The COIN-OR Open Solver Interface 2.0 Redux

(Zombies Attack!)

Lou Hafer[1] Matthew Saltzman[2]

[1]Department of Computer Science
Simon Fraser University

[2]Department of Mathematical Sciences
Clemson University.

INFORMS Annual Meeting
Austin, Texas
November 10, 2010

What is OSI?

- A cross-solver 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)

A Fractured History

- Original goal for OSI: “solver independent”
  - Too ambitious—can’t reproduce outcomes reliably with different solvers

A Fractured History

- Original goal for OSI: “solver independent”
  - Too ambitious—can’t reproduce outcomes reliably with different solvers
- More achievable: “solver agnostic”
  - Common way to interact with solvers, but don’t enforce solver behavior
  - Success (more or less) at source code level

Source Level Control

- `OsiXxxSolverInterface` derived from `OsiSolverInterface` for various values of `Xxx` ($\texttt{Xxx} \in \{\texttt{Clp}, \texttt{Cpx}, \texttt{Grb}, \ldots\}$).
- User program instantiates concrete `OsiXxxSolverInterface` object
- So `Xxx` needs to be specified in source
- Selection can be controlled by compiler directives (`#ifdef`)
- Specified solver engine library required at link time

Source Level Control

- `OsiXxxSolverInterface` derived from `OsiSolverInterface` for various values of `Xxx` ($\texttt{Xxx} \in \{\texttt{Clp}, \texttt{Cpx}, \texttt{Grb}, \ldots\}$).
- User program instantiates concrete `OsiXxxSolverInterface` object
- So `Xxx` needs to be specified in source
- Selection can be controlled by compiler directives (`#ifdef`)
- Specified solver engine library required at link time
- Solver engine library must ship with user's binary!
  - For example, our CBC binary must ship with CLP as LP solver, because we can't ship other LP solvers.
  - An end user who wants another solver engine must build from source

Shared Library for Solver Engine

- If solver libraries are shared (`libxxx.so` on Unix, DLL on Windows)
  - We can ship without solver libs—they are linked at load time
  - Still need to specify solver in source code
  - User still needs to have solver libs
  - CPLEX isn't shipped as shared lib!

Dynamic Loading of Solver Engine

- `dlopen()`, `dlsym()`, etc., in Linux, other calls in Windows and other Unix systems
- There are cross-platform libraries for this task (GNOME glib, GNU libtool)
- Solver Engine loaded at runtime, not needed at link time
- But still tied to solver in source

Plugins: An Alternate Definition of “Solver Independence”

- Delay decision of what solver to instantiate until runtime
  - User declares abstract interface object (`OsiSolverInterface`)
  - Asks factory object to create a specified concrete implementation
- Solver engine loaded dynamically when implementation object is created
- Now we can ship CBC (say) and let the end user decide at runtime which LP engine to use
- Plugin builder and user need access to solver engine

Plugins: An Alternate Definition of “Solver Independence”

- Delay decision of what solver to instantiate until runtime
  - User declares abstract interface object (`OsiSolverInterface`)
  - Asks factory object to create a specified concrete implementation
- Solver engine loaded dynamically when implementation object is created
- Now we can ship CBC (say) and let the end user decide at runtime which LP engine to use
- Plugin builder and user need access to solver engine
- Now we're breaking source code compatibility...

Plugins: An Alternate Definition of "Solver Independence"

- Delay decision of what solver to instantiate until runtime
  - User declares abstract interface object (`OsiSolverInterface`)
  - Asks factory object to create a specified concrete implementation
- Solver engine loaded dynamically when implementation object is created
- Now we can ship CBC (say) and let the end user decide at runtime which LP engine to use
- Plugin builder and user need access to solver engine
- Now we're breaking source code compatibility...
- ...so we can think about what we would do if we were starting with a clean slate

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?