The COIN-OR
Open Solver Interface 2.0

Lou Hafer\(^1\) Matthew Saltzman\(^2\)

\(^1\)(retired)

\(^2\)School of Mathematical and Statistical Sciences
Clemson University

COIN fORgery, IMA, October 18, 2018
Outline

Introduction

The Plugin API

The Control API

The Feature API: Models and Solvers

The Frontier: Open Design Decisions
What is OSI?

- A cross-solving API
- Lower level than most solver APIs
  - Instance management
  - Algorithm control (e.g., pivot-level simplex) is a goal (honored more often than not in the breach)
  - Intended as a “crossbar switch” to connect applications to solvers
- 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

Introduction

The Plugin API

The Control API

The Feature API: Models and Solvers

The Frontier: Open Design Decisions
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
The singleton 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
  - Pulls in solver library with native linkage

- **Lightweight shim**
  - Load functions from solver library on first use as needed by each shim call
  - Requires ‘C’ linkage to dynamically retrieve function pointer
Outline

Introduction

The Plugin API

The Control API

The Feature API: Models and Solvers

The Frontier: Open Design Decisions
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 API selects based on capability registry)
- Control API maintains object identification information
- Utility functions
The main loop:

```cpp
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;
}
```
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

Introduction

The Plugin API

The Control API

The Feature API: Models and Solvers
  Proof of Concept: Incorporating OSI V1 Solver Interfaces

The Frontier: Open Design Decisions
Design Concepts

- Composition or inheritance — your choice!
- Bridge architecture
- Factory and abstract API
Concepts in Action: Osi1API

- Multiple inheritance
- Backward compatibility
- Ease of implementation when the user API matches the solver API
A new class that declares all OsiSolverInterface methods as pure virtual methods

```cpp
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

Introduction

The Plugin API

The Control API

The Feature API: Models and Solvers

The Frontier: Open Design Decisions
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.
- 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 synchronisation 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 synchronisation.
- 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?
What Else is Wrong?

- Front and back ends can get out of sync
- Interface changes break everything
- Extensions are difficult
- Feature-complete new shims are painful to implement
- No upgrade path
- Too many tasks are implemented in the shim layer (e.g., caching)—no way to implement common code
- No way for caller to know what capabilities are available or missing
- ...

What do we want?

- Writing shims should be straightforward (not much harder than other APIs)
- Using the interface should be straightforward (not much harder than using an unwrapped solver)
- Performance penalty should be minimal
- The interface should provide a useful set of capabilities
- The interface should be extensible
  - New capabilities should be easy to offer through the interface
  - Hooking the solver directly should truly be a last resort
Decouple Interface from Implementation

- In C++, this is a matter of programmer discipline
- Necessary to implement plugins
- Necessary to implement extensions
Decouple Interface from Implementation

- In C++, this is a matter of programmer discipline
- Necessary to implement plugins
- Necessary to implement extensions

- Abstract base class exposes only public interface—defines module semantics
- Concrete implementation object derived from abstract base class
- User asks factory method to return concrete object
- All implementation details and private data are hidden
Modularization

- Clusters of related methods to accomplish tasks
  - Core instance management
  - Presolve
  - Linear algebra, basis management
  - Simplex
  - B&C control
  - ...

- Capabilities managed via a map (name, version, factory)
  - Name defines semantics via abstract base class
  - Loading an interface returns a pointer to a concrete object
  - Upgrades (for developer)
  - Fallbacks (for user)

- Modules associated with an instance need to match solver engine
Inter-Module Communication

- Modules are users too
  - Need access to capabilities
- Incoming module responsible for replacing existing module with the same functionality
  - Extract data
  - Replace capabilities
  - Unload old module
Callbacks

- C engine callback handled by registering a function with specific signature
- C++ engine callback is method derived from virtual base method
- Different engines define different categories of callbacks, provide different types of information, and allow different sets of actions
- OSI could implement a limited set of callback actions (check for abort flag and abort) in a common set of hooks
- Much more than that requires exposing solver-specific interfaces
Parameter Management

- Infinite variety
- Might be able to identify a set of common ones
- Map/table?
- Probably need to expose solver-specific interface for less common settings
Other matters

- Message handling
- Interactions with other COIN-OR components
- ???
Lawyers, Guns, and Money

- Plugins mitigate license compatibility issues
- GPL requires any program that “includes” GPL code must be GPL
- But plugins are not “included” in programs that use them
Questions? Suggestions?