For string data (especially long ones like descriptions), the relation can be "contains" or "same." For example the following example finds all the services whose publication date are before (`relation="leq"`) 2005-03-14 and whose abstract contains (`relation="contains"`) each of the three words "interior point method."

```
<OSqL xmlns="os.optimizationservices.org" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="os.optimizationservices.org http://www.optimizationservices.org/schemas/OSpL.xsd">
   <standard>
      <service>
         <publicationDate relation="leq">2005-03-14</publicationDate>
         <abstract relation="contains"> interior point method</publicationDate>
      </service>
   </standard>
</OSqL>
```

8.8 Optimization Services uri Language (OSuL, representation)

The OSuL schema is located at http://www.optimizationservices.org/schemas/OSuL.xsd. OSuL is a specification of the discovery result (in URI) sent back from the OS registry. OSuL is the opposite of the OSqL query (§8.7) in the discovery process. Based on the OSqL instance, the OS registry returns the locations of the services that match the query.

The OSuL schema is shown in Figure 8-10 .



```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema targetNamespace="os.optimizationservices.org" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="os.optimizationservices.org"
elementFormDefault="qualified">
    <xs:element name="OSuL" type="OSuL"/>
    <xs:complexType name="OSuL">
        <xs:sequence>
            <xs:element name="uri" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                    <xs:attribute name="match" use="optional" default="exact">
                        <xs:simpleType>
                            <xs:restriction base="xs:string">
                                <xs:enumeration value="exact"/>
                                <xs:enumeration value="moregeneral"/>
                                <xs:enumeration value="approximate"/>
                                <xs:enumeration value="guess"/>
                            </xs:restriction>
                        </xs:simpleType>
                    </xs:attribute>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

Figure 8-10: `<OSuL>` root element in OSpL.

`<OSuL>` is simply a sequence of 0 (if no matches) or more `<uri>` children. Each `<uri>` has an optional `match` attribute and by default it is an "exact" match of the services. The OS registry may return service locations of other match types such as "`moreGeneral`," "`approximate`," and "`guess`." An example of the "`moreGeneral`" case is a nonlinear solver service for a linear program. An example of the "`approximate`" case is a convex nonlinear solver service for an almost convex nonlinear program. An example of the "`guess`" case is when there is not enough information in OSqL. In general the OS registry returns the URI locations ordered by fitness, with the best and exact matches in the beginning. An example of the OSuL discovery result may look like the following:

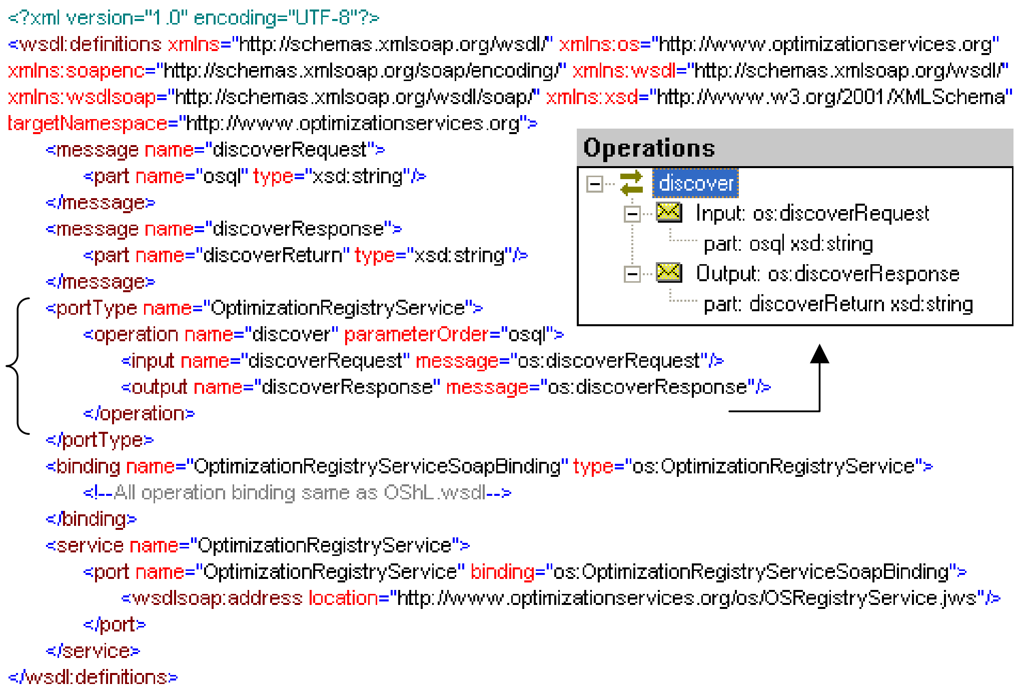
```
<?xml version="1.0" encoding="UTF-8"?>
<OSuL xmlns="os.optimizationservices.org" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="os.optimizationservices.org http://www.optimizationservices.org/schemas/OSuL.xsd">
    <uri>http://www.abc.com/lpsolver.jws</uri>
    <uri match="exact">http://www.edf.net/lpsolverservice.vb</uri>
    <uri match="moreGeneral">http://www.ghij.org/mpservice.cs</uri>
    <uri match="approximate">http://www.klmn.gov/os/nlpsolver.jws</uri>
    <uri match="guess">http://www.klmn.gov/os/minlpsolver.py</uri>
</OSuL>
```

8.9 Optimization Services discover Language (OSdL, communication)

The OSdL document is located at http://www.optimizationservices.org/wsdl/OSdL.wsdl. OSdL is a WSDL description of how a client sends the OSqL (query) information (§8.7) to the OS registry and discovers matched registered Optimization Services.

To make the *discover* communication, both the client and the OS registry have to follow the rules specified in the `OSdL.wsdl` document. The communication is just like any other OSxL client-service style communication on an OS network, with the same underlying networking process described in the previous sections and chapters.

Figure 8-11 shows the interface part of the OSdL WSDL document. The other part (protocol) of the WSDL document, like all other OSxL WSDL documents, uses the same specifications as OShL shown in Figure 7-3. The service address of the OS registry is shown at the beginning of this chapter.



```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:os="http://www.optimizationservices.org"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.optimizationservices.org">
    <message name="discoverRequest">
        <part name="osql" type="xsd:string"/>
    </message>
    <message name="discoverResponse">
        <part name="discoverReturn" type="xsd:string"/>
    </message>
    <portType name="OptimizationRegistryService">
        <operation name="discover" parameterOrder="osql">
            <input name="discoverRequest" message="os:discoverRequest"/>
            <output name="discoverResponse" message="os:discoverResponse"/>
        </operation>
    </portType>
    <binding name="OptimizationRegistryServiceSoapBinding" type="os:OptimizationRegistryService">
        <!--All operation binding same as OShL.wsdl-->
    </binding>
    <service name="OptimizationRegistryService">
        <port name="OptimizationRegistryService" binding="os:OptimizationRegistryServiceSoapBinding">
            <wsdlsoap:address location="http://www.optimizationservices.org/os/OSRegistryService.jws"/>
        </port>
    </service>
</wsdl:definitions>
```

```
Operations
discover
    Input: os:discoverRequest
        part: osql xsd:string
    Output: os:discoverResponse
        part: discoverReturn xsd:string
```

Figure 8-11: Illustration of OSdL (interface part); other parts are the same as OShL in Figure 7-3.

The `<wsdl:portType>` element in Figure 8-11 has only one operation whose name is `discover`. The `discover` operation's input is required to be of message type "`discoverRequest`" and its output is required to be of message type "discoverResponse." The discoverRequest message has one part element (i.e. one argument), osql, which is of string type. Simply put, the WSDL document in Figure 8-11 specifies the following single operation:

```
String discover(String osql);
```

that is, the OS registry service implements a method called "discover" that takes one input string and returns one string. The input string has to follow the OSqL schema (§8.7). The output string has to follow the OSuL schema (§8.8). In Table 8-3, we list the operations specified in the OSdL WSDL document (currently only one).

Operation	Description
String discover (String)	1st input is an OSqL query instance. Output is an OSuL for service URI locations.

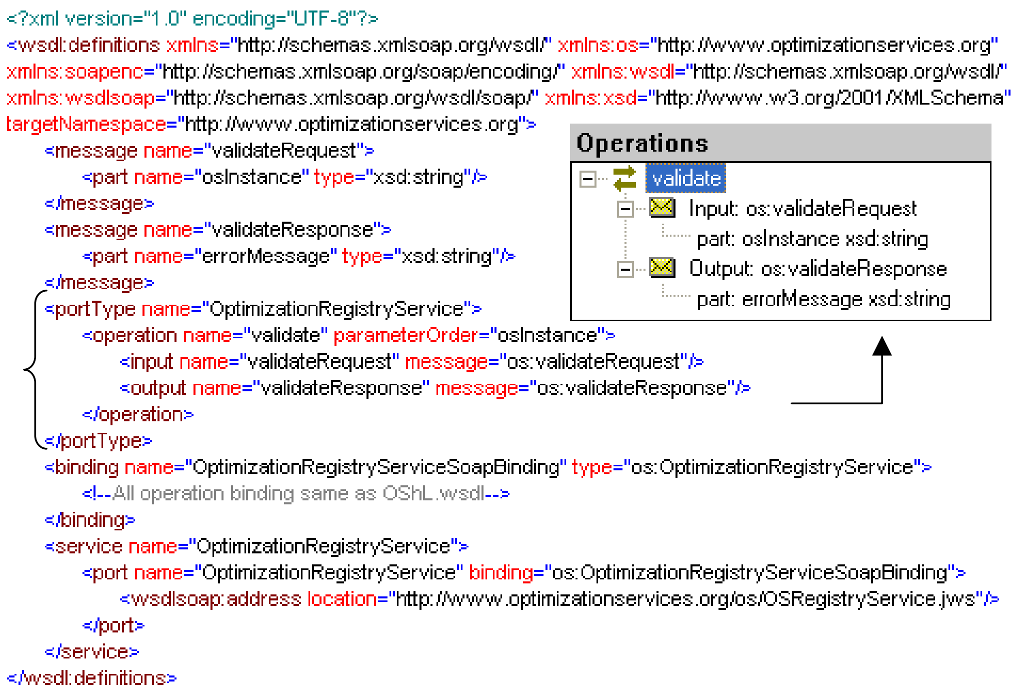
Table 8-3: Operations in OSdL (currently only one).

8.10 Optimization Services validate Language (OSvL, communication)

The OSvL document is located at http://www.optimizationservices.org/wsdl/OSvL.wsdl. Besides the "join" (OSjL, §8.5) service for service providers and the "discover" (OSdL, §8.9) service for service clients, the OS registry also provides a validation service through OSvL for any client on the OS network. OSvL is a WSDL description of how the OS registry is used to validate any OSxL instance. The OS registry returns an error message if there is any warning or error in the OSxL instance submitted. Otherwise it returns a null or empty string.

To make the *validate* communication, both the client and the OS registry have to follow the rules specified in the `OSvL.wsdl` document. The communication is just like any other OSxL client-service style communication on an OS network, with the same underlying networking process described in the previous sections and chapters.

Figure 8-8 shows the interface part of the OSdL WSDL document. The other part (protocol) of the WSDL document, like all other OSxL WSDL documents, uses the same specifications as OShL shown in Figure 7-3. The service address of the OS registry is shown at the beginning of this chapter.



```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:os="http://www.optimizationservices.org"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.optimizationservices.org">
    <message name="validateRequest">
        <part name="osInstance" type="xsd:string"/>
    </message>
    <message name="validateResponse">
        <part name="errorMessage" type="xsd:string"/>
    </message>
    <portType name="OptimizationRegistryService">
        <operation name="validate" parameterOrder="osInstance">
            <input name="validateRequest" message="os:validateRequest"/>
            <output name="validateResponse" message="os:validateResponse"/>
        </operation>
    </portType>
    <binding name="OptimizationRegistryServiceSoapBinding" type="os:OptimizationRegistryService">
        <!--All operation binding same as OShL.wsdl-->
    </binding>
    <service name="OptimizationRegistryService">
        <port name="OptimizationRegistryService" binding="os:OptimizationRegistryServiceSoapBinding">
            <wsdlsoap:address location="http://www.optimizationservices.org/os/OSRegistryService.jws"/>
        </port>
    </service>
</wsdl:definitions>
```

```
Operations
validate
    Input: os:validateRequest
        part: osInstance xsd:string
    Output: os:validateResponse
        part: errorMessage xsd:string
```

Figure 8-12: Illustration of OSvL (interface part); other parts are the same as OShL in Figure 7-3.

The `<wsdl:portType>` element in has only one operation whose name is `validate`. The `validate` operation's input is required to be of message type "`validateRequest`" and its output is required to be of message type "`validateResponse`." The `validateRequest` message has one `part` element (i.e. one argument), `osInstance`, which is of `string` type. Simply put, the WSDL document in Figure 8-12 specifies the following single operation:

```
String validate(String osInstance);
```

that is, the OS registry service implements a method called "`validate`" that takes one input string and returns one string. The input string can be any OSxL instance. The registry can distinguish the type of the instance from its root element. The output string is an error message if there is any warning or error in the OSxL instance submitted. Otherwise, the output is a null or empty string. In Table 8-4, we list the operations specified in the OSvL WSDL document (currently only one).

Operation	Description
String validate (String)	1st input is any OSxL instance. Output is a string that contains an error or warning information; the string is null or empty if there is no error or warning.

Table 8-4: Operations in OSvL (currently only one).

Along with the join operation from OSjL and the discover operation OSdL, the validate operation from OSvL is the third operation offered by the OS registry service.

CHAPTER 9 OPTIMIZATION SERVICES MODELING LANGUAGE (OSML)

OSmL is a computer modeling language for mathematical optimization. This language allows mathematical model developers to formulate complex and large-scale optimization problems in a concise and efficient way. OSmL is based upon the W3C XQuery standard and is designed to convert raw data in XML format into problem instances that conform to the Optimization Services instance Language (OSiL) standard. An optimization instance represented in OSiL can be solved with any standard solver that is Optimization Services compatible. Thus, OSmL is particularly well suited for optimization over distributed systems. OSmL is an Optimization Services project designed to facilitate the adoption of Optimization Services.

In this chapter we briefly describe the motivations behind designing the OSmL modeling language. We list four paradigms of combining XML with optimization modeling. The OSmL approach is the fourth paradigm. It takes raw data files in XML format and transforms them using XQuery and XPath (a subset of XQuery since XQuery 2.0) into a single XML OSiL instance. Notice that OSmL itself is *not* an XML dialect, but rather a customized implementation of XQuery. XQuery provides a concise query language and unlike style sheets, XQuery is not designed to transform the entire structure of an XML document. It is designed to quickly and efficiently extract chunks of data – much like SQL for relational databases. Unlike other OSxL languages, OSmL is *not* a low-level instance language. OSmL is a high-level user friendly *modeling* language.

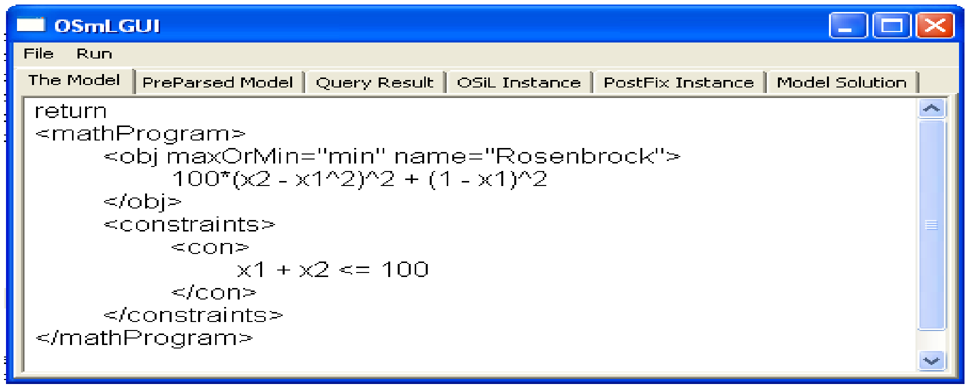
An advantage of the OSmL approach is that people can easily share and reuse OSmL models regardless of computing platform. All of the required software is open source and available on all major platforms. Also, this is the natural way to work with XML data, which is becoming an increasingly popular standard for storing and transmitting data. After a brief introduction, we describe four different paradigms of combining XML with mathematical programming. At the end of this chapter, we show various features and some examples of the OSmL modeling language.

9.1 Introduction and Motivation

As discussed in Chapter 4, XML is popular and powerful. Whether in the technical press or mainstream business press, each day is filled with new articles about XML-related technologies. XML is rapidly becoming an accepted format for transferring and storing data. Currently almost all databases and spreadsheets are xml-enabled. This means that even if the stored data are not in native XML form, they can at least be exported in XML formats. The XML output can then be retrieved with the standard XPath and XQuery language (§4.4).

One might argue that mathematical modeling is also about data. Indeed, a mathematical programming modeling language, and associated solver tools, will not be used unless they are closely integrated with corporate data. One reason for the success of Excel based solvers is their close integration with all the existing spreadsheet data.

This creation of the OSmL modeling language is based on two premises. First is the ubiquity of XML and the existence of tools to easily transform the non-XML data into an XML format if the data of interest are not in an XML format. Second is the availability of many powerful open source platform independent tools for taking XML data stored in one format and transforming it into another XML format (such as XPath, XQuery, XSLT; see Chapter 4). Since we have already designed a set of XML standards for math program instance representation, we can use the transformation tools to transform the raw XML data into the XML instance format. In this respect, the OSmL approach is similar to the work of Atamtürk et al. [7], where these authors demonstrate that SQL is sufficient for generating linear programming problem instances. Earlier, Choobineh [22] developed SQLMP, an SQL based modeling system for linear programming. Figure 9-1 shows the OSmL GUI with the modified Rosenbrock problem used throughout this thesis. We illustrate advanced features such as set, indices, loops and other features in the following sections.



```
return
<mathProgram>
        <obj maxOrMin="min" name="Rosenbrock">
                100*(x2 - x1^2)^2 + (1 - x1)^2
        </obj>
        <constraints>
                <con>
                        x1 + x2 <= 100
                </con>
        </constraints>
</mathProgram>
```

Figure 9-1: OSmL GUI with an OSmL model of the modified Rosenbrock problem.

9.2 Four Paradigms of Combining XML with Optimization

It is common practice to store data in a relational database system. Two aspects of commercial relational database systems are 1) the data are stored in multiple tables or relations, and 2) the files containing the data are typically binary files. XML data is 1) stored using a tree structure, and 2) stored as a text file containing both tags and the data.

There are four paradigms to incorporating XML into mathematical modeling of optimization problems:

1. Use XML to represent the instance of a mathematical program
2. Develop an XML modeling language dialect
3. Enhance modeling languages with XML features such as XPath
4. Use XML technologies to transform XML data into a problem instance

9.2.1 Use XML to represent the instance of a mathematical program

The Optimization Services instance Language (OSiL, Chapter 6) is an example of the first paradigm. Besides Optimization Services, several other projects also take this approach ([15][19] [53] [69]).

This approach differs from the rest of the three approaches in that it is incorporating the XML technologies at the *low level* instances, whereas the rest are positioned at the modeling language level (Figure 9-2). This approach requires no changes to current mathematical programming modeling languages. It does require drivers to interface with various solvers and modeling languages. The native format for representing a problem instance in each modeling language must be converted to the XML instance. Then the XML instance must be converted to the native format required by a solver; see §2.3 for more details.

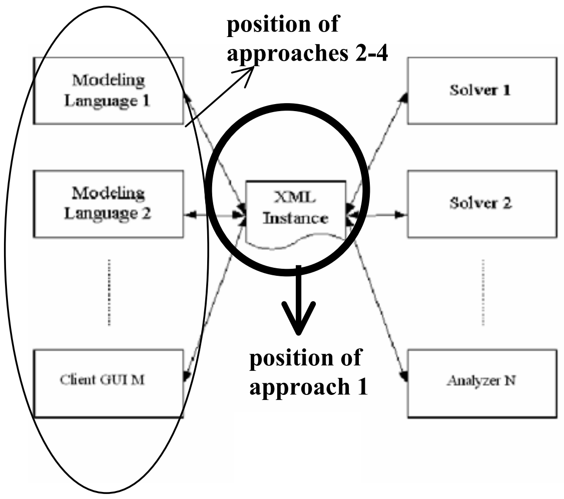


Figure 9-2: Using XML to represent the instance of a mathematical program (1st approach).

9.2.2 Develop an XML modeling language dialect

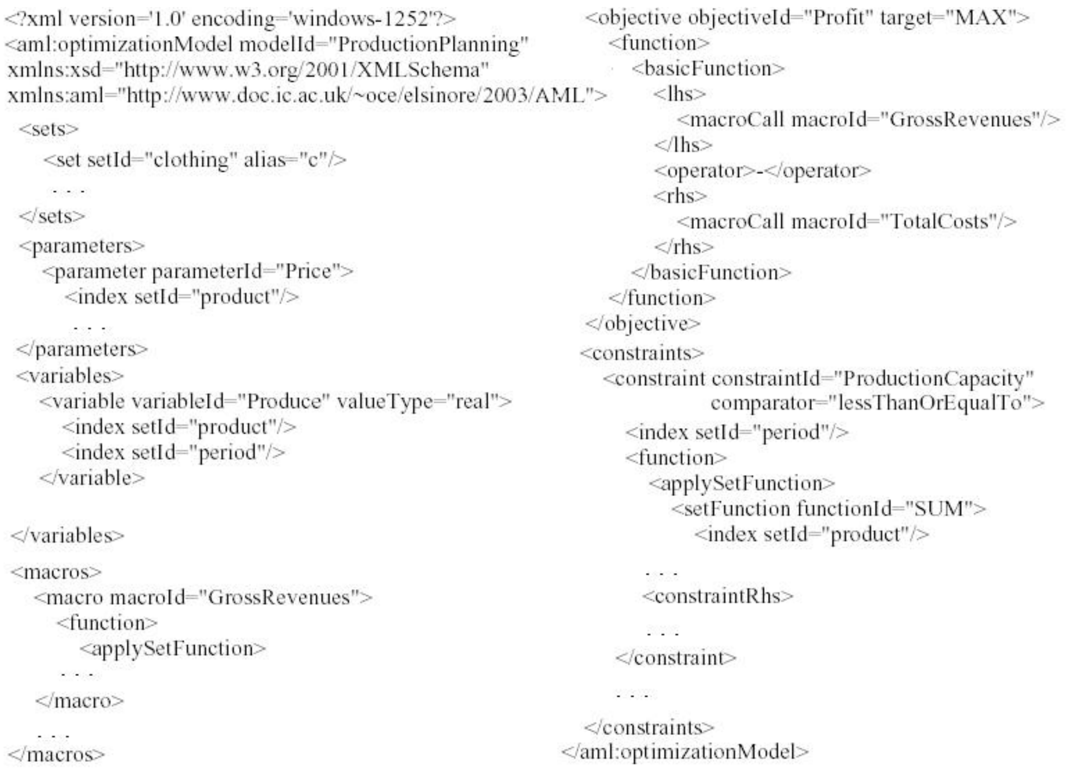
This approach represents the entire *high level* mathematical model in XML, by designing tags for model constructs such as sets, indices, summation, and looping. The only current research project that takes this approach comprises the Algebraic Markup Language (AML, [34]) and Optimization Reporting Markup Language (ORML, [35]) by Ezechukwu and Maros.

With this approach, an XML input file contains both raw data and information about the algebraic structure of the model. For example, sets and indices are defined within the XML input file. This approach requires the development of a new XML based modeling language syntax. Although feasible, this approach may require consensus on the syntax of such an XML dialect. Currently there is a proliferation of various modeling languages and thus it is hard to get everybody to agree with the XML tags. Moreover, as XML is wordy, this approach leads to a verbose language. Indeed, one reason the Optimization Services project is only involved in low-level instance representation, rather than an XML based modeling language, is that the instance is the lowest common denominator and requires the least amount of agreement.

As AML/ORML is currently the only research project in this area, it is appropriate for us to quote below the project's own description and motivation from its Web site [35]:

> … So how do you solve the problem [of model representation]. Well the first step is to ensure that the solution doesn't require any form of support or compliance by vendors. Secondly ensure that it is an open source initiative or at least is as widely and freely available as possible. Thirdly and most importantly use established and widely supported software standards such as XML and XSL.
>
> This is exactly how the framework works. What we have done is taken a typical software engineering approach to solving this problem. In the first place we invented the Algebraic Modeling Language (AML) which is an abstract XML based representation of mathematical models. Secondly we created the Optimization Reporting Markup Language (ORML) which is used to represent optimization analysis and solution results. Finally we utilize a translation process such as XSLT (or program constructs) to transform AML and ORML data to target formats. …
>
> This is basically what we view as a common-sense solution to the problem, because XML solves the problem of varying formats, and XSL provides a means of converting the XML representations to the appropriate target format. We have no intention of pushing for a new standard, as that would only result in a lot more debate, and truth be told vendors of algebraic modeling systems would probably view the idea with a great deal of hostility (not that we would blame them for that), because it would obviously affect their marketing strategies. Not to mention the fact that it may become more difficult to retain customers.
>
> One view of the framework is not so much as a solution to the problem of model representation but rather as a way to side-step the problem. However we would prefer to view it as the former. Whichever view you adopt though, the bottom line is that it provides you with a portable means of representing optimization models. … There is no need for endless discussions on standardization, neither is there any need for support from vendors. It is quite simply very easy to use or plug in.
>
> … Is it a modeling language? Absolutely and categorically not! … Asking this question would almost be the same as asking of XML itself is a programming language which very clearly it is not. … The framework does not come with any modeling system or similar executable and at present, it isn't actually possible to execute a model in the AML format directly i.e. it has to be ported to a target/native format by translation in order to be executed. It is purely a representation format. …

Figure 9-3 shows the sketch of a production planning model written in AML [34].



```
<?xml version='1.0' encoding='windows-1252'?>
<aml:optimizationModel modelId="ProductionPlanning"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:aml="http://www.doc.ic.ac.uk/~oce/elsinore/2003/AML">
 <sets>
   <set setId="clothing" alias="c"/>
   . . .
 </sets>
 <parameters>
   <parameter parameterId="Price">
     <index setId="product"/>
     . . .
 </parameters>
 <variables>
   <variable variableId="Produce" valueType="real">
     <index setId="product"/>
     <index setId="period"/>
   </variable>

</variables>
<macros>
  <macro macroId="GrossRevenues">
    <function>
      <applySetFunction>
    . . .
  </macro>
  . . .
</macros>
```

```
<objective objectiveId="Profit" target="MAX">
  <function>
    <basicFunction>
      <lhs>
        <macroCall macroId="GrossRevenues"/>
      </lhs>
      <operator>-</operator>
      <rhs>
        <macroCall macroId="TotalCosts"/>
      </rhs>
    </basicFunction>
  </function>
</objective>
<constraints>
  <constraint constraintId="ProductionCapacity"
              comparator="lessThanOrEqualTo">
    <index setId="period"/>
    <function>
      <applySetFunction>
        <setFunction functionId="SUM">
          <index setId="product"/>

      . . .
      <constraintRhs>

      . . .
    </constraint>

    . . .
  </constraints>
</aml:optimizationModel>
```

Figure 9-3: The sketch of a math programming model written in AML [34].

9.2.3 Enhance modeling languages with XML features such as XPath

Current algebraic modeling languages such AMPL, LINGO, and MPL provide capabilities for interfacing with relational databases. This is usually done through ODBC [85] drivers that are database specific. This third approach to using XML technologies is incorporating into these modeling languages the ability to access data stored in XML format in a manner analogous to the access of data stored in a relational database. With this approach we are not suggesting changing the basic syntax of the algebraic modeling language used to represent sets, loop, perform sums, etc.

We illustrate this approach with a multiproduct dynamic lot sizing model; seeWagner and Whitin [118]. We assume that the input data for the model is in a single XML file. This assumption is not necessary; it is made only for ease of exposition. Assume there are two products with a four period planning horizon and that inventory holding cost, marginal production cost, and fixed production cost depend on product but not time period. The model is illustrated in Figure 9-4 and the corresponding XML data are represented in Figure 9-5 and graphically illustrated in Figure 9-6 . The costs (`fixedCost`, `holdCost`, `prodCost`) are depends on only `productID`. The `demand` data are functionally dependent on `productID` and `periodID`. The `capacity` data are functionally dependent on only `periodID`.

$$
\begin{array}{lrcll}
\min \sum_i \sum_t (c_{it}x_{it} + h_{it}I_{it} + f_{it}y_{it}) & & & & \text{-- minimization of the sum of the production, inventory holding} \\
\text{s.t.} \quad \sum_i x_{it} & \le & g_t, & \forall t & \text{-- capacity constraint} \\
I_{i,t-1} + x_{it} - I_{it} & = & d_{it}, & \forall i,t & \text{-- conservation of flow or sources and uses requirement} \\
x_{it} & \le & M_{it}y_{it}, & \forall i,t & \text{-- fixed charge or set forcing constraint} \\
x_{it}, I_{it} & \ge & 0, & \forall i,t & \text{-- nonnegative constraint} \\
y_{it} & \in & \{0,1\}, & \forall i,t & \text{-- binary on open or not open of a facility}
\end{array}
$$

Sets:

TIME = 1 .. T
PROD = { i }
LINKS = { (i, t) }

Parameters:

d_{it} – demand for product i in period t
f_{it} – fixed cost associated with production of product i in period t
h_{it} – marginal cost of holding one unit of product i in inventory at the end of period t
c_{it} – marginal production cost of one unit of product i in period t
g_t – production capacity available in period t
M_{it} – an upper bound on the production of product i in time period t

Variables:

x_{it} – units of product i produced in period t
I_{it} – units of product i held in inventory at the end of period t
y_{it} – a binary variable which is fixed to 1 if there is nonzero production of product i in period t, otherwise 0

Figure 9-4: Multiproduct dynamic lot sizing problem.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<lotSizeData>
    <product productID="1" holdCost="1" prodCost="7" fixedCost="150">
        <period periodID="1">
            <demand>60</demand>
        </period>
        <period periodID="2">
            <demand>100</demand>
        </period>
        <period periodID="3">
            <demand>140</demand>
        </period>
        <period periodID="4">
            <demand>200</demand>
        </period>
    </product>
    <product productID="2" holdCost="2" prodCost="4" fixedCost="100">
        <period periodID="1">
            <demand>40</demand>
        </period>
        <period periodID="2">
            <demand>60</demand>
        </period>
        <period periodID="3">
            <demand>100</demand>
        </period>
        <period periodID="4">
            <demand>40</demand>
        </period>
    </product>
    <periodCapacity>
        <capacity periodID="1">200</capacity>
        <capacity periodID="2">200</capacity>
        <capacity periodID="3">200</capacity>
        <capacity periodID="4">200</capacity>
    </periodCapacity>
</lotSizeData>
```

Figure 9-5: Dynamic lot sizing data (lotsizedata.xml)

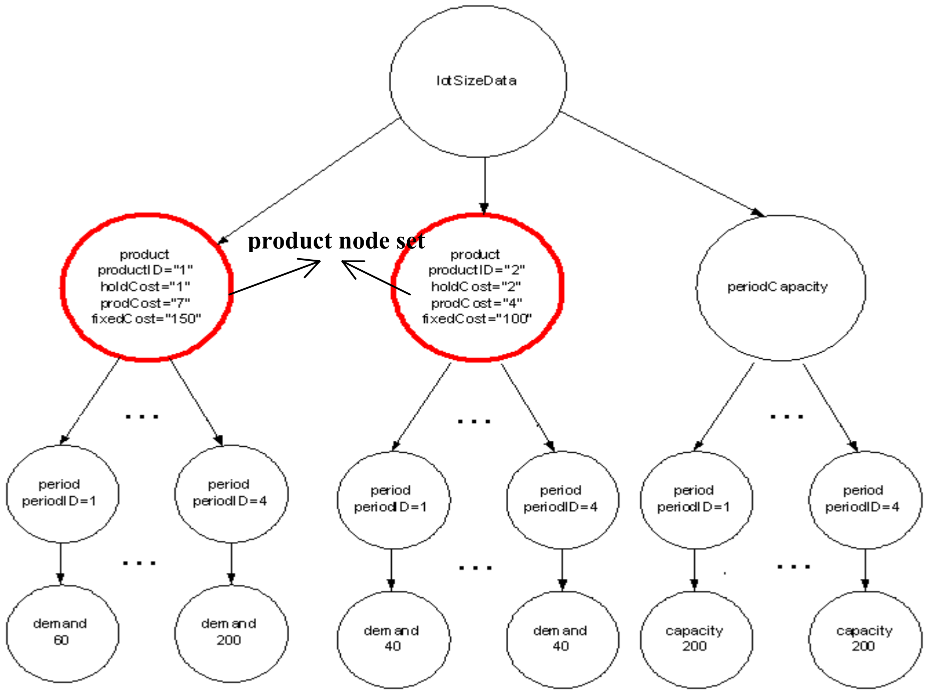


Figure 9-6: Graphic illustration of the lot sizing data in Figure 9-5; two highlighted circles indicate the product set.

As discussed in Chapter 4, XPath is used to locate data in an XML database. The function of XPath is similar to the SELECT command in SQL. However, the syntax of XPath is similar to the syntax used to locate files in a directory with a tree structure. The typical use of an XPath command is a location path to locate a set of nodes in a tree. This is called the *node-set*. The node-sets is manipulated (e.g. set difference, intersection, union) and used to generate indices.

For example, the location path, `/lotSizeData/product[(1, 2)]`, on the XML data shown in Figure 9-5, locates the node-set `{`<1st product>`, `<2nd product>`}`. XPath is also used to perform set operations such as union, intersection, and set difference; these provide the necessary language features to meet the requirements of modeling language design (see §2.2). .

For example, in the LINGO modeling language [74], one might declare a set of time periods and a capacity for every time period. Denote the set of capacities by CAP. In LINGO, this set of capacities is populated from an ODBC database in the DATA section. This is illustrated below:

```
DATA:
CAP = @ODBC( 'capacitydata', 'capacity');
ENDDATA
```

However, if these data were in the XML file illustrated in Figure 9-5 we might instead incorporate an XPath command in LINGO like:

```
DATA:
CAP = @XML(/lotSizeData/periodCapacity/capacity);
ENDDATA
```

A command is also needed to locate the XML file with the data. Developers of algebraic modeling languages could also add features allowing the software to read and write an XML instance based on an accepted W3C XML Schema. The following hybrid approaches and suggestions may also be possible:

1. Making XQuery/XPath work in the same way as ODBC/SQL
2. Supporting the concept of a node set as an alternative to a relational database table
3. Adding XQuery syntax to an algebraic modeling language

A further level of standardization might be to have all algebraic modeling languages use a common underlying syntax, based upon XQuery/XPath, for database access.

9.2.4 Use XML technologies to transform XML data into a problem instance.

The focus of this chapter and the Optimization Services modeling Language (OSmL) is this 4th approach. We show how to use XML technologies to generate math programming models.

The OSmL approach is to take as input the XML files that contain the problem instance *data* and then transform the input data files into an output file that is an instance of a math program. The most natural way to do this is to use something expressly designed to transform one XML file into another. OSmL uses XQuery (and XPath, a subset of the XQuery language 2.0) to generate instances from math programs. XQuery provides very powerful algebraic modeling features, e.g. sets, for loops, if-then, union, intersection, and library modules. These are already accepted W3C standards and are what makes the OSmL modeling language symbolic, general, concise and understandable [45].

The output of any OSmL model is an OSiL instance. As long as all modeling languages use OSiL, a total of only `N` software drivers are necessary, where `N` is the number of solvers. For each of the `N` solvers, a driver is required to translate the XML data instance into a format acceptable to the solver API. We illustrate in the next section various features and some examples of this OSmL approach.

9.3 OSmL Features and Examples

9.3.1 Sets, indices and data

The first step in building an algebraic model using a modeling language is to identify the primitive sets; see [57] for a discussion of sets and indices in mathematical programming modeling. The primitive sets often correspond to the indices on the decision variables. In the relational database world these are often attributes that correspond to keys in a relation. Algebraic modeling languages have commands to create sets. Sets may be either primitive or derived sets through such operations as Cartesian product or set union. In the dynamic lot sizing example introduced in §9.2, primitive sets correspond to products and time periods. A derived set is the Cartesian product of the product and time period sets. Here is an example of set declarations in LINGO:

```
SETS:
product /1, 2/;
period /1..4/;
demand(product, period);
ENDSETS
```

The analogous concept in the XML world is the XPath node-set. Node sets corresponding to `product`, `period`, and `demand` are:

```
/lotSizeData/product
/lotSizeData/periodCapacity/capacity[@periodID]
/lotSizeData/product/period/demand
```

In an algebraic modeling language, once the sets are identified, parameters and variables are associated with the sets and referenced by indices. For example, considering only parameters, in the lot sizing example we have in LINGO:

```
SETS:
product /1, 2/: holdCost, prodCost, fixedCost;
period /1..4/: capacity;
prodperiod(product, period): demand;
ENDSETS
```

For example, `holdCost(1)` is the holding cost of the first product. The holding cost node-set is referenced in XPath by:

```
/lotSizeData/product/@holdCost
```

The position() function in XPath is then used as an index. For example, the holding cost of the first product is:

```
/lotSizeData/product[position()=1]/@holdCost
```

or, if we define `$product = /lotSizeData/product`, we can write:

```
$product[position()=1]/@holdCost
```

Similarly, the demand for product 2 in periods 3 and 4 is given by:

```
/lotSizeData/product[position()=2]/period[position()>2]/demand
```

or, in terms of `$product`, we can write:

```
$product[position()=2]/period[position()>2]/demand
```

One advantage of nesting time period nodes within product nodes (Figure 9-5) over a more traditional tabular approach is that we can use the position function to easily index both the product and time periods. An important aspect of this approach is that we are using the input XML for data only; the input files do not contain any information about constraints or variables. The input XML files need only contain all of the model sets and parameters (or sufficient information to generate them).

9.3.2 OSmL examples and comparison with other modeling languages

Figure 9-7 shows the dynamic lot sizing model introduced in § 9.2.3 in AMPL. AMPL currently does not have built in support to retrieve the dynamic lot sizing XML data shown in Figure 9-5, so the model is non-working. It is shown here for illustration and comparison with the OSmL language.

```
#SET, PARAMETER, AND VARIABLE CONSTRUCTIONS
param T;
set PROD;
set LINKS = {PROD, 1..T};
param HC {PROD} ;
param FXC {PROD} ;
param CAP {1..T} ;
param DEM {LINKS};
param PCOST {PROD, 1..T} ;

#VARIABLE DECLARATION
var x  {PROD, 1..T} >= 0;
var I  {PROD, 0..T} >=0;
var y  {PROD, 1..T}binary;

#OBJECTIVE CONSTRUCTION
minimize Total_Cost:
 sum {i in PROD} I[i, 0]  +  sum {i in PROD, t in 1..T} (PCOST[i, t]*x[i, t] + HC[i]*I[i, t] + FXC[i]*y[i, t]);

# INITIAL INVENTORY CONSTRAINTS
subject to Init_Inv {i in PROD}:
   I[i, 0] = 0.0;

# DEMAND CONSTRAINTS
subject to Balance {i in PROD, t in 1..T}:
   x[i, t] + I[i, t - 1] - I[i, t] = DEM[i, t];

# FIXED CHARGE CONSTRAINTS
subject to Fixed_Charge {i in PROD, t in 1..T}:
   x[i, t] <= CAP[ t]*y[i, t];

# CAPACITY CONSTRAINTS
subject to Capacity {t in 1..T}:
  sum {i in PROD} x[i, t] <= CAP[ t];
```

Figure 9-7: Dynamic lot sizing model in AMPL (nonworking with the dynamic lot size XML data).

Figure 9-8 shows the dynamic lot sizing model in OSmL and how it retrieves the data shown in Figure 9-5. It is a working model.

```
(: SET AND PARAMETER CONSTRUCTIONS:)
let $capacity := doc("./lotsizeData.xml")/lotSizeData/periodCapacity/capacity
let $products := doc("./xml/ds800m.xml")/lotSizeData/product
let $N := count($products)
let $T := count($capacity[periodID])
let $FXC := data($products/@fixedCost)
let $HC := data($products/@holdCost)
let $PCOST := data($products/@prodCost)
let $CAP := data($capacity/text())
let $DEM := $products/period/demand
let $PROD := (1 to $N)
return  <mathProgram>
(: VARIABLE DECLARATION :)
<variables>{ for $i in (1 to $N),  $t in (1 to $T) return
  (<var name="X[{$i},{$t}]"/>,
  <var name="I[{$i},{$t}]"/>,
  <var name="Y[{$i},{$t}]" type="B"  />) }
</variables>

(: OBJECTIVE FUNCTION   :)
<obj maxOrMin="min" name="Total_Cost">
SUM(for $i in (1 to $N), $t in (1 to $T) return
{$PCOST[$i]}*X[{$i},{$t}] + {$FXC[$i]}*Y[{$i},{$t}] + {$HC[$i]}*I[{$i},{$t}])
</obj>

<constraints>
(: INITIAL INVENTORY CONSTRAINTS :)
{for $i in $PROD return
<con name="inventory[{$i}]"> I[{$i},0]  = 0 </con>  }

(: DEMAND CONSTRAINTS :)
{for $i in $PROD,  $t in (1 to $T)
let $demand := ($products[$i]/period[@periodID=$t]/demand/text())    return
<con name="demand[{$i},{$t }]">  X[{$i},{$t}] + I[{$i},{$t - 1}] - I[{$i},{$t}] = {$demand} </con>  }

(: FIXED CHARGE CONSTRAINTS :)
{for $t in (1 to $T),  $i in (1 to $N)    return
<con name="fixed_charge[{$i},{$t }]" > X[{$i},{$t}]-{$CAP[$t]}*Y[{$i},{$t}] <= 0</con> }

(:  CAPACITY CONSTRAINTS  :)
{for $t in (1 to $T) return
<con name="capacity[{$t}]">  SUM(for $i in (1 to $N)  return  X[{$i},{$t}])<= {$CAP[$t]} </con>}
</constraints> </mathProgram>
```

Figure 9-8: Dynamic lot sizing model in OSmL (working with the dynamic lot size XML data).

Table 9-1 gives a side-by side comparison between AMPL and OSmL on different constructs.

	AMPL (no data retrieval)	**OSmL (and XML data retrieval)**
Sets and parameters,	param **T**; set **PROD**; set **LINKS** = {PROD, 1..T}; param **HC** {PROD} ; param **FXC** {PROD} ; param **CAP** {1..T} ; param **DEM** {LINKS}; param **PCOST** {PROD, 1..T} ;	let $capacity := doc("lotsizedata.xml")/lotSizeData/periodCapacity/capacity let $**T** := count($capacity) let $products := doc("lotsizedata.xml")/lotSizeData/product [(1, 2)] let $N := count($products) let $**PROD** := (1 to $N) let $**HC** := $products/@holdCost let $**FXC** := $products/@fixedCost let $**CAP** := $capacity/text() let $**DEM** := $products/period/demand let $**PCOST** := data($products/@prodCost)

Variables	var **x** {PROD, 1..T} >= 0;
	var **I** {PROD, 0..T} >=0;
	var **y** {PROD, 1..T}binary;
		<variables>{for $i in (1 to $N), $t in (1 to $T) return
		(<var name="**X**[{$i},{$t}]"/>,
		<var name="**I**[{$i},{$t}]"/>,
		<var name="**Y**[{$i},{$t}]" type="B" />) }
		</variables>

Objective	minimize Total_Cost: **sum {i in PROD} I[i, 0] +** **sum {i in PROD, t in 1..T}** **(PCOST[i, t]*x[i, t] +** **HC[i]*I[i, t] +** **FXC[i]*y[i, t]);**	<obj maxOrMin="min" name="Total_Cost"> **SUM(for $i in (1 to $N), $t in (1 to $T) return** **{$PCOST[$i]}*X[{$i},{$t}] +** **{$FXC[$i]}*Y[{$i},{$t}] +** **{$HC[$i]}*I[{$i},{$t}])** </obj>
Initial inventory constraints	subject to Init_Inv {i in PROD}: **I[i, 0] = 0.0;**	{ for $i in $PROD return <con name="inventory[{$i}]"> **I[{$i},0] = 0** </con>}

demand constraints (or balance constraints)	subject to Demand {i in PROD, t in 1..T}:
	x[i, t] + I[i, t - 1] - I[i, t] =
	DEM[i, t];
		{for $i in $PROD, $t in (1 to $T)
		let $demand := ($products[$i]/period[@periodID=$t]/demand/text())
		return <con name="demand[{$i},{$t }]">
		X[{$i},{$t}] + I[{$i},{$t - 1}] - I[{$i},{$t}] = {$demand} </con>}

Fixed charge constraints	subject to Fixed_Charge {i in PROD, t in 1..T}: **x[i, t] <= CAP[t]*y[i, t]**;	{for $t in (1 to $T), $i in (1 to $N) return <con name="Fixed_charge[{$i},{$t }]"> **X[{$i},{$t}] <= {$CAP[$t]}*Y[{$i},{$t}]** </con>}
Capacity constraints	subject to Capacity {t in 1..T}: sum {i in PROD} **x[i, t] <= CAP[t]**;	{for $t in (1 to $T) return <con name="capacity[{$t}]"> SUM(for $i in (1 to $N) return **X[{$i},{$t}]) <= {$CAP[$t]}** </con>}

Table 9-1: Comparison between AMPL and OSmL.

The basic "set" in the XQuery-based OSmL language is an ordered sequence. All "XQuery variables" begin with a "$" sign. An XQuery engine evaluates what is in { }. Decision variables are declared in `<variables> ... </variables>`. OSmL, however, does not require declaring variables; any variable not declared assumes certain default features (e.g. `type="C", lb="0"`). An objective function is constructed in `<obj> ... </obj>` and each constraint is added in `<con> ... </con>`. To make mathematical modeling easier, we added several macros to the standard XQuery language. For example, the **SUM** function as we use it is not provided by XQuery. A preprocessor inside the OSmL compiler converts the macros into standard XQuery language.

Since OSmL is XQuery-based, we can automatically inherit many powerful features from the XQuery language. For example, we can use built-in Java functions:

```
declare namespace math="java:java.lang.Math";
```

The objective function in OSmL with a "square root function" on fixed cost may look:

```
<obj maxOrMin="min" name="Total_Cost">
SUM(for $i in (1 to $N), $t in (1 to $T) return
{$PCOST[$i]}*X[{$i},{$t}] +
{math:sqrt($FXC[$i])}*Y[{$i},{$t}] +
{$HC[$i]}*I[{$i},{$t}])
</obj>
```

We can also use many other built-in features of XQuery such as the "`where`" clause to put conditions on sets or "`if-then`" logic for more complex data manipulation.

9.3.3 Model compilation, instance generation and auxiliary software

The OSmL model compilation and OSiL instance generation process are illustrated in Figure 9-9. We could use the XQuery to transform the OSmL model into an instance that validates against the OSiL Schema. However, the OSiL Schema is designed for minimizing file size and for easy integration with solver APIs. So rather than use XQuery to directly generate an instance file in the OSiL format, we generate an intermediate instance file that has a syntax that makes the XQuery-based OSmL language very easy to construct. Then the intermediate XML instance is transformed into a final OSiL instance file.

We have been emphasizing the fact that OSmL is XQuery based, but OSmL is not exactly an XQuery language. OSmL has extra pre-built constructs tailored for optimization problems. For example, the relational operators "<", "<=", ">" and ">=" are represented in XQuery as "`&gt;`", "`&gt;=`", "`&lt;`" and "`&lt;=`" to avoid conflicts with the XML tags (< >). As the relational operators appear very often in optimization, OSmL allows users to use "<", "<=", ">" and ">=" directly and it has a "preprocessor" to detect these operators and convert them to the XQuery language specification. Also OSmL adds some macros such as the "`SUM`" function, as again these macros provide extra convenience for mathematical modeling. The OSmL preprocessor expands these macros to XQuery equivalents.

Of course if a modeler is sophisticated in the XQuery language, he can directly use the standard XQuery representations to construct an optimization model and avoid using OSmL macros. In this situation, OSmL is a pure XQuery language.

A pure XQuery (original or after preprocessing) is sent to an XQuery processor and is compiled into an intermediate XML instance. The immediate instance is parsed and converted into the standard OSiL instance and sent out to an Optimization Service using an OS communication agent.

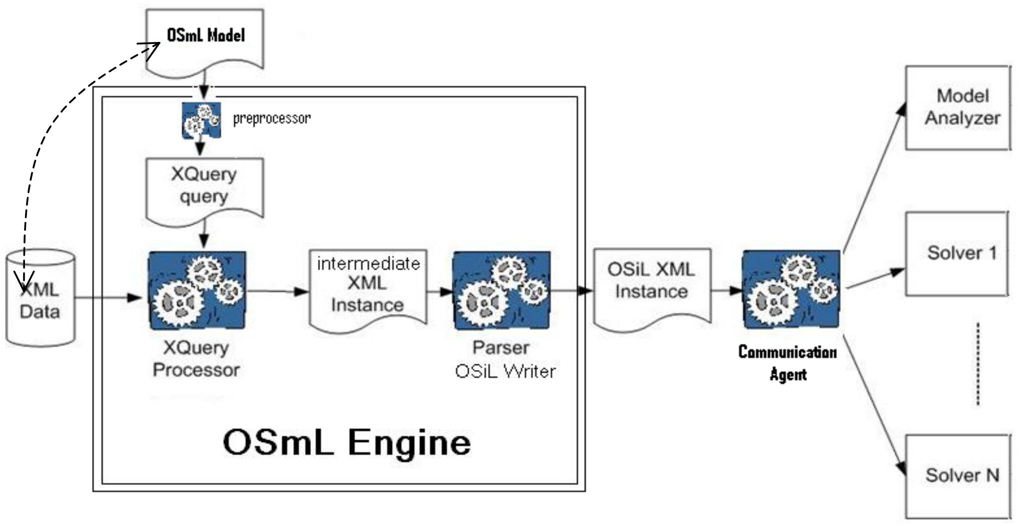


Figure 9-9: The OSmL process.

Numerous auxiliary software packages are available that implement XQuery and XPath. There are two major camps: Microsoft .NET and Java. The most recent release of the Microsoft development tool, Visual Studio .NET [84], contains numerous classes for manipulating and transforming XML data. These classes are available to all of the .NET languages. Indeed, a major advantage of using .NET software is that Microsoft has done such an excellent job of integrating XML into Visual Studio .NET. The downside of .NET is that .NET software runs on the Windows platform (although Ximian has announced the launch of the Mono project [120] to create an open source implementation of the .Net development framework). However, the actual XQuery files are platform independent. There is no problem with sharing model files among users of different platforms.

A number of Java open source XQuery and XPath tools are also available. There is Saxon (for XQuery and XPath) written by Michael Kay [18] and Xalan (for XSLT, a C++ version is also available) by the Apache organization [3]. Both Saxon and Xalan can be used from the command line or called from a Java Servlet or a standalone Java program. Both Xalan and Saxon implement the Java API for XML Processing (JAXP). This makes it very convenient to write portable software that can call either Saxon or Xalan to transform XML. There is also XML Spy from Altova [1] and Stylus Studio from Progress Software [96], which are a proprietary XML development environments. Both are equipped with some very nice graphical tools for constructing XML-related files.

9.3.4 Getting data

XQuery and XPath are designed to work with input data in an XML format. In this section we show that there are numerous tools for transforming non-XML data into XML data. Most of the data used in a math program will reside in a

• spreadsheet
• desktop database (e.g. Microsoft Access)
• ASCII flat file
• enterprise database (e.g. DB2, Oracle, SQL Server)
• XML file

We discuss converting each source into XML. There are several options with a spreadsheet or desktop database. If the spreadsheet or database is part of Microsoft Office 2002 (or later) it is possible to directly export each table in the database, or range in the spreadsheet, as an XML file. If the desktop spreadsheet or database are ODBC or OLE-DB compliant, then one can write a program in a procedural language such as C++ or Java to access the data using ODBC or OLE-DB, read it into memory, and then use DOM (document object module) to create an XML representation of the data. There is some overhead in creating the DOM and storing it in main memory. An alternative approach is to write a custom SAX parser and feed the information directly into a JAXP compliant XSLT processor. DOM and SAX are alternative APIs for processing XML.

If the flat file is an ASCII flat file, several options exist. First, one could import the flat file into a desktop database such as Microsoft Access and then save it as an XML file. A second option is to write a C++ or Java program to parse the file and then use DOM or SAX create an XML representation of the data.

Much of the data for large models is stored in enterprise corporate databases. Fortunately, the major database vendors are adding features to their products that allow the user to submit an SQL query to the database and get the result back in XML format. There are JDBC drivers for the most widely used databases. Thus, one could write a Java program and use JDBC and SQL to query the database, get the result as XML, and then transform the XML using a JAXP transformation engine such XALAN or Saxon. This process is also easily carried out using Visual Studio .NET. There are many classes available to any of the .NET languages for reading data in XML format from a relational database and then transforming it to XML.

Ideally, the input data is initially in XML format. However, some XML structures are more amenable to transformation into a mathematical model than others. Of course most XML transformation tools are designed to transform one XML file into another without much difficulty.

There are products expressly for the purpose of accessing data stored in different formats and viewing the data as XML. Two such products include BEA's *Liquid Data* [8]and IBM's *XPeranto* [62]. The trend is obvious: make it easy to gather data from various sources and convert it into XML. This makes the OSmL methodology we are proposing even more viable over time.

CHAPTER 10 FUTURE WORK AND DERIVED RESEARCH FROM OPTIMIZATION SERVICES

10.1 The Optimization Services Project

Optimization Services is a young research area that has potential for many benefits to operations research and the optimization community. Motivated by a vision of the next generation of optimization software and standards, Optimization Services deals with a wide variety of issues that have accumulated over the past few decades in computing and optimization. This work addresses design as well as implementation issues by providing a general and unified framework for such tasks as standardizing problem representation, automating problem analysis and solver choice, working with new Web service standards, scheduling computational resources, benchmarking solvers, and verification of results – all in the context of the special requirements of large-scale computational optimization. The criteria required of Optimization Services must therefore be very high. Improving the quality of Optimization Services related standards, tools and systems should be a constant effort for our future work. Adapting to the new needs of researchers and developers and best serving the ultimate users should always be the goal of Optimization Services, which should therefore be highly scalable for future extensions and very simple to use. In the next sections, we briefly describe the most imminent future work of the OS project and some of the derived research projects and business models.

10.2 Standardization

Optimization Services involves a large set of standard protocols that need to be adopted quickly and universally. The standardization process can start from working group notes, and go through stages such as working drafts, candidate recommendations, and finally become recommended as standards. Such a process not only requires further research efforts such as new optimization problem extensions but also entails more organizational efforts that require formal establishment of collaborations under the Optimization Services framework.

10.3 Problem Repository Building

With the standardization of various problem representations naturally comes the task of building repositories of optimization problem instances using the standard schemas. Problem repositories no longer need to be categorized by the format the problems are using. Rather they are only classified by the different optimization types supported in the OS standards.

10.4 Library Building

The OS library and the OS server described in Appendix B are provided to facilitate the adoption and use of the OS standards. Besides the original OS designers, other researchers and developers are free to develop their own OS-compatible libraries, such as parsers (readers and writers) of standard instances, and communication agents to transmit these instances.

10.5 Derived Research in Distributed Systems

A distributed system leaves open many questions in coordination, job scheduling and congestion control. One distinct issue for example is how optimization "jobs" should best be assigned to run on available registered services after the optimization types are determined. The usual centralized scheme of an optimization server maintains one queue for each solver/format combination, along with a list of the workstations on which each solver can run. In a decentralized environment, we may still want to maintain this scheduling control, while at the same time making the scheduling decisions more distributed, i.e. transferring some controls to the solver service sides.

Further study is needed to better understand how categorization of optimization problem instances together with statistics from previous runs can be used to improve scheduling decisions. As just one example, an intelligent scheduler should not assign two large jobs to a single-processor machine, since they will only become bogged down contending for resources; but a machine assigned one large job could also take care of a series of very small jobs without noticeable degradation to performance on either kind of job. Both the kind and size of optimization instances must be assessed in order to determine which should be considered "large" and which "very small" for purposes of this scheduling approach.

10.6 Derived Research in Decentralization

The central issue in a decentralized architecture is the design of a registration and discovery mechanism for acting on service requests. For example the optimization registry could assign requests based on some overall model of solver performance and resource availability. Requests can be scheduled after they are matched to some services, or scheduling could be made an integral part of the assignment process. Pricing could involve agent "rents" as well as charges determined by various measures of resource use.

Besides keeping and maintaining information on optimization solvers and other services, one critical and more complex role of an optimization registry in a decentralized environment is a "more confident" determination of appropriate solvers. A relatively easy and straightforward scheme can rely on a database that matches solvers with problem types they can handle. Characteristics of a problem instance, determined from the analyzers, can be used to automatically generate a query on the database that will return a list of appropriate solver services. But how should solver recommendations deal with problem types (e.g. bound-constrained optimization) that are subsets of other problem types (e.g. nonlinear optimization)? Or how can recommendations be extended to solver options?

For these purposes, a straightforward database approach for a server or registry may not be adequate. Developers will consider more sophisticated ways of determining recommendations, such as through business rules systems. A more complicated and advanced scheme may consider extensions to generate lists ranked by degree of appropriateness.

10.7 Derived Research in Local Systems

In §2.4, we listed the interface and communication agent as a distinct component in an optimization system. The Optimization Services framework standardizes all the communications between *any* two Optimization Services components on an OS *distributed* system. The framework does *not* standardize *local* interfacing.

As mentioned in the previous chapters, related projects such as COIN [23] and derived research from Optimization Services such as the Optimization Services instance Interface (OSiI), Optimization Services option Interface (OSoI), and Optimization Services result Interface (OSrI) are intended to do this job. The COIN project includes the OSI (Open Solver Interface) library which is an API for linear programming solvers, and NLPAPI, a subroutine library with routines for building nonlinear programming problems. Another proposed nonlinear interface by Halldórsson, Thorsteinsson, and Kristjánsson is MOI (Modeler-

Optimizer Interface [60]), which specifies the format for a callable library. This library is based on representing the nonlinear part of each constraint and the objective function in post-fix (reverse Polish) notation [2] and then assigning integers to operators, characters to operands, integer indices to variables and finally defining the corresponding set of arrays. The MOI data structure then corresponds to the implementation of a stack machine. A similar interface is described in the LINDO API manual [74]. The Optimization Services framework is complementary to all of the standardization of local interfaces. The connection between Optimization Services and local interfacing is illustrated in Figure 7-1, shown again below.

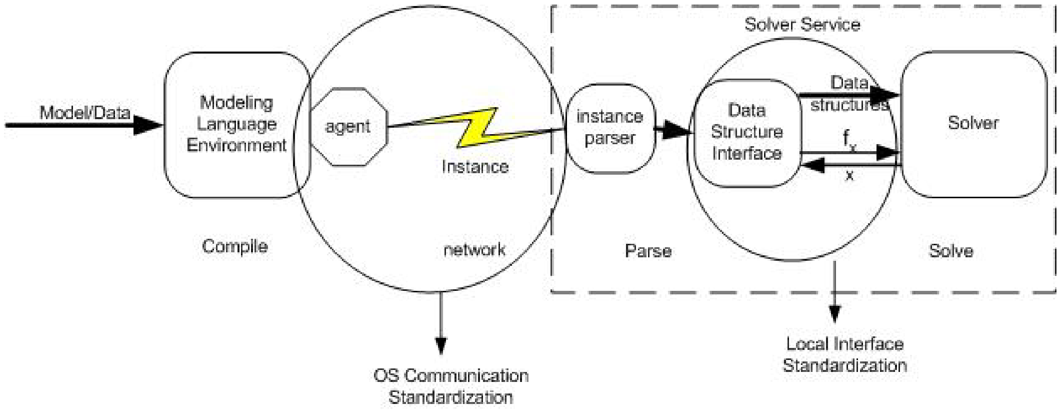


Figure 10-1: Relationship between OS Communication and local interface standardization.

10.8 Derived Research in Optimization Servers

Optimization Services is motivated by the current issues faced by many optimization servers. More specifically Optimization Services is intended to provide the next-generation NEOS [29]. As mentioned in §3.1.4, the effects of Optimization Services on NEOS are multifaceted:

- The NEOS server and its connected solvers will communicate using the Optimization Services framework, e.g. using standard representation for data communication.
- External optimization submissions can still be kept as flexible as possible and may become even more flexible. At least one more networking mechanism will be provided, i.e. the communication based on the Optimization Services Protocol (OSP). That means NEOS will add an interface so that it can be invoked exactly as what's specified by the Optimization Services hook-up Language (OShL, Chapter 7). It will also accept OSiL as a standard input, and may gradually deprecate the other formats.

- The entire Optimization Services system over the Internet can be viewed as a new decentralized NEOS. In effect the old NEOS will become another OS-compatible solver in the new system. The "NEOS server" can then solve more types of optimization by delegating the job further to different solvers behind it. We therefore regard the old NEOS as a "meta-solver" registered on the new Optimization Services system.

10.9 Derived Research in Computational Software

With the advent of Optimization Services and its standard OSP protocol, related software developers may need to think about how to best adapt to the OS framework and be "OS-compatible." The issues are detailed in Chapter 2.

There have already been two immediate projects that are related to the Optimization Services framework. One is the Optimization Services modeling Language described in Chapter 9 and the other is the IMPACT solver development project that is under development by Professor Sanjay Mehrotra's group at the Industrial Engineering and Management Sciences department at Northwestern University. The two projects are the two sides of Optimization Services: client and service. Both are natively built *for* the Optimization Services framework and strictly follow the Optimization Services Protocol.

There are existing modeling languages and solvers that are or will be adapted (by writing wrapper classes) to the Optimization Services framework such as the AMPL modeling language [49], the Lindo solver [74] and Knitro solver [121]. Solvers from the NEOS system [29] are the next target of the Optimization Services project.

10.10 Derived Research in Computational Algorithms

The design of effective and efficient computational algorithms that fit the Optimization Services design is important. Optimization Services immediately opens up the questions of how to best utilize the available services on the OS network. Following are some of the potential research areas in computational algorithms related to Optimization Services:

- Parallel computing where many registered services can simultaneously solve the same type of optimization problems.
- Optimization via simulation where simulation services are located remotely from the optimization solver service.
- Optimization job scheduling at the registry side and queuing at the service side.
- Analyzing optimization instances according to the needs of the OS registry.

- Modeling and compilation that generates OSiL instances quickly and accurately.
- Efficient OSxL instance parsing and preprocessing algorithms.
- Effective Optimization Services process orchestration.
- Also as the OS standards allow representations of various optimization types, optimization algorithm development (e.g. in stochastic programming) that has been lagging due to lack of good representations can hopefully get a boost.

10.11 Commercialization and Derived Business Models

Optimization Services, though itself an open framework, does not prevent registered services and related business to be commercialized. Following are some of the related business models:

- Modeling language developers leverage on using Optimization Services to provide more and better solver access to their customers and become more competitive.
- Solver developers concentrate on developing better algorithms to increase their competitiveness without worrying about representation, communication and interfacing that are taken care by the OS standards.
- Developers can commercialize the libraries that they build for the Optimization Services, e.g. readers and writers of standard instances.
- Registry/server developers can provide auxiliary services such as storage services, BPEL-related flow orchestration services (§7.3), advertisement services and consulting services.
- Auxiliary services and software such as analyzer services and benchmarkers may possibly charge fees to involved parties.
- Solver service owners may adopt a "computing on demand" model by charging the user for using their solver services.
- A solver service owner may also adopt a "result on demand" model by reporting the objective results that his solver service has found but hiding the solutions that are only to be revealed when the user agrees to pay. For example in the OSrL result instance (§6.4) that the solver service returns, it may write out only the `<objectiveValue>` value and in the `<solverMessage>` element it may provide the instructions for obtaining the `<variableSolution>` values.

REFERENCES

[1] Altova XML Spy, http://www.altova.com/ (2005).

[2] Aho, A.V., R. Sethi, J.D. Ullman, Compilers: Principles, Techniques and Tools, Addison-Wesley, Reading, MA (1986).

[3] Apache Software Foundation, http://www.apache.org/ (2005).

[4] Apache Software Foundation, Apache Jakarta Tomcat, http://jakarta.apache.org/tomcat/ (2005).

[5] Apache Software Foundation, Axis, http://ws.apache.org/axis/ (2005).

[6] Apache XML Project, Apache Xindice, http://xml.apache.org/xindice/ (2005).

[7] A. Atamtürk, E.L. Johnson, J.T. Linderoth, and M.W.P. Savelsbergh., A relational modeling system for linear and integer programming. Operations Research (2000) 4:263–283.

[8] BEA Inc. BEA Liquid Data forWebLogic, http://www.bea.com/products/weblogic/liquiddata/index.shtml (2005).

[9] J. Bigus and J. Bigus, Constructing Intelligent Agents with Java, John Wiley & Sons (1997).

[10] J.R. Birge, M.A.H. Dempster, H.I. Gassmann, E.A. Gunn, A.J. King and S.W. Wallace, A Standard Input Format for Multiperiod Stochastic Linear Programs. COAL Newsletter 17 (1987) 1-19.

[11] J.R. Birge, F. Louveaux, Introduction to stochastic programming, Springer Series in Operations Research, Springer Verlag, New York (1997).

[12] J.J. Bisschop and A. Meeraus, On the Development of a General Algebraic Modeling System in a Strategic Planning Environment. Mathematical Programming Study 20 (1982) 1-29.

[13] R.E. Bixby, Solving Real-World Linear Programs: A Decade and More of Progress. Operations Research 50 (2002) 3-15.

[14] S. Brin, L. Page, Anatomy of a Large-Scale Hypertextual Web Search Engine, Proceeding 7th International World Wide Web Conference (1998).

[15] Bradley, G., Network and graph markup language (NaGML) – data file formats. Tech.Rep. NPS-OR-04-007, Department of Operations Research, Naval Postgraduate School, Monterey, CA, USA. Available from the author, bradley@nps.navy.mil. (2004).

[16] A. Brooke, D. Kendrick and A. Meeraus, GAMS: A User's Guide, Release 2.25. Scientific Press/Duxbury Press (1992). See also http://www.gams.com.

[17]T. Berners-Lee, etc., W3C, http://www.w3c.org (2003).

[18] T. Berners-Lee, J. Hendler, O. Lassila, The Semantic Web, Scientific American (05 2001). See also http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21&catID=2.

[19] Business Integration Journal Online, http://bijonline.com (2005).

[20] Chang, T.-H., Modeling and presenting mathematical programs with xml:lp. Masters thesis, Department of Management, University of Canterbury, Christchruch, NZ (2003).

[21] J.W. Chinneck, Analyzing Mathematical Programs Using MProbe. Annals of Operations Research 104 (2001) 33-48.

[22] J. Choobineh, SQLMP: a data sublanguage for representation and formulation of linear mathematical models. ORSA Journal of Computing (1991) 3:358–375.

[23] COmputational INfrastructure for Operations Research (COIN-OR), http://www.coin-or.org (2005).

[24] A. R. Conn, N. I. M. Could and Ph. L. Toint, LANCELOT: A FORTRAN Package for Large-Scale Nonlinear Optimization. Springer Verlag (1992).

[25] J. Czyzyk, M.P. Mesnier and J.J. Moré, The NEOS Server. IEEE Journal on Computational Science and Engineering 5 (1998) 68-75.

[26] G. B. Dantzig, Linear Programming and Extensions. Princeton University Press, Princeton, NJ (1963).

[27] E.D. Dolan, NEOS Server 4.0 Administrative Guide. Technical Memorandum ANL/MCS-TM-250, Argonne National Laboratory, Argonne, IL (2001).

[28] E.D. Dolan, R. Fourer, J.-P. Goux and T.S. Munson, “Kestrel: An Interface from Modeling Systems to the NEOS Server. “ Technical report, Mathematics and Computer Science Division, Argonne National Laboratory (September 2002).

[29] E.D. Dolan, R. Fourer, J.J. Moré and T.S. Munson, “Optimization on the NEOS Server.” SIAM News 35, 6 (2002) 4, 8-9.

[30] E.D. Dolan and J.J. Moré, Benchmarking Optimization Software with Performance Profiles. Mathematical Programming 91 (2002) 201-213.

[31] E.D. Dolan, J.J. Moré and T.S. Munson, Measures of Optimality for Constrained Optimization. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory (April 2002).

[32] elipse.org, Eclipse IDE, http://www.eclipse.org (2005).

[33] Ezechukwu, O., I. Maros, OOF: open optimization framework. Tech. Rep. ISSN 1469-4174, Department of Computing, Imperial College of London, London, UK (2003).

[34] Ezechukwu, O., I. Maros, AML: Algebra Markup Language. Tech. Rep. ISSN 1469-4174, Department of Computing, Imperial College of London, London, UK (2003).

[35] Ezechukwu, O., I. Maros, what is the Open Optimization Framework (Elsinore), http://www.doc.ic.ac.uk/~oce/elsinore/introduction.htm, Department of Computing, Imperial College of London, London, UK (2003).

[36] eXist, Exist Open Source Native XML Database, http://exist.sourceforge.net (2005).

[37]B. Dominguez-Ballesteros, G. Mitra, C. Lucas and N.-S. Koutsoukis, Modeling and Solving Environments for Mathematical Programming (MP): A Status Review and New Direction. Journal of Operational Research Society 53 (2002) 1072-1092.

[38] M.D. Ferris, M. Mesnier and J.J. Moré, NEOS and Condor: Solving Optimization Problems over the Internet. ACM Transactions on Mathematical Software 26 (2000) 1-18.

[39] T. Finin and Y. Labrou, eds., UMBC agentWeb, http://agents.umbc.edu (2003).

[40] I. Foster, Designing and Building Parallel programs, Addison Wesley (1994).

[41] I.Foster and C. Kesselman, eds., Open Grid Services Architecture (OGSA), http://www.globus.org/ogsa/ (2003).

[42] I.Foster and C. Kesselman, eds., The Globus Alliance, http://www.globus.org (2003).

[43] I.Foster and C. Kesselman, eds., Open Grid Services Architecture (OGSA), http://www.globus.org/ogsa/ (2005).

[44] R. Fourer, Next Generation Servers for Optimization as an Internet Resource, http://users.iems.nwu.edu/~4er/NEOSprop.pdf (2003).

[45] R. Fourer, Modeling Languages Versus Matrix Generators for Linear Programming. ACM Transactions on Mathematical Software 9 (1983) 143-183.

[46] R. Fourer, Optimization Frequently Asked Questions. Optimization Technology Center of Northwestern University and Argonne National Laboratory, www-unix.mcs.anl.gov/otc/ Guide/faq/ (2003).

[47] R. Fourer and D.M. Gay, Extending an Algebraic Modeling Language to Support Constraint Programming. Technical Report, Department of Industrial Engineering and Management Sciences, Northwestern University (2001).

[48] R. Fourer, D.M. Gay and B.W. Kernighan, A Modeling Language for Mathematical Programming. Management Science 36 (1990) 519-554.

[49] R. Fourer, D.M. Gay and B.W. Kernighan, AMPL: A Modeling Language for Mathematical Programming, 2nd edition. Duxbury Press, Pacific Grove, CA (2002). See also www.ampl.com.

[50] R. Fourer and J.-P. Goux, “Optimization as an Internet Resource.” Interfaces 31, 2 (2001) 130-150.

[51] R. Fourer and L. Lopes, A management System for Decompositions in Stochastic Programming. Under revision for Annals of Operations Research (2002).

[52] R. Fourer and L. Lopes, A filtration-Oriented System for Modeling Stochastic Programming. Draft Paper, Department of Industrial Engineering and Management Sciences, Northwestern University (2003).

[53] R. Fourer, L. Lopez, K. Martin, FMLLP: A W3C XML Schema for Linear Programming. Draft Paper, Department of Industrial Engineering and Management Sciences, Northwestern University (2003).

[54] Robert Fourer, Jun Ma, Kipp Martin, Optimization Services, www.optimizationservices.org (2005).

[55] Robert Fourer, Jun Ma, Kipp Martin, Optimization Services, www.optimizationservices.net (2005).

[56] D.M. Gay, Hooking Your Solver to AMPL. Technical report, Bell Laboratories, Murray Hill, NJ (1997); http://www.ampl.com/REFS/abstracts.html#hooking2.

[57] A.M. Geoffrion. Indexing in modeling languages for mathematical programming, Management Science, (1992) 38:325–344.

[58] H.J. Greenberg, A functional Description of ANALYZE: A Computer-Assisted Analysis System for linear programming Models. ACM Transations on Mathematical Software 9 (1983) 18-56.

[59] W. Gropp and J.J. Moré, Optimization Environments and the NEOS Server. In Approximation Theory and Optimization, M.D. Buhmann and A. Iserles, eds., Cambridge University Press (1997) 167-182.

[60] B.V. Halldórsson, E.S. Thorsteinsson and B. Kristjánsson, A modeling Interface to Nonlinear Programming Solvers – An Instance: *x*MPS, the Extended MPS Format. Technical report, Department of Mathematical Sciences and Graduate School of Industrial Administration, Carnegie Mellon University (2000).

[61] IBM Inc., BPEL http://www-128.ibm.com/developerworks/library/specification/ws-bpel/ (2005).

[62] IBM Inc., Xperanto, http://www.almaden.ibm.com/software/dm/Xperanto/index.shtml (2005).

[63] IBM Inc., Web Services flow Language (WSFL), http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf (2001).

[64] IBM Inc., Web Services Inspection Language (WSFL), http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html (2001).

[65] IBM Inc., WebSphere MQSeries, http://www.ibm.com/software/mqseries (2005).

[66] M. Kay. Saxon the XSLT processor, http://saxon.sourceforge.net/ (2005).

[67] W.K. Klein Haneveld, Duality in Stochastic Linear and Dynamic Programming, Lecture Notes in Economics and Mathematical Systems, Vol. 274, Springer-Verlag, Berlin (1986).

[68] T. Hastie, R. Tibshirani, J. H. Friedman, The Elements of Statistical Learning. Springer-Verlag (2001).

[69] Kristjánsson, B., Optimization modeling in distributed applications: how new technologies such as XML and SOAP allow OR to provide web-based services, http://www.maximal-usa.com/slides/Svna01Max/index.htm (2001).

[70] C.A.C. Kuip, Algebraic Languages for Mathematical Programming. European Journal of Operational Research 67 (1993) 25-51.

[71] D. Lange and M.Oshima, Programming and Deploying Java Mobie Agents with Aglets, Addison-Wesley (1998).

[72] M. Litzkow, M. Livny, and M.W. Mutka, Condor – A Hunter of Idle Workstations. Proceedings of the 8th International Conference of Distributed Computing Systems (1998) 104-111.

[73] J.P.Lewis and Ulrich Neumann, Performance of Java versus C++, http://www.idiom.com/~zilla/Computer/javaCbenchmark.html (2004).

[74] Lindo Systems, Inc., API user's manual, Tech. Rep., Lindo Systems, Inc. http://www.lindo.com/lindoapi_pdf.zip (2002).

[75] M.S. Lobo, L. Vandenberghe, and S. Boyd, Application of second-order cone programming, Linear Algebra Application, 284 (1998) 193-228.

[76] Boris Lublinsky, An Introduction to Business Process Execution Language, Business Integration Journal, October Issue (2004) 58-60.

[77] Markowitz, H., Portfolio Selection, Efficient Diversification of Investments. John Wiley & Sons, New York (1957).

[78] Marriott, K. and P. Stuckey, Programming with Constraints An Introduction. The MIT Press, Cambridge, MA (1957).

[79] David Megginson, Simple API for XML (SAX), http://www.saxproject.org (2003).

[80] Microsoft Inc., BPEL, http://msdn.microsoft.com/library/en-us/dnbiz2k2/html/BPEL1-1.asp (2005).

[81] Microsoft Inc., BizTalk server, http://www.microsoft.com/biztalk/default.mspx (2004).

[82] Microsoft Inc., MSMQ, http://www.microsoft.com/windows2000/technologies/communications/msmq/ (2000).

[83] Microsoft Inc., XLANG, http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm (2001).

[84] Microsoft Inc., Visual Studio .NET, http://msdn.microsoft.com/library/default.asp?url=/vs/techinfo/Default.%asp (2002).

[85] Microsoft Inc., ODBC, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/odbc/htm/dasdkodbcoverview.asp (2004).

[86] B.A. Murtagh, Advanced Linear Programming: Computation and Practice, McGraw-Hill (1981).

[87] Napster.com, P2P Technology, http://www.napster.com.

[88] NEOS Server for Optimization, http://www-neos.mcs.anl.gov/neos/ (2005).

[89] NEOS Server: SDPA, http://www-neos.mcs.anl.gov/neos/solvers/SDP:SDPA/ (2005).

[90] OASIS, http://www.oasis-open.org (2005).

[91] OASIS, BPEL, http://www.oasis-open.org/committees/wsbpel/charter.php (2005).

[92] Optimization Services, www.optimizationservices.org (2005).

[93] Optimization Services, www.optimizationservices.net (2005).

[94] Oracle Inc. http://www.oracle.com/bpel (2005).

[95] M.J.D. Powell, An efficient method for finding the minimum of a function of several variables without calculating derivatives, Computer J. 7 (1964) 155-162.

[96] Progress Software Stylus Studio, http://www.stylusstudio.com/ (2005).

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

[98] P. Sandhu, The MathML Handbook, Charles River Media, MA (2003).

[99] Aaron Skonnard and Martin Gudgin, Essential XML Quick Reference, Pearson Education (2002).

[100] Sun Microsystems, Jini Network Technology, http://www.sun.com/software/jini/ (2005).

[101] T. Tirpak, L. Lach, J. Lopez, J. Ma, W. Xiao, Virtual Prototyping, Motorola Inc., http://www.motorola.com/content/0,3306,263,00.html (2003).

[102]UDDI.org, Universal Description, Discovery, and Integration (UDDI), http://www.uddi.org (2003).

[103] E. Van der Vlist, XML Schema. O'Reilly & Associates (2002).

[104] W3C, http://www.w3.org (2005).

[105] W3C, Document Object Model (DOM), http://www.w3.org/DOM (2003).

[106] W3C, Namespaces in XML, http://www.w3.org/TR/REC-xml-names (1999).

[107] W3C, XML, http://www.w3.org/XML (2003).

[108] W3C, XML Link Language (XLink) http://www.w3.org/TR/xlink (2001).

[109] W3C, Mathematical Markup Language (MathML) http://www.w3.org/Math/ (2005).

[110] W3C, XML Path Language (XPath) http://www.w3.org/TR/xpath (1999).

[111] W3C, XML Pointer Language (XPointer) http://www.w3.org/TR/xptr (2002).

[112] W3C, XML Query Language (XQuery) http://www.w3.org/TR/xquery (2005).

[113] W3C, XML Schema, http://www.w3.org/XML/Schema.html (2004).

[114] W3C, XSL, http://www.w3.org/TR/xsl (2001).

[115] W3C, XSLT, http://www.w3.org/TR/xslt (1999).

[116] W3C, Web Services, http://www.w3.org/2002/ws/ (2005).

[117] Woflfram Research In.c. Multivariate ARMA Models, http://documents.wolfram.com/applications/timeseries/UsersGuidetoTimeSeries/1.2.5.html (2002).

[118] H. M.Wagner and T. M. Whitin, Dynamic version of the economic lot size model, Management Science (1958) 5:89–96.

[119] P.Walmsley, Definitive XML Schema. Prentice-Hall (2001).

[120] Ximian, Inc. Ximian and the Mono Project, http://developer.ximian.com/projects/mono/ (2002).

[121] Ziena Optimization Inc., Kintro Solver, http://www.ziena.com (2005)

APPENDIX A OPTIMIZATION SERVICES REPRESENTATION EXTENSIONS

The optimization services representation extensions in this appendix are mostly at a very early development stage and are changing constantly. They are described for initial reviews and complete references. But the primary design philosophies and main features of these extension should remain approximately the same over the time.

A.1 `<cones>` for cone programming

The cone programming extension in OSiL (Chapter 6) mainly addresses second-order cone programming (SOCP). SOCP is usually solved with some kind of primal-dual interior point method. The objective function is usually linear, while the constraints are an intersection of an affine set and the direct product of quadratic cones; see [75] for more details. In general, an SOCP can be expressed as

$$\begin{aligned} &\underset{\mathrm{x}}{\text{minimize}} \quad cx \\ &\text{subject to} \;\; \mathrm{Ax = b} \\ &\qquad\qquad\quad \mathrm{x} \in \boldsymbol{K} \end{aligned} \tag{A-1}$$

where $\boldsymbol{K}$ is a closed and convex cone. The second order cone $\boldsymbol{K}$ is more formally defined as a direct product $\boldsymbol{K} = \boldsymbol{K}^1 \times \boldsymbol{K}^2 \times \ldots \times \boldsymbol{K}^k$, where $\boldsymbol{K}^i$ can be any type of quadratic cones. There are different types of cones used in the literatures. Currently we define three widely used cones:

1). $\boldsymbol{K}^i$ (nonnegativeCone) =
$\boldsymbol{R}^+ = \{x^i \in \boldsymbol{R} | x \geq 0)$. If every $\boldsymbol{K}^i$ is a nonnegativeCone, then (A-1) is basically a regular linear program that can be simply expressed as (A-2). Including nonnegativeCone is for the purpose of completeness.

$$\begin{aligned} &\underset{x}{\text{minimize}} \quad cx \\ &\text{subject to} \quad Ax = b \\ &\qquad\qquad\quad \mathrm{x} \geq 0 \end{aligned} \tag{A-2}$$

2). $\boldsymbol{K}^i$ (quadraticCone) =
$\boldsymbol{K}^{quad} = \{x^i \in \boldsymbol{R}^{n_i} | \| x^i_{2:n_i} \|^2 \leq (x^i_1)^2, x^i_1 \geq 0)$

3). $\boldsymbol{K}^i$ (rotatedQuadraticCone) =
$\boldsymbol{K}^{rotatedQuad} = \{x^i \in \boldsymbol{R}^{n_i} | \| x^i_{3:n_i} \|^2 \leq 2x^i_1 x^i_2)^2, x^i_1, x^i_2 \geq 0)$

As OS representations are a set of evolving standards, more cone types may be added in the future. With the standard cone definitions, the `<cones>` element can simply be expressed as shown in Figure A-1.

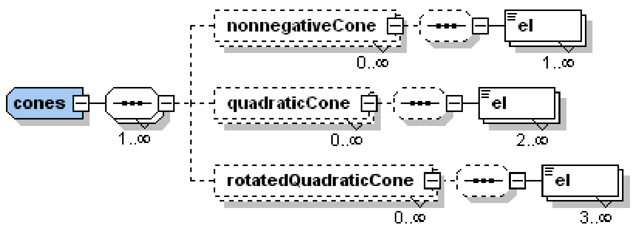


Figure A-1: `<cones>` element in OSiL.

`<Cones>` can have a sequence of different child cones defined above. Each type of cone is similarly defined. For example `<quadraticCone>` can appear 0 or more times as shown below:

```
<xs:element name="quadraticCone" minOccurs="0" maxOccurs="unbounded">
    <xs:complexType>
        <xs:sequence minOccurs="0">
            <xs:element name="el" minOccurs="2" maxOccurs="unbounded">
                <xs:complexType>
                    <xs:simpleContent>
                        <xs:extension base="xs:nonNegativeInteger">
                            <xs:attribute name="mult" type="xs:positiveInteger" use="optional" default="1"/>
                            <xs:attribute name="incr" type="xs:int" use="optional"/>
                        </xs:extension>
                    </xs:simpleContent>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
        <xs:attribute name="startIndex" type="xs:nonNegativeInteger" use="optional"/>
        <xs:attribute name="endIndex" type="xs:nonNegativeInteger" use="optional"/>
    </xs:complexType>
</xs:element>
```

Each `<quadraticCone>` can have 2 or more `<el>` child elements representing the variables that belong to the quadratic cone. Each `<el>` element is a nonnegative integer to indicate a variable index. The `multi` and `incr` attributes of `el` are similar to those defined in the `intVector` element described in the OSgL section (§6.1). If all the variable indexes are continuous, the sequence of `<el>` elements becomes optional; instead we can use the `startIndex` and `endIndex` attributes of `<quadraticCone>`.

Suppose there are 11 variables in the optimization problem and their domains are $x_0 \geq 0$ (i.e. $x_0 \in$ nonnegativeCone), $x_{1,3} \in$ quadraticCone, $x_{2,4,5,6} \in$ rotatedQuadraticCone, $x_{7,8,9} \in$ quadraticCone, $x_{10} \geq 0$. The cone programming representation in OSiL can look like:

```
<cones>
    <nonnegativeCone>
        <el>0</el>
        <el>10</el>
    </nonnegativeCone>
    <quadraticCone>
        <el>1</el>
        <el>3</el>
    </quadraticCone>
    < quadraticCone>
        <el>2</el>
        <el mult="3" incr="1">4</el>
    </quadraticCone>
    < rotatedQuadraticCone startIndex="7" endIndex="9"/>
</cones>
```

A.2 `<stages>` for math programs using stage information

Information of stages is used in several optimization types, such as dynamic programming, and stochastic programming. The `<stages>` element is shown in Figure A-2.

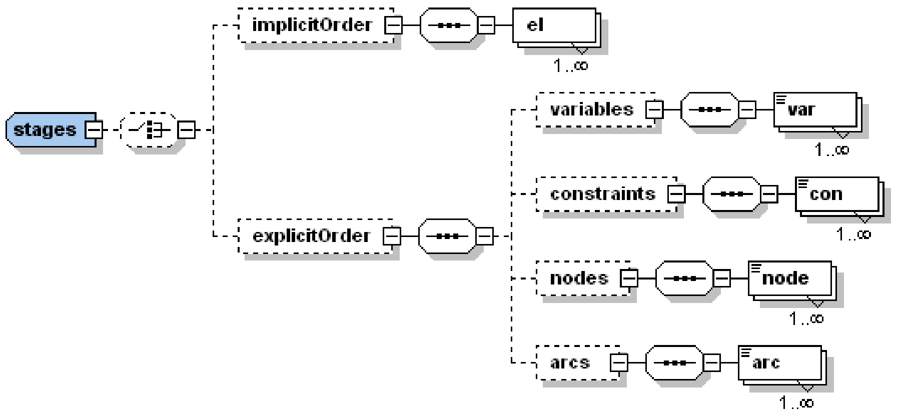


Figure A-2: `<stages>` element in OSiL.

`<stages>` has an optional `name` attribute and a required `number` attribute. As with many other array-type elements in OSiL, stages are referred by the indexes, *not* by the names. The start stage is always 0 and the end stage is always `number-1`. Stages can be implicitly listed using `<implictOrder>` if the rows and columns in the base program data part are listed in time order, or otherwise explicitly stated using `<explicitOrder>`. `<implicitOrder>` contains a sequence of `<el>` elements, each one a nonnegative integer. Each `<el>` has two required attributes: `startRowIdx` and `startColIdx`. Each `<el>` also has two optional attributes: `endRowIdx` and `endColIdx`. For example if we want to indicate that all the elements from row 0 to 4 and from column 0 to column 3 belong to stage 0, we can write the information down as `<el startRowIdx="0" startColIdx="0" endRowIdx ="4" endColIdx="3">0<el>`. If `endRowIdx` and `endColIdx` are missing, stage 0 ends just before `startRowIdx` and `startColIdx` of the next `<el>` element (stage 1).

Alternatively the stage can be explicitly specified on each variable (`<var>`) and constraint (`<con>`) in `<explicitOrder>`. Both `<var>` and `<con>` are nonnegative integers indicating stages. `<var>` has a required `idx` attribute for variable index references and `<con>` also has a required `idx` attribute for constraint index references.

A.3 `<stochastic>` for stochastic programming

For a complete review of stochastic programming, refer to [11]. The OSiL stochastic programming extension is designed to make it convenient and powerful to transform existing deterministic linear or nonlinear programs into stochastic programs by adding dynamic and stochastic structure information. It was first designed totally independent of the SMPS format[10] and later, through working with Horand Gassmann, one of the coauthors of the original SMPS format, added many new ideas. The OSiL stochastic extension is highly comprehensive and is evolving at a faster pace than most other OSxL schemas. Describing the entire stochastic extension is out of the scope of this thesis. We hereby illustrate the main features in the current `<stochastic>` element (Figure A-3).

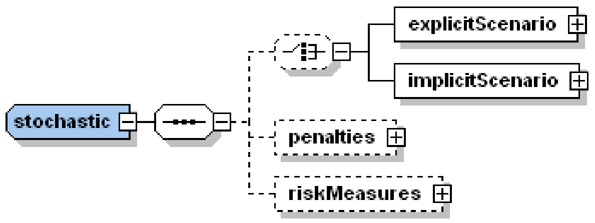


Figure A-3: `<stochastic>` element in OSiL.

The `<stochastic>` element is the next child after `<stages>` in `<programData>`. The most commonly used child in practice is the scenario child which can be either an `<explicitScenario>` (Figure A-4) or <implicitScenario> (Figure A-6). With scenarios, we can model a variety of dependencies, both within and across stages. Explicit scenarios are mostly for modeling stochastic processes with discrete distributions or discrete approximation. Implicit scenarios can be used to model continuous distributions.

Scenario based stochastic programs can be mixed with penalty-related (`<penalties>`, Figure A-7) and/or risk-measure-related (`<riskMeasures>`, Figure A-8) stochastic problems. The best-known penalty-related stochastic problem is simple recourse. Risk-measure-related problems are mostly about chance constraints and probabilistic objectives.

Figure A-4 shows the two alternatives to represent an explicit scenario: <scenarioPaths> and <scenarioTree>.

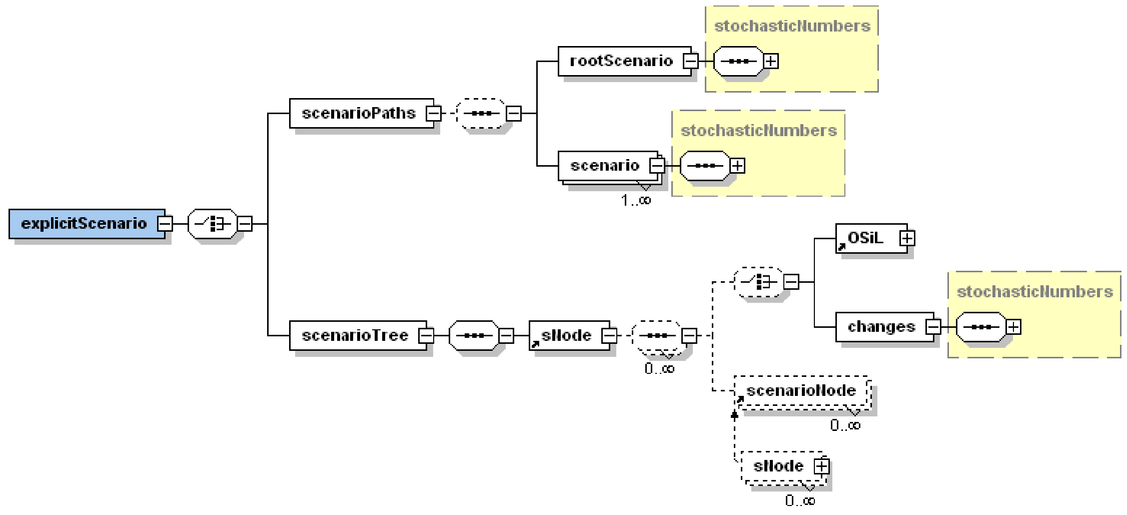


Figure A-4: <explicitScenario> element in OSiL.

The scenario path (<scenarioPaths>) approach views every scenario as a path from the root of the scenario tree to one of its leaves. There has to be exactly one root scenario (<rootScenario>, the first child of <scenarioPaths>). Every other scenario (the subsequent <scenario> children of <scenarioPaths>) is a path that branches either directly from the root scenario or indirectly from a branch of the root scenario. So each scenario has a *parent* scenario. The root scenario's parent is usually the OSiL core program. Each scenario inherits all of the values from its parent scenario and makes changes on the *stochastic* numbers that are different from the parent scenario in the stochasticNumbers child section. Basically any number in the entire OSiL core program can be stochastic. Figure A-5 shows the different types of stochastic numbers.

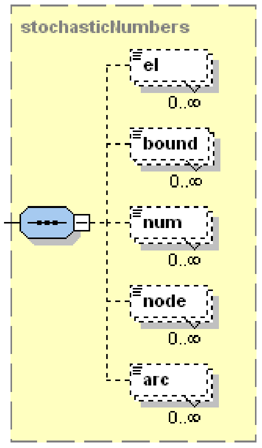


Figure A-5: `stochasticNumbers` type in OSiL.

`<el>` has a `rowIdx` and a `colIdx` attribute for references to linear coefficients. If `rowIdx` is negative, it is an objective function. If `colIdx` is -2, it is a lower bound (or left-hand side) of a constraint. If `colIdx` is -1, it is an upper bound (or right-hand side) of a constraint. `<var>` is used to vary different aspects of a variable in the math program, such as lower bound, upper bound, and type. `<num>` is used to reference a nonlinear number that is identified with an `id`. `<node>` and `<arc>` are used to refer to nodes and arcs in a network or graph. These are mainly used for future extensions if network and graph extension is added.

The scenario tree (`<scenarioTree>`) approach allows a node by node construction of the event tree. It has one and only only child (`<sNode>` of type `scenarioNode`) as the root of the event tree. An `<sNode>` element contains its own data information. There are two alternatives to specify the information: 1) by changing the information from one node to another through the `<changes>` element; 2) by specifying an entire sub-optimization problem through the `<OSiL>` element. Each <sNode> can in turn have 0 (if a leaf node) or more (if an internal node) <sNode> children. The idea is similar to the construction of nonlinear expression trees in the `<nl>` elements (§6.3). The recursive definition allows an entire scenario tree to be constructed cleanly and flexibly.

Figure A-6 shows the `<implicitScenario>` element.

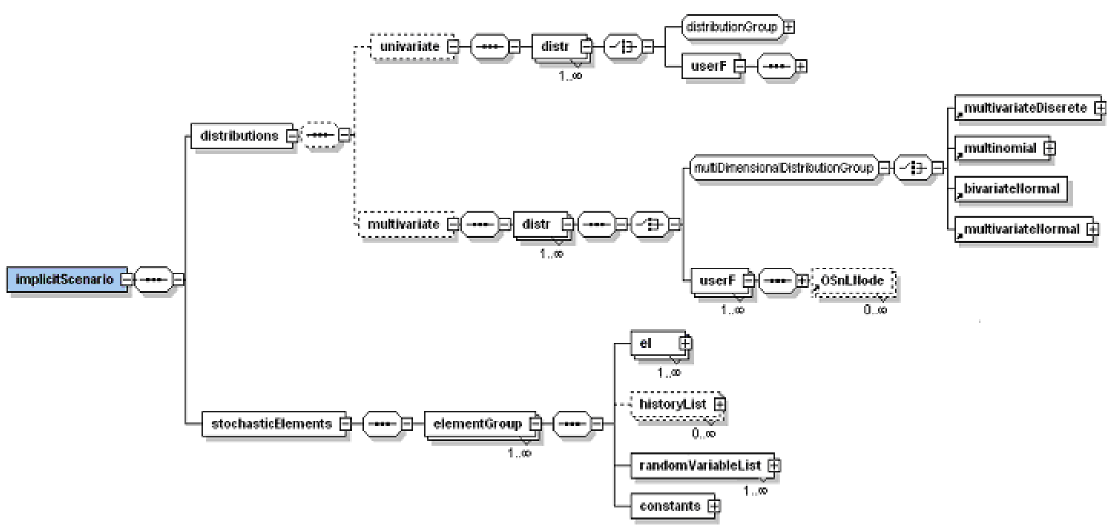


Figure A-6: `<implicitScenario>` element in OSiL.

In the `<distributions>` child, we can specify various univariate and multivariate distributions. Many standard distribution functions are built in the OSgL schema (§6.1). User-defined distribution functions are allowed through the `OSnLNode` in the OSnL schema (§6.3), just like defining any nonlinear expression in an OSiL instance.

`<stochasticElements>` can have a sequence of `<elementGroup>` children. In each `<elementGroup>`, we can specify history-dependent parameters in a stochastic process of the form

$$Y_t = \sum_{i=1}^{p} M_i Y_{t-i} + \sum_{j=1}^{q} N_j v_{t-j} + c_t \qquad \text{(A-3)}$$

where $M_i, i = 1,..., p$ and $N_j, j = 1,..., q$ are given matrices, $v_t, v_{t-1},...v_{t-q}$ are serially uncorrelated and identically distributed random vectors and c_t is a constant vector. This process is known as the ARMA(p,q) process. For more information on the ARMA process, refer to [117]. The incorporation of ARMA(p,q) into the OSiL stochastic extension is suggested by H.I Gassmann. One special case of (A-3) is $p = 0$, so (A-3) turns into $Y_t = Nv_t$, a simple linear transformation. So `<stochasticElements>` is a more generalized transformation of stochastic numbers. Another special case of (A-3) is $Y_t = Y_{t-1} + v_t$, where v_t is +1 or -1 with probability 0.5. So the model turns into a random walk.

In the `<elementGroup>` element, a sequence of `<el>` elements are used to identify elements of the Y_t vector. The subscript t of Y is specified by the `stage` attribute of

`<elementGroup>`. $M_i Y_{t-i}$ is specified in `<historyList>` (which contains a matrix and a vector) for each i. $N_j v_{t-j}$ is specified in `<randomVariableList>` (which contains a matrix and a vector) for each j. c_t is specified in `<constants>`.

Figure A-7 shows the `<penalties>` element.

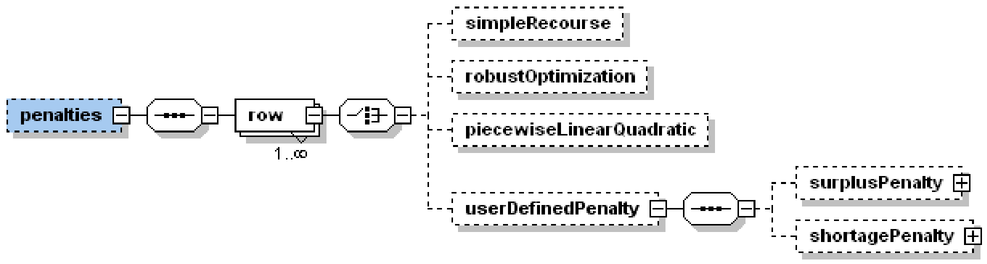


Figure A-7: `<penalties>` element in OSiL.

The `<penalties>` element can contain one or more `<row>` elements, each one having a rowIdx (≥ 0, constraints only) attribute. Penalties are imposed on violation of a constraint (either shortage or surplus). The best-known penalty-related stochastic problem is simple recourse. The `<simpleRecourse>` element has a (linear) `shortagePenalty` and a (linear) `surplusPenalty` attribute. There are other kinds of standard penalties. For example the `<robustOptimization>` element has quadratic penalties and the `<piecewiseLinearQuadratic>` element has both linear and quadratic penalties. The `<userDefinedPenalty>` element can be used to define customized penalty functions for both surplus and shortage, through the `OSnLNode` in the OSnL schema, just like defining any nonlinear expression in an OSiL instance.

Figure A-8 shows the `<riskMeasures>` element.

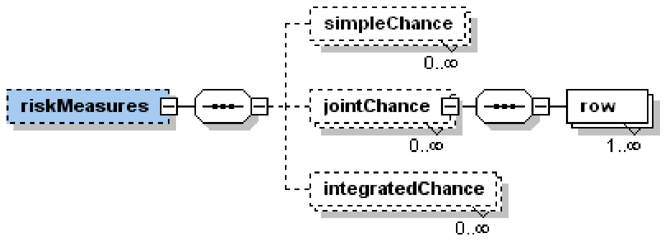


Figure A-8: `<riskMeasures>` element in OSiL.

Risk-measure-related problems are mostly about chance constraints and probabilistic objectives. There are three children of `<riskMeasures>`: `<simpleChance>`,

<jointChance>, and <integratedChance>; each one represents a different type of risk-measure-related problem and each one is associated with one (simple chance) or more (joint chance) rowIdx attributes. If rowIdx ≥ 0, it is a chance constraint, specifying the probability that a constraint or some joint constraints are satisfied. If rowIdx < 0, it is a probabilistic objectives, changing the minimization or maximization of the objective to minimization of maximization of the *probability* of the objective function value with respect to $(=,\leq,\geq)$) the objective constant. Integrated chance constraints (ICC) is introduced by Klein Haneveld. See [67].

A.4 <networkAndGraph> for network and graph problems

In the first version of OSiL, we excluded the <networkAndGraph> extension for network and graph definition. For review, a test version schema OSiL_NaG.xsd includes the network and graph extension and can be found at http://www.optimizationservices.org/schemas/OSiL_NaG.xsd. We here briefly describe the features of network and graph extension.

Like defining many data structures and elements, The OSgL schema (§6.1) also defines a <networkAndGraph> element (Figure A-9), and then gets included in the OSiL schema. The <networkAndGraph> element is used to comprehensively describe a network and graph *topology* through a set of nodes and arcs elements and definitions of nodeProperties and arcProperties.

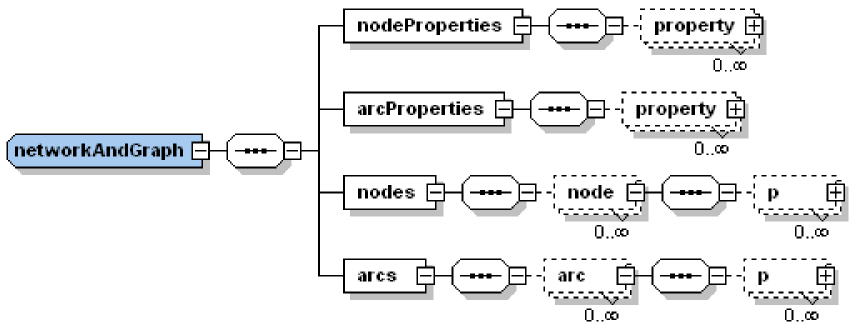


Figure A-9: <networkAndGraph> data type in OSgL.

Also like pre-defining standard functions, The OSgL schema (§6.1) also defines many standard *problems*; OSgL is then included in the OSiL schema. Most of these standard problems are heuristics based on a network or graph. Here is a predefined standard shortest path problem:

```
<xs:element name="shortestPath">
```

```
    <xs:complexType>
        <xs:attribute name="costPropName" type="xs:IDREF" use="required"/>
        <xs:attribute name="start" type="xs:nonNegativeInteger" use="required"/>
        <xs:attribute name="end" type="xs:nonNegativeInteger" use="required"/>
    </xs:complexType>
</ xs:element>
```

The shortest path problem defines a `start` and an `end` attribute which refer to nodes (identified by a nonnegative integer) in a network and graph topology (Figure A-9). Since a network can have many properties defined on arcs or nodes, the `costPropName` attribute in the above `shortestParth` element specifies which arc property the shortest path algorithm should be carried out on. This mechanism fully defines everything about a standard shortest path problem. Other problems such as maximum flow problem, minimum spanning tree, minimum cost flow, traveling sales person, vehicle routing problem, are similarly defined. All these standard heuristics are grouped in `<networkAndGraphHeuristicsGroup>` (Figure A-10). Along with the network and graph topology definition in Figure A-9, the group can potentially be used in a future extension of OSiL to network and graph problems.

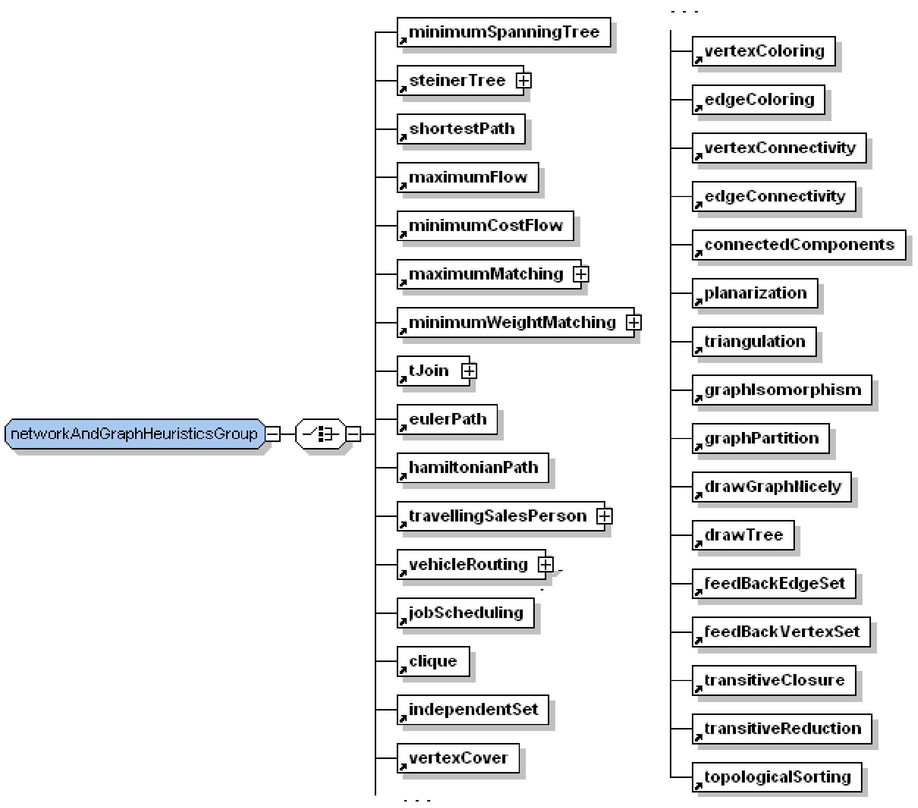


Figure A-10: `<networkAndGraphHeuristicsGroup>` group in OSgL.

A.5 Special nonlinear nodes in OSnL

A.5.1 `<complements>` for complementarity problems

The `<complements>` schema from OSnL is shown below:

```
<xs:complexType name="OSnLNodeComplements">
    <xs:complexContent>
        <xs:extension base="OSnLNode">
            <xs:sequence minOccurs="2" maxOccurs="2">
                <xs:element ref="OSnLNode"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name="complements" type="OSnLNodeComplements" substitutionGroup="OSnLNode"/>
```

The `<complements>` element allows complementarity problems to be constructed for solvers to search for a feasible solution. Linear or smooth nonlinear optimization problems can be viewed as special cases of complementarity problems. Complementarity more or less means that at least one of a pair of logic expressions (e.g. two constraints) must hold with equality. For more details refer to [49].

The `<complements>` element is one of the few special elements that don't have attributes. It also has a definite number of 2 children, which are often constraints. The sequence of the 2 children does not matter. Both children can consist from 1 to 3 expressions separated by logic operators $=$, $\geq$, and $\leq$. Of the two children of `<complements>`, there must be either exactly two inequality operators or one equality operator. The `<complements>` element evaluates to true if both children are true and at least one inequality is tight. For example if $constraint_1$ is of the form $expression_1 \geq 0$ and $constraint_2$ is of the form $expression_2 \leq 5$, we can express the complementarity as

```
<complements>
    <geq>
        <constraint idx="1"/>
        <number value="0"/>
    </geq>
    <leq>
        <constraint idx="2"/>
        <number value="5"/>
    </leq>
</complements>
```

or more concisely as

```
<complements>
    <constraint idx="1" valueType="status"/>
    <constraint idx="2" valueType="status"/>
</complements>
```

because the bound information is already specified in the `<constraints>` element of OSiL. Of course it has to be made sure that in using the concise form, the constraint is only constrained on one side; otherwise there can be ambiguities. Notice if the `valueType` attribute of `constraint` is not specified, it defaults to the constraint value.

When one constraint $constraint_i$, $i \in \{1,2\}$, involves two inequalities and is of the form $lb \le expression_i \le ub$ or $ub \ge expression_i \ge lb$ (lb and ub are numbers), then the other $constraint_j$, $j \in \{1,2\}, j \ne i$, must be just of the free form $expression_j$. In this case, the `<complements>` element evaluates to true if $constraint_i$ is true and

$$expression_j = 0 \text{ if } lb < expression_i < ub$$

$$expression_j \le 0 \text{ if } expression_i = ub$$

$$expression_j \ge 0 \text{ if } expression_i = lb$$

For example if $constraint_1$ is of the form $2 \le expression_1 \le 7$ and $constraint_2$ is of the form $expression_2$, we can express the complementarity as

```
<complements>
    <and>
        <leq> <number value="2"/><constraint idx="1" valueType="value"/> </leq>
        <leq> <constraint idx="1" valueType="value"/><number value="7"/> </leq>
    </and>
    <constraint idx="1" valueType="value"/>
</complements>
```

or more concisely as

```
<complements>
    <constraint idx="1" valueType="status"/>
    <constraint idx="2" valueType="value"/>
</complements>
```

Of course it has to be made sure that in using the concise representation, $constraint_1$ is bounded on both sides, that is, both `lb` ($\ne -\infty$) are `ub` ($\ne \infty$) attributes have to be specified on $constraint_1$ in the `<constraints>` element of OSiL.

Child elements of `<complements>` do not always have to be constraints. For example the following is also valid:

```
<complements>
    <and>
        <leq> <number value="0"/><var idx="0"/> </leq>
        <leq> <var idx="0"/><number value="9"/> </leq>
    </and>
    <plus> <var idx="1"/><var idx="2"/> </plus>
</complements>
```

for $0 \le x_0 \le 9$ *complements* $x_1 + x_2$.

A.5.2 `<nodeRef>` and `<arcRef>` for network and graph problems

As the first release of OSiL does not include network and graph extension, the `<nodeRef>` and `<arcRef>` elements from OSnL, which are used to reference node and arc property values in a network, are reserved for future use. Briefly, the use of `<nodeRef>` and `<arcRef>` are similar to that of `<simInput>` and `<simOutput>` discussed in §6.3. Like `<simInput>` and `<simOutput>` which are used to reference values in a simulation definition in OSiL (`<simulation>`), `<nodeRef>` and `<arcRef>` can be used to reference values in a network and graph definition that can potentially be in OSiL (`<networkAndGraph>`). `<simInput>` and `<simOutput>` use their attributes (`simName`, `inputName`, `outputName`) and an optional child to take or supply values to and from a simulation; similarly `<nodeRef>` and `<arcRef>` also use attributes (`arcID`, `nodeID`, `propName`) and an optional child node to take or supply values to and from a network and graph.

APPENDIX B OPTIMIZATION SERVICES LIBRARY

The OS library (including the OS server software for hosting individual services) and related documents are located at http://www.optimizationservices.org. The OS library is an open-source Java library intended to simplify the implementation of various OS compatible services, enforce the requirements of standards, assist in the exchange of instances between components, and facilitate the adoption of Optimization Services.

The OS library has two types of distributions. The first type is one entire library file `os.jar`, which contains all the library classes. A "jar" file is a java equivalent to a Windows `.dll`, or UNIX `.so` or `.a` library file. A jar file should be appropriately set in the `CLASSPATH` environment before it can be properly included and used; check any major java tutorial for details.

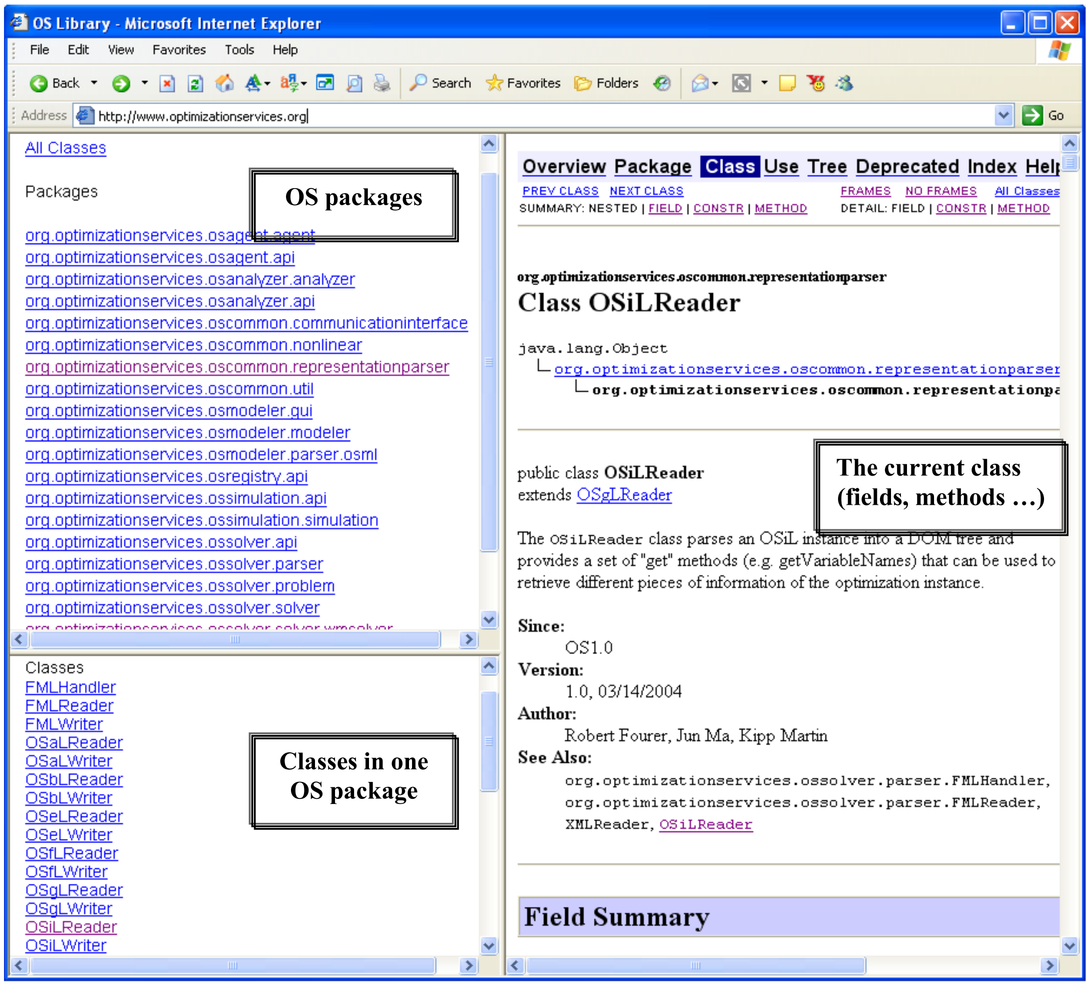
The second type of library distribution breaks the entire `os.jar` into seven smaller jar files: `osagent.jar`, `ossolver.jar`, `osmodeler.jar`, `osanalyzer.jar`, `ossimulation.jar`, and `osregistry.jar`, so that developers only need to download and include related and more light-weighted jar files. `oscommon.jar` and `osagent.jar` are almost always required.

- `oscommon.jar` contains parsers for reading and writing all the instances specified by the standard OS representation schemas (Chapter 6), interfaces specified by the OS communication WSDL documents (Chapter 7), and related computational and utility classes.
- `osagent.jar` contains communication agents that can be delegated to send and receive OS instances according to the protocols specified by the OS communication WSDL documents (Chapter 7).
- `ossolver.jar` contains sample solver services, public solver service APIs, local interfaces, sample problems, and customized parsers that use the standard parsers (`oscommon.jar)` to convert instances to and from the solver-specific formats or data structures.
- `osmodeler.jar` contains sample modeling language environments (MLE), especially the OSmL MLE, which includes the GUI, OSmL engine and associated tokenizers, parsers and compilers.
- `osanalyzer.jar` contains sample analyzer services, local interfaces, and customized interfaces.

- `ossimulation.jar` contains sample simulation services, local interfaces, and customized interfaces.
- `osregistry.jar` mainly contains the implementation of the OS registry.

Besides the library jar files, the OS server software for hosting individual services can also be downloaded from the OS Web site, along with tutorials and other documents. In Chapter 7, we showed some examples of using the library and explained the process of how the OS server software works.

Java classes (and interfaces) are grouped into *packages*, equivalent to the C++ namespaces, to avoid class name conflicts. A jar file can contain several packages and each package usually contains many java classes. Each OS java class file is documented, or commented in the "Javadoc" format in detail. Javadoc tools are then used to generate the java APIs and documentation comments to a set of HTML pages describing the classes, inner classes, interfaces, constructors, methods, and fields. These HTML documentation pages are also published at the OS Web site (Figure B-1). We describe the OS library and all its packages and classes at a relatively high level in the following sections. For details on using each class, refer to the OS Javadoc document on the OS Web site.


OS Library - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Folders
Address http://www.optimizationservices.org
Go
OS packages
The current class (fields, methods …)
Classes in one OS package
Overview Package Class Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

All Classes

Packages

org.optimizationservices.osagent.agent
org.optimizationservices.osagent.api
org.optimizationservices.osanalyzer.analyzer
org.optimizationservices.osanalyzer.api
org.optimizationservices.oscommon.communicationinterface
org.optimizationservices.oscommon.nonlinear
org.optimizationservices.oscommon.representationparser
org.optimizationservices.oscommon.util
org.optimizationservices.osmodeler.gui
org.optimizationservices.osmodeler.modeler
org.optimizationservices.osmodeler.parser.osml
org.optimizationservices.osregistry.api
org.optimizationservices.ossimulation.api
org.optimizationservices.ossimulation.simulation
org.optimizationservices.ossolver.api
org.optimizationservices.ossolver.parser
org.optimizationservices.ossolver.problem
org.optimizationservices.ossolver.solver

Classes
FMLHandler
FMLReader
FMLWriter
OSaLReader
OSaLWriter
OSbLReader
OSbLWriter
OSeLReader
OSeLWriter
OSfLReader
OSfLWriter
OSgLReader
OSgLWriter
OSiLReader
OSiLWriter

org.optimizationservices.oscommon.representationparser

Class OSiLReader

```
java.lang.Object
  └ org.optimizationservices.oscommon.representationparser
      └ org.optimizationservices.oscommon.representationpa
```

public class **OSiLReader**
extends OSgLReader

The `OSiLReader` class parses an OSiL instance into a DOM tree and provides a set of "get" methods (e.g. getVariableNames) that can be used to retrieve different pieces of information of the optimization instance.

Since:
OS1.0

Version:
1.0, 03/14/2004

Author:
Robert Fourer, Jun Ma, Kipp Martin

See Also:
`org.optimizationservices.ossolver.parser.FMLHandler`, `org.optimizationservices.ossolver.parser.FMLReader`, `XMLReader`, `OSiLReader`

Field Summary

Figure B-1: OS Java library document (Javadoc) at http://www.optimizationservices.org.

B.1 Library Design

When designing the OS library, we break up the library into seven projects: OSCommon, OSAgent, OSSolver, OSModeler, OSAnalyzer, OSSimulation, and OSRegistry. At design time, each corresponds to a folder with similar subfolder structures as shown in Figure B-2.

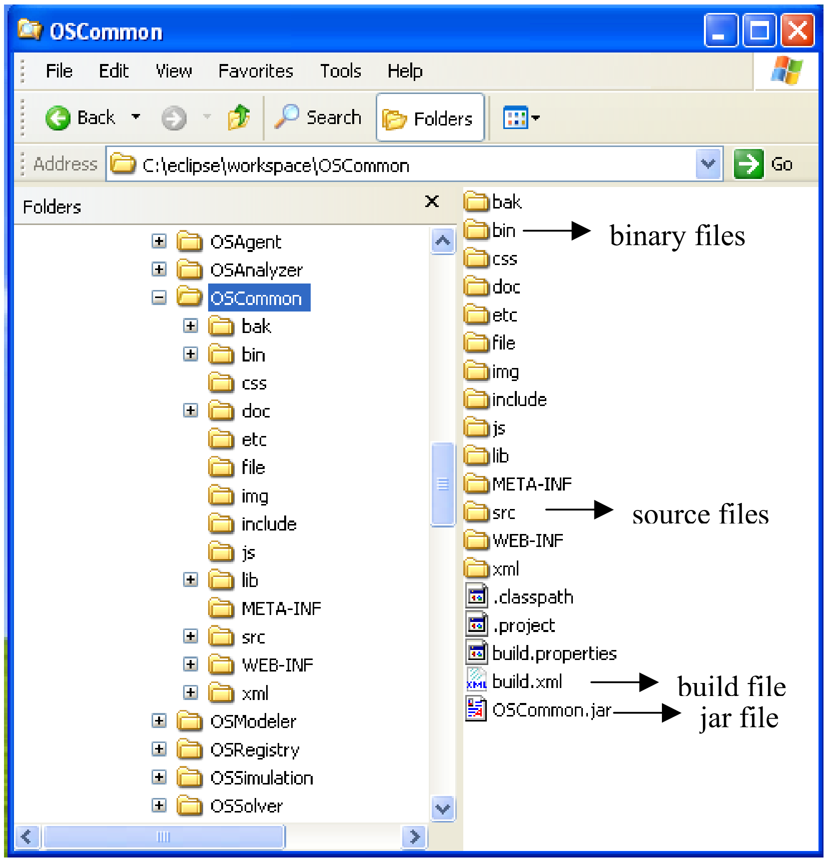
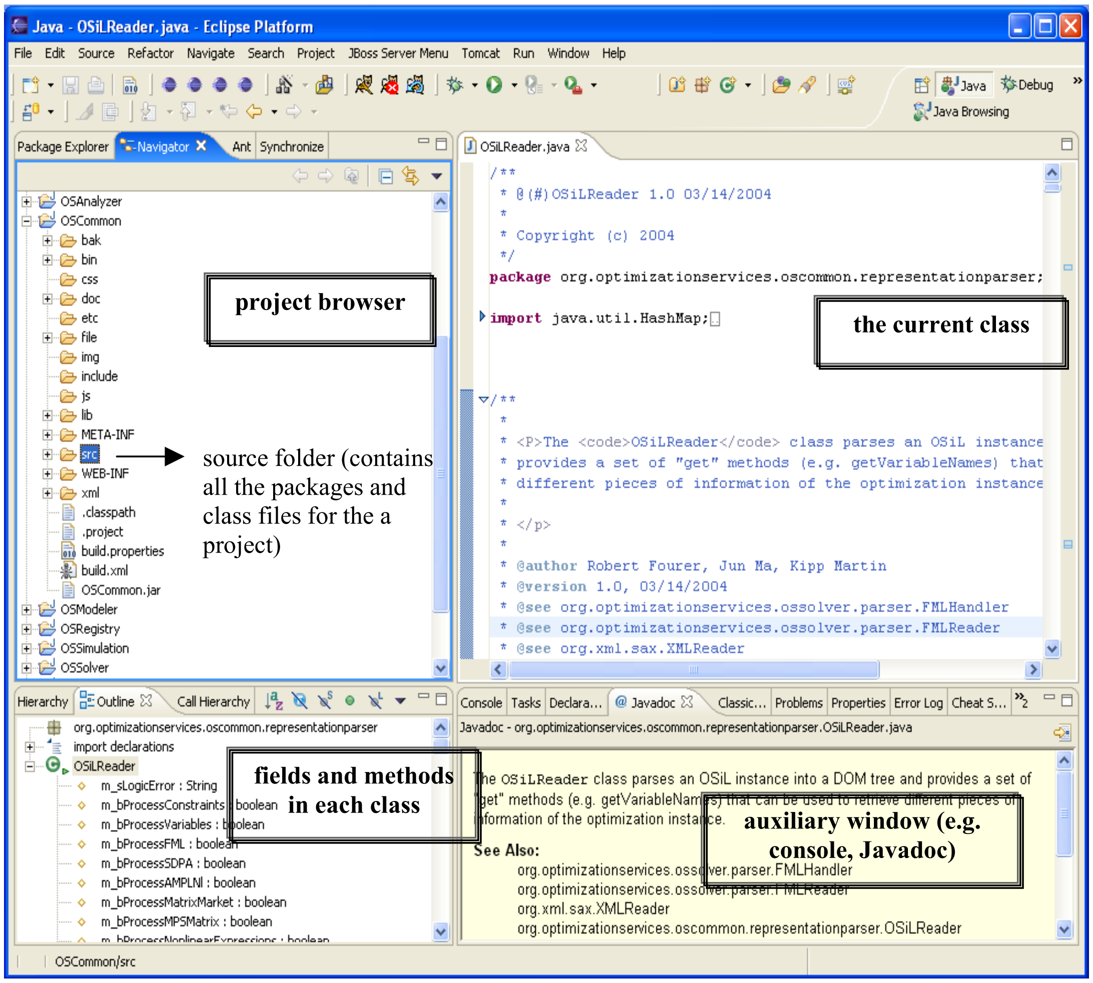


Figure B-2: The subfolder structure of the OSCommon project folder; other folders have similar subfolder structures.

We use the Eclipse IDE [32] as our Java development environment (Figure B-3). So each folder also corresponds to an Eclipse project. At distribution time each project is zipped into a respective jar file with the same name as the project's folder name. The `os.jar` file described in the introduction is a compilation of all the seven respective jar files. Java sources are included in the jar file distribution.

Each project (design time) or jar file (distribution time) contains several packages and the classes are grouped into a corresponding package. All the Optimization Services package names begin with the prefix: `org.optimizationservices.[projectname]` where `[projectname]` is the name of one of the seven projects. For example the package name for all the classes in the OSCommon project or jar begins with `org.optimizationservices.oscommon`, and then under this package sub-packages can be further created, for example, `org.optimizationservices.oscommon.util` or `org.optimizationservices.representationparser`.


Java - OSiLReader.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project JBoss Server Menu Tomcat Run Window Help
Java
Debug
Java Browsing
Package Explorer
Navigator
Ant
Synchronize
OSiLReader.java
project browser
the current class
source folder (contains all the packages and class files for the a project)
fields and methods in each class
auxiliary window (e.g. console, Javadoc)
Hierarchy
Outline
Call Hierarchy
Console
Tasks
Declara...
Javadoc
Classic...
Problems
Properties
Error Log
Cheat S...
OSCommon/src

```
OSAnalyzer
OSCommon
  bak
  bin
  css
  doc
  etc
  file
  img
  include
  js
  lib
  META-INF
  src
  WEB-INF
  xml
  .classpath
  .project
  build.properties
  build.xml
  OSCommon.jar
OSModeler
OSRegistry
OSSimulation
OSSolver
```

```java
/**
 * @(#)OSiLReader 1.0 03/14/2004
 *
 * Copyright (c) 2004
 */
package org.optimizationservices.oscommon.representationparser;

import java.util.HashMap;

/**
 *
 * <P>The <code>OSiLReader</code> class parses an OSiL instance
 * provides a set of "get" methods (e.g. getVariableNames) that
 * different pieces of information of the optimization instance
 *
 * </p>
 *
 * @author Robert Fourer, Jun Ma, Kipp Martin
 * @version 1.0, 03/14/2004
 * @see org.optimizationservices.ossolver.parser.FMLHandler
 * @see org.optimizationservices.ossolver.parser.FMLReader
 * @see org.xml.sax.XMLReader
```

```
org.optimizationservices.oscommon.representationparser
import declarations
OSiLReader
    m_sLogicError : String
    m_bProcessConstraints : boolean
    m_bProcessVariables : boolean
    m_bProcessFML : boolean
    m_bProcessSDPA : boolean
    m_bProcessAMPLNI : boolean
    m_bProcessMatrixMarket : boolean
    m_bProcessMPSMatrix : boolean
    m_bProcessNonlinearExpressions : boolean
```

Javadoc - org.optimizationservices.oscommon.representationparser.OSiLReader.java

The OSiLReader class parses an OSiL instance into a DOM tree and provides a set of "get" methods (e.g. getVariableNames) that can be used to retrieve different pieces of information of the optimization instance.

See Also:
org.optimizationservices.ossolver.parser.FMLHandler
org.optimizationservices.ossolver.parser.FMLReader
org.xml.sax.XMLReader
org.optimizationservices.oscommon.representationparser.OSiLReader

Figure B-3: The Eclipse Java IDE (Integrated Development Environment).

B.2 OSCommon Library

The OSCommon library (project or jar) currently contains the 5 packages described in Table B-1.

Package name	Brief description
org.optimizationservices.oscommon.**communicationinterface**	Interface classes that list the operations required by the OSP communication WSDL documents, and are to be implemented by corresponding Optimization Services.
org.optimizationservices.oscommon.**representationparser**	Parser classes that read and write standard OSP representation instances.
org.optimizationservices.oscommon.**nonlinear**	An expression tree class and all the OSnL node classes that correspond to the nonlinear nodes (operators, functions, etc.) specified in the OSnL schema.
org.optimizationservices.oscommon.**algebra**	Algebra (mainly linear algebra) utility classes used in optimization solvers.

org.optimizationservices.oscommon.**util**	General utility classes that handle I/O, XML, XPath, XQuery, XSLT, Web services, and other common routines.

Table B-1: OSCommon packages.

Table B-2 through Table B-6 list some of the important classes in each of the above 5 OSCommon sub-packages. For detailed usage (fields, methods, etc.) of each class refer to the Javadoc and other documents on the OS Web site.

Sample classes in the communicationinterface sub-package	**Brief description**
OScL	Interface class that lists all the operations specified in the OScL WSDL document: such as `String call (String ossl)`, and to be implemented by OS simulations.
OSdL	Interface class that lists all the operations specified in the OSdL WSDL document: such as `String discover (String osql)`, and to be implemented by the OS registry.
OShL	Interface class that lists all the operations specified in the OShL WSDL document: such as `String call (String osil)`, and to be implemented by OS solvers and OS analyzers.
OSjL	Interface class that lists all the operations specified in the OSjL WSDL document: such as `String join (String osel)`, and to be implemented by the OS registry.
OSkL	Interface class that lists all the operations specified in the OSkL WSDL document: such as `String knock ( )`, and to be implemented by OS solvers, OS analyzer and OS simulations.
OSvL	Interface class that lists all the operations specified in the OSvL WSDL document: such as `String validate (String osxl)`, and to be implemented by the OS registry.

Table B-2: Sample classes in org.optimizationservices.oscommon.communicationinterface.

Sample classes in the representationparser sub-package	**Brief description**
OSiLReader	Read an OSiL instance and generate certain standard data structures and methods such as array/vector of objective coefficients, coefficient matrices, and calculation of nonlinear objective/constraint functions or derivatives.
OSiLWriter	Write out an OSiL instance from standard data structures such as array/vector of objective coefficients, coefficient matrices, and nonlinear function expressions.
OSaLReader, OSaLWriter, OSbLReader, OSbLWriter, ….	Similar to the OSiLReader and OSilWriter. In the representationparser packages, each OSxL representation has two corresponding classes: OSxLReader and OSxLWriter.

Table B-3: Sample classes in org.optimizationservices.oscommon.representationparser.

Sample classes in the nonlinear sub-package	Brief description
ExpressionTree	The OSExpressionTree class represents an expression tree for a nonlinear function (linear ones being special cases) and provide convenience methods to process the contained nonlinear function. In essence it contains the root node (of OSnLNode type) of an expression and hides all the internal nodes. It is the only public class that interfaces with any component (e.g. a solver) that needs to manipulate the nonlinear functions in an instance. For example, it is mainly used in the osilReader class to parse a nonlinear optimization instance.
OSnLNode	The OSnLNode class represents a node in an expression tree for a nonlinear function (linear ones being special cases) and provide convenience methods to process the node. It is an abstract (or generic) node from which we derive concrete operator nodes.
OSnLNodeSin	The OSnLNodeSin class represents a sin node in an expression tree. It extends the abstract OSnLNode class and implements its abstract methods such as calculateFunction(double[]).
OSnLNodePI	The OSnLNodePI class represents a PI constant node in an expression tree. It extends the abstract OSnLNode class and implements its abstract methods such as calculateFunction(double[]).
OSnLNodeVar	The OSnLNodeVar class represents a variable node in an expression tree. The variable can be treated as a unary operator with its index as a subscript operand of the "variable operator". If the variable index is a number, there is no operand node. The number is treated as the variables attribute. If the variable index is an integer-valued function or a look up from some data source, it is treated no different from a unary operator. It extends the abstract OSnLNode class and implements its abstract methods such as calculateFunction(double[]).]).
OSnLAbs, OSnLAnd, OSnLArccos, OSnLArccosh …	Similar to OSnLNodeSin, OSnLNodePI and OSnLNodeVar. They are all concrete node classes that extend the abstract OSnLNode class and implements the required abstract methods.

Table B-4: Sample classes in org.optimizationservices.oscommon.nonlinear.

Sample classes in the algebra sub-package	Brief description
DoubleVector	Vector class with double precision entries.
DoubleSparseMatrix	Sparse matrix class with double precision entries.
BigDecimalDenseMatrix	Dense matrix class with arbitrary precision entries.
BigIntegerDenseMatrix, DoubleDenseMatrix …	Various kinds of matrix and vector classes that provide basic matrix and vector operations such as multiplications, factorizations, etc.

Table B-5: Sample classes in org.optimizationservices.oscommon.algebra.

Sample classes in the util sub-package	Brief description
CommonUtil	Contains methods for performing common basic operations used by many classes in the

	Optimization Services (OS) framework.
IOUtil	Contains methods for performing common basic input-output (I/O) operations, such as file reading/writing, used by various components in the Optimization Services (OS) framework.
MathUtil	Contains methods for performing mathematics related operations used by many classes in the Optimization Services (OS) framework.
ProcessUtil	A process and runtime (or terminal environment) related utility class. For example it provides methods to run commands (e.g. DOS or UNIX commands from within the programming codes.
WSUtil	Contains methods for performing common web services related operations, such as soap construction/web services invocation, used by various components in the Optimization Services (OS) framework.
XMLUtil	Contains methods for performing common basic XML-related operations used by various classes in the Optimization Services (OS) framework.
XPathUtil	Contains methods for performing common basic XPath-related operations used by various classes in the Optimization Services (OS) framework.
XQueryUtil	Contains methods for performing common basic XQuery-related operations used by various classes in the Optimization Services (OS) framework.
XSLTUtil	Contains methods for performing common basic XSLT-related operations used by various classes in the Optimization Services (OS) framework.

Table B-6: Sample classes in org.optimizationservices.oscommon.util.

B.3 OSAgent Library

The OSAgent library (project or jar) currently contains the 2 packages described in Table B-7.

Package name	**Brief description**
org.optimizationservices.osagent.**agent**	Various agent classes for communication to different Optimization Services. For example an OSSolverAgent is used to hook up to an OS solver.
org.optimizationservices.osagent.**parser**	Parser classes that convert one standard instance to another.

Table B-7: OSAgent packages.

Table B-8 through Table B-9 list some of the important classes in each of the above 2 OSAgent sub-packages. For detailed usage (fields, methods, etc.) of each class refer to the Javadoc and other documents on the OS Web site.

Sample classes in the agent sub-package	Brief description
OSSolverAgent	The class implements the OShL interface as specified by Optimization Services hook-up Language (OShL). It contains methods to help solver agents communicate with OS solvers. It hides all the SOAP protocol related technical details from an optimization user.
OSAnalyzerAgent, OSSimulationAgent, OSRegistryAgent	All similar to the OSSolver Agent.
OSFlowAgent	The class is invokes Optimization Services according to the process flow specified in an Optimization Services flow Language (OSfL). It may involve invoking separate agents listed above.

Table B-8: Sample classes in org.optimizationservices.osagent.agent.

Sample classes in the parser sub-package	Brief description
OSaLToOSqL	The class converts standard OSaL instance to standard OSqL instance. It is used if the OSFlowAgent involves automatically invoking and discover operation after it gets an OSaL analysis from an OSAnalyzer.
Other standard instance conversion classes needed by the agents and various agent-customized parser classes needed in communications.	

Table B-9: Sample classes in org.optimizationservices.osagent.parser.

B.4 OSSolver Library

The OSSolver library (project or jar) currently contains the 5 packages described in Table B-10.

Package name	Brief description
org.optimizationservices.ossolver.**api**	Contains sample solver services. These are the classes that implement the OShL (hook-up) and OSkL (knock) and are accessed on the OS network.
org.optimizationservices.ossolver.**localInterface**	Local interfaces that contains standard in memory data structures that can be directly accessed by the solver engines.
org.optimizationservices.ossolver.**parser**	Customized parsers that use the standard parsers from oscommon to convert instances to and from the solver-specific formats or data structures.
org.optimizationservices.ossolver.**solver**	Sample solvers that solve various optimization problems.
org.optimizationservices.ossolver.**problem**	Sample optimization problems using the standard data structures in the org.optimizationservices.oscommon.**localInterfac e** package.

Table B-10: OSSolver packages.

Table B-11 through Table B-15 list some of the important classes in each of the above 4 OSSolver sub-packages. For detailed usage (fields, methods, etc.) of each class refer to the Javadoc and other documents on the OS Web site.

Sample classes in the api sub-package	Brief description
KnitroSolverService	KnitroSolverService is an api that is public to the external world. It hides all the parsing, local interfacing, and solving processes. It implements the OShL and OSkL interfaces from OSCommon. It is called by an OS agent. The service solves continuous nonlinear optimization problems.
LindoSolverService	LindoSolverService is an api that is public to the external world. It hides all the parsing, local interfacing, and solving processes. It implements the OShL and OSkL interfaces from OSCommon. It is called by an OS agent. The service solves very general optimization problems.

Table B-11: Sample classes in org.optimizationservices.ossolver.api.

Sample classes in the localInterface sub-package	Brief description
OSiI	This is an Optimization Services instance Interface. It contains the standard optimization problem data structures that are generated from the OSiL instance and can be directly accessed by a solver.
OSoI	This is an Optimization Services option Interface. It contains the standard optimization option data structures that are generated from the OSoL instance and can be directly accessed by a solver.
OSrI	This is an Optimization Services result Interface. It contains the standard optimization result data structures that are returned by a solver and then used to generate the OSrL result instance.

Table B-12: Sample classes in org.optimizationservices.ossolver.localInterface.

Sample classes in the parser sub-package	Brief description
LindoOSiLReader	The LindoOSiLReader class uses the generic OSilReader to parse an OSiL instance into the Lindo's Instruction List format that can be inputted into the Lindo solver.

Table B-13: Sample classes in org.optimizationservices.ossolver.parser.

Sample classes in the solver sub-package	Brief description
KnitroSolver	Knitro optimization solver that solves continuous nonlinear problems.
LindoSolver	Lindo optimization solver that solves general nonlinear problems.

Table B-14: Sample classes in org.optimizationservices.ossolver.solver.

Sample classes in the problem sub-package	Brief description
OptProblem_Rosenbrock	The Rosenbrock problem constructed using the standard data structures in the org.optimizationservices.oscommon.**localInterface** package

Table B-15: Sample classes in org.optimizationservices.ossolver.problem.

B.5 OSModeler Library

The OSModeler library (project or jar) currently contains the 4 packages described in Table B-1.

Package name	Brief description
org.optimizationservices.osmodeler.**api**	Contains modeling language environment services that can be accessed publicly over the OS network.
org.optimizationservices.osmodeler.**gui**	Contains modeling language Graphical User Interface, which is usually used locally on a desktop.
org.optimizationservices.osmodeler.**modeler**	Contains the modeling language engines that compiles modeling languages into standard instances.
org.optimizationservices.osmodeler.**parser**	Contains relevant tokenizers and parsers using by the modeling language engine compilation process.

Table B-16: OSModeler packages.

Table B-17 through Table B-20 list some of the important classes in each of the above 4 OSModeler sub-packages. For detailed usage (fields, methods, etc.) of each class refer to the Javadoc and other documents on the OS Web site.

Sample classes in the api sub-package	Brief description
OSmLService	It is a public OSmL Web service that can be accessed over the OS network. It takes OSmL models, generates standard instances and delegates communication agents to solve the optimization instances.

Table B-17: Sample classes in org.optimizationservices.osmodeler.api.

Sample classes in the gui sub-package	Brief description
OSmLGUI	OSmLGUI provides the OSmL modeling language GUI.

Table B-18: Sample classes in org.optimizationservices.osmodeler.gui.

Sample classes in the modeler sub-package	Brief description
OSmLEngine	The OSmLEngine compiles the XQuery based Optimization Services Modeling Language model and compiles the model into an Optimization Services instance Language (OSiL) low level representation. When the optimization result is returned in Optimization Services result Language (OSrL), the engine takes the role of an OSrL parser.

Table B-19: Sample classes in org.optimizationservices.osmodeler.modeler.

Sample classes in the parser sub-package	Brief description
OSmLPreparser	It pre-parses an OSmL model into a pure XQuery Language.
OSmLQueryResultToOSiL	It parses the XQuery result intermediate XML instance generated by an XQuery engine and converts the intermediate XML instance into the standard OSiL instance.
InfixParser	It parses infix based expressions.

Table B-20: Sample classes in org.optimizationservices.osmodeler.parser.

B.6 OSAnalyzer Library

The OSAnalyzer library (project or jar) currently contains the 3 packages described in Table B-21.

Package name	Brief description
org.optimizationservices.oscommon.**api**	Contains OS analyzer services that can be accessed publicly over the OS network.
org.optimizationservices.oscommon.**analyzer**	Sample analyzers that analyze various optimization problems.
org.optimizationservices.oscommon.**parser**	Customized parsers that use the standard parsers from oscommon to convert instances to and from the analyzer-specific formats or data structures.

Table B-21: OSAnalyzer packages.

Table B-22 through Table B-24 list some of the important classes in each of the above 3 OSAnalyzer sub-packages. For detailed usage (fields, methods, etc.) of each class refer to the Javadoc and other documents on the OS Web site.

Sample classes in the api sub-package	Brief description
DrAMPLAnalyzerService	DrAMPLAnalyzerService is an api that is public to the external world. It hides all the parsing, local interfacing, and analyzing processes. It implements the OShL and OSkL interfaces from OSCommon. It is called by an OS agent. The service analyzes an optimization instance and returns an OSaL analysis result.

Table B-22: Sample classes in org.optimizationservices.osanalyzer.api.

Sample classes in the analyzer sub-package	Brief description
DrAMPLAnalyzer	Dr. AMPL analyzer that analyzes various optimization problems.

Table B-23: Sample classes in org.optimizationservices.osanalyzer.analyzer.

Sample classes in the parser sub-package	Brief description
DrAMPLOSiLReader	The DrAMPLOSiLReader class uses the generic OSilReader to parse an OSiL instance into the Dr. AMPL's format.

Table B-24: Sample classes in org.optimizationservices.osanalyzer.parser.

B.7 OSSimulation Library

The OSSimulation library (project or jar) currently contains the 3 packages described in Table B-25.

Package name	Brief description
org.optimizationservices.ossimulation.**api**	Contains OS simulation services that can be accessed publicly over the OS network
org.optimizationservices. ossimulation.**simulation**	Sample simulation engines that run simple or complex simulations.
org.optimizationservices. ossimulation.**parser**	Customized parsers that use the standard parsers from oscommon to convert instances to and from the simulation-specific formats or data structures.

	simulation-specific formats or data structures.

Table B-25: OSSimulation packages.

Table B-26 through Table B-28 list some of the important classes in each of the above 3 OSSimulation sub-packages. For detailed usage (fields, methods, etc.) of each class refer to the Javadoc and other documents on the OS Web site.

Sample classes in the api sub-package	**Brief description**
SampleSimulationService	It is an api that is public to the external world. It hides all the parsing, local interfacing, and simulation processes. It implements the OScL and OSkL interfaces from OSCommon. It is called by an OS agent. The sample simulation service runs various sample simulations and returns an OSsL simulation result.

Table B-26: Sample classes in org.optimizationservices.ossimulation.api.

Sample classes in the simulation sub-package	**Brief description**
SampleSimulation	It calculates various simple or complex functions and operations.

Table B-27: Sample classes in org.optimizationservices.ossimulation.simulation.

Sample classes in the parser sub-package	**Brief description**
SampleSimulationParser	The class uses the generic OSsLReader and OSsLWriter to read or write an OSsL instance to or from the SampleSimulation's format.

Table B-28: Sample classes in org.optimizationservices.ossimulation.parser.

B.8 OSRegistry Library

The OSRegistry library (project or jar) currently contains the 5 packages described in Table B-29.

Package name	**Brief description**
org.optimizationservices.oscommon.**api**	Contains the OS registry service that can be accessed publicly over the OS network
org.optimizationservices.oscommon.**parser**	Customized parsers that use the standard parsers from oscommon to convert instances to and from the OS registry-specific formats or data structures.
org.optimizationservices.oscommon.**registry**	The OS registry and provides join, discover, validate and other operations.
org.optimizationservices.oscommon.**util**	Utility classes used by the OS registry.
org.optimizationservices.oscommon.**web**	OS Web site development related classes such as Java servlets.

Table B-29: OSRegistry packages.

Table B-30 through Table B-34 list some of the important classes in each of the above 5 OSRegistry sub-packages. For detailed usage (fields, methods, etc.) of each class refer to the Javadoc and other documents on the OS Web site.

Sample classes in the api sub-package	Brief description
OSRegistryService	It is an api that is public to the external world. This is what people sees as an OS registry service. It hides all the parsing, local interfacing, and registry related processes. It implements the OSjL, OSdL and OSvL interfaces from OSCommon. It is called by an OS agent.

Table B-30: Sample classes in org.optimizationservices.osregistry.api.

Sample classes in the parser sub-package	Brief description
OSRegistryReader	The class uses the generic OS registry related reader to read a registry related OSxL instance to the OSRegistry's own format.
OSRegistryWriter	The class uses the generic OS registry related writer to write a registry related OSxL instance from the OSRegistry's own format.

Table B-31: Sample classes in org.optimizationservices.osregistry.parser.

Sample classes in the registry sub-package	Brief description
OSRegistry	The OS registry class that provides join, discover, validate and other operations.

Table B-32: Sample classes in org.optimizationservices.osregistry.registry.

Sample classes in the util sub-package	Brief description
OSRegistryCommonUtil	Common utility classes that provide various convenient methods used by the OSRegistry.

Table B-33: Sample classes in org.optimizationservices.osregistry.util.

Sample classes in the web sub-package	Brief description
OSRegistryJoinServlet	A java servlet class that is used with the OS join Web form. When the user clicks the submit button of the Web form, the servlet parses the form and generates an OSeL instance that is then sent to the OS registry database.

Table B-34: Sample classes in org.optimizationservices.osregistry.web.

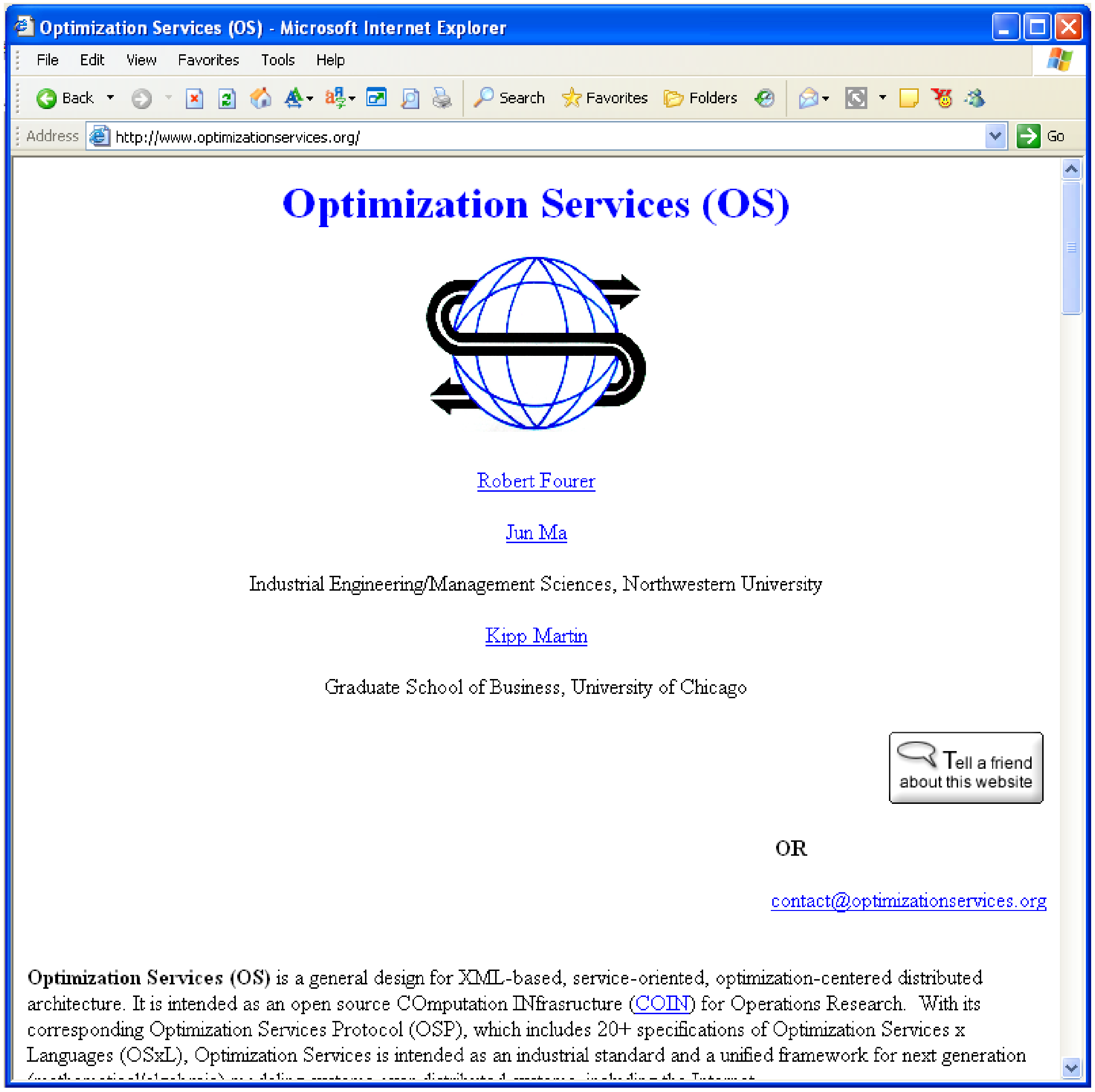
B.9 Optimization Services Server

We provide the OS server software that can be downloaded and installed on the OS service providers' computers and host their Optimization Services. The OS server uses the Tomcat Web server [4] from Apache for HTTP and Java servlet handling. It uses Axis [5] again from Apache for Web services SOAP handling. The OS server then adds OS related libraries and classes as plug-ins for OSP handling (such as OSxL representations and communication). User manuals are provided on the OS Web site.

B.10 www.optimizationservices.org and www.optimizationservices.net

Figure B-4 shows the OptimizationServices.org Web site. Currently the OptimizationServices.net is mirrored after the .org Web site and provides exactly the same information. Later contents on the two Web sites may diverge on different emphases. But the standards (schemas, WSDL documents, the OS registry) will always use the OptimizationServices.org address. OptimizationServices.net will provide auxiliary services that facilitate the use of Optimization Services. Various papers, presentations, user manuals, standards, software and other documents are published via the OS Web sites. For latest information always check the two Web sites:

http://www.optimizationservices.org and http://www.optimizationservices.net


Optimization Services (OS) - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Folders
Address http://www.optimizationservices.org/ Go
Tell a friend about this website

Optimization Services (OS)

Robert Fourer

Jun Ma

Industrial Engineering/Management Sciences, Northwestern University

Kipp Martin

Graduate School of Business, University of Chicago

OR

contact@optimizationservices.org

Optimization Services (OS) is a general design for XML-based, service-oriented, optimization-centered distributed architecture. It is intended as an open source COmputation INfrasructure (COIN) for Operations Research. With its corresponding Optimization Services Protocol (OSP), which includes 20+ specifications of Optimization Services x Languages (OSxL), Optimization Services is intended as an industrial standard and a unified framework for next generation

Figure B-4: The OS Web site at http://www.optimizationservices.org (or http://www.optimizationservices.net)