The COIN-OR Open Solver Interface 2.0

Lou Hafer[1] Matthew Saltzman[2]

[1]Department of Computer Science
Simon Fraser University

[2]Department of Mathematical Sciences
Clemson University.

INFORMS Annual Meeting
Charlotte, North Carolina
November 15, 2011

Outline

What is OSI?

- A cross-solver API
- Lower level than most solver APIs
  - Intended as a “crossbar switch” to connect applications to solvers
  - Instance management
  - Algorithm control (e.g., pivot-level simplex) is a goal
- One of the original COIN-OR projects (a product of impetuous youth and inexperience)

Design Objectives for OSI 2.0

- Transparent plugin framework for dynamic loading of back ends and solver libraries
- Good programming practice—clean separation of interface and implementation, based on standard design patterns, etc.
- Small, focused, reusable, extensible APIs
- Support for interaction of APIs
- Ease of use, ease of shim generation

Outline

Dynamic Loading of Solver Engine

- Solver Engine loaded at runtime, not needed at link time
- dlopen(), dlsym(), etc., in Linux, other calls in Windows and other Unix systems
- Cross-platform libraries for this task (GNOME glib, GNU libtool)
- Mechanics should be hidden from users

Plugin Management Architecture: The Manager

- The Osi2PluginManager class supports the loading and unloading of external libraries.
  - Get/set default path
  - Load/unload a single plugin library
  - Load all plugin libraries in a directory
  - Unload all plugin libraries
  - Provide services to plugin
- Plugin loading is atomic
- Multiple libraries can provide the same API

Plugin Management Architecture: The Plugin

- Plugin pulls in solver back-end libraries and provides a factory for API objects
- A plugin library needs an initPlugin () function with 'C' linkage.
  - Plugin manager sends pointers to functions for API registration and services, including plugin state management
  - Supports distinct states for plugin library and each API
  - initPlugin () registers API create and destroy functions, returns state management object and cleanup function
- Can provide shims written in C++ or C (wrapped in C++ adapters).

Loading Solver Back End

- Heavyweight shim
  - Link with solver library so all functions are loaded automatically at startup
  - Dynamic solver library?
- Lightweight shim
  - Dynamically load solver library
  - Load functions from solver library on first use as needed by each shim call
  - Requires ‘C’ linkage

Outline

The Control API—A User-Friendly PM Interface

- Loads and unloads plugin libraries
- Creates and destroys objects
- User can specify library that creates an object (or not)
- Control API maintains object identification information
- Utility functions

An Example: Create Several APIs

The main loop:

```
std::vector<std::string> solvers;
solvers.push_back("clp");
solvers.push_back("clpHeavy");
solvers.push_back("glpkHeavy");
std::vector<std::string>::const_iterator iter;
for (iter = solvers.begin(); iter != solvers.end();
     iter++) {
  std::string solverName = iter;
  retval = testControlAPI(solverName,dfltSampleDir);
  totalErrs += retval;
}
```

An Example: Create Several APIs (cont.)

The testControlAPI() function:

```
API apiObj = nullptr;
if (ctrlAPI.createObject(apiObj,"Osi1"))
  errcnt++;
else {
  Osi1API *osi = dynamic_cast<Osi1API *>(apiObj);
  std::string exmip1Path = dfltSampleDir+"/brandy.mps";
  osi->readMps(exmip1Path.c_str());
  Osi1API *o2 = osi->clone();
  if (ctrlAPI.destroyObject(apiObj)) errcnt++;
  o2->initialSolve();
  if (!o2->isProvenOptimal()) errcnt++ ;
  if (ctrlAPI.destroyObject(o2)) errcnt++;
}
```

Outline

Design Concepts

- Object interactions managed using inheritance or composition
- Composition:
  - an object of one type contains a reference to an object of another type
  - The owner object calls the owned objects methods
  - The owner can pass its **this** pointer to the owned object's methods if the owned object must act on the owner
- Both have roles in OSI2 design

Use of Inheritance: Factories

- User interacts with a pure virtual parent interface class
- A *factory* class provides a createObject() method that returns an appropriate child object
- Multiple implementations can be derived from the interface
- Child type can be decided at runtime
- Key design concept for plugin management

Use of Composition: Bridges

- Full set of primitive operations defined in owned class
- User interacts with a collection of high-level methods
- High-level methods implemented via calls to the owned object's primitives
- Different implementations of primitives can be instantiated using factories
- Different collections of high-level operations can share primitive implementations
- Concept used in ModelAPI and SolverAPI, which own solver shims to accomplish actions

Concepts in Action: Osi1API

- Multiple inheritance
- Backward compatibility
- Ease of implementation when the user API matches the solver API

Osi1API on the User Side

- A new class that declares all OsiSolverInterface methods as pure virtual methods
- ```
  class Osi1API : public API
  {
    ...
    virtual void initialSolve() = 0;
    ...
  }
  ```
  ```

Osi1API_ClpHeavy on the Plugin Side

- A new class that inherits all OsiSolverInterface methods.
- ```
  class Osi1API_ClpHeavy
  : public API, public OsiClpSolverInterface
  {
    ...
    inline void initialSolve()
    { OsiClpSolverInterface::initialSolve(); }
    ...
  }
  ```
  ```

Outline

Sample Use Case

Suppose we want to build and solve a model:

- Develop a model with a plugin specialized for model development. ModelAPI provides methods to create and maintain a model instance
- Solve the model with a plugin specialized for solving the model. SolverAPI provides methods to determine an optimal solution to a model instance
- Continue with cycles that modify and resolve the model. Need both APIs.

How should the objects implementing these APIs communicate?

Integrated Model

Suppose the solver plugin can support both the ModelAPI and the SolverAPI:

- OSI2 provides support for capability upgrades.
- Unload the model from the object implementing the ModelAPI and load it into the object implementing the ModelAPI and SolverAPI.
  - Explicitly invoke API load and unload operations. Implementation within OSI2.
  - Use a 'copy constructor'. Implemented by the solver shim.

Integrated Model

Suppose the solver plugin can support both the ModelAPI and the SolverAPI:

- OSI2 provides support for capability upgrades.
- Unload the model from the object implementing the ModelAPI and load it into the object implementing the ModelAPI and SolverAPI.
  - Explicitly invoke API load and unload operations. Implementation within OSI2.
  - Use a 'copy constructor'. Implemented by the solver shim.
- Coordination between the model and the solver is handled by the solver object and the shim, because the solver is holding the only representation of the model.
- This is a good match to the current state of the art in solver implementation.
- The disadvantage is that combinations of APIs are limited to strict augmentation.

Interacting Objects

Suppose that the solver does not support an integrated model; it expects to use an external model.

- OSI2 would provide support for synchronization between interacting objects. This is nontrivial, and there would be some overhead, but it's possible.
- If the solver were prepared to work with an external model, there would be no overhead beyond synchronization.
- In the current state of the art for solvers, there would be at least two copies of the model and multiple repetitions of copying the model from the ModelAPI object to the SolverAPI object.
- The advantage is that the interactions between objects are not constrained.

What Price Freedom?

The big questions:

- Will there be sufficient use cases to justify the costs of the interacting objects model?
- Will developers develop to this model if it's available?