NORTHWESTERN
UNIVERSITY

Industrial Engineering and
Management Sciences

A General and Unified Design and Framework for Distributed Optimization

Ph.D. Proposal

by

Jun Ma

Advisor: Robert Fourer

A thesis proposal submitted in partial fulfillment of the
requirements for the candidacy of

Doctor of Philosophy

Industrial Engineering and Management Sciences

Northwestern University

2003

Approved by Robert Fourer

Chairperson of Supervisory Committee

John R. Birge

Wei Chen

Thomas Tirpak

Program Authorized
to Offer Degree Industrial Engineering and Management Sciences

Date

Rev. No.	Release Date	Reviser(s)	Comments
1.0	November 01, 2003	Jun Ma	Initial Draft
1.1	December 05, 2003	Robert Fourer, Jun Ma	Revised Draft for Proposal on December 12, 2003

Table of Contents

ACKNOWLEDGMENTS

1 ABSTRACT

Large-scale optimization has been a subject of investigation for over 50 years, but the challenge of making it useful in practice has continued to the present day. Initially the primary difficulties were posed by *computation*. But as computational needs were addressed by breathtaking increases in computer power and algorithm sophistication, the more serious difficulties came to be posed by *representation*. Again the challenge was eventually met, by increasingly sophisticated modeling languages and systems.

The primary difficulty of large-scale optimization has now shifted again, to one of *communication*. Currently there exist of a plethora of optimization algorithm implementations, various formats to represent optimization problems and heterogeneous mechanisms to communicate with optimization components. Besides, there are plentiful research initiatives in developing supporting tools to analyze and benchmark optimization problems and solvers. Moreover different optimization components are implemented in different programming languages and located on different operating systems all over the network.

In this project, I will analyze the above issues under two real world scenarios. One is Motorola's Virtual Prototyping Intelligent Optimization System that I have participated in designing over the past 3 years, led by Thomas Tirpak at Motorola Advanced Technology Center. The other is Argonne's National Laboratory's Network Enabled Optimization System, which has been developed by researchers at Optimization Technology Center led by Robert Fourer and Jorge Moré.

We propose a general design for distributed optimization architecture to bring together the seemingly significantly distributed optimization systems. The general design will serve as the basis for our unified framework introduced under our concept of "Optimization Services", intended as guidance for designing future Optimization Services components and next-generation optimization systems. The introduction of Optimization Services framework can be regarded as an initiative to start a wider level of cooperation to move toward a final standardization and facilitate a healthier development environment for research in the area of Operations Research.

2 BACKGROUND AND INTRODUCTION

Large-scale optimization has been a subject of investigation for over 50 years. But the challenge of making it useful in practice has continued to the present day. Initially the primary difficulties were posed by *computation*, but breathtaking increases in computer power and algorithm sophistication combined to allow for routine solution of large problems arising in practical applications [4]. As computational needs were addressed, the more serious difficulties came to be posed by *representation*, as modelers found that they could solve larger problems than they could manage or understand [24]. This challenge, too, was eventually met, by increasingly sophisticated modeling languages, and systems for describing and working with optimization problems [17][37].

The primary difficulty of large-scale optimization has now shifted again, to one of *communication*. Increasing numbers of optimization algorithms are implemented increasingly well, but prospective users are unaware of these "solvers" or do not see the potential benefit that would justify obtaining and installing them. Only certain combinations of solvers and modeling systems work with each other, moreover, and modeling language support is slow to keep up with solver extensions to new problems types.

2.1 A Real World Example (Motorola)

In 2001, I helped design an optimization service (though the term "optimization service" was loosely defined then), based on a modified feasible direction algorithm [41] for the Virtual Prototyping (VP) group [48], led by Thomas Tirpak at Motorola Advanced Technology Center (MATC). The service, along with the development of several optimization solvers including linear and integer programming types was later integrated into the Virtual Prototyping system. The service is intended to solve general large scale nonlinear constrained optimization problems with discrete variables. This optimization service has since proved to be of great value to the Motorola engineering community. It has been applied in areas like print wiring board panel layout problem and embedded passives selection in circuit board design and has helped achieve great cost reductions.

At the beginning, the VP optimization service was only applied in single and local domain model services, i.e., the objective function is calculated by one model service that is located on the same machine as the optimization service. But in the real world, an objective function can consist of metrics from multiple and distributed model services, as illustrated in Figure 2-1.

The objective f of the optimization service is comprised of metrics y_i's calculated from the corresponding service i. The variable set **x** is shared among all the services. The arrows indicate flow of information for iterations throughout the optimization. At the higher level is the optimization engine or solver that, at each iteration, suggests new values for the variable set **x** to individual model services. At the lower level are the model services that supply the objective functional values, $y_i(\mathbf{x})$ and constraint functional values, $g_i(\mathbf{x})$. In nonlinear optimization, which goal is to find a local minimum or maximum, it is possible to generate an improved solution just by knowing the numeric values of the objective and constraint functions at current iteration.

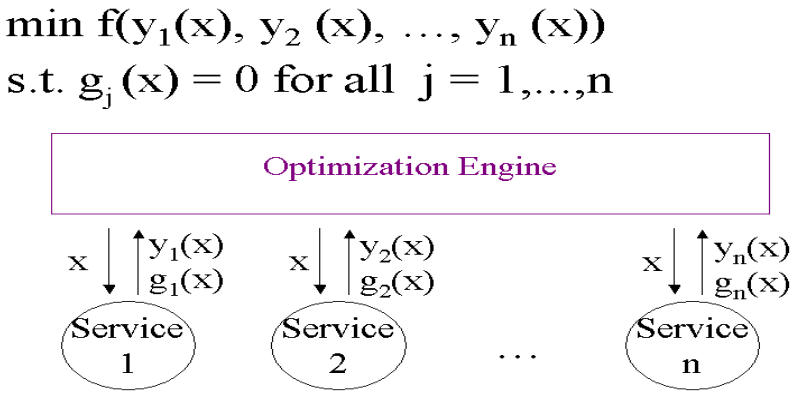

Figure 2-1: Dataflow of optimization with metrics calculated from distributed services

The problem of multidisciplinary optimization (MDO) with an objective function that incorporates metrics calculated by distributed model services arises from an ambitious multistage effort to develop and deploy enterprise-wide, suites of state-of-the-art tools that drastically reduce the cycle time for new or improved designs and technologies. The principal feature of this effort is the integration of design and development processes among various disciplines, e.g., mechanical engineering, electrical engineering, environment engineering, manufacturing, supply chain, etc.

In §1 we will address in detail the issues that we encounter in developing such a multidisciplinary optimization system from the perspective of scheduling procedure; and in §4 from the perspectives of architecture design and communication framework.

2.2 Another Real World Example (Argonne)

The Internet is now providing an increasingly practical way of addressing communication problems in large optimization [28]. Websites offer abundant solver information [25], to be sure, but the more significant advance is the ability to send optimization problems over the Internet for submission to a solver at some remote site. The remote optimization "server" can address numerous problem types and can provide varied solvers for problems of each type, giving modelers much more of a choice than they could hope to have locally. In previous work under the auspices of Optimization Technology Center (OTC) co-directed by Robert Fourer at Northwestern University and Jorge Moré at Argonne National Laboratory since its founding in 1994, member researchers have studied and experimented with the concept of an optimization server through the creation of the Network Enabled Optimization System (NEOS) Server [11][14][34].

The continuing goal of the NEOS project is to make optimization a part of the worldwide software infrastructure that supports science and commerce. To this end, the NEOS Guide (http://www-fp.mcs.anl.gov/otc/Guide) includes online examples of optimization problems, listings of test problem collections, and surveys of publications and software. The complementary NEOS Server (http://www-neos.mcs.anl.gov) provides remote access to solvers and so is the focus of this project.

The NEOS Server currently supports nearly 70 solvers. Collectively these solvers accept about a dozen different kinds of input, ranging for example from function definition in programming languages (Fortran, C, Matlab) to explicit problem instance descriptions (MPS, LP, sparse SDPA) to symbolic modeling language descriptions (AMPL [26], GAMS [6]). A callable interface, Kestrel [13], also permits direct access to many of the NEOS solvers from within modeling systems' environments.

Usage of the NEOS Server (Figure 2-2) has grown to an average level of about 10000 submissions per month; peak loads of 5000 in a week have been handled without difficulty. Submissions have, however, leveled off a bit in the past few months; this has motivated us to direct some of the proposed research, particularly in §4, toward making the Server easier to use for those who are not solver experts.

The current NEOS Server only begins to address the communication difficulties of large-scale optimization, however. The Server cannot tell users which solvers are appropriate for a problem that has been submitted, or choose a solver host based on the expected resource needs of a problem. Connections from modeling languages to solvers are still incomplete, and support for benchmarking is limited. Because NEOS has evolved along with the Web and the Internet – its first interface, through e-mail, dates back to 1996 – it is limited to some degree by early design decisions.

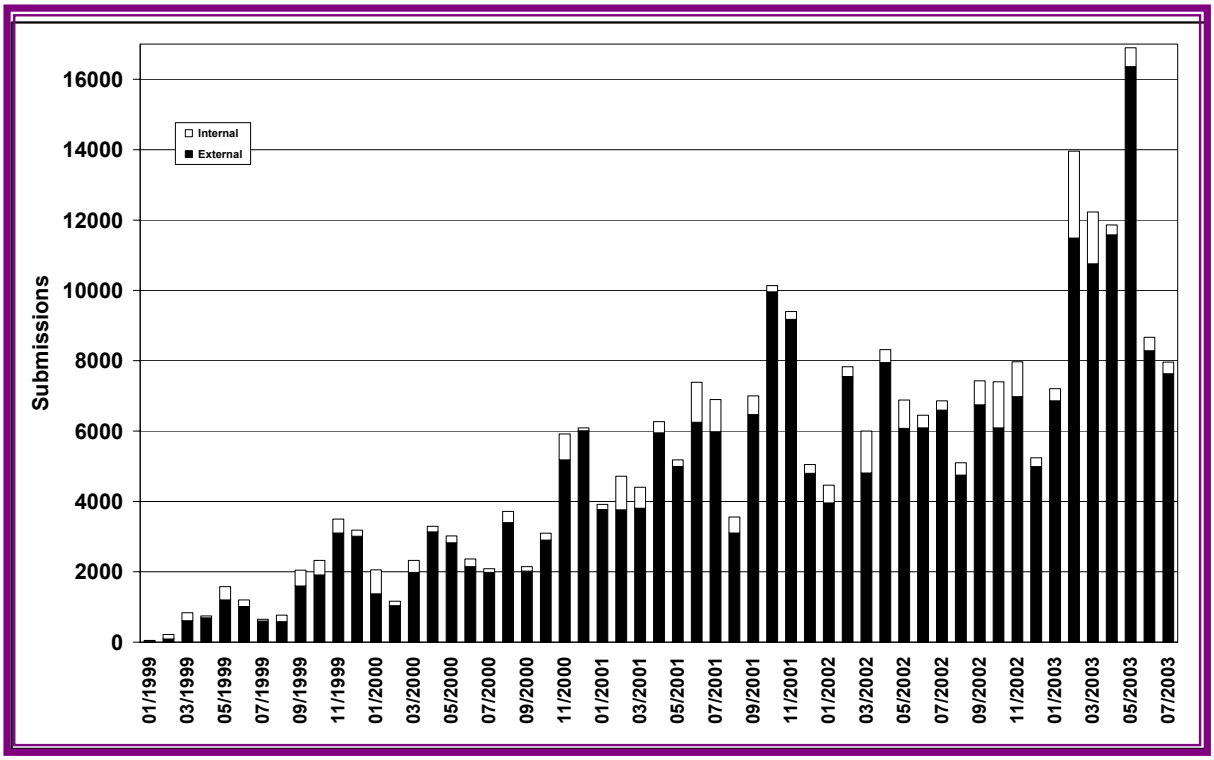

Figure 2-2: Monthly total submissions to the NEOS Server since 1999. "Internal" submissions are those from the domains of Argonne (anl.gov) and Northwestern (nwu.edu).

3 TWO DISTRIBUTED OPTIMIZATION SYSTEMS

In this section we will discuss two distributed optimization systems – Motorola's Virtual Prototyping (VP) Intelligent Optimization System (§3.1) and Argonne National Laboratory's Network Enabled Optimization System (NEOS, §3.2). Issues in designing and implementing the two systems will be raised and discussed in detail. The two seemingly significantly different systems provide us with the motivations for a general and unified design and framework for distributed optimization. As will be seen in §6, these two systems can be viewed as special cases of our general design of distributed optimization architecture. Our general and unified design and framework (see §4) is intended to resolve the issues regarding architectures, communications, and representations and help build robust optimization over distributed systems.

3.1 Motorola VP Multidisciplinary Intelligent Optimization System

3.1.1 General Background

The VP optimization system is a critical step in a multistage effort to develop and deploy enterprise-wide, suites of state-of-the-art tools that drastically reduce the cycle time for new or improved designs and technologies. The principal feature of this effort is the integration of design and development processes among various disciplines. The goal is to plan, design, construct and manage knowledge-based systems for the transfer, application and execution of knowledge, usually highly specialized. 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.

3.1.2 Knowledge Flow

Knowledge derives originally from customers, who express in the form of specifications of their needed product. 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 departments or groups for proof-of-concept designing or prototyping. Without the Multidisciplinary Intelligent Optimization System, the engineering solutions that have been developed in a separate manner finally are combined together into a complete prototype in a more or less mechanical 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 until the final complete product is free from design conflicts.

In contrast, the optimization system takes the responsibility of coordinating the design solutions that originate from separate departments, finds a feasible solution and possibly optimizes within the feasible choices to find the best combination of design. As shown in Figure 2-1, the optimization system architecture leverages on knowledge flow in the real engineering world. The system is broken up into two levels. The higher level is the one that assumes the role of coordination and the lower level is all the individual functional modules or simulation services that keep on feeding their separate solutions to the higher level.

3.1.3 Properties of the Model Services

Optimization services and solvers mostly need users to submit all the data of the problem, at least including mathematical formulas for objectives and constraints. Such requirements cannot be met due to the properties of our model services:

1. The final objective and constraint functions consist of multiple services.

2. 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.
3. Some model services are so complicated that no simple mathematical representation can be formulated.
4. Some of the model services are proprietary and thus their formulas cannot be revealed.
5. Most importantly, some model services do not return results instantaneously (see §3.1.4). The delays make it difficult to integrate the model services into the optimization solver.

3.1.4 Initial Modeling of Computational Complication

In our modeling (Figure 3-1), different optimization solvers are extended from a standard optimizer interface. All solvers interact with optimization problems with a common interface. The optimization problem interface is connected with a simple accelerator, which purpose is to simulate the behavior of remote services, and provide estimated function values to the solvers locally, thus avoiding networking anomalies. Each remote service has a corresponding local optimization problem client connected with the interface.

Model services are simulated with arbitrarily chosen and relatively simple functions they 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 sleep or wait according to the parameters specified for each service, before the function values are returned to the optimization solver. To speed up the modeling 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 day, the complexity of the scenario represented by the inputs ($\mathbf{x}$), etc. Services may be unavailable at certain scheduled and/or unscheduled times; there may be a delay in transmitting the inputs to the services and/or the outputs from the services or even the model service may itself be an optimization process.

The model services in Motorola's Virtual Prototyping System can be characterized mainly according to three factors, which determine the time each optimization iteration takes: 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/or when there are difficulties running the service for a given set of inputs ($\mathbf{x}$). Communication time between the optimization engine and model services is insignificant. 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.

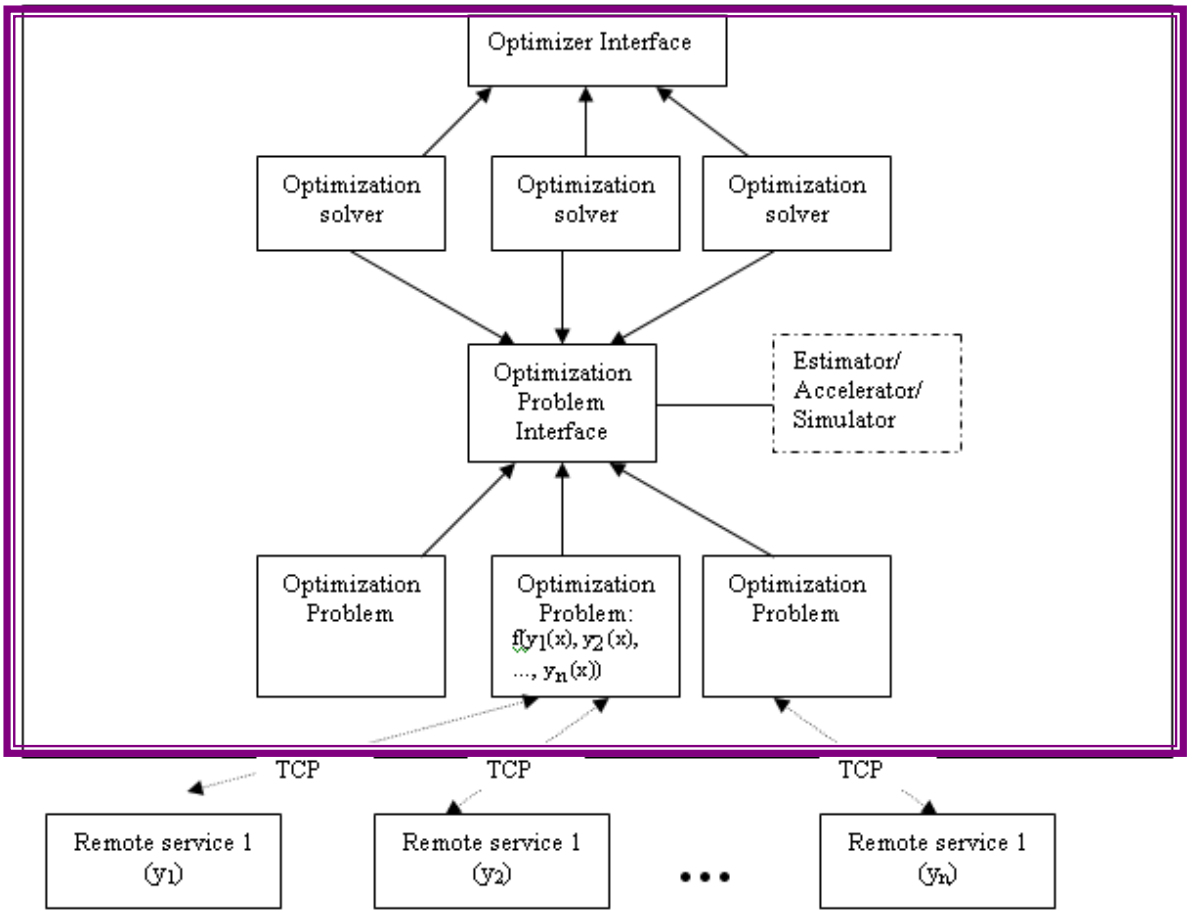

Figure 3-1: Architecture of Proof-of-Concept Modeling of Optimization with Metrics Calculated from Distributed Simulation Services.

Moreover when engineers in other areas design and construct their model services, they do not have the intention 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 has to be based on a direct method, that is, an algorithm not using derivatives.

Benchmarking has been conducted on different optimization algorithms, and a method based on Powell's algorithm [45] with quadratic step length estimation was tested and implemented in the prototype modeling system.

Our initial tests have proceeded as follows. Benchmark problems are first tested with their objective functions unbroken 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 based on 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 step. Primitive estimations or acceleration techniques, for example quadratic fitting, smoothing splines, have been used. Estimation of execution time is given by the following formula and data:

$$T = (Ts\ )\ (LF(t)) + DT, \qquad \textbf{(Equation 3-1)}$$

Where:

Ts = 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:

Ts = 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:

Ts = 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:

Ts = 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.

Through our initial modeling, we have shown that without any estimation and acceleration techniques, the optimizations in distributed system are solved with the same accuracies and same number of iterations, but the time taken to solve each problem is significantly longer, since the

optimization solver always has to wait for the last and slowest machine to respond with a functional value.

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), 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 scheduling procedure.

3.1.5 An Approach on Robust Design of Distributed Optimization

The next sections introduce our research effort on more advanced architecture and intelligent methods of search and acceleration. Special procedures are being developed along with optimization, in areas of statistical learning and artificial intelligence including data mining and machine learning. The real world challenge is how the optimization engine should simultaneously use information about information such as rate of improvement of the objective function and the computational performance characteristics of a set of distributed model services, to efficiently manage the evaluation of the objective function, so that the "best" solution can be found in the "shortest" possible time.

3.1.6 Design and Architecture

Figure 3-2 shows the Virtual Prototyping Multidisciplinary Intelligent Optimization System. The upper right part of the figure is the solver architecture. Listed are the major component modules.

Remote Central Server – This is mainly used to connect to different distributed services offered by the Virtual Prototyping System, and maintain administrative routines.

Simulation Engines – These contain the major Virtual Prototyping services in different engineering domains.

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 and constraints.

Client – This is usually any engineer who wants to use the services connected through the central server. From the client's view, model constructor is simply another simulation model.

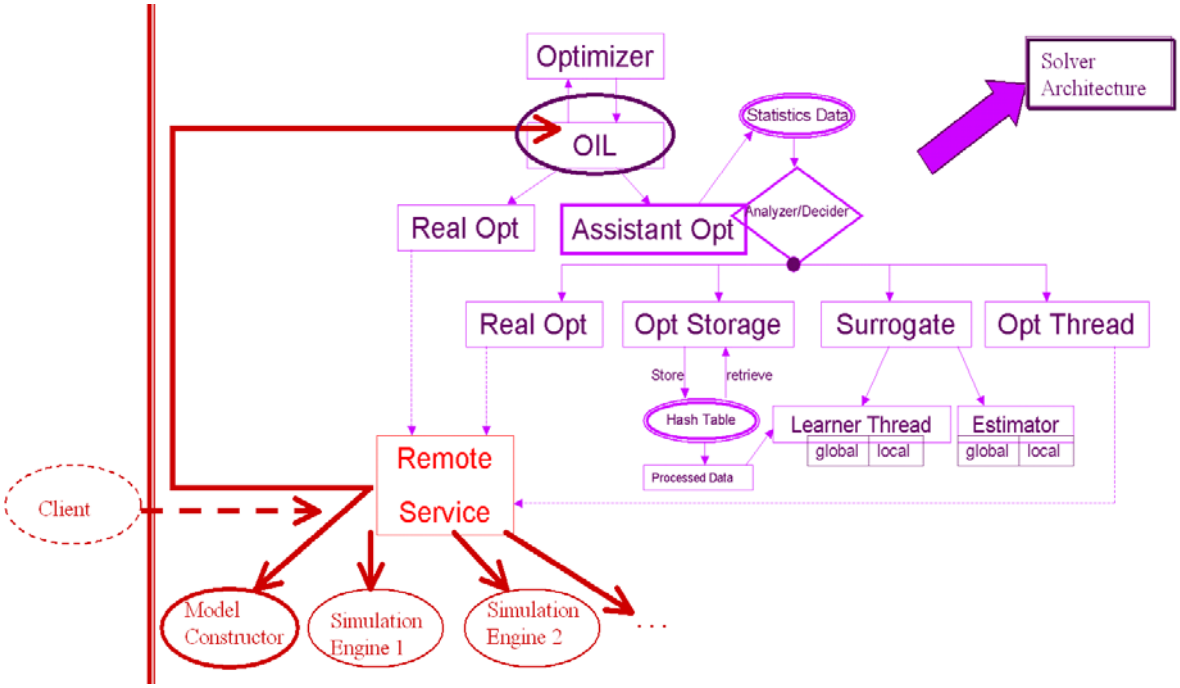

Figure 3-2: Architecture of the VP Multidisciplinary Intelligent Optimization System

Following are the modules related specifically to the solver architecture.

Solver – This module contains optimization solvers of different types, including linear programming, nonlinearly constrained programming, integer programming, etc.

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 services offered through the central server. It also helps generate gradient information such as Hessian matrices needed by the gradient-based solvers.

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.

Real Opt – This is the module that routes solver requests to real simulation engines.

Assistant Opt – This is the module that routes solver requests to a set of "intelligent" components and surrogates for optimization acceleration and robustness.

Analyzer/Decider – This is the module that Assistant Opt uses to branch to different optimization processes.

Opt Storage – This is actually an interface that provides accesses to retrieval and storage of online optimization data, for example the variable points and objective values on the optimization path.

Hash Table – This is basically a database that stores all the evaluated variable points in a special way.

Surrogate – This is the module that acts as an approximate deputy for a simulation model.

Processed Data – This module is a data structure that processes the data stored in Hash Table into a format accessible by Surrogate Learner.

Learner – This module takes the processed data from Hash Table, and learns functions that approximate response behaviors of the simulation engines.

Estimator – This module takes the learned function from Learner and responds to the solver with an estimated function value.

Opt Thread – The purpose of Opt Thread is that solver does not need to wait or just wait a short time for a response from simulation engines because it is launched as a separate process from the general optimization process. On one hand function values are still to be returned. Solver can just carry on its iterative optimization progression. One main advantage is that when a simulation engine returns an error, the thread can simply be aborted without affecting the solver process.

3.1.7 Service Requirements and Non-generic Solutions

In designing an intelligent multidisciplinary optimization system that involves pre-built or legacy simulation engines never intended to be optimized and distributed all over the network, 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 are solved on an *ad hoc* base. Many of these serve as a motivation for a general design and framework for distributed optimization.

- **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 are required of each variable in the form of **(default value, mostly likely value, lower bound, upper bound)**, after consulting with domain engineers. Default values can be customized for each optimization run by the client. In case when multi-start optimizations are carried out [38], distribution functions (for example triangular distribution based on mostly likely value, lower and upper bounds) can be 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 specifications. Originally, the situation is handled by constructing interdisciplinary constraints forcing different variables to be of the same values. But the

optimization problem size is unnecessarily large due to redundant variable declarations. An overhaul thus 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 "paring" table has been constructed to support the Model Constructor module in Figure 3-2, so common variables are detected and variables are declared only once. But still other issues exist.

Clients may be unaware of the common variable situations by supplying different default values to two differently named copies of the same variable. In cases like these, model constructor takes the average of the two default values. Most likely values, lower and upper bounds may also assume different values when a single domain simulation is run. When constructing a multi-domain model, the largest lower bound, the smallest upper bound, and the average of the mostly likely values are assumed by the model constructor.

- **Objective Construction**

Multidisciplinary objective function usually takes the form of a weighted sum. Different simulation engines are chosen by the clients and corresponding weights are specified. Weights are solely based on a subjective judgment base reflecting importance of different simulation metrics deemed by the client. But the client has to tell whether a smaller value or a bigger value of a metrics is better or not, so that model constructor can build a consistent maximization or minimization objective function. Metrics of different simulation engines are of different unit, thus the constructed multi-objective function is unitless and only useful for relative comparisons. Meaningful reports for each simulation are constructed based optimal variable values. Metrics of different simulations engines are of different scales. Normalization techniques such as arctangential and sigmoidal transformation are taken to bring component metrics on to the same scale.

- **Constraint Enforcement**

Constraints of the multi-disciplinary optimization are a combination of all constraints from individual domain constraints of each simulation and all variable boundaries. All the interdisciplinary constraints are hard coded in an assistant module that accompanies the Model Constructor module. The Model Constructor module first detects which simulation engines and which variables are chosen, and it then incorporates into the optimization model the interdisciplinary constraints that contain the simulations and the variables.

- **Result Interpretation**

Though Motorola has a proprietary data format to standardize results from different simulation engines, but they were never intended to be combined with each other to, say construct a multi-objective function. Name confliction is one major issue. Efforts have to be taken in setting distinctions between names. One way is to rename, but this causes tremendously many unforeseeable bugs. The other way is to group results into subsections and use combination of

simulation names, subsection names and result names. This issue will be elegantly solved by the introduction of XML namespaces as we will see in our general design and framework.

Another issue, though 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, as introduced in §3.1.8, this problem is naturally avoided. Another technique is on a situation by situation base. In one circumstance [51], 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 that accompanies the Model Constructor module. The Model Constructor module first detects whether the simulations that have discrete objectives are chosen, and then incorporates into the optimization model the corresponding interdisciplinary objective terms.

- **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 by putting a barrier at the end of all the request calls. Other typical techniques are also employed depending on situations. Any major text books on designing and building parallel programs cover some most popular and practical algorithms, see [20]. For our purpose in Virtual Prototyping, most of the time, the multi-objective function can only be constructed with the returns of all the component objectives from distributed simulations, robust design with some acceleration techniques are needed for further speed up. This will be discussed in §3.1.8.

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**

All processes cannot just be launched in a parallel version. Some simulations (e.g. [50]) may contain variables that are results from other simulations (e.g. [51]). Flows are hard coded when the Model Constructor encounters a combination of simulations that need to in sequence. Processes that have to wait for results from other processes are waited in a queue to be notified later. Some kind of standard service flow coordinating system is needed here.

- **Input Parsing/Output Reporting**

All input parsing and output reporting are specified in a Motorola proprietary format. Though standardized, yet complicated enough to be understood by just a few. It was not built for multi-disciplinary optimization constructions. Special efforts have to be taken to scale it up for accommodations. In the case of process sequencing, where one simulation's variable takes a value from another simulation's result during run time, the effort is extremely laborious. In the

case of generating reports of multidisciplinary results and mapping multi-dimensional space onto two-dimensional graphs, the procedure is even more painstaking.

3.1.8 Procedure and Reasoning

Figure 3-3 shows the processes of an entire intelligent optimization system, in an effort to build a robust distributed optimization system, with reasonable accelerations.

Normal Flow

On the left part of the figure are processes (Processes **0-10**) with bold borders that represent a normal nonlinear optimization flow: roughly starting with an optimization problem instance, entering an iterative process of finding directions and step lengths, updating variables and terminate and return results based upon certain conditions. The major characteristic in this flow is that processes **2** and **5**, when requesting a function value to determine directions and step lengths do not get them locally. Instead they have to go through process **11**, the solver interface, which when no intelligence is needed in Process **12**, always goes through the central server (process **13**) and asks its connected simulation engines (process **14**) to return function results F(**x**).

Processes on the right part of the figure (Processes **15-25**), with dotted borders represent the intelligent components. Notice the total separation between the solver and the intelligent part. None of the intelligent components are built within the solver, that is, the optimization algorithm remains untouched. The idea is that any Virtual Prototyping solver can leverage on the intelligent system with no alteration and also any intelligent system that is compatible with the solver interface can be plugged with no extra effort. Solver interfacing will be unified in our general design and framework introduced in §5, so under the framework, any supporting tool suite can be integrated with solvers seamlessly.

Processes **2** and **3** are intended to find moving directions. A large number of requests are made to obtain information on function values and gradients in all variable dimensions. Thus arrows leading out of process 2 and leading into process 3 are in **bold**. Processes **5** and **6** are intended to find step lengths along the decided moving direction. Relatively much smaller number of function requests is needed. Functional evaluations are computationally expensive in our distributed optimization scenario. Thus the specific solver that we favor has a loop back mechanism from process **6** to process **4**, intended to do a very accurate linear search on step length. In practice, all the loops combined to find a step size take a fraction of time of finding a direction.

The major decision branching is Process **12**. If no intelligence is needed, it goes through a regular distributed optimization process. Otherwise, it leverages on the estimation and acceleration techniques in the intelligent part. Process **15** is used to store evaluated data in the Hash Table module. It can be turned off when no intelligence is needed.

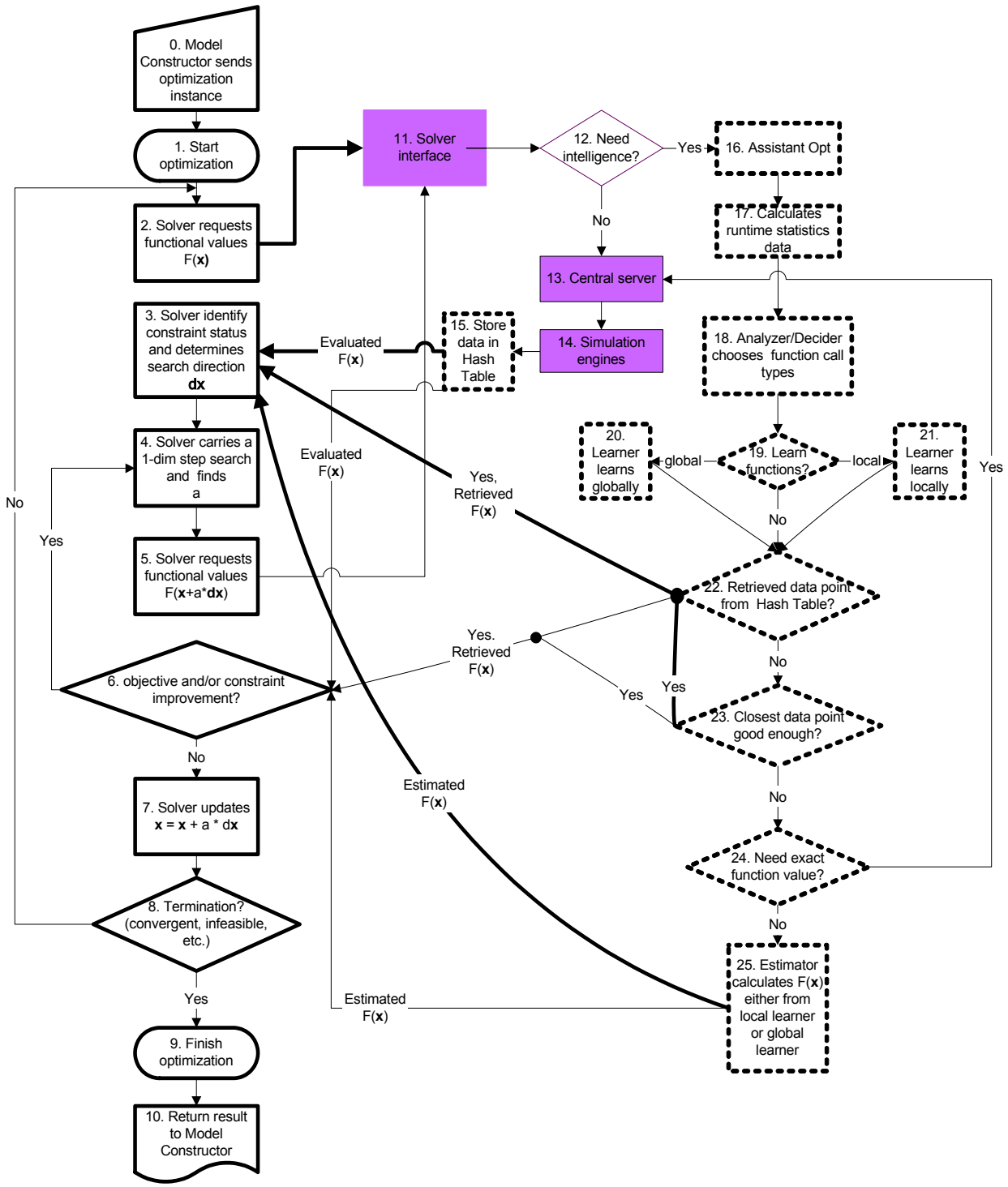

Figure 3-3: Flowchart of intelligent optimization process

Intelligence Flow – Analysis

When intelligence is tuned on, the process always goes through an Assistant Opt module (Process **16**).

The first thing that an Assistant Opt module does is to analyze statistics of run time information, including:

- those related to optimization process, for example current iteration number, variable change rate, objective convergence rate, constraint improvement rate
- evaluated data points in database
- finishing status of a simulation
- time it takes between requests and responses of a simulation over recent iterations
- access types of recent runs – retrieved through database, estimated through an approximate 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 value and real value

Statistics are constantly updated on finishing of corresponding processes that provide such information.

Intelligence Flow – Learning

Process **18** is a decision to learn a function based on all the collected points that have been evaluated by real simulation engines so far. The decision to learn a function is based on one fact, namely when there are enough new data points. The choice of the number of data points is quite empirical. It can be further studied and on an adaptive base. All learners are launched in separate processes, so that the flow can move on to the next three decisions (Processes **22, 23, 24**).

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, Gene Programming, though potentially more accurate, can take time comparable to the optimization process itself. Motorola Advanced Technology Center has developed some advanced though proprietary or patented learning tools that take a short time, which the intelligent optimization system leverages on. But the main purpose here is not to describe the algorithms inside these tools. The intention is to illustrate that with the help of well designed learning tools that are properly coordinated with an intelligent optimization system, decent acceleration can be achieved. In addition to the proprietary tools, a range of other algorithms are incorporated into our stack of learning tools. Learning tools are grouped into global learners and local learners separately. In practice, local learners are relatively fast.

Global learners include standard statistic regressions, neural network, gene programming, etc. Global learners are launched when an optimization first starts. 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 roughly smooth shape (that is, 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 launching local learners. Global and local learners, in our optimization system, are launched separately.

Local learners include basis expansions 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 global to local learner, we have accumulated more points. Many algorithms in local learning need a large number of points to fit functions in high dimension variable space.

Just as in optimization that no solvers always perform 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 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 dimension. If a learner takes an extremely long time, it may just be dropped.

We are also developing optimization-specific learners that leverage on information from the optimization path and runtime optimization performance. They will be illustrated after further works at Northwestern University and Motorola Inc.

The following decisions are based on the three ways that the solver can get functional values: retrieval, evaluation, and estimation.

Intelligence Flow – Retrieval

Function value retrieval from database happens quite often in practice. Our database is in essence a hash table with the hash key being the **x** variable and the hash value being the function value f(**x**) combined with an access index. Access index measures recentness of variables, useful in cases where only recent points are needed for learning, estimation and validation. Admittedly, hash table takes up memory. Our reasoning is that memories are abundant and inexpensive, and in practice we never have to face memory overflow due to the accumulation of data points. The growth is only linear. Our main concern is *speed* rather than *space*. The greatest advantage of a hash table is that 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 in the table. Data precision is kept to certain decimal points and digits after that are truncated to avoid numerical ill-conditioning.

There are mainly 3 reasons that same points are being retrieved. The first is due to searching algorithm going back to the same region. The second is due to algorithms using finite difference 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, after first calculation, 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 a same scale before optimization, a "closeness" measure is set to a very small fraction of 1 multiplied by the number of variables. The closest point is returned if the distance between the closest point and current point is (1) smaller than the "closeness" measure, *and* (2) 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 guard against finite difference based gradient estimation, in which the last point is almost surely the closest point, thus generating gradient value of 0.

Intelligence Flow – Evaluation

If no previous data point or closest data point can be retrieved, Analyzer/Decider may choose to get the evaluation (Process **24**) from the real simulation engine (Process **14)** through Central Server (Process **13)**. This process is always launched in a separate process, 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 the simulation result is obtained fine, it is first stored in the database or Hash Table (Process **15**). If there is error returned or the maximum wait time expires, the flow moves on to the next process (process **24**) to return an estimated function value. This is a major step toward robust optimization design against simulation anomaly. If the process is alive after the maximum wait time, it can still store the result in database. This stored result is of special interest in validation and comparison of learners, because this point is both estimated by a learner (actually returned to solver, too) and evaluated by the real simulation engine.

Intelligence Flow – Estimation

If 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 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 out 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 Analyzer/Decider has an *ad hoc* mechanism to guarantee convergence or termination. Estimation cannot be made in a row for some number of times. After convergence rate is slow or iteration number exceeds a certain number, Analyzer/Decider will just choose to always get evaluation from real simulation. Due to the small convergence and the large iteration number that we set, this mechanism is seldom used in practice.

3.1.9 Benchmarks

Figure 3-4 shows an initial benchmarking between the Virtual Prototyping distributed optimization system with and without using intelligent techniques, both using exactly the same solver and on the same set of distributed machines.

Comparisons are made only on solution accuracies and time each system takes to achieve such accuracies. No comparisons are made between iteration numbers, because it naturally takes less iteration for optimization without intelligence since the optimization is always carried on the exact value retuned by real simulation. Optimization with intelligence can potentially take a detour in searching, but the time saved from getting function values through retrieval and estimation is worth such a detour.

Problem set includes typical nonlinear problems such as Rosenbrock, Beale, Powell, Helix, Cube, Box etc. Testing results on other problems are not listed because comparison results are extremely similar to those conducted on the Rosenbrock problem. Real Motorola simulation services are used too. More will be included later. The initial benchmarking results are quite encouraging.

First of all without any intelligence, distributed optimization service will simply not be able to finish if simulation engines crash. The Intelligent optimization system can sustain up to 50% simulation errors, that is one out every two times a simulation engine will crash. Though the time it takes to finish optimization (naturally) increases with errors, but most of the time it increases at a slower rate than the errors, especially when the number of errors is small around 1%.

With simulations whose function evaluations are quick, there is no advantage of using intelligent optimization, due to all the overhead needed to getting a simple function value. But with longer service delay (that is, when simulations take more time), intelligent optimization system turns out to be saving more time with about the same accuracies achieved. Usually accuracies achieved are always about the same, because standards used for termination in the solver are not changed at all. In practice, a client has the choice to set optimization system to using intelligence or not. Most clients have a good idea about the behaviors of the simulation services. If the simulations are instant and robust, they are suggested to set the intelligence off.

problem	w/o Intelligence		w/ Intelligence	
	time	solution (X1*, X2*,..)-> Y*	time	solution (X1*, X2*,..)-> Y*
Rosenbrock (<1mm excecution)	200 ms	(0.97, 0.94) -> 0	401 ms	(0.97, 0.94) -> 0
Rosenbrock (w/ 1% service error)	not able to finish		400 ms	(0.99 0.98) -> 0
Rosenbrock (w/ 10% service error)	not able to finish		641 ms	(1.0, 1.0) -> 0
Rosenbrock (w/ 20% service error)	not able to finish		1012 ms	(0.97, 0.94) -> 0.001
Rosenbrock (w/ 30% service error)	not able to finish		2000 ms	(1.0, 1.0) -> 0.0
Rosenbrock (w/ 40% service error)	not able to finish		>10000 ms	(1.04, 1.09) -> 0.002
			can have a tolerance with up to 70% or service error	
Rosenbrock (w/ average 10ms delay)	26327 ms	(0.97, 0.94) -> 0	26077	(0.97, 0.94) -> 0
Rosenbrock (w/ average 20ms delay)	51693 ms	(0.97, 0.94) -> 0	48068	(0.97, 0.94) -> 0
Rosenbrock (w/ average 30ms delay)	105869 ms	(0.97, 0.94) -> 0	88195	(0.97, 0.94) -> 0
	w/ longer delay (i.e. service take longer time), intelligent optimizer turns out to be saving more time (juan will show this in real scenario when services are much longer)			
For all other problems in the test set including beale, Powell, Helix, Cube, Box etc. the comparisons are extremely similar to those conducted in Rosenbrock problems.				
Panel (7x9) (w/ 0% service error)	2333 ms	6642 totoal board	6619 ms	6642 totoal board
Panel (7x9) (w/ 1% service error)	not able to finish		46125 ms	6642 totoal board
Panel (7x9) (w/ 10% service error)	not able to finish		50000 ms	6642 totoal board
Panel (7x9) (w/ 20% service error)	not able to finish		>60000 ms	6642 totoal board
Panel (7x9) (w/ 30% service error)	not able to finish		>100000 ms	6642 totoal board
Panel (7x9) (w/ 1ms delay)	41048 ms	6642 totoal board	36000 ms	6642 totoal board
Panel (7x9) (w/ 10ms delay)	420000 ms	6642 totoal board	370000 ms	6642 totoal board
Panel (7x9) (w/ 10ms delay)	850000 ms	6642 totoal board	600000 ms	6642 totoal board
	w/ longer delay, intelligent optimizer turns out to be saving more time			

Theoretical optimal solution for Rosenbrock is (1, 1) -> 0

Figure 3-4: Benchmarking between distributed optimization with and without intelligence

3.2 AMPL and Network Enabled Optimization System (NEOS)

3.2.1 Standalone AMPL Architecture

AMPL is a modeling language for mathematical programming. For detailed description, refer to the book in [27]. Figure 3-5 shows the standalone optimization modeling system architecture of AMPL interacting with a locally connected solver.

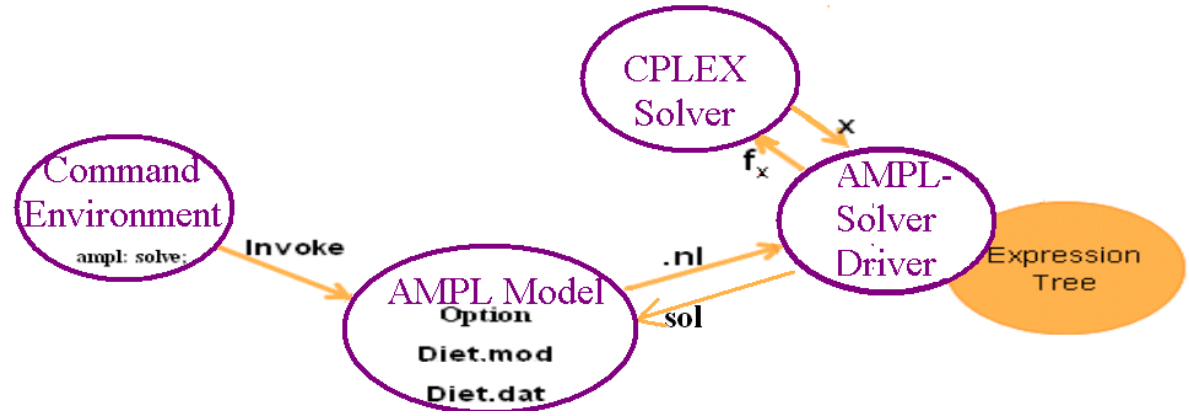

Figure 3-5: Standalone AMPL-Solver Architecture

A user begins in a command environment. After starting AMPL, the first thing the user sees is 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 3-5, 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 lower level optimization instance representation in the AMPL ".nl" format. The ".nl" instance file is then sent to a solver for optimization through the AMPL-Solver Driver, which is basically an interface between the AMPL modeling language and the hooked solver.

For nonlinear objectives and constraints, the AMPL-Solver Driver has at its back corresponding expression trees for calculating function values. Throughout optimization iterations, solver asks for function (f_x) values from the expression trees by providing the current variable ($\mathbf{x}$) values, all through the AMPL-Solver Driver.

Finally optimization results are sent back by solvers, which again go through the AMPL-Solver Driver interface, and get converted into the AMPL ".sol" format to be finally interpreted and presented by the AMPL modeling environment.

3.2.2 AMPL-NEOS Architecture

The NEOS Server at Argonne National Laboratory currently provides nearly 70 optimization solvers through some types of networking interfaces, including e-mail, World Wide Web, and socket-based graphical user interfaces. Though the server's location is fixed, optimization solvers can be on any workstation on the Internet that is registered with NEOS through a standard procedure [12].

The Kestrel interface augments the interfaces currently available on NEOS by providing a mechanism that enables remote optimization solution from within the AMPL modeling environment. For detailed description, refer to the paper in [13]. As a result, the locally running AMPL modeling system can have access to a wide variety of the remote NEOS solvers. Users don't notice significant differences between local and remote accesses to solvers. Moreover, optimization results are provided within the AMPL modeling language so that users do not need to parse the text file to use the generated answers. The introduction of the Kestrel interface does not require significant changes to the NEOS server either.

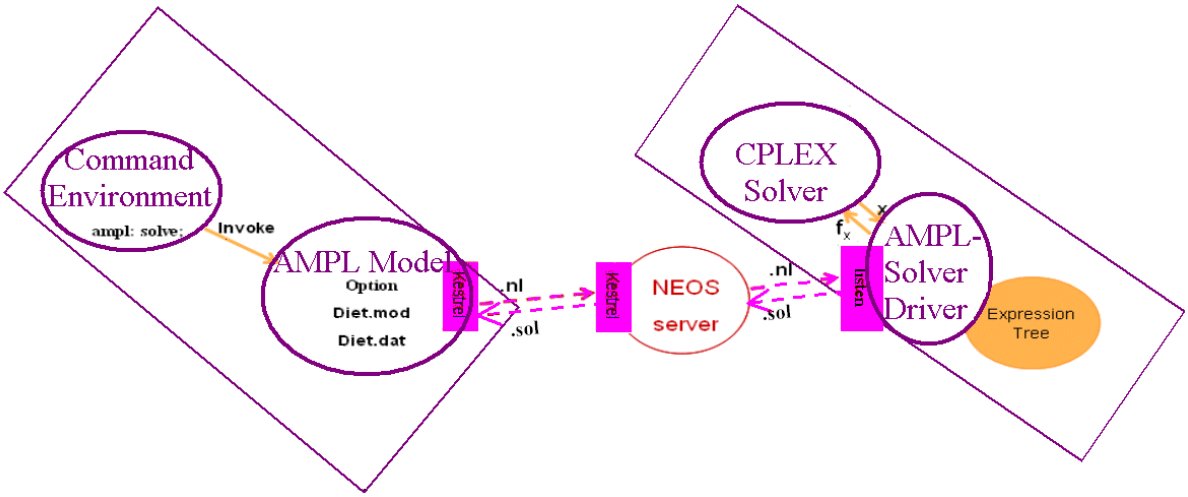

Figure 3-6: AMPL-NEOS Architecture through Kestrel

In terms of architecture, there are no major differences between the standalone AMPL and the AMPL-NEOS system shown in Figure 3-6. They are essentially the same at the two ends of the optimization process, that is, the command environment invocation and the solver-driver interaction. The AMPL-NEOS system adds a Kestrel client and a Kestrel server between the AMPL modeling environment and the NEOS server and connects the two Kestrel interfaces with a CORBA (http://www.corba.org) interconnection. The ".nl" and ".sol" file are transmitted via the Kestrel

interfaces onto the Internet through the NEOS server to and from the registered solver in Figure 3-6 rather than locally on the same operating systems in Figure 3-5. As will be seen in §6.2, the seemingly complex optimization system is just an example of our general design of decentralized distributed optimization architecture.

3.2.3 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. As a matter of fact, neither AMPL nor NEOS precludes any text or binary file format to be passed to a solver. For example, if there are N solvers on NEOS, then N different drivers are required to be implemented by the AMPL developers for total compatibility. There several algebraic modeling languages supported by NEOS. Suppose there are M modeling languages and N solvers, then M × N drivers are required for complete interoperability over NEOS.

Even a cursory look at the NEOS Server's list of solvers (Figure 3-7) reveals the babble of input formats recognized by current optimization software. There are about 10 different low-level formats – ones that describe problems instances – recognized by one or another solver in the NEOS lineup, including MPS [43] formats for linear and integer programming, SMPS [2] extensions to the MPS format for stochastic programming, SIF [10] 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 [3][6] and AMPL [26][27] 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.

In our general and unified design and framework for distributed optimization, we propose a new low-level format (Optimization Services Template Language – OSTL, see §7.1.1) that will be flexible enough to represent a broad variety of the optimization problems currently handled by the NEOS Server. Our presentation will address problems that are not application-specific, but that are as specialized as network linear programs or as generalized as nonlinear-constrained nonlinear 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, moreover; the overall effect will be to decouple language and solver choice, letting the user pick the best tools for any project.

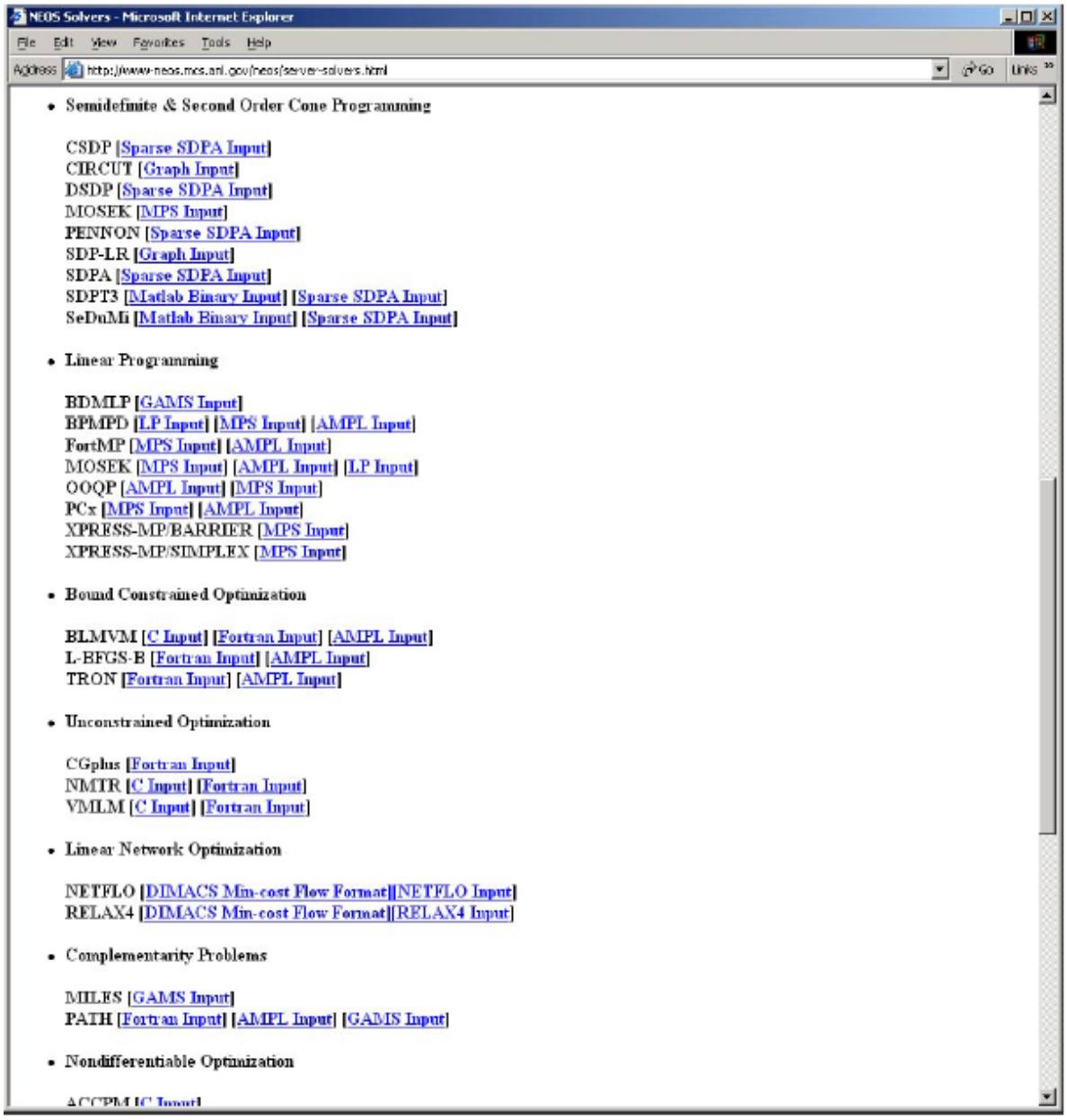

Figure 3-7: Part of the NEOS Server's list of solvers and problems formats

Currently circumstances are particularly favorable for a study of this sort. It is not only that services such as the NEOS Server demand more standardization. New principles and tools, such as XML, described in §4.3.2, have emerged over the past few years to guide the design of standard forms for Internet communication of all kinds. The XML Schema described in §4.3.3, 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 [43] vary in details between implementations, interpreters of the SMPS standard [2] are even more varied, and no proposal for nonlinear extensions (see, for instance [35]) has caught on at all. The proposed optimized service representation consisting of Optimization Services Template Language (OSTL, §7.1.1), Optimization Services Result Language (OSSL, §7.1.2) Optimization Services Option Language (OSOL, §7.1.3), Optimization Services Simulation Language (OSSL, §7.1.4) and Optimization Services Analysis Language (OSAL, §7.1.5), undertake an ambitious project to design a standard representation that addresses all of the problems types supported through the NEOS Server, with sufficient flexibility to be extended to new types. These Optimization Services representation standards can provide diverse higher-level modeling languages with a standard way of reaching solvers.

This work is also complementary to the design of OSI, a standard procedural interface to solvers currently being implemented under the auspices of the COIN-OR project [36]. OSI provides a way of calling optimizers directly from applications, whereas our standard is to be a representation of the content of optimization problem instances, which could be communicated to solvers in a variety of ways. We intend to use COIN-OR to publicize our work on this project, to attract additional collaborators and reviewers, and to distribute the interface library for our XML-based standard.

3.2.4 AMPL-NEOS Optimization Communication Issues

Solving large optimization problems may require computational power far beyond regular desktop workstations can offer. Due to increasing performance of computing and networking power, typical users now have access to more resources than ever before. When the 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 [13] (see also http://www-neos.mcs.anl.gov/neos/kestrel.html). Nowadays, a user can typically submit an optimization problem to NEOS via any of the above-mentioned interfaces. NEOS Server then locates the specified solver in its data bank and schedules the user's entire data on a remote computation resource that is currently available and equipped to process jobs of the given type. Registered solver providers must provide both software and hardware. Solver administrators have to write implementations to check data consistency, solve optimization and return appropriate results. The NEOS Communication Package – a Perl application, is provided to facilitate communications between NEOS Server and solver computers.

Still, the current NEOS Server only begins to address the communication difficulties of large-scale optimization with respect to the combinatorial effect of the plethora of solver types, interface choices, scheduling, benchmarking, and connection to modeling languages and services that calculate function values. The Server has evolved along with the Web and the Internet, moreover, it is limited to some degrees by early design decisions and showing a so-called “second-system effect.”

We are now seeing a new generation of standards that are designed to make Web Services (see §4.3.5) more flexible in design and easier to build and maintain. With tools like XML (see §4.3.2), SOAP (see §4.3.5), WSDL (see §4.3.6), WSIL (see §4.3.7), UDDI (see §4.3.8) and OGSA (see §4.3.9), we can think about a more general and flexible Optimization Services environment for developers and researchers to make their models, solvers and simulations available and easily interact with each other on the Internet.

4 SETTINGS FOR THE DISTRIBUTED OPTIMIZATION DESIGN AND FRAMEWORK

4.1 A General Picture – The Future of Computing

Figure 4-1 shows a likely future of computing where semantic Web Services and software agents interact with each other. A user, or maybe more appropriately a "consumer" plugs his computer into a so-called "computing socket" or may be 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. The corresponding utility or power company is the consumer's application service provider that rents computing power and resources and charges with 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/or with Universal Description, Discovery and Integration (UDDI, §4.3.8) that are either general registries which keep information of all kinds of Web Services or specialized registries like the NEOS Registry (see §6.4) that only serves Optimization Services (see §5) Facilities like Condor [18][40] will also help in finding computers to provide idle computing power.

Admittedly most of these tasks could be achieved by an arrangement of manual labor and customized software tools using existing technologies, although it 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 that are used to achieve the tasks described in the above scenario, some mature enough to be commercialized, whereas others still in research phase:

- Peer to Peer (P2P) [44]
- Software Agents [1][19]
- Ontologies and the Semantic Web [8]
- Grid Computing [21]
- Embedded Web Services [7]

Although the argument is true that many of the technologies already existed, 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, as a first non-standard step, the Google search engine for web *pages* [5], with its automated web crawlers and state-of-the-art file storage design with inverse indexing technologies.

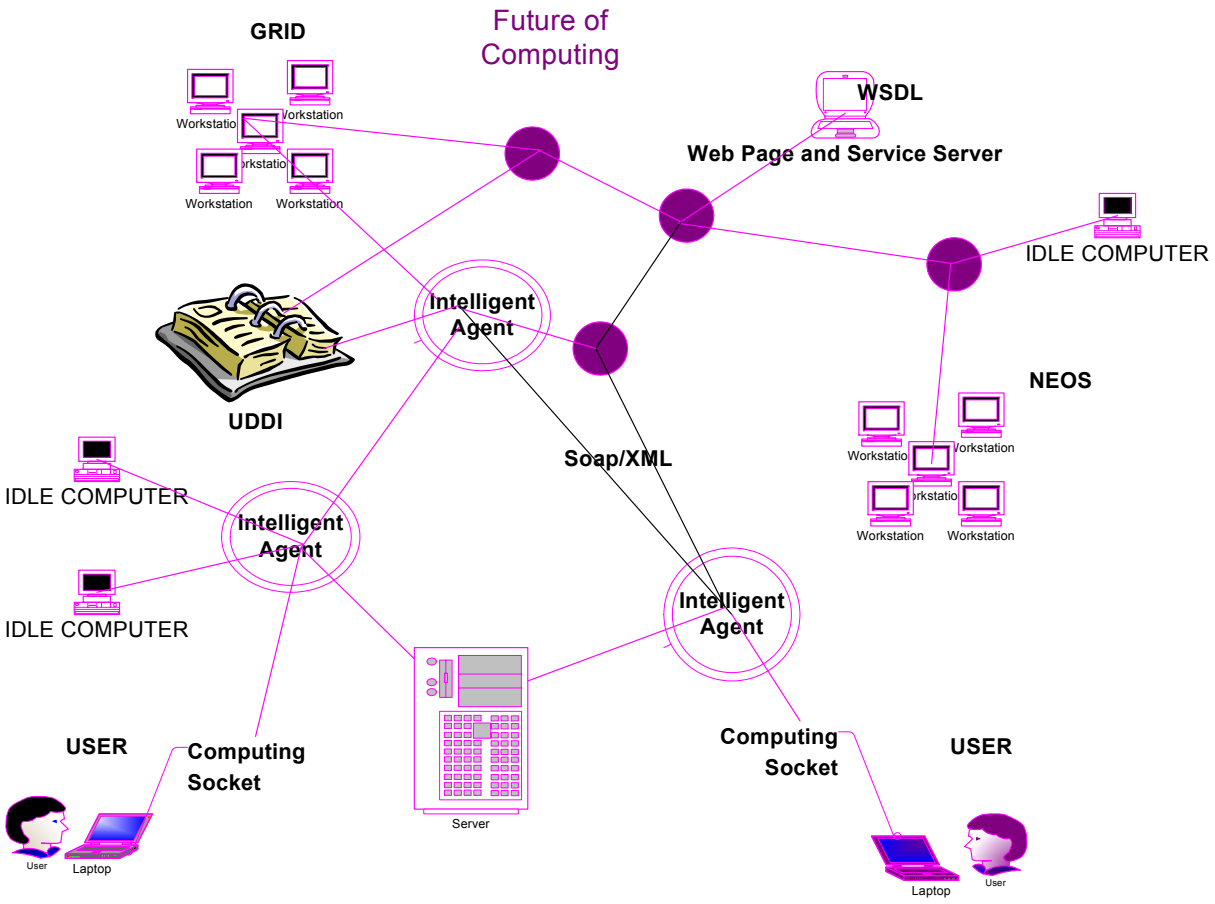

Figure 4-1: A general picture – The Future of Computing

4.2 Our Positioning – The Hierarchy of Operations Research (OR)

Ideally, researchers in all areas that involve computing or have a need for computing, should have a similar vision as illustrated in §4.1, thus from now on working toward or helping to achieve the same goal ultimately. Researchers in the area of operations research especially fit in this category as illustrated in Figure 4-2. Operations research, viewed by many as a branch of applied mathematics, naturally lies on the foundations of mathematics and computing theory, on which basic tools like statistics, optimization and simulation are built. We apply these tools to model many of the industrial engineering and management sciences areas that concern design, analysis and implementation of any system in order to improve quality and productivity. The areas can be in any sector of the economy – manufacturing, distribution, finance, marketing etc.

The highest level in the hierarchy, which concerns modeling, is the part that mostly interfaces with regular consumers who use models for daily analysis. Our project's positioning is in the middle of the Operations Research hierarchy, which is concerned with things like communication infrastructures, modeling languages and systems. It is an interface part that bridges OR modeling with the basic OR tools. When implemented smoothly, it is the part that does not need to be known or noticed by modelers or "consumers" in a daily sense. When planned generally, it is the part that can fit in the general picture of the future of computing (see §4.1), thus contributing, as well as itself benefiting from, the largest possible synergy generated within the computing world. When designed simply enough (without sacrifice of power), it is the part that can be quickly adopted by both the modelers and the tool builders, thus facilitating a healthier environment for OR development as a whole. Our general and unified design and framework for distributed optimization takes account of these goals.

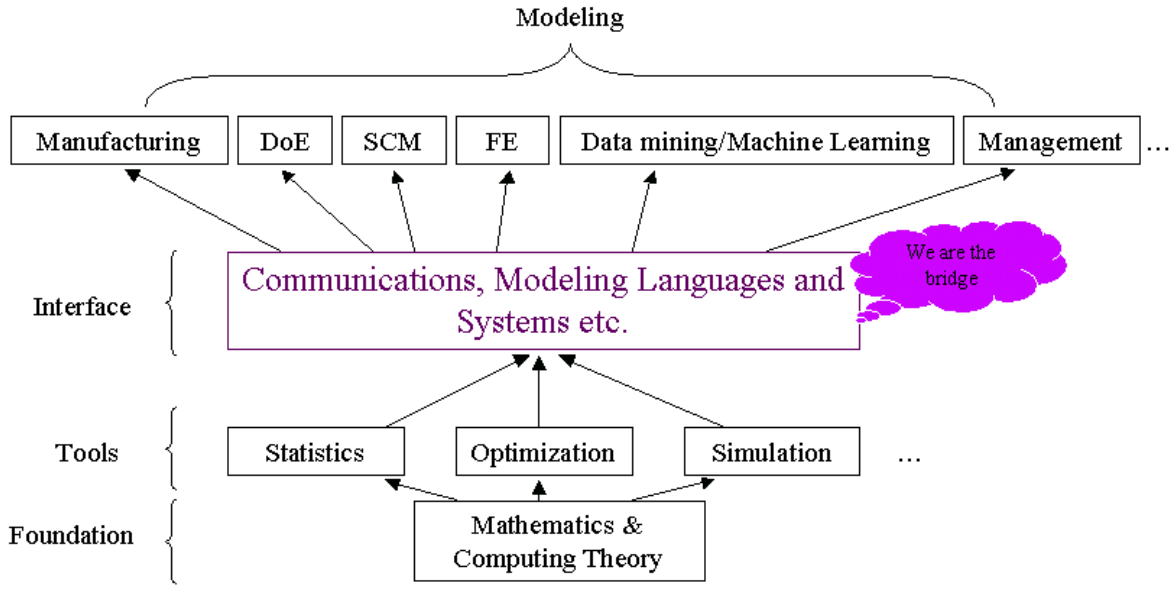

Figure 4-2: A rough sketch of operations research and this proposal's positioning within this hierarchy

4.3 Technologies, Terminologies, Current States of Optimization Services Related Research

This proposal uses some knowledge that does not necessarily pertain to the field Operations Research. This necessitates a section devoted to a general introduction to related research, and clarifications of certain concepts and terminologies that can sometimes cause confusion.

4.3.1 Parallel/Distributed/Grid Computing

There are many definitions attempted to make distinctions between the three. In short, parallel computing is about a process or an algorithm to parallelize a program; distributed computing is more about building a computing architecture; whereas grid computing is to provide the underneath environment or mechanism to facilitate parallel and distributed computing. Below are three definitions that I think should be clear enough to set the differences:

- Parallel Computing

"Process by which a problem is solved using multiple resources working concurrently and collaboratively." [Class Notes on Parallel Computing, ECE Department, Northwestern University, P. Banerjee]

- Distributed Computing

"Computing on networked computers which is deeply concerned with problems such as reliability, security, and heterogeneity that are generally regarded as tangential in parallel computing." [Designing and Building Parallel Programs [20], I. Foster]

- Grid Computing

"An ambitious and exciting global effort to develop an environment in which individual users can access computers, databases and experimental facilities simply and transparently, without having to consider where those facilities are located." [RealityGrid, Engineering & Physical Sciences Research Council, UK 2001]

Our project more fits in the sphere of distributed computing, which is "deeply concerned with" "heterogeneity". It does, however, also leverage on the "environment" provided by Grid Computing, as well as Web Services, which we will talk about more in the following sections.

4.3.2 XML

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 [47]. For a complete reference, see [56]. SGML is a standard for how to specify a document markup language or tag set. HTML is another example of SGML.

Forms based on XML, in particular, are being used for a wide variety of purposes, and we propose to investigate their application for communicating instances of optimization problems. An XML representation consists of data delimited by **<tags>**, much like an html representation of the content of a web page. New collections of XML tags can be defined for any specialized purpose, however, by specifying a schema (see §4.3.3). One perceived disadvantage of 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.

An example of XML is given in Figure 4-3, expressed in MathML [46][58], a dialect of XML that is of particular interest in this paper. A dialect is basically 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 so forth. We will use MathML for simple nonlinear function representation in optimization problems. We will also introduce many Optimization Services (OS, §5) related dialects in the later sections, including most notably, Optimization Services Template Language (OSTL, §7.1.1), Optimization Services Result Languages (OSRL, §7.1.2) and Optimization Services Option Language (OSOL, §7.1.3).

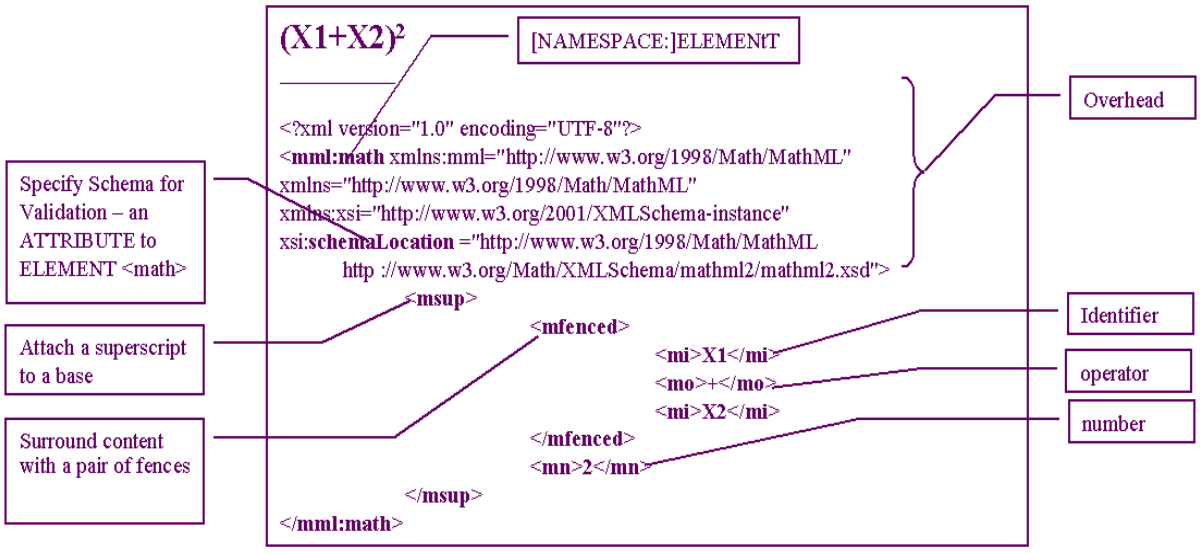

Figure 4-3: Expression $(X_1 + X_2)^2$ in MathML – a dialect of XML

The example shown in Figure 4-3 expresses $(X_1 + X_2)^2$ in XML. The root element is <**math**>, which is ended with a corresponding **</math>** element**,** as should any element in an XML document.

<math> has an XML namespace tagged in the front and separated with a "**:**". Namespaces (see §4.3.4) are used to qualify the elements and avoid potential naming conflicts. Any element may also have some attributes. In the case of the **<math>** element, it has some **xmlns** attributes to declare namespace abbreviations. Between **$** and **$** can be contained other elements and in this example just one, namely the **<msup>** element. Under **<msup>** are again contained two elements: **<mfenced>** to contain the base expression $X_1 + X_2$, and **<mn>** to contain the exponent number **2.**

4.3.3 XML Schema

XML Schema is a database-inspired method for specifying constraints on XML documents, itself using an XML-based language. There are other popular XML specification methods, including DTD, standing for Document Type Definition. The reasons we do not choose to use DTD are:

1. It is not as expressive as XML Schema.
2. It is not expressed in XML.
3. It is not a WC3 recommendation.
4. Most importantly it is not supported in SOAP (see §4.3.5), which our Optimization Services (OS, §5) heavily leverages on.

For a complete reference on XML Schema, see [62]. 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 [53][66]. For each Optimization Services instance language that we introduce, we will specify representation rules in XML Schema.

```
<xs:complexType name="msup.type">
        <xs:group ref="Presentation-expr.class" minOccurs="2" maxOccurs="2" />
        <xs:attributeGroup ref="msup.attlist" />
</xs:complexType>

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

Figure 4-4: MathML Schema specifying constraints on tag <msup>

Figure 4-4 shows a section of the MathML Schema, specifying constraints on the **<msup>** tag. Basically it is saying that the element **<msup>** has to follow a predefined "**msup.type**", and any **"msup.type"** should contain exactly 2 elements, one indicating a base, while the other indicating a superscript. Both the base and superscript elements have to be a **group** defined in the **Presentation-**

expr.class, which is not shown here. In our MathML example in Figure 4-3, the **group** is **<mfenced>…</mfenced>** for the base and **<mn>…</mn>** for the superscript. Any element can have attributes. In our MathML example, element **<msup>** does not have any attributes. But if it does, it can take any attributes specified in the **attributeGroup** of **msup.aatlist.**

4.3.4 Other XML Technologies

In this section, we give a list of other XML technologies used in this project and their corresponding references.

- **XML Authoring** tools assist in editing XML documents or validating XML syntaxes. XML documents can be XML Schemas as well as regular XML dialects.
- **XML Transformation** tools assist in transforming XML into something that can be displayed in a browser or other rendering device. **XSL** [63], and its associated language **XSLT** [64], is the main tool here. XSLT stands for Extensible Stylesheet Language Transformation, is itself an XML based declarative (as versus imperative languages such as C/C++) programming language to transform XML files into other HTML files, or XML files or any other plain text files. Figure 4-5 shows how the combination of XML and XSLT can serve as at least the same purpose as HTML. XSLT can be used for example to display optimization results formatted in Optimization Services Result Language (OSRL, §7.1.2).
- **XML Parsing Models** include mainly **Document Object Model (DOM)** [54] and **Simple API for XML (SAX)** [42]. Both are language APIs that can be used to translate XML documents to some format suitable for use by computer programs. DOM is a set of traversal interfaces that can decompose the XML documents into a hierarchal tree of generic objects or nodes. SAX is a set of streaming interfaces that can decompose the XML documents into a sequence of predefined method calls. To construct an XML document, DOM has to be used. To parse an XML document, both DOM and SAX can be used, though SAX is less memory intensive. DOM is mainly used by algebraic modeling systems like AMPL to construct low level optimization problem instances and by solvers to construct low level optimization results. SAX is mainly used by solvers to parse low level optimization problem instances and by algebraic modeling systems to parse low level optimization results.
- **XPath** [59] is a declarative language used to identify subsets (nodes and fragments) of an XML document. It is used in XSLT (for pattern matching), XPointer (for addressing), XQuery (for selection and iteration) and XML Schema (for uniqueness and scope description).
- **XLink** [57] and **XPointer** [60] are used to link and reference information within an XML. XLink is a generalization of the HTML link concept, though it is more 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 linkbases. XPointer is sort of an extension to XPath

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 when some of the function evaluations in optimization problems can only be obtained from a remote Web Service.

- **XQuery** [61] is a query language for retrieving data items from an XML document. XQuery is to XML what SQL is to relational databases. As of December 2003, it is still in progress under the auspices of the W3C's XML Query working group. It may turn out to be useful in designing our Optimization Services Query Language (OSQL, §7.3.4).

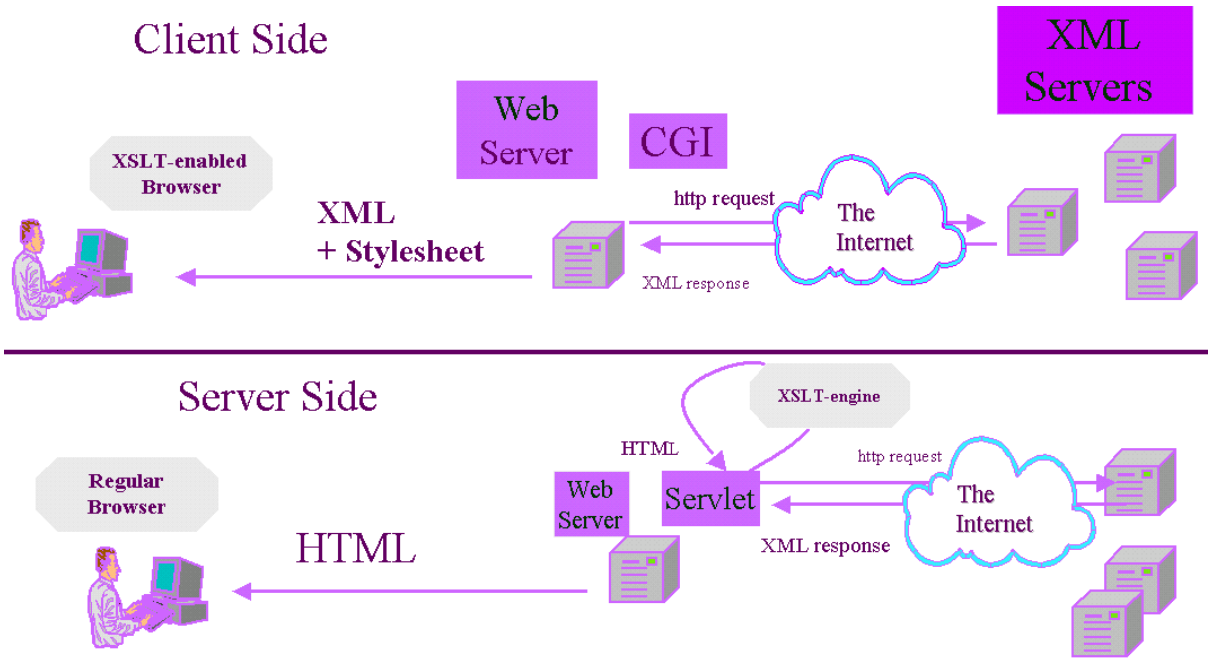

Figure 4-5: An illustration of how the combination of XML and XSLT Stylesheet can serve as the same purpose of HTML

- **XML Namespace** [55] provides a simple method for qualifying element and attribute names used XML documents by associating them with namespaces identified by URI references. It is mainly used to avoid naming potential conflict of XML tags. Important namespaces that need to be standardized include "**OSCL**" and **"OSDL"** for qualifying the element **<definition>** in our Optimization Services Client Language (OSCL §7.2.1) and Optimization Services Definition Language (OSDL §7.2.2) and **"OSIL"** for qualifying the element **<inspection>** in our Optimization Services Inspection Language (OSIL §7.3.1).

4.3.5 Web Services and Simple Object Access Protocol (SOAP)

W3C's official definition of Web Services [65] is as follows as of August 2003:

> "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 can be **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 can not tell what language, operating system, or computer type was used. It is achieved through the Simple Object Access Protocol (SOAP, see this section below).

"**Described**" means that a Web Service must describe itself, mainly what requests can be made, what the arguments are and what transport it uses. It is achieved through the protocol of Web Services Description Language (WSDL, §4.3.6).

"**Published**" means that a Web Service must tell a registry service where it is located (like "yellow pages"). It is achieved through the protocol of Web Services Inspection Language (WSIL, §4.3.7) and Universal Description, Discovery and Integration (UDDI §4.3.8).

"**Discovered**" means that a potential client can find it in a registry service. It is also achieved through the protocol of WSIL and UDDI.

"**Invoked**" means that the arguments and return types are known. It 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.

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 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 favor 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. Figure 4-6 gives

an illustration from architecture view, protocol view, SOAP envelope structure view and 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. SOAP message is then transmitted via 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. Response is returned in a same way.

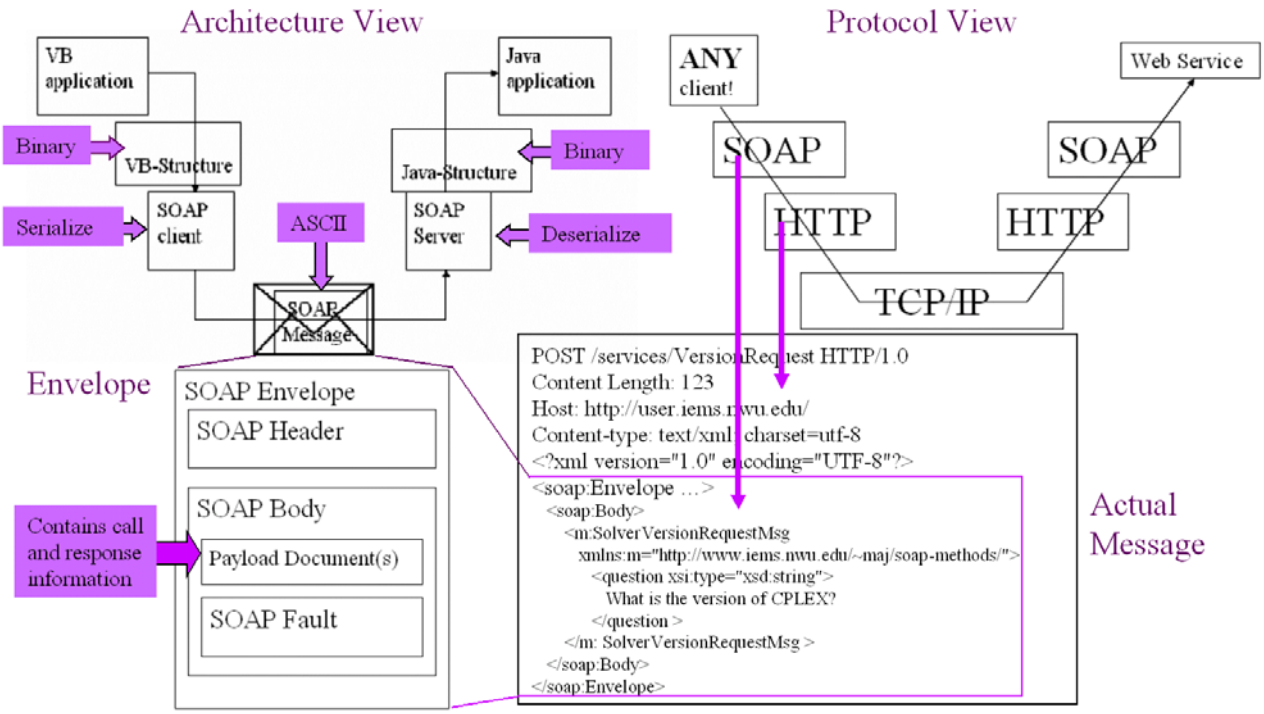

Figure 4-6: SOAP illustration from high to low level

In the network view protocol, all the information needed for the client call is stored in a SOAP envelope. 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.

A SOAP envelope contains two sections: SOAP Header and SOAP Body. SOAP Header mainly has some administrative information to complete a call. SOAP body contains the major request and response information, for example call methods and arguments. SOAP Body also contains a subsection of SOAP Fault, which contains exception error returned by the called Web Service. As shown in the actual message part of Figure 4-6, 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 and may start a standard for newly developed network protocols.
All our Optimization Services networking mechanism is based on SOAP.

4.3.6 Web Services Description Language (WSDL)

Web Services Description Language (WSDL) [65] is another XML document type that defines the XML tags to be used in accessing a Web Service. But, for example, in case where a user knows exactly where an Optimization Service is and how the Optimization Service should be invoked, WSDL is optional. WSDL helps significantly in registering, discovering and automation of Web Services. Links to WSDL descriptions can be given through Universal Discovery and Integration (UDDI §4.3.8) listings.

Two types of information in WSDL are specified. One is that about interface semantics and the other 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 Optimization Services Description Language (§7.2.2), we will enforce a standard on call interface and arguments, fix certain values by default and suggest recommendations that are most suitable for Optimization Services, thus simplifying the entire mechanism.

Figure 4-7 shows an abbreviated WSDL definition. Illustrated elements about method, interface, protocol and address are of most relevance to our design of an Optimization Services framework. The entire program, called "**SimpleSolver**" in this example contains (in **<portType**>) only one operation (or function, method, 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>** element specifies that the SOAP call is to be a remote procedure call (**rpc**, a request and response model) and is to be transported over HTTP. The **<Service>** element specifies an address (in **<port>**) which tells where the actual Web Service is.

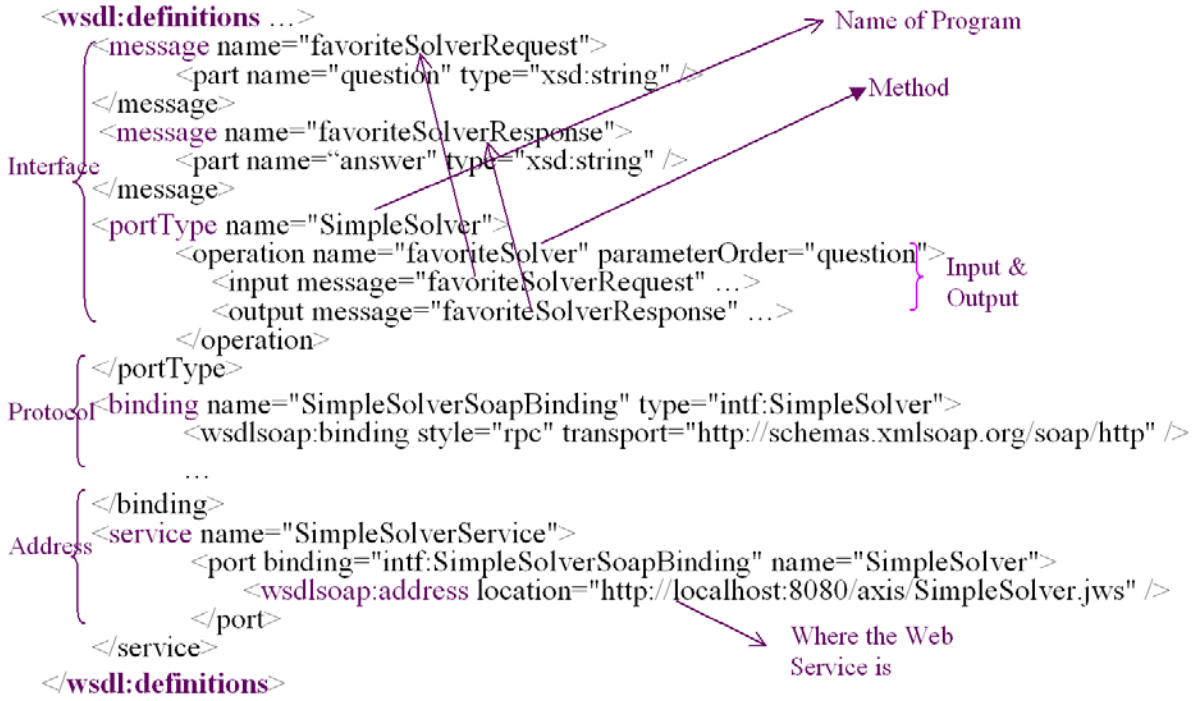

Figure 4-7: An abbreviated WSDL document

4.3.7 Web Services Inspection Language (WSIL)

After a Web Service is deployed, potential users must have a way to find and use that service For web pages/sites, search engines like Google and Yahoo do this function, though search information is of non-standard form. Web Services Inspection Language (WSIL), as well as Universal Description, Discovery, and Inspection (UDDI) in the next section handle the situations for Web Services.

UDDI is a specification for an online registry of Web Services. WSIL is similar in scope to UDDI, but intended to be complementary rather than competitive. 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, it 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. 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. Figure 4-8 shows an abbreviated example of a WSIL document. Most information is self-explanatory in this example. 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. Our Optimization Services Inspection Language (OSIL §7.3.1) is essentially a WSIL document.

```
<inspection …>
    <abstract>Solver Web Services</abstract>
    <service>
            <name>Solver Service</name>
            <abstract>A solver that provides solver info.</abstract>
            <description
                    referencedNamespace=http://schemas.xmlsoap.org/wsdl/
                    location="http://localhost:8080/axis/SimpleSolver?wsdl">
            </description>
    </service>
    <link location="http://localhost:8080/axis /othersolvers.wsil" >
            <abstract>Other Solver Services</abstract>
    </link>
</inspection>
```

Figure 4-8: An abbreviated WSIL document

4.3.8 Universal Description, Discovery and Integration (UDDI)

Universal Description, Discovery and Integration (UDDI) [52] is a specification for an online registry of Web Services. Providers can list their services in this registry, and users can seek out services by searching the registry in a standard way.

Compared with WSIL, it is more heavyweight, and is intended to be maintained by centralized registries. Unlike WSIL, it also concerns itself with business entity information. 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. UDDI usually requires infrastructure to be deployed with substantial overhead and costs. Two main parts of functions are provided. Vendors *register* data via SOAP. Users *discover* the services via SOAP query requests. NEOS or other designated Optimization Services will, in the long run, evolve into a registry based on the UDDI model containing many OSIL documents.

4.3.9 Open Grid Services Architecture (OGSA)

The Globus Alliance [22] 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 & 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) [23] 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 first 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 is currently 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 (QoS), 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 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, many 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.

5 A GENERAL AND UNIFIED DESIGN AND FRAMEWORK FOR DISTRIBUTED OPTIMIZATION (PART I – PROPOSING OPTIMIZATION SERVICES)

Optimization Services (temporary definition, abbreviated as OS) are SOAP based Web Services (potentially also leveraging on grid computing technologies) with specified interfaces and behaviors under the general framework of distributed optimization, including the following OSXL's:

for representing optimization instances

- Optimization Services Template Language (OSTL, §7.1.1): used to construct optimization problems
- Optimization Services Result Language (OSRL, §7.1.2): used to construct optimization results returned from solvers
- Optimization Services Option Language (OSOL, §7.1.3): used to construct simulation inputs and outputs for function evaluations
- Optimization Services Simulation Language (OSSL, §7.1.4): used to set solver options
- Optimization Services Analysis Language (OSAL, §7.1.5): used to provide meta-knowledge of optimization problems though optimization analyzers

for controlling optimization accesses, flows and operations

- Optimization Services Client Language (OSCL, §7.2.1): used for solvers to call simulation services
- Optimization Services Description Language (OSDL, §7.2.2): used for modelers to invoke solvers
- Optimization Services Flow Language (OSFL, §7.2.3): used to coordinate Optimization Services components
- Optimization Services Endpoint Language (OSEL, §7.2.4): used to manage non-functional characteristics of Optimization Services

and for discovering and inspecting Optimization Services

- Optimization Services Inspection Language (OSIL, §7.3.1): used to describe any Optimization Services components, but mainly solvers
- Optimization Services Process Language (OSPL, §7.3.2): used to describe run time information of solvers
- Optimization Services Benchmark Language (OSBL, §7.3.3): used for an authoritative benchmarker to evaluate existing solvers
- Optimization Services Query Language (OSQL, §7.3.4) :used to construct queries to discover Optimization Services ■

.

Such an arrangement in the Optimization Services definition has the potential to substantially *decentralize* the registry of solver characteristics currently maintained by the NEOS Server at Argonne National Laboratory. The remaining work of the centralized NEOS Server would be focused on activities not specific to individual solvers, such as analyzing problems and recommending solvers and on providing multi-solver services such as benchmarking and translation (as with the current GAMS-to-AMPL modeling language translator).

This vision of a next-generation NEOS Server leaves open the question of how optimization "jobs" will be scheduled to run on available workstations. The current centralized scheme maintains one queue for each solver/format combination, along with a list of the workstations on which each solver can run. We will want to maintain this scheduling control, while at the same time making the scheduling decisions more distributed. We will also investigate extending the power of the NEOS scheduling schemes to take advantage of Grid Computing [21], both in making use of idle use of computing power (as provided, for instance, by Condor [18][40]) and in supporting the use of multi-processor optimization methods. In the case of the latter our work has especially great potential to stimulate new applications, by saving potential users the considerable difficulty of setting up the required hardware and networking software. Figure 5-1 shows a tree view of Optimization Services.

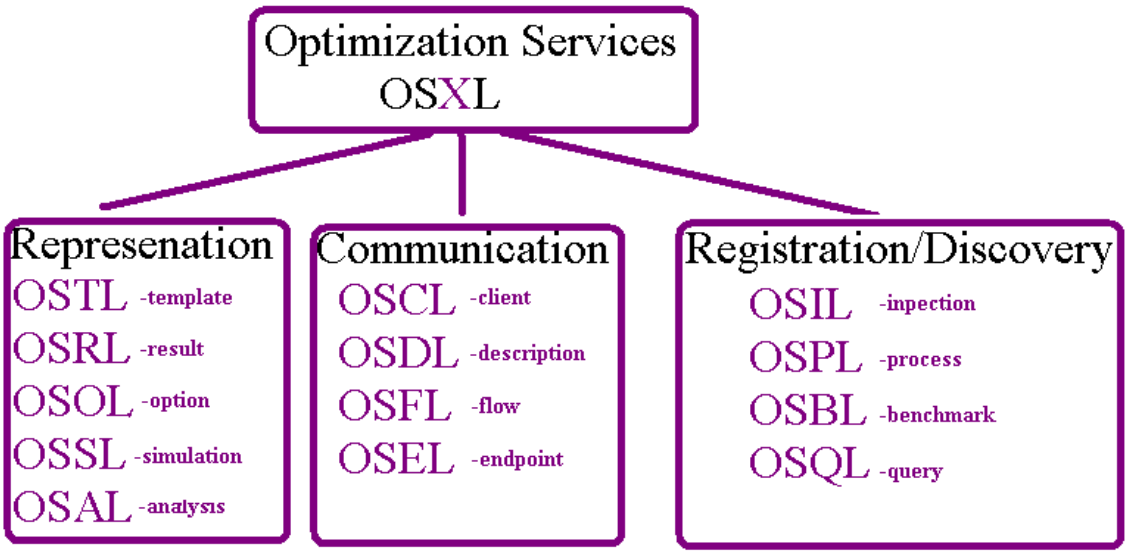

Figure 5-1: A tree view of Optimization Services

6 A GENERAL AND UNIFIED DESIGN AND FRAMEWORK FOR DISTRIBUTED OPTIMIZATION (PART II – ARCHITECTURE DESIGNS)

From our experiences in designing and developing optimization systems, we realized that in general most optimization systems can be decomposed into five distinct optimization components – **Client**, **Model**, **Solver**, **Simulation** and **Server/Registry**, whether they are distributed on a network or "distributed" on the same operating system. The latter can be regarded as a special case in the general distributed architecture. There are, however, two types of general design – the centralized version and the decentralized version, as we will respectively investigate in §6.1, and §6.2. In the centralized version, the central component is a server, whereas in the decentralized version, the central component is more of a registry. The decentralized architecture is envisioned as the trend of the future, while the centralized architecture is more suitable in a corporate environment, in which companies want to take control through this central server. We will revisit Motorola's VP Intelligent Optimization System and Argonne's AMPL-NEOS System under the two general architectures in §6.1, and §6.2. The decentralized design serves as the basis for deciding the necessary pieces for our general and unified framework for distributed optimization introduced in the following sections. The main guiding principles for our design and framework are:

- When implemented smoothly, it does not need to be known or noticed by modelers.
- When planned generally, it fits in the general picture of the future of computing (see §4.1), thus contributing, as well as itself benefiting from, the largest possible synergy generated within the computing world.
- When designed simply enough (without sacrifice of power), it can be quickly adopted by both the model builders and the algorithmic tool builders, thus facilitating a healthier environment for operations research development as a whole.

6.1 The Centralized Architecture

Figure 6-1 shows the five components in our general design of centralized distributed optimization architecture and their interactions. Dotted arrows indicate data flow. Circles indicate components: optimization Client, optimization Model, optimization Solver, Simulation for optimization and in the center optimization Server (in this example NEOS).

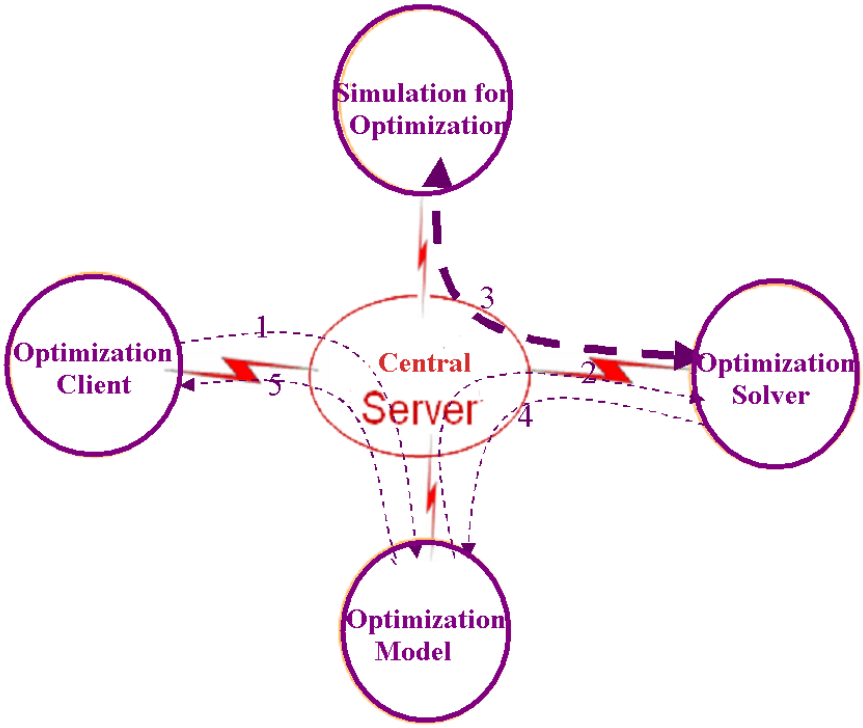

Figure 6-1: General design of centralized distributed optimization architecture

Data Flow (All Through the Central Server)

Numbers below correspond numbers in Figure 6-1.

1. Client invokes optimization Model.
2. Model establishes an optimization session with Solver. It can first set solver options and then invoke optimization.
3. Solver, as a client, asks Simulation for function values by providing current variable and parameter values. This is potentially a highly iterative process, thus the data flow arrow in **bold**.
4. Solver sends back optimization results.
5. Model forwards back optimization results to Client.

Comments

Communication and representation specifications introduced later in the general and unified framework can be used as references in this centralized design, but do not have to be enforced. Optimization Client and Model are usually together, that is, a client locally constructs a model and sends the model instance to a remote solver. Between optimization Model and Solver, session should be maintained. This is because there is typically a sequence of calls between the two. Calls made previously (e.g. setting solver options) may affect calls that follow (e.g. solving an optimization problem). Simulation can be thought of as a set of function value calculators, be them objective function or constraint function calculators. Simulation can return more than one value in its result. No distinctions are set between deterministic and stochastic simulations. For example, both values of the expected mean and variance of the mean can be returned as metrics. Simulation may be provided within the Model sent to the Solver**,** like an expression tree that is hooked to the AMPL-Solver Driver situated locally with the Solver.

6.2 The Decentralized Architecture

Figure 6-2 shows the five components in our general design of decentralized distributed optimization architecture and their interactions. All the components remain the same except that the central Server is replaced by a central Registry. This is the architecture we envision for the future. It serves as the basis for the design and analysis of our general and unified framework for distributed optimization.

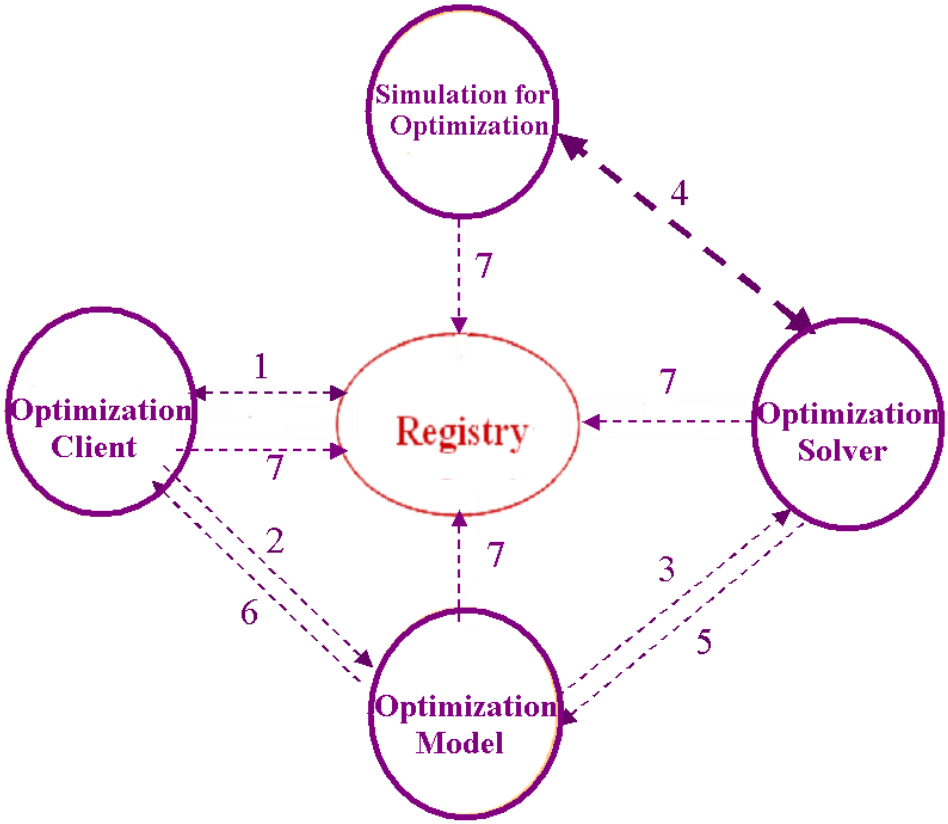

Figure 6-2: General design of decentralized distributed optimization architecture

Data Flow (All Peer to Peer)

Numbers below correspond numbers in Figure 6-2.

1. Client discovers Optimization Services through NEOS Registry, through the protocols of Optimization Services Inspection Language (OSIL, §7.3.1). Query can be constructed in the format of Optimization Services Query Language (OSQL, §7.3.4). Run time information may be obtained from Optimization Services Process Language (OSPL, §7.3.2).

2. Client invokes optimization Model. This invocation can be in any form or it can take references from the protocol of Optimization Services Client Language (OSCL §7.2.1) and provide input similar to the Optimization Services Simulation Language (OSSL §7.1.4).
3. Model establishes an optimization session with Solver, through the protocol of Optimization Services Description Language (OSDL §7.2.2). It can first set solver options following the format specified in the Optimization Services Option Language (OSOL, §7.1.3) and then invoke optimization by providing an optimization problem instance following the format specified in the Optimization Services Template Language (OSTL, §7.1.1).
4. Solver, as a client, asks Simulation for function values by providing current variable and parameter values, through the protocol of Optimization Services Client Language (OSCL §7.2.1). This is potentially a highly iterative process, thus the data flow arrow in **bold**. Both input (variables and parameters) and output (function values or metrics) are to follow the format specified in the Optimization Services Simulation Language (OSSL, §7.1.4).
5. Solver sends back to Model the optimization results following the format specified in the Optimization Services Result Language (OSRL, §7.1.2).
6. Model returns optimization results to Client depending on the nature of the initial client call. If the client call is based on a "request and response" model, then the optimization results is sent back though the "response" part of the model. Like in 2, no restriction is specified on the return mechanism.
7. Different components can send individual information to registry through some feedback mechanism. For example Client can register an optimization Solver, through the protocol of Optimization Services Inspection Language (OSIL, §7.3.1). Queries can be constructed in the format specified in the Optimization Services Query Language (OSQL, §7.3.4). Solver can report its current status through the protocol of Optimization Services Process Language (OSPL, §7.3.2). {This part needs to be further investigated.}

Comments

All components are not controlled by NEOS Registry. Optimization Client and Model are usually on the same machine. Whether there should be an optimization specific protocol governing the communication between Client and Model is not or may never be considered. Between optimization Model and Solver, an active session should be maintained. Session maintenance and other generic resource management functionalities that are not optimization specific should be leveraged upon either Web Services or Grid Services protocols. For example a "stop" call, intended to end an optimization session, should be handled by the notification functionality provided in the Grid Services protocol. Simulation can be thought of as a set of function value calculators, whether they are objective function or constraint function calculators.

Simulation can return more than one value in its result. No distinctions are set between deterministic and stochastic simulations. For example, both values of expected mean and variance of the mean can be sent back from a stochastic simulation in the output section of Optimization Services Simulation Language (OSSL, §7.1.4). Simulation may be provided within the Model sent to the Solver**,** like an expression tree that is hooked to the AMPL-Solver Driver situated locally with the Solver. The exact mechanism of invoking Simulation, is specified in the Optimization Services Template Language (OSTL, §7.1.1).

6.3 Motorola VP Optimization System Revisited (Centralized Architecture)

Figure 6-3 shows how the Motorola VP Intelligent Optimization System fits in the general design of the centralized distributed optimization architecture.

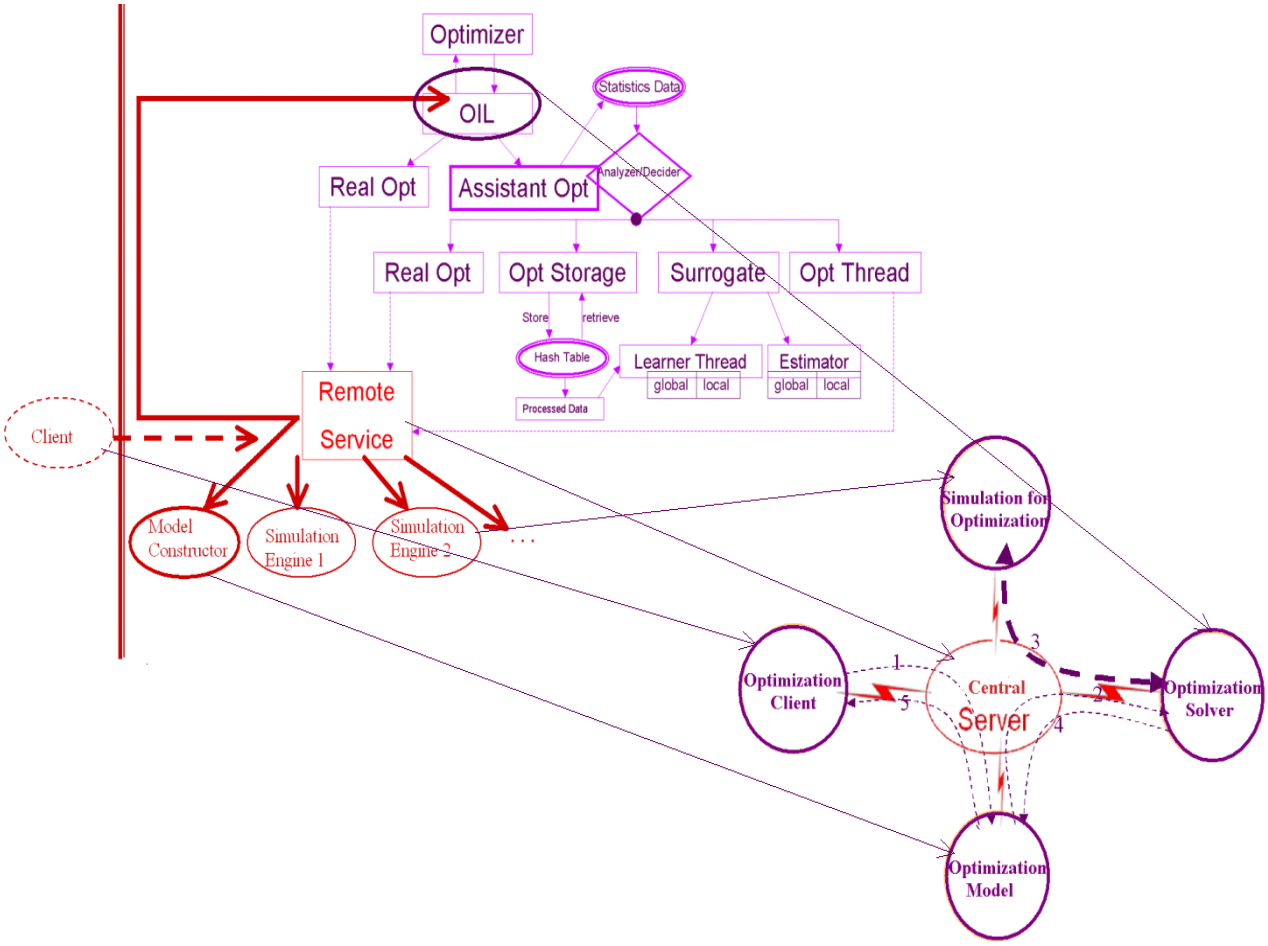

Figure 6-3: Motorola VP Optimization system mapped under the centralized architecture

1. The remote central server maps to Server. All the information has to go through the central server to control logins and keep track of usage statistics.
2. The client maps to Client. All clients are within the Motorola Intranet.
3. The model constructor maps to Model. In the Motorola's VP optimization system, the client is separate from the model constructor. Client provides necessary information to the model constructor and the model constructor creates a model during run time.

4. The solver interface maps to Solver. All the other auxiliary pieces in facilitating "intelligence" are behind the solver interface. They are system specific.
5. All the simulation engines maps to Simulation. They are not Web Services, but the VP optimization system has its own proprietary standard in coordinating these simulations on the network and parsing inputs and outputs.

Data flow follows exactly the process described in §6.1.

6.4 AMPL-NEOS Revisited (Decentralized Architecture)

Figure 6-1 shows how the AMPL-NEOS System fits in the general design of the decentralized distributed optimization architecture.

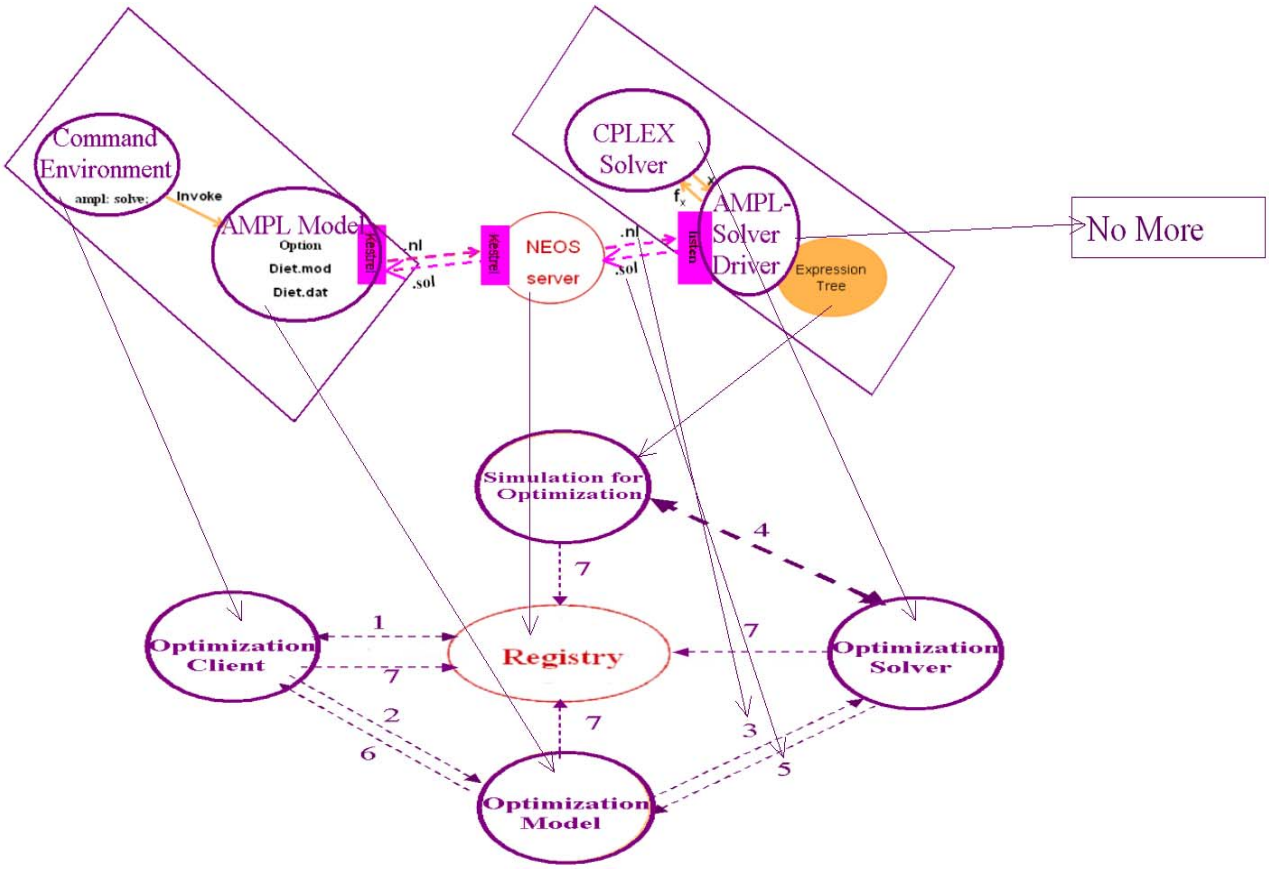

Figure 6-1: AMPL-NEOS system mapped under the decentralized architecture

1. The NEOS Server will become the NEOS Registry, or will be replaced by other Optimization Service Registries. It contains records of Optimization Services Inspection Language (OSIL, §7.3.1) documents. It provides discovery and registration mechanisms.
2. The AMPL command environment maps to Client. Client can be any user on the Internet, be it a human user or a piece of modeling software.

3. The AMPL model maps to Model. In the AMPL-NEOS System, Client is together with Model in the same AMPL modeling environment. Client constructs a model and AMPL converts it into an optimization instance and sends the instance to a remote solver.
4. The solver maps to Solver.
5. The AMPL-constructed expression trees maps to Simulation. In this case, Simulation is located on the same machine with Solver, connected with the AMPL-Solver Driver interface. AMPL assumes that expressions have explicit functional forms. AMPL may need to extend its syntax functionalities to allow, for example, Simulation Web Services that do not have closed forms.

Data flow follows exactly the process described in §6.2. One significant achievement of the general decentralized design is that it gets rid of the AMPL Solver Driver. There is no longer a need for such a one-to-one modeling language-solver interface, because all the components talk in standardized languages. The AMPL ".nl" file will be replaced by Optimization Services Template Language (OSTL, §7.1.1) and the ".sol" file will be replaced by Optimization Services Result Language (OSRL, §7.1.2). The Kestrel interface may still exist, but its communication with remote objects needs to follow the Optimization Services protocol, rather than CORBA.

7 A GENERAL AND UNIFIED DESIGN AND FRAMEWORK FOR DISTRIBUTED OPTIMIZATION (PART III – OPTIMIZATION SERVICES FRAMEWORK)

The Optimization Services framework is based on the general design of decentralized distributed optimization architecture discussed in §6.2. It addresses issues in communications between pairs of the five components in the decentralized architecture. Major aspects include *representing* with new XML standard forms for optimization problem instances, *scheduling* with new Optimization Services standards and their use in distributed, intelligent assignment of optimization requests to resources, *categorizing* with standard procedures for guiding prospective users in their choice of solvers, and incorporation of *analyzing and benchmarking* information from a suite of optimization related supporting tools into the general framework. For the sake of standardization and uniformity, the whole framework is intentionally specified in standard 4-letter acronyms of the form OSXL, standing for Optimization Services X Language, where "X" is to be replaced by any other defined alphabetical letter. For a quick reference, refer to Figure 8-1. OSXL's are grouped into 3 main categories: Optimization Services Representation, Optimization Services Communication and Optimization Services Inspection and Discovery.

7.1 Optimization Services Representation

In this section we introduce a set of low level formats for transmission between different Optimization Services component, including Optimization Services Template Language (OSTL, §7.1.1) for representing optimization problems, Optimization Services Result Language (OSRL §7.1.2) for representing optimization results, Optimization Option Language (OSOL, §7.1.3) for representing solver options, Optimization Services Simulation Language (OSSL, §7.1.4) for representing input/output between simulation and optimization and Optimization Services Analysis Language (OSAL, §7.1.5) for representing analysis results of an optimization problem. They can all be regarded as dialects of XML (see §4.3.2 for definition of "XML dialect").

Before we move on to the proposed optimization services representation languages we should explain more about high level and low-level optimization representations. High-level optimization representation refers to representing an optimization *model*, whereas low-level optimization representation refers to representing an optimization *instance*. The distinction between *model* and *instance* should be clarified. By *model* we mean an abstract algebraic representation of a problem. It can be represented in a modeling language such as AMPL, GAMS, LINGO, ILOG OPL, etc., all of which separate model from data. An optimization problem *instance* is generated by filling a model with corresponding data. Examples of instance representations include MPS standard [43] for linear programming, SMPS standard [2] for stochastic programming, the ".nl" format [32] used in AMPL, and numerous other proprietary formats used in commercial solvers. The Motorola VP Intelligent Optimization System also uses its own proprietary formats.

Low-level optimization representations, that is, optimization instance representations are to be transmitted in our distributed optimization network system and our framework for this research will be concerned only with low-level representations.

7.1.1 Optimization Services Template Language (OSTL)

Background and Purposes

Optimization Services Template Language (OSTL) is the first ambitious step toward a general framework of optimization services representation that addresses all of the problem types supported through the NEOS Server, with sufficient flexibility to be extended to new types. This is an undertaking of a breadth and difficulty not undertaken previously in the area of optimization, and as a result, OSTL may be viewed as one of the most significant parts of our research.

It's worth mentioning the choice of the letter "T" in OSTL. Initially we used "M" for "Modeling." This causes misunderstanding that "OSML" is a high level optimization representation, though it is not. Also "ML" is widely used to indicate "Markup Language" as in "XML." Possible

naming conflicts with other areas can be avoided if we just take away the temptation to use the letter "M" by forbidding it in any OSXL's.

On the other hand, the word "Template" better indicates a unified approach toward representing low level optimization instances. The goal of OSTL is not to introduce another totally new instance representation format and replace all the others. Rather it is to combine and leverage on the best practices of current instance representations, while at the same time phase out the less popular, less powerful and nonstandard formats. For example for linear programming instances, OSTL is potentially leveraging more on the XML based format such as FMLLP [31] than plain text based format such as MPS [43]. One advantage of XML based formats is that additional schemas can be included to provide optional extensions. Thus a standard for optimization can be enforced and can grow in a well-defined way to accommodate new problem types. This contrasts with the current situation, where for example parsers for the MPS standard vary in details between implementations, and interpreters of the SMPS standard for stochastic programming are even more varied.

The construction of OSTL also takes into account that functions may not have a closed form. They may be provided in a binary code or from a remote Web Service. {This part needs to be further investigated.}

Specification Descriptions

OSML Schema is given in A.1.

Figure 7-1 shows an OSML example in its simplest form, in which it contains just one type of representation. Within the **<singleFormat>** element, any standard format can be embedded depending on the "type" attribute of the element. For example if type = "FMLLP", an FMLLP representation can be included. **<singleFormat>** can be suitable for linear programming problems.

```
<OSML>
    <singleFomrat type="...">
        <!--anyformat-->
    </singleFomrat>
</OSML>
```

Figure 7-1: OSML example with a single format

Figure 7-2 shows an OSML example that contains a **<mixedFormat>** element. **<mixedFormat>** can be suitable for general mathematical programming problems, e.g. mixed integer nonlinearly

constrained problems with some non-closed form functions. Following is a descriptive list for the **<mixedFormat>** element:

- **<mixedFormat>** contains three elements: <**variables>**, **<objective>**, and **<constraints>**. **<objective>** is of singular form. It is assumed that in each optimization problem only one objective function can exist.
- **<variables>** contains a collection of **<variable>** elements.
- Each **<variable>** contains an optional initial value
- Each **<variable>** is required to have a name attribute, whose value should be unique. In general a name attribute is required of any variable, objective and constraint. One of the purposes is that names are used to match with elements in other components of the Optimization Services. If no names are specified in high level modeling, default values should be "padded" into the low level instance representation.
- Each **<variable>** can optionally have a type attribute, which can only take the value of either "integer", "binary", or "continuous." By default the value is "continuous" if the attribute is not specified.
- Each **<variable>** can optionally contain a lowerBound attribute and an upperBound attribute to indicate a minimum and a maximum value that the variable can take.
- **<objective>** is required to have a name attribute. It contains four elements : **<direction>**, **<function>**, **<lowerBound>** and **<upperBound>**.
- **<objective>** can optionally have a type attribute. The purpose is for OSML to leverage on existing formats for expressing coefficients of linear objective function. For example, the format can take the objective section of a full linear programming format of MPS. {This part needs further investigation.}
- **<direction>** can only contain one of the two values, namely "minimize" or "maximize."
- **<function>** is of functionType to be discussed below.
- **<lowerBound>** and **<upperBound>** are optional elements to specify lower value and upper bound for the objective function.
- **<constraints>** can contain both a collection of **<constraint>** elements and **<constraintSet>** elements.
- Each <constraint> is required to have a name attribute. It contains three elements: **<function>**, **<lowerBound>** and **<upperBound>**, which are of exactly the same types as those contained in the **<objective>** element.
- **<constraintSet>** is intended to express a set of linear constraints, so that coefficients can be specified in a more compact form. It is required to have a name attribute.
- **<constraintSet>** can optionally have a type attribute. The purpose is for OSML to leverage on existing formats for expressing coefficients of linear constraints. For example, the format

can take the constraint section of a full linear programming format of MPS. {This part needs further investigation.}

- **<function>** can contain any one of the elements: **<webService>, <MathML>** and **<binary>.** {This part needs further investigation}
- **<webService>** is intended for simulation or function evaluation on a remote network. It has to have two elements, **<URI>** for specifying where the service is and **<OSSL>** for specifying input variables and parameters for the Web Service. Specifications such as call operations and input/output arguments are already standardized in the Optimization Services Description Language (OSDL, §7.2.2) and therefore necessary information is kept to a minimum.
- **<binary>** is intended for locally attached binary executable codes. Each **<binary>** has to have a language attribute to indicate what programming language the binary code is generated from. Each **<binary>** also has to have a platform to indicate what computer system the binary code is generated on. Like **<webService>,** it has to have two elements **<URI>** for specifying where the binary code is and **<OSSL>** for specifying input variables and parameters for the Web Service. Solvers don't need to support this. Users can initially discover through the Optimization Services Registry which solvers support the binary code they provide before invoking the solvers. {This part needs further investigation.}
- **<MathML>** is to follow the MathML Schema. Only that its terms can be of any function types described above, besides simple values and defined variables. Constructing multidisciplinary objective function with metrics calculated from remote simulations may become easy. Usually a multi-objective functions are of simple forms such as weighted sum or ratio of metrics, which can be easily expressed in MathML. Metrics can be in the form of **<webService>** elements, which are easy to invoke.

<OSML> can contain an optional element **<OSAL>** at its end. **<OSAL>** element contains meta-information on the analysis of the optimization problem. It is constrained by the Optimization Services Analysis Language (OSAL, §7.1.5) Schema.

```
<OSML>
    <mixedFormat>
        <variables>
            <variable name="X1" type="countinuous" lowerBound="1" upperBound="3">1.2</variable>
            <variable name="X2" type="binary">0</variable>
            <variable name="X3" type="integer"/>
        </variables>
        <objective name="totoalCost">
            <direction>minimize</direction>
            <function>
                <webService>
                    <URI>
                    </URI>
                    <OSSL>
                    </OSSL>
                </webService>
            </function>
            <lowerBound>-100</lowerBound>
            <upperBound>-100</upperBound>
        </objective>
        <constraints>
            <constraint name="c1">
                <!--for expressing nonlinear constraints-->
                <function>
                    <binary language="java" platform="unix">
                        <URI>
                        </URI>
                        <OSSL>
                        </OSSL>
                    </binary>
                </function>
                <lowerBound>-3</lowerBound>
                <upperBound>5</upperBound>
            </constraint>
            <constraint name="c2">
                <function>
                    <MathML>
                        <!--terms can be functions, which in term can be binary, webservice, or mathml, besides values and defined variables-->
                    </MathML>
                </function>
                <upperBound>5</upperBound>
            </constraint>
            <constraintSet name="cset" type="MPS">
                <!--for expressing linear constraints-->
            </constraintSet>
        </constraints>
    </mixedFormat>
    <OSAL>
        <!--analysis data-->
    </OSAL>
</OSML>
```

Figure 7-2: OSML example with a mixed format

7.1.2 Optimization Services Result Language (OSRL)

Background and Purposes

Optimization Services Result Language (OSRL) is intended to represent results generated by optimization solvers. It is a counterpart to OSTL. OSTL will be used as an input format in the "solve" function specified in Optimization Services Description Language (OSDL, §7.2.2) whereas OSRL will be used as an output format returned by the "solve" function. Compared with OSTL, OSRL is more straightforward. The separation of OSRL from OSTL helps in reducing network traffics and enhancing flexibility, among many other benefits. The standardization of OSRL may be most valuable to the modelers for the purpose of presentation. It can also help in facilitating benchmarking as discussed in §7.3.3.

Specification Descriptions

Figure 7-3 shows an OSRL example. Following is a descriptive list for the **<OSRL>** element:

- **<OSRL>** contains four elements: **<status>,** <**variables>**, **<objective>**, and **<constraints>**.
- **<status>** is to contain general information on the optimization solution, such as "unbounded", "solution found", "infeasible", etc. Types of status are to be exhausted. Naming is to be standardized. Possible numeric coding standard for representation of status can also be introduced. This is comparable to the standardization of the status code definitions of the HTTP protocol. For example, in HTTP, code "404" indicates "Not Found", meaning "The web server has not found anything matching the Request-URI."
- **<variables>** contains a collection of **<variable>** elements.
- **<constraints>** contains a collection of **<constraint>** elements.
- All **<variable>, <objective>** and **<constraint>** elements are required to have a name attribute, which value is to be unique.
- All **<variable>, <objective>** and **<constraint>** elements can have two elements: **<standard>** and **<specific>.**
- Both **<standard>** and **<specific>** contain a collection of **<R>** elements to contain individual results.
- **<R>** elements under **<standard>** are standardized across solvers in terms of naming and usage.
- Individual solvers can have solver specific **<R>** elements under **<specific>.**
- Each **<R>** element is required to have a name attribute and a value.
- Each **<R>** element can have an optional **<description>** element.
- **<R>** elements under **<specific>** are suggested to have a **<description>** element.
- Exactly what other elements are to be contained in the **<R>** element depends on the meaning of each result. But it should be kept as simple as possible. {This part needs further investigations.}

```
<OSRL>
    <status>
        <!--unbounded, found, infeasible, error-->
    </status>
    <variables>
        <variable name="x1">
            <standard>
                <R name="value">12</R>
            </standard>
            <specific>
                <R name="weirdValue">
                    <description>some weird values only calculated by this solver</description>
                </R>
            </specific>
        </variable>
        <variable name="x2">
            ...
        </variable>
</variables>
    <objective name="totalCost">
        <standard>
            <R name="value">3</R>
        </standard>
        <specific>
            ...
        </specific>
    </objective>
    <constraints>
        <constraint name="c1">
            <standard>
                <R name="body">12</R>
            </standard>
            <specific>
                ...
            </specific>
        </constraint>
        <constraint name="c2">
            ...
        </constraint>
    </constraints>
</OSRL>
```

Figure 7-3: OSRL example

7.1.3 Optimization Services Option Language (OSOL)

Background and Purposes

Before invoking the "solve" function specified in Optimization Services Description Language (OSDL, §7.2.2), certain solver options can be set through the "set" function also specified in OSDL. OSOL is separate from OSTL for the simple reasoning that OSOL is solver specific whereas OSTL is

not. Options, especially the standard ones in OSOL can be included in Optimization Services Inspection Language, so that user can choose solver based on the availability of options provided by solvers. A standard set of options need to be regulated among all solvers regarding naming and usage. Solvers can choose not to support some options. But as long as they do, they should use the standard names with the same intended uses. There may also be possible naming conflicts between different types of solvers. For example "maxIter" in Figure 7-4 can mean differently in linear programming solvers and nonlinear programming solvers. But this issue may be solved with a standard XML Namespace (see §4.3.4) introduction in Optimization Services world.

Specification Descriptions

Figure 7-4 shows an OSOL example. Following is a descriptive list for the **<OSOL>** element:

- **<OSOL>** contains two elements: **<standard>** and **<specific>.**
- Both **<standard>** and **<specific>** contain a collection of **<O>** elements to contain individual options.
- **<O>** elements under **<standard>** are standardized across solvers in terms of naming and usage.
- Individual solvers can have solver specific **<O>** elements under **<specific>.**
- Each **<O>** element is required to have a name attribute and a value.
- Each **<O>** element can have an optional **<description>** element.
- **<O>** elements under **<specific>** are suggested to have a **<description>** element.
- Exactly what other elements are to be contained in the **<O>** element depends on the meaning of each option. But it should be kept as simple as possible. {This part needs further investigations.}

```
<OSOL>
    <standard>
        <!--standard can be used for inspection and discovery-->
        <O name="maxIter">
            <description>maximum number of iterations</description>
            1.0
        </O>
        <O name="maxTime">
            2.0
        </O>
    </standard>
    <specific>
        <!--solver specific-->
        <O name="weirdSolverOption">
            <description>...</description>
        abc
    </O>
        <O name="...">
            ...
    </O>
    </specific>
</OSOL>
```

Figure 7-4: OSOL example

7.1.4 Optimization Services Simulation Language (OSSL)

Background and Purposes

Optimization Services Simulation Language (OSSL) is used as an input/output format for a client to call a simulation. Each simulation can be thought of as a function, be it an objective or constraint function. It will return some values given a set of input values. No distinctions are made between deterministic and stochastic simulations. OSSL contains an input and/or an output section. Input section contains two types of elements: variables and parameters. From perspective of simulations, they are both input arguments. The reasons to distinguish between two types are that parameters are fixed, whereas variables change. Through the iterative process of optimization, parameters may only need to be sent on the first call to the simulation, thus reducing networking traffic. Also different simulation may choose to treat variables and parameters differently. For example, variable may be represented more accurately for calculating derivatives. Variable names in OSSL should match variable names in the optimization problem specified in OSTL. By separating variable and parameter types, the variable section in OSSL may just keep a simple reference to the corresponding variable section in the OSTL.

Specification Descriptions

Figure 7-5 shows an OSSL example. Following is a descriptive list for the **<OSSL>** element:

- **<OSSL>** contains two elements: **<input>** and **<output>.**
- **<input>** contains a collection of **<param>** and **<var>** elements indicating parameter inputs and variable inputs.
- **<output>** contains a collection of **<metrics>** elements indicating individual simulation results.
- All **<param>, <var>** and **<metrics>** elements are typeless.
- All **<param>, <var>** and **<metrics>** elements are required to have a name attribute and a value.
- Usually under **<Output>**, there is just one **<metrics>** element indicating a functional value. But a simulation can return more than one metrics. In this case, when a functional value is requested by client and the metrics name is not specified, the first metrics is assumed by default.

```
<OSSL>
    <Input>
        <param name="p1">23</param>
        <param name="p2">abc</param>
        <var name="X1">34</var>
        <var name="X2">44</var>
    </Input>
    <Output>
        <metrics name="m1">33</metrics>
        <metrics name="m1">abc</metrics>
    </Output>
</OSSL>
```

Figure 7-5: OSSL example

7.1.5 Optimization Services Analysis Language (OSAL)

Background and Purposes

Optimization Services Analysis Language (OSAL) is used to describe meta-knowledge or extracted characteristics of an optimization problem. As shown in §7.1.1, it may be included as a section in OSTL. OSAL is separate from OSTL, because OSAL is analyzer specific, as well as problem specific.

Currently, a NEOS Server user typically begins at the website index screen, which presents a list of 13 problem types in Figure 7-6.

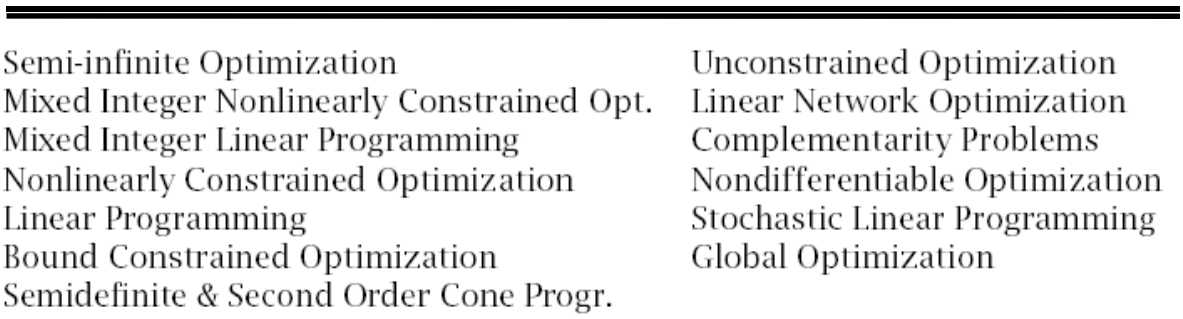

Figure 7-6: Optimization Problem Types at NEOS

Each type links into a list of solvers and input formats (Figure 3-7). The choice among solvers is then up to the user. To provide some assistance in the choice, each solver has a main page with links to the NEOS Guide and to solver-specific documentation (Figure 7-7).

Although this arrangement has proved adequate for many purposes, unavoidably it burdens users with the job of determining a problem type and choosing a solver. Requests to the NEOS help line (neos-comments@mcs.anl.gov) suggest, in particular, that many potential users are analysts who have the training to build a model using a high-level modeling language, but who do not have the expertise to determine what category of model they have produced and what solvers are appropriate for it. The previously remarked leveling off of NEOS Solver requests (Figure 2-2) may reflect the difficulty of broadening the user base to include modeling and application domain experts who are not also algorithm and solver experts.

A description of an optimization problem instance already contains, at least implicitly, all of the information needed to properly categorize the problem. This principle underlies the design of *interactive* problem analyzers such as ANALYZE [33] for linear problems and MProbe [9] for nonlinear problems. Interactive analyzers rely on fairly sophisticated users, however, who are looking to better understand their problems with the aim of making their own determination of how best to solve them. In the context of the NEOS Server, we cannot be sure of as high a level of sophistication on the user's part, nor can we assume that the user is available to interact with the system online. We want to make an automated determination of problem characteristics, and of solver choice based on those characteristics.

Our general framework for distributed optimization is not intended to analyze the optimization problems. Rather it relies on analysis work done by other researchers, and provides a framework specified under the Optimization Services Analysis Language (OSAL) that enforces a standard XML output format of analysis results, thus an automated discovery process can be carried under the Optimization Services inspection and discovery framework.

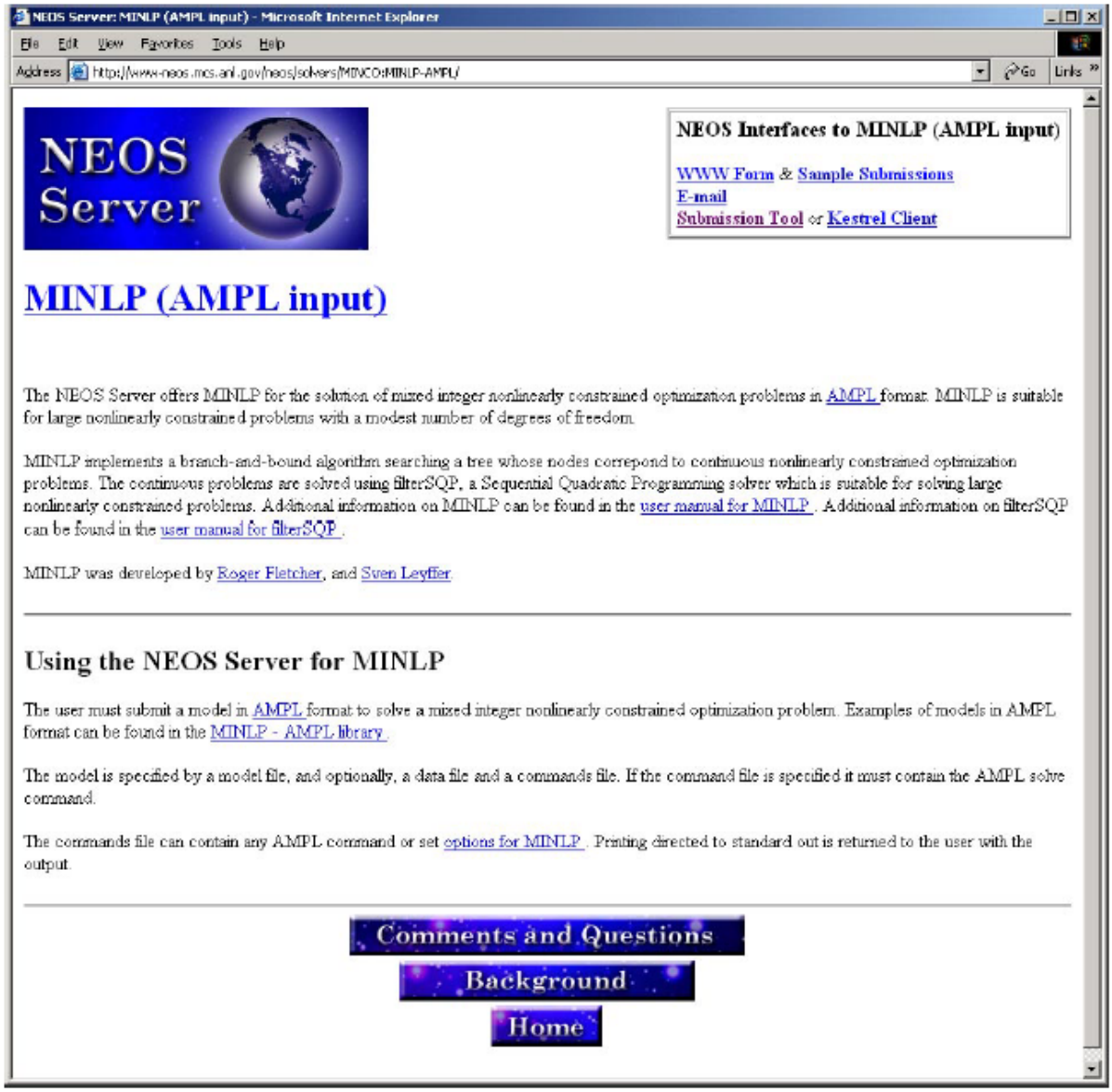

Figure 7-7: An example of a NEOS Server web page for a particular solver, with links to the NEOS Guide and to solver-specific documentation. The box at the top right provides links to the web interface and to instructions for other interfaces.

The collaborative research in this area will initially concentrate on design of a problem analyzer for the NEOS Server and then concerns the determination of appropriate solvers given a list of problem properties from the analyzer. At the beginning, the analyzers will likely be based on the ".nl" format of AMPL [32], which is already recognized by two dozen varied NEOS solvers. Later as the Optimization Services representation framework becomes finalized, the analyzers will switch to the format specified under this framework, taking OSTL as input format and outputting in OSAL format.

Analyzers on the network are to be called under the Optimization Services communication framework, specified by Optimization Services Client Language (OSCL §7.2.1), and possibly also Optimization Services Flow Language (OSFL) and Optimization Services Endpoint Language (OSEL).

Analyzers are in a special ways, solvers. Both types take OSTL as an input parameter, only that analyzers return information in an OSAL format while solvers return information in an OSRL format. Analyzers may also have a set of options to be set before carrying out analysis. Thus it is possible for analyzers to leverage on the access framework specified by Optimization Services Description Language (OSDL §7.2.2) rather than OSCL.

Determination of appropriate solvers based on the meta-knowledge generated by analyzers, is to be carried out under the Optimization Services inspection and discover framework specified by Optimization Services Inspection Language (OSIL §7.3.1).

Specification Descriptions

Figure 7-8 shows an OSAL example. Following is a descriptive list for the **<OSAL>** element:

- **<OSAL>** contains two elements: **<standard>** and **<specific>.**
- Both **<standard>** and **<specific>** contain a collection of **<A>** elements to contain individual analyses.
- **<A>** elements under **<standard>** are standardized across analyzers in terms of naming and usage.
- Individual analyzers can have analyzer specific **<A>** elements under **<specific>.**
- Each **<A>** element is required to have a name attribute and a value.
- Each **<A>** element can have an optional **<description>** element.
- **<A>** elements under **<specific>** are suggested to have a **<description>** element.
- Exactly what other elements are to be contained in the **<A>** element depends on the meaning of each analysis. But it should be kept as simple as possible. {This part needs further investigations.}

```
<OSAL>
    <standard>
        <A name="OptimizationType">
            <description>describing optimization problems type</description>
            Nonlinearly Constrained Optimization
        </A>
        <A name="...">
            ...
        </A>
    </standard>
    <specific>
        <A name="weirdAnalysis">
            <description>...</description>
        abc
        </A>
        <A name="...">
            <description>...</description>
            ...
        </A>
    </specific>
</OSAL>
```

Figure 7-8: OSAL example

7.2 Optimization Services Communication

As mentioned in the introduction, the primary difficulty now facing large-scale optimization has now shifted to *communication*. Increasing number of optimization algorithms are implemented increasingly well. But every algorithm has its own way of naming interfaces, operations, methods, arguments, data types etc. These algorithms, when implemented in software, are programmed in different languages and compiled on different platforms. Furthermore, when the software is put on the network, thy can be located in various places, and numerous mechanisms are employed to invoke them. Due to such an enormous heterogeneity, users become unaware of these "solvers" or do not see the potential benefit that would justify using them. Even if the users do realize the benefit, they may have a hard time obtaining, installing and interfacing with the solvers.

Moreover, only certain combinations of solvers and modeling systems work with each other and modeling language support is slow to keep up with solver extensions to new problem types due to the combination effect of interacting component.

Internet is now providing an increasingly practical way of addressing communication problems in large-scale optimization, especially with the advent of Web Services technologies and the establishment of the recommended SOAP 1.2. The Optimization Services Communication Framework is motivated by the vision that the next-generation Network Enabled Optimization System will be able to address and simplify all the above-mentioned design and implementation issues.

Unlike the Optimization Services Representation framework, languages specified under the Optimization Services Communication framework, are not XML dialects. Rather they are a set of specifications written in XML format based on Web Services Definition Languages (WSDL, §4.3.6). The main purpose of these languages is to further constrain and simplify certain functionalities in a Web Service invocation to tailor to our Optimization Services world. Communication between Model and Solver is to follow Optimization Services Description Language (OSDL, §7.2.2). Communication between Solver and Simulation is to follow Optimization Services Client Language (OSCL, §7.2.1). Theoretically, there can also be communication between Client and Simulation, for example for testing purposes. This is in essence a typical Web Services SOAP call – a user calls a service to get a functional value. If it is to happen, it follows exactly the same communication mechanism as that between Solver and Simulation. Communication between Client and Model is left open. Heterogeneous invocation mechanisms will not affect the distributed optimization process as a whole. Client and Model are usually together. Flexible ways should exist to allow customized modeling environments to meet diverse customer needs. Distributed Client and Model communication can take references from Optimization Services Client Language and specify input/output according to Optimization Services Simulation Language (OSSL, §7.1.4). Communications between all components and the central NEOS Registry are left open for now. Likely the two communications to be specified are the one between Solver and Registry for reporting solver runtime status

and the one between Client and Registry for registering and querying Optimization Services. Likely both can leverage on other specifications that are already defined, for example Optimization Services Client Language (OSCL).

7.2.1 Optimization Services Client Language (OSCL)

Background and Purposes

Optimization Services Client Language (OSCL) is mainly intended to call a standard Web Service used as a simulation for optimization. Put a different way, any simulation that will be called by an optimization solver to get functional values should be interfaced as a Web Service specified by OSCL. When a solver needs a function from a simulation, the solver is considered as a client.

Specification Descriptions

OSCL is in essence a WSDL document. It has a root element **<definitions>** prefixed with an **"OSCL"** namespace. OSCL stipulates only one operation for client interface:

string call (string input)

Both return value and input value should be of the XML format specified in Optimization Services Simulation Language (OSSL, §7.1.4). Default binding should be SOAP to HTTP. Other needs of transport bindings are not seen as of immediate necessities in Optimization Services. It should by default be a remote procedure call (rpc) based on the request and response synchronous model. To mimic an asynchronous call, the rpc can just be launched in a separate process or thread other than the general optimization process. Port addresses (locations of simulations) should be specified by modelers when constructing OSTL. Figure 7-9 shows an OSCL example. As mentioned earlier, the simple OSCL may be leveraged upon and tailored toward communications between other components in our general decentralized design. For example, when a client queries an optimization service from NEOS Registry, the same call can be made, only that the input and output have to be of the format specified in Optimization Services Query Language (OSQL, §7.3.4). When a solver reports its current run time status to NEOS Registry, the same call can still be made, only that the input and output have to be of the format specified in Optimization Services Process Language (OSPL, §7.3.2). But in general, between any two components, the **"call"** can only assume *one* type of pair of input and output formats.

```
<OSCL:definitions 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">
    <types/>
    <message name="callRequest">
        <part name="OSSLRequest" type="xsd:string"/>
    </message>
    <message name="callResponse">
        <part name="OSSLResponse" type="xsd:string"/>
    </message>
    <portType name="client">
        <operation name="call" parameterOrder="OSSLRequest">
            <input message="callRequest" name="callRequest"/>
            <output message="callResponse" name="callResponse"/>
        </operation>
    </portType>
    <binding name="clientSoapBinding" type="client">
        <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="call">
            <wsdlsoap:operation soapAction=""/>
            <input name="callRequest">
                <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
            </input>
            <output name="callResponse">
                <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" use="encoded"/>
            </output>
        </operation>
    </binding>
    <service name="clientService">
        <port binding="clientSoapBinding" name="client">
            <wsdlsoap:address location="http://localhost:8080/axis/client.jws"/>
        </port>
    </service>
</OSCL:definitions>
```

Figure 7-9: OSCL example

7.2.2 Optimization Services Description Language (OSDL)

Background and Purposes

Optimization Services Definition Language (OSDL) is used by modelers to call solvers, including initiating the solver, setting options and solving the optimization. It is the communication between Model and Solver in our general decentralized design of distributed optimization. Other OSDL functionalities are possible but should only be optimization specific. Between optimization Model and Solver, session should be maintained so that options set through a previous call should remain in effect when a later call is initiated for optimization. Session maintenance and other generic resource management functionalities that are not optimization specific should be leveraged upon either Web Services or Grid Services protocols. For example a “stop” call, intended to end an

optimization session, should be handled by the notification functionality provided by the Grid Services protocol.

Specification Descriptions

Like OSCL, OSDL is in essence a WSDL document. OSCL has a root element **<definitions>** prefixed with an **"OSDL"** namespace. OSDL stipulates the following client interface:

int solver (binary bSolve) – for initiating the solver

string set (string optionInput) – for setting solver options

string solve (string problemInput) – for solving the optimization

In the "**solver**" operation, input value specifies whether a caller just wants to check status ("false") or finally needs to solve optimization ("true"). Output reports solver status. For example "-1" can indicate solver not ready and a positive integer can indicate optimization job number for later retrieval. Integer encodings need to be standardized. It is similar the standardization of the status code definitions of the HTTP protocol. In practice, the **"solver"** operation can be used by NEOS registry to check status of the solver.

In the "**set**" operation, both input and output values should be of the XML form specified in OSOL (see 7.1.3). Option values in OSOL are set to empty or some equivalent but descriptive encodings indicating input request cannot be resolved. Again encodings here should be standardized. Based on the retuned status of option settings, modelers can chose to further initiate the **"solve"** operation or not.

In the "**solve**" operation, input should be of the XML format specified in OSTL (see §7.1.1) and output should be of the XML format specified in OSRL (see §7.1.2). Figure 7-10 shows an OSDL example.

```
<OSDL:definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
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">
    <types/>
    <message name="solverRequest">
        <part name="jobRequest" type="xsd:boolean"/>
    </message>
    <message name="solverResponse">
        <part name="jobResponse" type="xsd:int"/>
    </message>
    <message name="setRequest">
        <part name="OSOLRequest" type="xsd:string"/>
    </message>
    <message name="setResponse">
        <part name="OSOLResponse" type="xsd:string"/>
    </message>
    <message name="solveRequest">
        <part name="OSMLRequest" type="xsd:string"/>
    </message>
    <message name="solveResponse">
        <part name="OSRLResponse" type="xsd:string"/>
    </message>
    <portType name="solver">
        <operation name="solver" parameterOrder="jobRequest">
            <input message="solverRequest" name="solverRequest"/>
            <output message="solverResponse" name="solverResponse"/>
        </operation>
        <operation name="set" parameterOrder="OSOLRequest">
            <input message="setRequest" name="setRequest"/>
            <output message="setResponse" name="setResponse"/>
        </operation>
        <operation name="solve" parameterOrder="OSMLRequest">
            <input message="solveRequest" name="solveRequest"/>
            <output message="solveResponse" name="solveResponse"/>
        </operation>
    </portType>
    <binding name="solverSoapBinding" type="impl:solver">
        <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="set">
            <wsdlsoap:operation soapAction=""/>
            <input name="setRequest">
                <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" use="encoded"/>
            </input>
            <output name="setResponse">
                <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" use="encoded"/>
            </output>
        </operation>
        <operation name="solver">
            <wsdlsoap:operation soapAction=""/>
            <input name="solverRequest">
                <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" use="encoded"/>
            </input>
            <output name="solverResponse">
                <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" use="encoded"/>
            </output>
        </operation>
```

```
        <operation name="solve">
            <wsdlsoap:operation soapAction=""/>
            <input name="solveRequest">
                <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" use="encoded"/>
            </input>
            <output name="solveResponse">
                <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" use="encoded"/>
            </output>
        </operation>
    </binding>
    <service name="solverService">
        <port binding="solverSoapBinding" name="solver">
            <wsdlsoap:address location="http://localhost:8080/axis/solver.jws"/>
        </port>
    </service>
</OSDL:definitions>
```

Figure 7-10: OSDL example

7.2.3 Optimization Services Flow Language (OSFL)

Background and Purposes

The term Optimization Services Flow Language (OSFL) is reserved for now. The exact purpose is not clear and may well be covered with the future development of Web Services and Grid Services technologies. It is not of an immediate design issue. Our informal intention is to organize analyzers, solvers, optimization simulations and other Optimization Services components, orchestrate information (e.g. input and output), sequence optimization process, resolve common variables etc. OSFL may prove to be useful in multi-objective optimization, multi-start optimization, multi-level optimization, multi-disciplinary optimization, Multi-task optimization, Multi-processor optimization and Pareto-set optimization. It is likely that OSFL will highly leverage on the interfaces specified in OSDL (see §7.2.2). It may also need to collaborate with OSPL (see §7.3.2). OSFL will probably wait to see the success and popularity of other OSXL's.

Specification Descriptions

At this point the term OSFL remains as a concept and is not investigated in detail.

7.2.4 Optimization Services Endpoint Language (OSEL)

Background and Purposes

The term Optimization Services Endpoint Language (OSEL) is reserved for now. The exact purpose is not clear and may well be covered with the future development of Web Services and Grid Services technologies. It is not of an immediate design issue. Our informal intention is to be compatible with certain grid computing features. OSEL may be used to describe non-functional characteristics of an Optimization Service, including quality of service, privacy policy, auditing policy. The design of OSEL should not affect the core syntax of OSDL (see §7.2.2). OSEL may affect whether the solver requestor chooses to collaborate with a particular solver provider. It can be important for asynchronous message flows (that is not request and response model), expected optimization time, possible duration estimates for interaction or number of acceptable retires, basis on which solver requestor could establish time-out behavior and execute rollback or other interaction/compensation mechanism. OSEL should mainly deal with run time information and it may need to collaborate with OSPL (see §7.3.2).

Specification Descriptions

At this point the term OSEL remains as a concept and is not investigated in detail.

7.3 Optimization Services Inspection and Discovery

Traditionally, distributed optimization systems use a straightforward scheme that relies on a database that pairs solvers with problem types they can handle. Characteristics of a problem instance, determined from either a manual search or an analysis phase, will be used to automatically generate a query on the database that will return a list of appropriate solvers. More advanced scheme will consider extensions to generate lists ranked by degree of appropriateness. In a subsequent stage of research, a more sophisticated mechanism can be developed to take account of additional that would be used by a solver expert.

The special features of optimization serve to distinguish our research in this area from the routine design of new Web Services. Optimization runs are characterized by their huge and hard-to-predict consumption of processor time and memory space; only a modest increase in the instance size generated from an integer programming model, for example, can cause the solution time to increase from minutes to days, with a corresponding increase in the maximum size of the branch-and-bound tree. Predictions of resource requirements must take account of problem characteristics, since for instance a continuous linear program in hundreds of thousands of variables is generally much more tractable than an integer or nonlinear program of the same size.

Collaborated research, as well as our research outlined in §3.1 can be used to study how categorization of optimization problem instances together with statistics from previous run can be used to improve upon the current scheduling decision of the NEOS server. 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 of 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. Some of the above information can be retrieved off-line, meaning available before solving of a problem. Offline information on solvers is mostly specified in Optimization Services Inspection Language and Optimization Services Benchmark Language (OSBL). Offline information on optimization instances are mostly specified in Optimization Services Analysis Language (OSAL). Other information can only be retrieved on-line, meaning available when a solver is solving an optimization problems. Online information is specified in Optimization Services Process Language (OSPL) and can be conveyed in a feedback system to a registry though mechanisms like Optimization Services Client Language (OSCL). Query formats are specified by Optimization Services Query Language (OSQL).

Again the Optimization Services inspection and discovery framework is not intended to find good schemes to pair solvers and problems. Rather it relies on appropriate schemes found already and provides a mechanism to facilitate the automation of inspection and discovery process of Optimization Services through OSIL, OSPL, OSBL and OSQL.

7.3.1 Optimization Services Inspection Language (OSIL)

Background and Purposes

Optimization Services Inspection Language (OSIL) is mainly used to find and register optimization solvers. It certainly includes categorization information illustrated in the NEOS Optimization Tree (Figure 7-11). OSIL is to be treated like a "database record" in the Optimization Services Registries, only that the record is in XML format rather than a row, and it is to be queried by OSQL (see §7.3.4). OSIL can contain optimization information in the form of keywords, abstracts and descriptions. It can publish functionalities including supported solver options specified in OSOL (see §7.1.3), NEOS authoritative benchmarking (e.g. NEOS solver rankings) specified in OSBL (see §7.3.3), OSPL (see §7.3.2) and accepted function types that it supports (see §7.1.1). OSIL can even contain links to other valuable information, like a pointer to a compatible solver. OSIL is the part that heavily needs authorities' involvements, for example INFORMS, OTC/NEOS, and W3C.

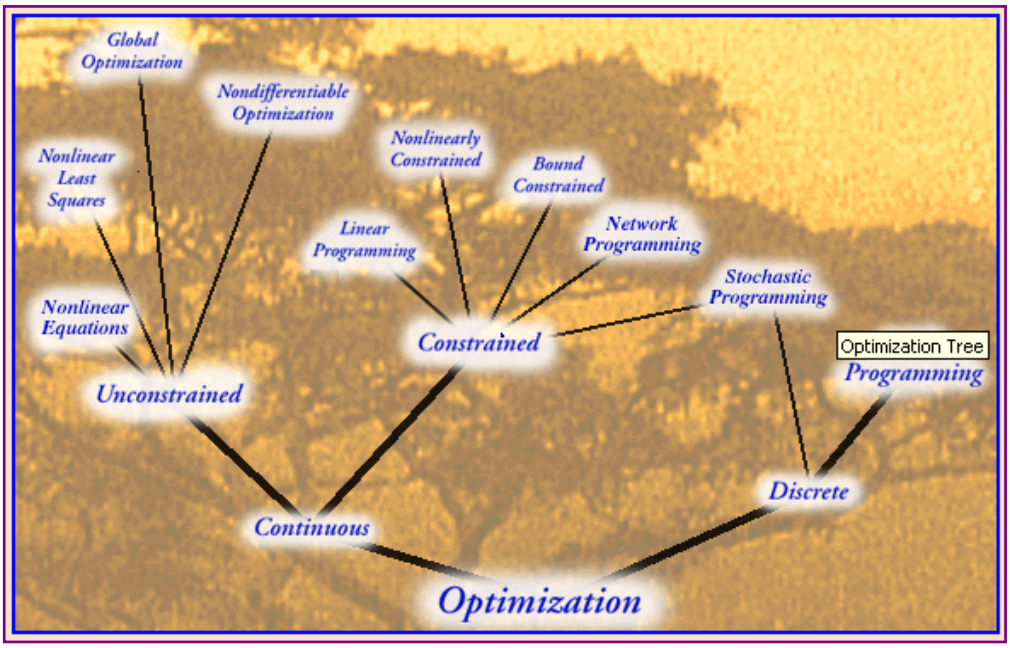

Figure 7-11: NEOS Optimization Tree

Specification Descriptions

Figure 7-12 shows an OSIL example. Following is a descriptive list for the **<OSAL>** element:

```
<OSIL:inspection xmlns="http://schemas.xmlsoap.org/ws/2001/10/inspection/">
    <!--siimlar to a record in a table-->
    <!--can use OSQL to search the registry table of OSIL-->
    <abstract/>
    <service>
        <name>CPLEX</name>
        <abstract>A solver that solves linear problem</abstract>
        <description referencedNamespace="http://schemas.xmlsoap.org/wsdl/" location="http://localhost:8080/cplex?wsdl">
    </description>
        <solverCategory>Linear Programming</solverCategory>
        <OSOL>
            <standard/>
            <specific/>
        </OSOL>
        <OSBL/>
        <FunctionTypesAccepted>
            <MathML/>
            <binary>
                <languages>
                    <language>java</language>
                    <language>c++</language>
                </languages>
                <platforms>
                    <platform>win2000</platform>
                    <platform>UNIX</platform>
                </platforms>
            </binary>
            <webService/>
        </FunctionTypesAccepted>
        <NEOSRank>1</NEOSRank>
    </service>
    <link referencedNamespace="http://schemas.xmlsoap.org/ws/2001/10/inspection/" location="http://localhost:8080/othersolver.wsil">
        <abstract>Other solver</abstract>
    </link>
</OSIL:inspection>
```

Figure 7-12: OSIL example

- **<OSIL:inspection>** contains at least three elements including, **<abstract>**, **<service>** and <**link>.**
- **<abstract>** is to provide a brief description and key words about the optimization solver.
- **<link>** contains a set of locations that the that can be linked from the current OSIL. It also contains a set of **<abstract>** to describe briefly what the links are.
- **<service>** can contain many elements including **<name>, <abstract>**, **<description>, <solverCategory>, <OSOL>, <OSBL>, <OSPL>, <FunctionTypesAccepted>.**
- **<OSOL>, <OSBL>, <OSPL>** elements are to follow the formats specified respectively in Optimization Services Option Language, Optimization Services Benchmark Language and Optimization Services Process Language.

- The exact inclusion of all elements can only be finalized on the satisfactory development of the other OSXL's.

7.3.2 Optimization Services Process Language (OSPL)

Background and Purposes

Optimization Services Process Language (OSPL) is mainly used to keep runtime or dynamic online information about solvers, such as whether is solver is busy or not, and the number of optimization jobs waiting in the solver queue. To this end, it can be regarded as a counterpart to OSBL (see §7.3.1), which mainly keeps static solver information. It is possible, as discussed in §7.3.1, that OSPL can be embedded in OSIL. On the other hand it may not be feasible, due to the constantly changing nature of OSPL. Further details need to be investigated.

Specification Descriptions

Figure 7-13 shows an OSPL example. Following is a descriptive list for the **<OSPL>** element:

```
<OSPL>
    <standard>
        <P name="status">
            <description>indicating the status of a solver server</description>
        busy
    </P>
    </standard>
    <specific>
        <P name="serverS1">
            <description>...</description>
        abc
    </P>
    </specific>
</OSPL>
```

Figure 7-13: OSPL example

- **<OSPL>** contains two elements: **<standard>** and **<specific>.**
- Both **<standard>** and **<specific>** contain a collection of **<P>** elements to contain individual online process information.
- **<P>** elements under **<standard>** are standardized across solvers in terms of naming and usage.

- Individual solvers can have solver specific **<P>** elements under **<specific>.**
- Each **<P>** element is required to have a name attribute and a value.
- Each **<P>** element can have an optional **<description>** element.
- **<P>** elements under **<specific>** are suggested to have a **<description>** element.
- Exactly what other elements are to be contained in the **<P>** element depends on the meaning of each process information. But it should be kept as simple as possible. {This part needs further investigations.}

7.3.3 Optimization Services Benchmark Language (OSBL)

Background and Purposes

The availability of more than one solver for many classes of problems makes the NEOS Server an obvious choice as a benchmarking tool. In fact the Server is potentially useful both in choosing a solver for a particular application and in comparing solvers generally. There are significant barriers to achieving these potentials, however, which motivate this part of the proposed research.

Someone who has developed a new model, but who is not sure which of the several applicable solver packages to apply, is often advised that the only way to be sure is to carry out some test runs on typical problems instances. The straightforward way to do this is to send each test instance to each candidate solver. But as NEOS makes no guarantee that separate runs will be done on comparable machines under comparable conditions, the results may say little about the relative efficiency of the solvers. The results may say more about the reliability of the solvers, but even so they may be distorted by differences in the memory available on the workstations devoted to different solvers, or by differences in time limits imposed by the owners of different workstations on which NEOS Server jobs run. There is not necessarily any obvious way to compensate for the differences between runs, moreover, because in general each solver is available on any of a number of dissimilar workstations, among which one is selected by the Server according to the load at the time a job is submitted.

As a first step in addressing these difficulties, NEOS has added a kind of "benchmarking solver." A user tells this benchmarker which solvers are to be compared (Figure 7-14) and which problem (in AMPL or GAMS) they are to be compared on. The benchmarker then applies all the requested solvers – on the same computer – and returns concatenated listing of their results, along with a summary of problem statistics. For the case of smooth nonlinear problems, the benchmarker also optionally assesses the quality of each solver's solution with respect to complementarity, feasibility and optimality tolerances (which may be adjusted by the user) [16]. This innovative approach to solution verification is independent of any correctness claims or statistics made by individual solvers. Benchmarking on only one problem can be misleading, so a number of sample problems from an application are often tested at the time. Benchmark tests on large sets of problems from diverse applications are also common, for purposes of comparing the overall quality of different solvers. For

this purpose, a concept of a *performance profile* [15], has been developed, which clearly shows the tradeoffs between speed and reliability of alternative solvers applied to a test problems set (Figure 7-15). This device has been favorably received and is being increasingly adopted by researchers for their computational comparisons of new algorithmic ideas. We will investigate the incorporation of this approach into the NEOS Server environment, or more generally the Optimization Services framework, with the aim of producing a benchmarker that takes a set of problems as input and produces statistics and performance profiles for appropriate solvers. The benchmarking tools are intended to accept but not require guidance form the user, so that it is appropriate for us by practitioners as well as researchers. The measures of reliability reflected in the resulting performance profiles will make use of our verification approach to ensure that consistent standards are applied in comparing of solvers. The NEOS Server might then be able to automatically maintain benchmark results on available solvers for public test problem sets, re-running the benchmarker periodically to take account of updates or newly available solvers.

Like analysis on optimization instances (see §7.1.5), our general framework for distributed optimization is not intended to benchmark the optimization solvers. Rather it relies on analysis work done by other researchers, and provides a framework specified under the Optimization Services Benchmark Language (OSBL) that enforces a standard XML output format of benchmark result, thus an automated discovery process can be carried under the Optimization Services inspection and discovery framework. Benchmark information is likely to be imbedded in Optimization Services Inspection Language (OSIL, §7.3.1).

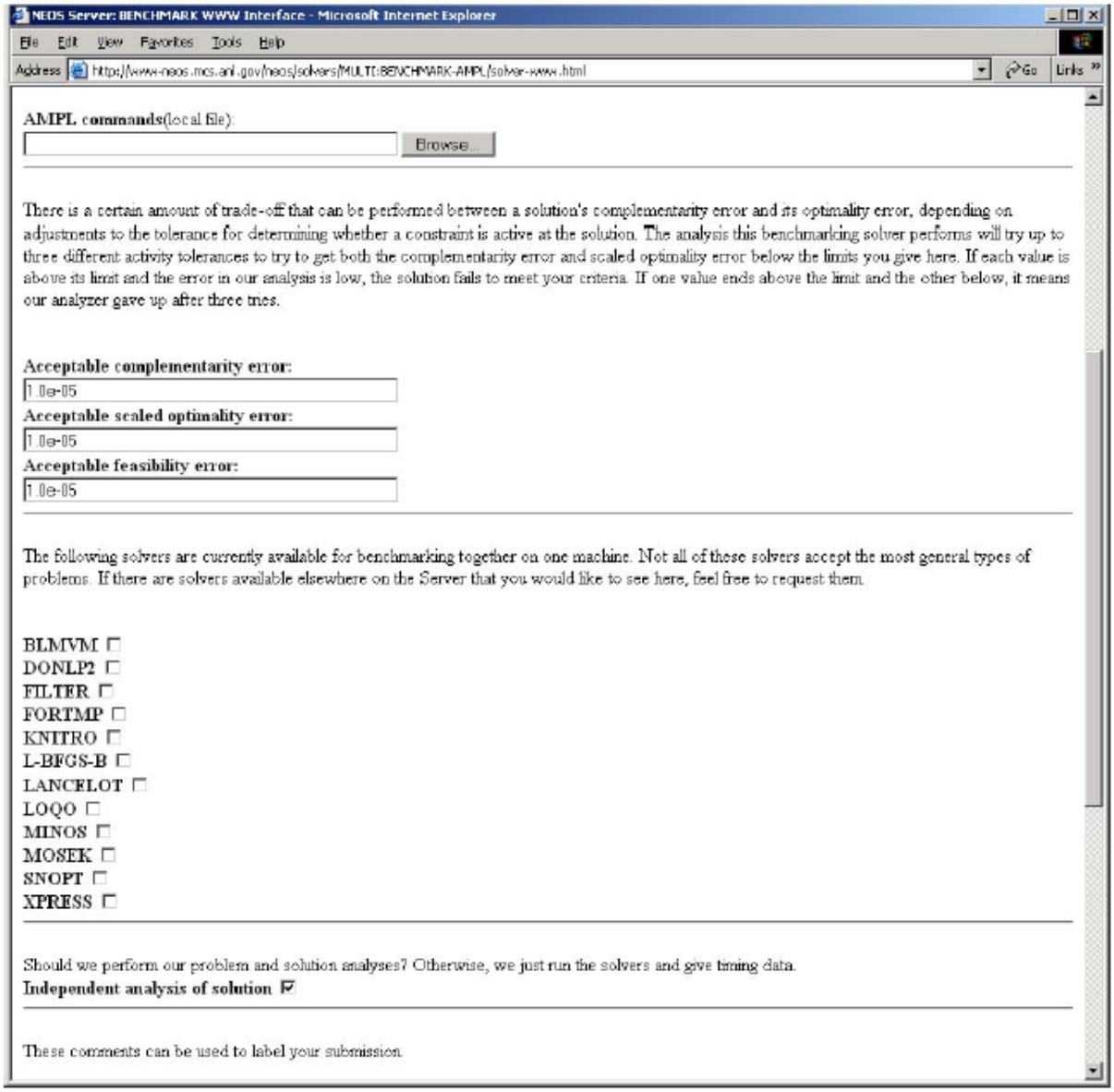

Figure 7-14: Part of the web interface for the special benchmarking solver of the NEOS Server

The collaborative research in this area will initially investigate connecting the analyzer described in §7.1.5 to the current benchmarker, so that the user is asked to choose only among solvers that are appropriate for the problem to be solved. Concurrently, the collaborative research will further test and refine the verification methods in [16] and will extend them to handle a broader variety of problems and situations.

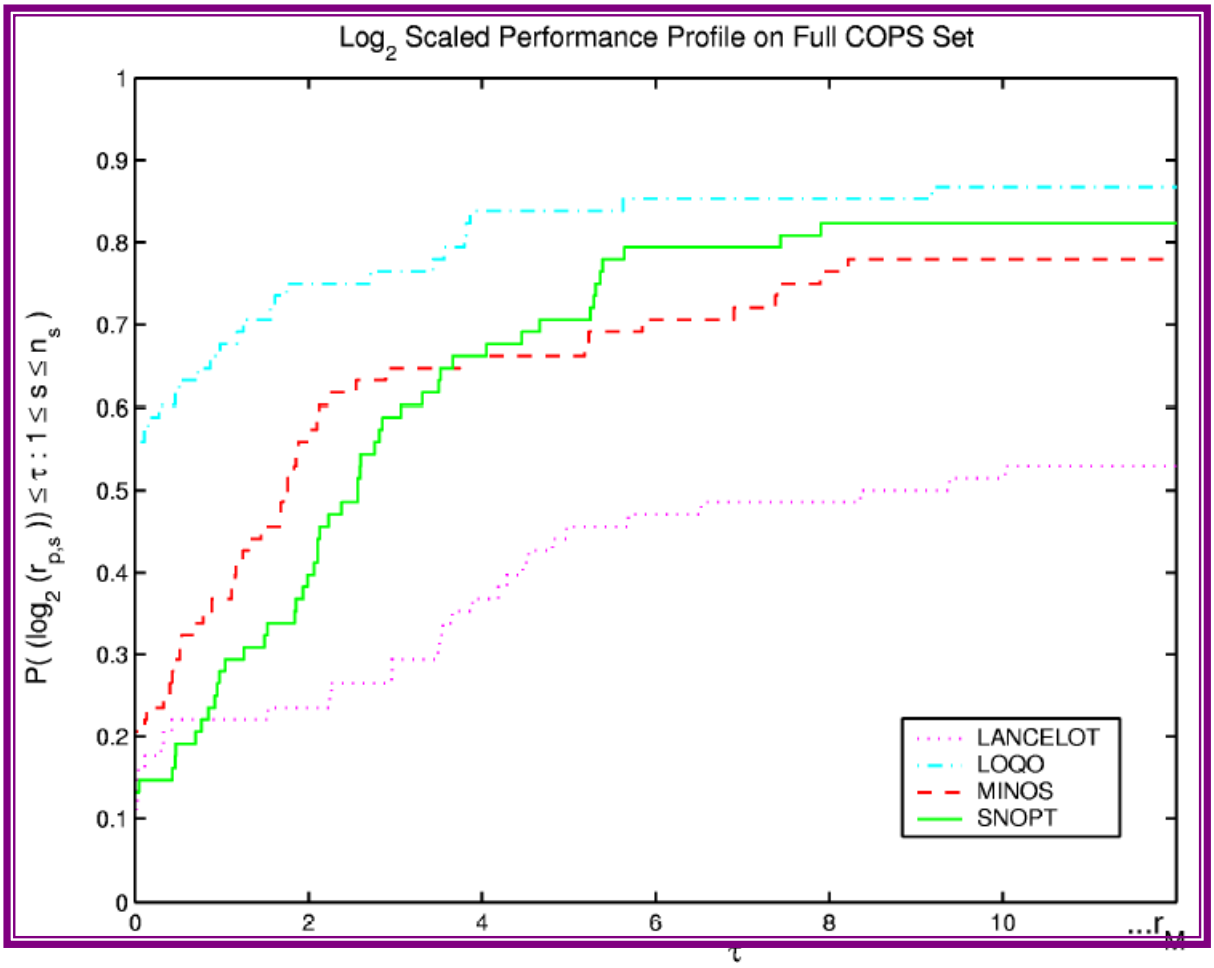

Figure 7-15: A performance profile [15] summarizing benchmark results from four solvers on a variety of test problems. Toward the left the curves emphasize speed of the solvers, while toward the right they place greater emphasis on reliability.

Specification Descriptions

Figure 7-16 shows an OSBL example. Following is a descriptive list for the <OSBL> element:

- **<OSBL>** probably does not need to contain two elements: **<standard>** and **<specific>,** since benchmarking is supposed to be carried out against one single authoritative benchmarker.
- Contents in **<OSBL>** are to be designed by researchers who do benchmarking analysis.

```
<OSBL>
    <!--TBD by benchmarking researchers-->
</OSBL>
```

Figure 7-16: OSBL example

7.3.4 Optimization Services Query Language (OSQL)

Background and Purposes

Optimization Services Query Language (OSQL) is intended as an optimization query language to search for OSIL (see §7.3.1) documents in Optimization Services registries. It may not be needed depending on the final draft of XMLQuery from W3C.

Specification Descriptions

OSQL is in essence an XMLQuery. As of December 2003, it is still in progress under the auspices of the W3C's XML Query working group. Specifications on OSQL need to wait after the W3C's final recommendation.

8 CONCLUSIONS AND FUTURE WORK

The research that we proposed is motivated by our vision of a next-generation distributed optimization, which we call "Optimization Services", characterized by a set of four-letter acronyms of the form OSXL, where X is a defined alphabetical letter in our framework (Figure 8-1). Our Optimization Services design and framework is intended to deal with outstanding challenges of communication in large-scale optimization. This work addresses design as well as implementation issues by providing a general and unified framework for 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. Our research in these areas is timely, being motivated by new standards for Web Services, grid-computing technologies, and the recent success of both the Virtual Prototyping Optimization System at Motorola and the NEOS Server at Argonne National Laboratory.

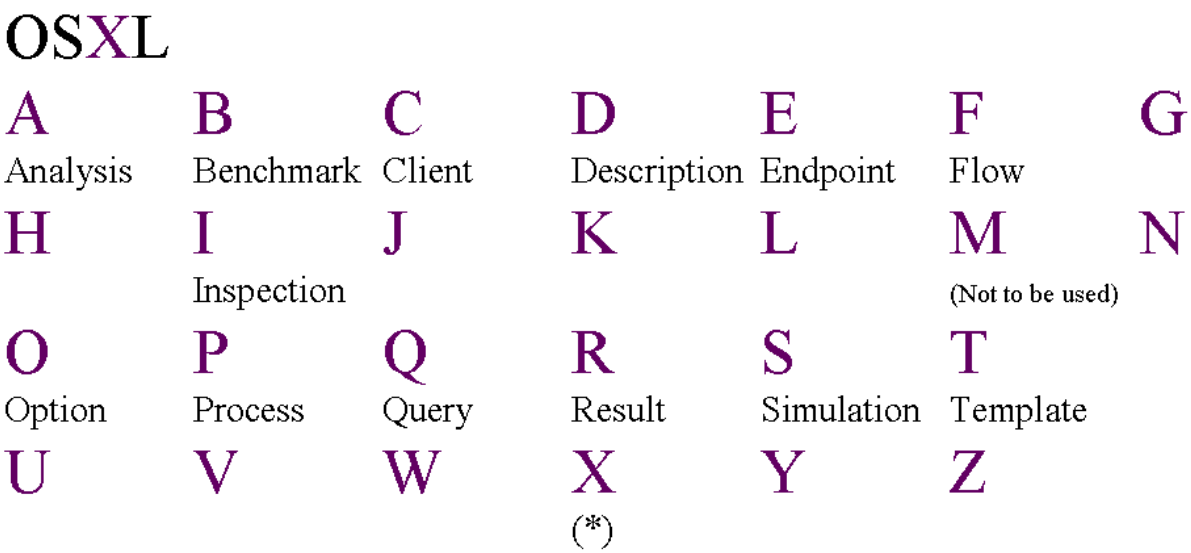

Figure 8-1: Optimization Services X Languages, where X is to be replaced by any of the other 25 letters that have been defined. OSXL's are used to specify the general and unified framework for distributed optimization proposed in this paper.

We still need further improvement on Motorola Virtual Prototyping group's intelligent optimization system, to be carried out in the summer of 2004. Optimization Services framework discussed in this paper may possibly follow a process similar to W3C's model: starting from working group notes, through working drafts, candidate recommendations, proposed edited recommendations,

proposed recommendations and finalized with recommendations. But before there is such a possible process, we need to be more thoughtful and have to further elaborate on certain details. Our design and framework need be more general, systematic and prepared for scalability. More formal and tighter collaborations, under the proposed framework, with researchers in mentioned areas need to be established.

APPENDIX

A.1 Optimization Services Template Language (OSTL) Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
    <xs:include schemaLocation="./OSAL.xsd"/>
    <xs:element name="OSML">
        <xs:annotation>
            <xs:documentation>Opitimization Service Modeling Language schema
    </xs:documentation>
        </xs:annotation>
        <xs:complexType>
            <xs:sequence>
                <xs:choice>
                    <xs:group ref="mixedFormat" minOccurs="0"/>
                    <xs:element ref="singleFormat" minOccurs="0"/>
                </xs:choice>
                <xs:element name="OSAL"/>
            </xs:sequence>
            <xs:attribute name="format" type="OSMLFormatType" use="required"/>
        </xs:complexType>
    </xs:element>
    <xs:simpleType name="OSMLFormatType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="MIXED"/>
            <xs:enumeration value="FMLLP"/>
            <xs:enumeration value="MPS"/>
            <xs:enumeration value="SMPS"/>
            <xs:enumeration value="AMPL.nl"/>
            <xs:enumeration value="other"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:group name="mixedFormat">
        <xs:all>
            <xs:element ref="variables"/>
            <xs:element ref="objective" minOccurs="0"/>
            <xs:element ref="constraints"/>
        </xs:all>
    </xs:group>
    <xs:element name="variables">
        <xs:complexType>
            <xs:sequence maxOccurs="unbounded">
                <xs:element name="variable" type="variableType"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:complexType name="variableType">
        <xs:all>
            <xs:element name="lowerBound" type="xs:decimal" minOccurs="0" maxOccurs="1"/>
            <xs:element name="upperBound" type="xs:decimal" minOccurs="0" maxOccurs="1"/>
        </xs:all>
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="type" default="continuous">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:enumeration value="interger"/>
                    <xs:enumeration value="binary"/>
                    <xs:enumeration value="continuous"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
    </xs:complexType>
    <xs:element name="objective">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="direction" minOccurs="1" maxOccurs="1">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
```

```
                        <xs:enumeration value="minimize"/>
                        <xs:enumeration value="maximize"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:element>
            <xs:element name="function" type="functionType"/>
            <xs:element name="lowerBound" type="xs:decimal" minOccurs="0" maxOccurs="1"/>
            <xs:element name="upperBound" type="xs:decimal" minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
        <xs:attribute name="name" type="xs:string" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="constraints">
    <xs:complexType>
        <xs:all minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="constraint" type="constraintType"/>
            <xs:element ref="constraintSet" type="constraintSetType"/>
        </xs:all>
    </xs:complexType>
</xs:element>
<xs:complexType name="functionType">
    <xs:choice>
        <xs:element name="webservice" type="WSType"/>
        <xs:element name="binary" type="binaryType"/>
        <xs:element name="MathML"/>
    </xs:choice>
</xs:complexType>
<xs:complexType name="WSType">
    <xs:sequence>
        <xs:element name="URI"/>
        <xs:element name="OSSL"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="binaryType">
    <xs:sequence>
        <xs:element name="URI"/>
        <xs:element name="OSSL"/>
    </xs:sequence>
    <xs:attribute name="language" type="xs:string" use="required"/>
    <xs:attribute name="platform" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="constraintType">
    <xs:sequence>
        <xs:element name="function" type="functionType"/>
        <xs:element name="lowerBound" type="xs:decimal" minOccurs="0" maxOccurs="1"/>
        <xs:element name="upperBound" type="xs:decimal" minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="constraintSetType">
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="type" type="xs:string" use="required"/>
</xs:complexType>
<xs:element name="singleFormat" type="xs:string">
</xs:element>
</xs:schema>
```

BIBLIOGRAPHY

[1] J. Bigus and J. Bigus, Constructing Intelligent Agents with Java, John Wiley & Sons (1997).

[2] 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.

[3] 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.

[4] R.E. Bixby, Solving Real-World Linear Programs: A Decade and More of Progress. Operations Research **50** (2002) 3-15.

[5] S. Brin, L. Page, Anatomy of a Large-Scale Hypertextual Web Search Engine, Proceeding 7th International World Wide Web Conference (1998).

[6] 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.

[7]T. Berners-Lee, etc., W3C, http://www.w3c.org (2003).

[8] 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.

[9] J.W. Chinneck, Analyzing Mathematical Programs Using MProbe. Annals of Operations Research **104** (2001) 33-48.

[10] A. R. Conn, N. I. M. Could and Ph. L. Toint, LANCELOT: A FORTRAN Package for Large-Scale Nonlinear Optimization. Springer Verlag (1992)

[11] J. Czyzyk, M.P. Mesnier and J.J. Moré, The NEOS Server. IEEE Journal on Computational Science and Engineering **5** (1998) 68-75.

[12] E.D. Dolan, NEOS Server 4.0 Administrative Guide. Technical Memorandum ANL/MCS-TM-250, Argonne National Laboratory, Argonne, IL (2001).

[13] 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).

[14] 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.

[15] E.D. Dolan and J.J. Moré, Benchmarking Optimization Software with Performance Profiles. Mathematical Programming **91** (2002) 201-213.

[16]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).

[17] 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.

[18] 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.

[19] T. Finin and Y. Labrou, eds., UMBC agentWeb, http://agents.umbc.edu (2003).

[20] I. Foster, Designing and Building Parallel programs, Addison Wesley (1994).

[21] I.Foster and C. Kesselman, eds., Open Grid Services Architecture (OGSA), http://www.globus.org/ogsa/ (2003).

[22] I.Foster and C. Kesselman, eds., The Globus Alliance, http://www.globus.org (2003).

[23] I.Foster and C. Kesselman, eds., Open Grid Services Architecture (OGSA), http://www.globus.org/ogsa/ (2003).

[24] R. Fourer, Modeling Languages Versus Matrix Generators for Linear Programming. ACM Transactions on Mathematical Software **9** (1983) 143-183.

[25] 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).

[26] R. Fourer, D.M. Gay and B.W. Kernighan, A Modeling Language for Mathematical Programming. Management Science **36** (1990) 519-554.

[27] 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.

[28] R. Fourer and J.-P. Goux, "Optimization as an Internet Resource." Interfaces **31,** 2 (2001) 130-150.

[29] R. Fourer and L. Lopes, A management System for Decompositions in Stochastic Programming. Under revision for Annals of Operations Research (2002).

[30] 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).

[31] 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).

[32] D.M. Gay, Hooking Your Solver to AMPL. Technical report, Bell Laboratories, Murray Hill, NJ (1997); http://www.ampl.com/REFS/abstracts.html#hooking2.

[33] 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.

[34] 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.

[35] 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).

[36] IBM, COmputational INfrastructure for Operations Research (COIN-OR), http://www-124.ibm.com/developerworks/opensource/coin (2003).

[37] C.A.C. Kuip, Algebraic Languages for Mathematical Programming. European Journal of Operational Research 67 (1993) 25-51.

[38] L. Lach, J. Lopez, J. Ma, T. Tiprak, W. Xiao, A Method for Automated Concept Exploration, http://www.motorola.com/content/0,1037,299,00.html (2003).

[39] D. Lange and M.Oshima, Programming and Deploying Java Mobie Agents with Aglets, Addison-Wesley (1998).

[40]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.

[41] J. Ma, L. Lach, T. Tirpak, A, W. Xiao, A Method for Large Scale Mixed Integer Nonlinear Optimization for Virtual Prototyping, Motorola Inc., http://www.motorola.com/content/0,1037,299,00.html (2001).

[42] David Megginson, Simple API for XML (SAX), http://www.saxproject.org (2003).

[43] B.A. Murtagh, Advanced Linear Programming: Computation and Practice, McGraw-Hill (1981).

[44] Napster.com, P2P Technology, http://www.napster.com.

[45] 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.

[46] P. Sandhu, The MathML Handbook, Charles River Media, MA (2003).

[47] Aaron Skonnard and Martin Gudgin, Essential XML Quick Reference, Pearson Education (2002).

[48] 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).

[49]T. Tirpak, J. Ma, J. Savic, R. Crosswell, Cost-Efficient Selection of Devices and Resistive Inks for Embedded Passive Board Designs, Motorola Inc. http://www.motorola.com/content/0,3306,359,00.html, http://www.motorola.com/content/0,3306,283,00.html (2001).

[50] T. Tirpak, J. Ma, J. Savic, R. Crosswell, Device Selection Method for Embedded Passive Design, Motorola Inc. http://www.motorola.com/content/0,3306,359,00.html, http://www.motorola.com/content/0,3306,283,00.html (2002).

[51] T. Tirpak, J. Ma, J. Savic, R. Crosswell, Optimization of Array and Panel Area Utilization, Motorola Inc., http://www.motorola.com/content/0,3306,299,00.html (2001).

[52] UDDI.org, Universal Description, Discovery, and Integration (UDDI), http://www.uddi.org (2003).

[53] E. Van der Vlist, XML Schema. O'Reilly & Associates (2002).

[54] W3C, Document Object Model (DOM), http://www.w3.org/DOM (2003).

[55] W3C, Namespaces in XML, http://www.w3.org/TR/REC-xml-names (1999).

[56] W3C, XML, http://www.w3.org/XML (2003).

[57] W3C, XML Link Language (XLink) http://www.w3.org/TR/xlink (2001).

[58] W3C, Mathematical Markup Language (MathML) http://www.w3.org/TR/REC-MathML (1999).

[59] W3C, XML Path Language (XPath) http://www.w3.org/TR/xpath (1999).

[60] W3C, XML Pointer Language (XPointer) http://www.w3.org/TR/xptr (2003).

[61] W3C, XML Query Language (XQuery) http://www.w3.org/TR/xquery (2003).

[62] W3C, XML Schema, http://www.w3.org/XML/Schema.html (2003).

[63] W3C, XSL, http://www.w3.org/TR/xsl (2001).

[64] W3C, XSLT, http://www.w3.org/TR/xslt (1999).

[65]W3C, Web Services, http://www.w3.org/2002/ws/ (2002).

[66] P.Walmsley, Definitive XML Schema. Prentice-Hall (2001).