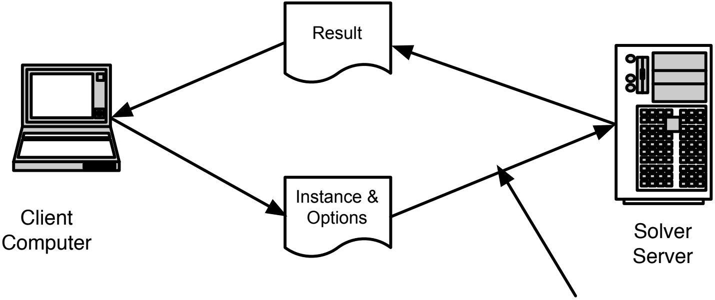
The Optimization Services Solver Interface

Horand Gassmann Dalhousie University
Jun Ma Northwestern University
Kipp Martin University of Chicago
(kmartin@chicagobooth.edu)

November 10, 2010

Outline

Motivation



BIG PROBLEM!

Unless the solver can directly read the optimization instance and options an API is needed.

Motivation

An ideal world – no solver interface!

If the instance interface and option interface are robust enough then we have:

```
load(problem_instance);
solve(options_list);
```

It is up to the solver to implement the problem_instance and interpret what is in the options_list

Motivation

From Coin-discuss:
On Sat, 8 Jul 2006, Matthew Galati wrote:

Hi,

Several of the LP solvers in Osi have interior point methods. Can this be added as an option to solve with interior vs simplex? I guess the OSI2 design - where **model and algorithm are split would fix this...** but, is OSI2 still going to happen? and, if there is no real timeline for OSI2, can this be added to OSI?

Matt

Basic Philosophy

Design Concept 1: a solver interface should distinguish between

1. the problem **instance**
2. the solver **options** used for a given run
3. the **result** of the solver run
4. **modifications** to a problem instance

Basic Philosophy

Design Concept 2: a solver interface should have

1. strings and file representation
2. in-memory representation

In a distributed environment we need strings and files (e.g. a client machine want to pass options to a server and get results back).

When the solver reads a file you need to work in-memory.

Basic Philosophy

Design Concept 3: Combine concepts 1 and 2

	Standard	**In-Memory Object**
Instance	OSiL	OSInstance
Option	OSoL	OSOption
Result	OSrL	OSResult
Modification	OSmL	OSModification

By *standard* I mean an XML that the string or file can be validated against. Obviously useful for error checking early in the game.

Basic Philosophy

Status Report:

- OSiL/OSInstance – 100% for linear, integer, general nonlinear. Still working on disjunctive, stochastic, cone, etc.
- OSrL/OSResult – 100% complete
- OSoL/OSOption – 85% complete
- OSmL/OSModification – 0 % – well you can modify variable bounds, constraint bounds, and objective function bounds – *in the linear case you can use Osi that sits under OS.*

Basic Philosophy

Example:

Step 1: Create a solver

```
DefaultSolver *solver  = NULL;
solver = new CouenneSolver();
```

`DefaultSolver` is an abstract class with pure virtual functions. The `CouenneSolver` class inherits from the `DefaultSolver` class.

Basic Philosophy

Example (Continued):

Step 2: Read the option and instance **file** into a **string**

```
std::string osil;
osilFileName =   dataDir +  "p0033.osil";
osil = fileUtil->getFileAsString( osilFileName.c_str() );
```

```
std::string osol;
osolFileName =   dataDir +  "p0033.osol";
osol = fileUtil->getFileAsString( osolFileName.c_str() );
```

Basic Philosophy

Example (Continued):

Step 3: Put the option and instance **strings** into the corresponding **in-memory** solver objects

```
osilreader = new OSiLReader();
solver->osinstance = osilreader->readOSiL( osil);

osolreader = new OSoLReader();
solver->osoption   = osolreader->readOSoL( osol);
```

Step 4: Give the solver the instance and set the options

```
solver->buildSolverInstance(); //a pure virtual function
solver->setSolverOptions(); //a pure virtual function
```

Basic Philosophy

Example (Continued):

Step 5: Solve the model

```
solver->solve(); //a pure virtual function
```

Step 6: Print the result!

```
std::cout << solver->osrl << std::endl;
```

Basic Philosophy

The `OSDefaultSolver` abstract class has three key virtual functions:

- virtual void solve() $=$ 0 ;
- virtual void buildSolverInstance() $=$ 0 ;
- virtual void setSolverOptions() $=$ 0 ;

The above functions get implemented in each of our solver interfaces.

Basic Philosophy

Interface with Osi: The OS interface **wraps around** Osi.

In OS there is a `CoinSolver` class that inherits from the `DefaultSolver` class. It has an additional member **sSolverName**.

```
solver = new CoinSolver();
solver->sSolverName ="clp"; //DyLP, SYMPHONY, Cplex, etc
```

The `CoinSolver` class has another member, not in the base class

```
OsiSolverInterface *osiSolver;
```

When you call `solver->buildSolverInstance();` it instantiates the proper Osi solver class.

Basic Philosophy

Interface with Osi (continued): Once you have `CoinSolver` you have the Osi interface.

Example: Now use the Osi Solver interface to add a column

```
OsiSolverInterface *si = solver->osiSolver;



for(k = 0; k < numColumns; k++){

   si->addCol(numNonz[ k], rowIdx[k], values[k],
        collb, colub,   cost[ k]) ;

}
```

Basic Philosophy

Summary: With the OS interface you get Osi plus:

- Nonlinear solver interfaces
- A separate Option interface
- A separate Result interface
- The ability to make remote solve calls

Basic Philosophy

Supported COIN-OR Solvers:

- Bonmin
- Clp (through Osi)
- Cbc (through Osi)
- Couenne
- Dip (see Application Templates) – **a different solver for each block**
- DyLP (through Osi)
- Ipopt
- SYMPHONY (through Osi)
- Vol (through Osi)

OS Instance Interface

$$\begin{array}{ll} \text{Minimize} & (1 - x_0)^2 + 100(x_1 - x_0^2)^2 + 9x_1 \\ \text{Subject to} & x_0 + 10x_0^2 + 11x_1^2 + 3x_0x_1 \leq 25 \\ & \ln(x_0x_1) + 7x_0 + 5x_1 \geq 10 \\ & x_0, x_1 \geq 0 \end{array}$$

OS Protocols: OSiL

The variables: $x_0, x_1 \geq 0$

```
<variables number="2">
     <var lb="0" name="x0" type="C"/>
     <var lb="0" name="x1" type="C"/>
</variables>
```

The objective function: $\text{minimize} \quad 9x_1$

```
<objectives number="1">
     <obj maxOrMin="min" name="minCost">]
          <coef idx="1">9</coef>
     </obj>
</objectives>
```

OS Instance Interface

The linear terms are stored using a sparse storage scheme

$$\begin{aligned} x_0 + 10x_0^2 + 11x_1^2 + 3x_0x_1 &\leq 25 \\ 7x_0 + 5x_1 + \ln(x_0x_1) + &\geq 10 \end{aligned}$$

```
<linearConstraintCoefficients>
     <start>
          <el>0</el><el>2</el><el>3</el>
     </start>
     <rowIdx>
          <el>0</el><el>1</el><el>1</el>
     </rowIdx>
     <value>
          <el>1.0</el><el>7.0</el><el>5.0</el>
     </value>
</linearConstraintCoefficients>
```

OS Instance Interface

Representing quadratic and general nonlinear terms

$$\begin{aligned} x_0 + 10x_0^2 + 11x_1^2 + 3x_0x_1 &\leq 25 \\ 7x_0 + 5x_1 + \ln(x_0x_1) + &\geq 10 \end{aligned}$$

```
<quadraticCoefficients numberOfQuadraticTerms="3">
     <qTerm idx="0" idxOne="0" idxTwo="0" coef="10"/>
     <qTerm idx="0" idxOne="1" idxTwo="1" coef="11"/>
     <qTerm idx="0" idxOne="0" idxTwo="1" coef="3"/>
</quadraticCoefficients>

<nl idx="1">
     <ln>
          <times>
               <variable coef="1.0" idx="0"/>
               <variable coef="1.0" idx="1"/>
          </times>
     </ln>
</nl>
```

OS Instance Interface

Key idea a **schema**. How do we know how to write proper OSiL? Similar to the concept of a class in object orient programming. Critical for parsing!

Schema $\Longleftrightarrow$ **Class**

XML File $\Longleftrightarrow$ **Object**

We need a schema to define the OSiL instance language.

```
<constraints number="2">
   <con name=“row0” ub="10.0"/>
   <con name=“row1” lb="10.0"/>
</constraints>
```

OS Instance Interface

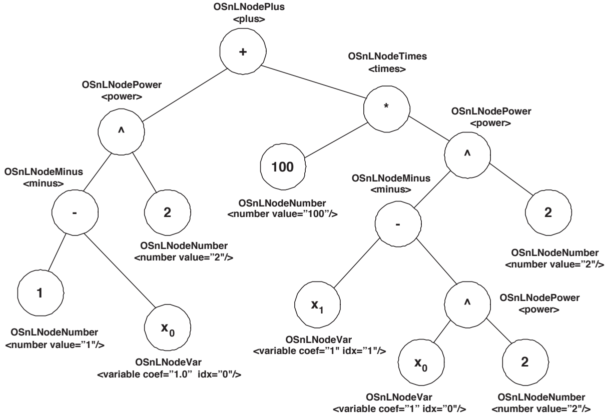
Schema a Constraints and Con Class

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

OSInstance Nonlinear Interface

$$(1 - x_0)^2 + 100(x_1 - x_0^2)^2$$

How do we validate this? Designing the schema is a huge problem!



OSInstance Nonlinear Interface

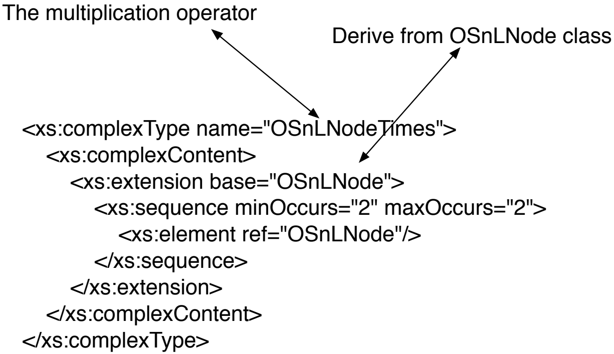
Design Goal: *represent a comprehensive collection of optimization problems while keeping parsing relatively simple.* Not easy!!!

- For purposes of validation, any schema needs an explicit description of the children allowed in a `<operator>` element
- It is clearly inefficient to list every possible nonlinear operator or nonlinear function allowed as a child element. If there are n allowable nonlinear elements (functions and operators), listing every potential child element, of every potential nonlinear element, leads to $O(n^2)$ possible combinations.
- This is also a problem when doing function and gradient evaluations, etc. a real PAIN with numerous operators and operands.
- We avoid this by having EVERY nonlinear node an OSnLNode instance.

OSInstance Nonlinear Interface

Solution: *Use objected oriented features of the XML Schema standard.*

```
<xs:complexType name="OSnLNode" mixed="false"/>
<xs:element name="OSnLNode" type="OSnLNode"
        abstract="true">
```



The multiplication operator

Derive from OSnLNode class

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

OSInstance Nonlinear Interface

- The code for implementing this is written in C++.

- The C++ code “mimics” the XML schema

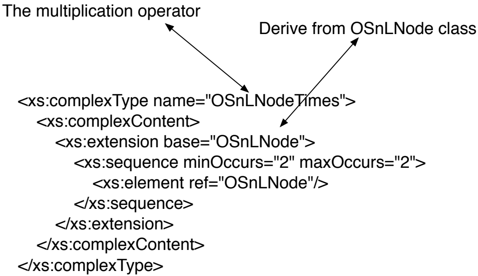
- In C++ there is an abstract class **OSnLNode** with pure virtual functions for function and gradient calculation.

- There are operator classes such as **OSnLNodePlus** that inherit from **OSnLNode** and *do the right thing* using polymorphism.

OSInstance Nonlinear Interface

- Each XML schema complexType corresponds to a class in `OSInstance`. Elements in the actual XML file then correspond to objects in the `OSInstance` class.
- An attribute or element used in the definition of a `complexType` is a member of the corresponding in-memory class; moreover the type of the attribute or element matches the type of the member.
- A schema sequence corresponds to an array. For example, the complexType `Constraints` has a sequence of `<con>` elements that are of type `Constraint`.

OSInstance Nonlinear Interface



The multiplication operator

Derive from OSnLNode class

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

```
OSnLNodePlus::OSnLNodePlus()
{
     snodeName = "plus";
     inumberOfChildren = 2;
     m_mChildren = new OSnLNode*[2];
     m_mChildren[ 0] = NULL;
     m_mChildren[ 1] = NULL;
     inodeInt = 1001;
}//end OSnLNodePlus
```

OSInstance Nonlinear Interface

The `OSnLNode` class mimics the complexType `OSnLNode` in the schema. It is an **abstract class** with virtual functions,

```
virtual double calculateFunction(double *x) = 0;
```

Here is the implementation of the virtual function in the `OSnLNodePlus` class that is derived from the `OSnLNode` class.

```
double OSnLNodePlus::calculateFunction(double *x){
   m_dFunctionValue = m_mChildren[0]->calculateFunction(x)
   + m_mChildren[1]->calculateFunction(x);
   return m_dFunctionValue;
}// end OSnLNodePlus::calculate
```

The OSInstance API

The OSInstance is then 1) a data structure (including a nonlinear expression tree), 2) a set of `get()` methods, and 3) a set of `set()` methods.

Some `get()` methods:

- get instruction lists in postfix or prefix
- get a text version of the model in infix
- get function and gradient evaluations
- get information about constraints, variables, objective function, the A matrix, etc.
- get the root node of the OSExpression tree

Using the OSInstance API

How can a solver use the API:

- the solver can work directly with the OSInstance data structure
- the solver can use the $\mathrm{get}()$ methods to convert the OSInstance structure into its own data structure
- the solver can use OSInstance to perform function and gradient calculations.

Work Directly with OSInstance

Scatter column 0 of the A matrix into a dense vector:

```
OSiLReader *osilreader ;
osilreader = new OSiLReader();
OSInstance *osinstance;
osinstance = new OSInstance();
osinstance = osilreader->readOSiL( osil);
double *aColSparse;
aColSparse = osinstance->instanceData->linearConstraintCoefficients->value->el;
int *rowIdx;
rowIdx = osinstance->instanceData->linearConstraintCoefficients->rowIdx->el;
int *start;
start = osinstance->instanceData->linearConstraintCoefficients->start->el;
int numConstraints;
numConstraints = osinstance->instanceData->constraints->numberOfConstraints;
double *aColDense = new double[ numConstraints ];
for(int i = start[0]; i < start[ 1]; i++){
     aColDense[ rowIdx[ i] ] = aColSparse[ i];
}
```

Convert an OSInstance into Solver Data Structure

CoinSolver – take an OSInstance object and create an instance using the OSI.

```
osinstance->getVariableUpperBounds();
osinstance->getConstraintLowerBounds();
```

LindoSolver – take an OSInstance object and create an instance using the Lindo API.

```
allExpTrees = osinstance->getAllNonlinearExpressionTrees();
for(posTree = allExpTrees.begin(); posTree != allExpTrees.end();
++posTree){
postFixVec = posTree->second->getPostfixFromExpressionTree();
```

Use OSInstance for Callback Functions

Use the OSInstance object to:

Provide function evaluations.

Provide gradient evaluations (AD)

Provide Hessian of the Lagrangian (AD)

Using the OSOption API

Design Goal: *maximize flexibility in representing solver options but keep the design simple!*

With OSiL we try to be as encompassing and complete as possible. We try to represent all interesting optimization instances.

With OSoL we do the opposite – take a minimalist approach.

A linear program is a well defined entity, a set of solver options is not. We can't have a linear program without constraints; however, we can have a set of solver options without an initial feasible point.

Using the OSOption API

The schema for a **SolverOption**

```
<xs:complexType name="SolverOption">
    <xs:sequence>
      <xs:element name="item" type="xs:string" minOccurs="0" maxOccurs="unbounded">
      </xs:element>
   </xs:sequence>
   <xs:attribute name="name" type="xs:string" use="required"/>
   <xs:attribute name="value" type="xs:string" use="optional"/>
   <xs:attribute name="solver" type="xs:string" use="optional"/>
   <xs:attribute name="category" type="xs:string" use="optional"/>
   <xs:attribute name="type" type="xs:string" use="optional"/>
   <xs:attribute name="description" type="xs:string" use="optional"/>
   <xs:attribute name="numberOfItems" type="xs:nonNegativeInteger" default="0" use="optional"/>
</xs:complexType>
```

Using the OSOption API

An actual OSoL instance:

```
<solverOptions numberOfSolverOptions="5">
   <solverOption name="tol" solver="ipopt" type="numeric"
      value="1.e-9"/>
   <solverOption name="print_level" solver="ipopt"
      type="integer" value="5"/>
   <solverOption name="OsiDoReducePrint" solver="osi"
      type="OsiHintParam"  value="false"/>
   <solverOption name="LS_IPARAM_LP_PRINTLEVEL" solver="lindo"
      category="model"  type="integer" value="0"/>
   <solverOption name="LS_IPARAM_LP_PRINTLEVEL" solver="lindo"
      category="environment" type="integer" value="1"/>
</solverOptions>
```

Using the OSOption API

A few things:

- An OSoL file can have options for multiple solvers, e.g. Ipopt, Osi, Lindo
- An Osi solver is one where we can set options through the Osi interface
- You can specify types for the options, e.g. numeric, int, string, etc.
- An option can have the *same* name, but apply to different categories, e.g. LINDO.

Using the OSOption API

The **SolverOption** class “mimics” the XML.

```
class SolverOption {

std::string name;
std::string value;
std::string solver;
std::string category;
std::string type;
std::string description;
}
```

Using the OSOption API

Using the OSOption API

```
std::vector<SolverOption*> solverOptions;
std::vector<SolverOption*>::iterator vit;

solverOptions = osoption->getSolverOptions("symphony");

for (vit = solverOptions.begin(); vit != solverOptions.end();
      vit++) {

         //get options you need

}
```

Using the OSOption API

For flexibility we have the **other** option. Here is how we get an initial solution into Dip for the blocks.

```
<other name="initialCol" solver="Dip" numberOfVar="6"
   value="2" >
   <var idx="10" value="1"/>
   <var idx="11" value="1"/>
   <var idx="12" value="1"/>
   <var idx="13" value="1"/>
   <var idx="14" value="1"/>
   <var idx="17" value="1"/>
</other>
```