OSI2 Design and Architecture

This document describes the design and architecture plans for OSI2.

OSI2 defines application programming interfaces (APIs) for common optimisation tasks (maintaining a model, obtaining an optimal solution to the model *etc.*). Libraries which supply objects to implement specific APIs can be dynamically loaded and unloaded by a plugin manager. When a library is loaded, it registers the set of APIs that it supports with the plugin manager.

Clients use the Osi2::ControlAPI interface to load libraries and create objects that implement the APIs supported by the libraries. The same interface is used to destroy objects and unload libraries.

In response to a request to load a plugin library, the Osi2::ControlAPI object interacts with the plugin manager to load and initialise the library. In response to a request for an object implementing a specific API or combination of APIs, the Osi2::ControlAPI object interacts with the plugin manager to invoke the appropriate factory method to produce an object supporting the requested API(s) and hands this object over to the client. Clients interact with these objects according to the interface defined by the API. The Osi2::API class serves as the base class for all APIs and provides some common services for interacting with plugins and mediating interactions between API objects.

There are a number of overarching design goals:

- ❋ Dynamic loading of solvers and other functional modules in support of specific APIs.
- ❋ A lightweight and transparent plugin manager implementation.
- ❋ Small, focused APIs, so that individual APIs are kept to a manageable size and implementation of an object in support of an API is a manageable task.
- ❋ Support for interaction between API objects to provide complex functionality: at minimum, composition and replacement.

API Architecture

OSI2 APIs derive from a common top-level virtual class `Osi2::API` as shown in Figure 1.

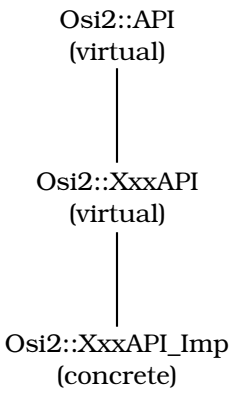

Figure 1: Class Hierarchy for Osi2 APIs

At present, Osi2::API provides support for a single capability: object identification, in the form of storage and retrieval of a pointer to an opaque data block.

Osi2::ControlAPI

The Osi2::ControlAPI API provides a simple user interface for loading and unloading plugin libraries and for creating and destroying objects.

The methods that support plugin library loading (`load`) associate a 'short name' string with a plugin library. Subsequent references to the library for object creation or unloading use only the short name. A default name for the plugin library can be constructed from the short name or the full name can be specified by the user. The full path to the plugin library is constructed using a default path or a path specified by the user. All interaction with the plugin manager is hidden from the client.

To create an object (`createObject`), the client specifies the API(s) that the object should support using the name(s) registered with the plugin manager. An optional parameter allows the user to specify that the object must be created by a particular plugin library.

To destroy an object (`destroyObject`), the client specifies the object to be destroyed. It is possible to use the standard C++ `delete` operator to destroy an object, but clients should use the interface's `destroyObject` method to ensure that any additional actions required by the plugin library are performed.

To unload a library (`unload`), the client specifies the short name of the library.

Because one library's implementation of an API can differ from another library's implementation of the same API, it is necessary that destruction of an object be performed by the same library that created it. The Osi2::ControlAPI implementation uses the identification hook in Osi2::API to store identification information when the object is created. The client need only provide a pointer to the object when asking for it to be destroyed.

Osi2::Osi1API

As a proof-of-concept for the OSI2 architecture, an Osi2::Osi1API API has been implemented. This API allows any implementation of the `OsiSolverInterface` API to be used as a plugin in the OSI2 architecture. Osi2::Osi1API 1 is declared as

```
class Osi1API : public API { ... }
```

and declares all `OsiSolverInterface` methods to be pure virtual.

Shims have been implemented for the clp and glpk solvers using a multiple inheritance strategy. Using the clp shim as an example, the declaration is

```
class Osi1API_ClpHeavy : public Osi1API, public OsiClpSolverInterface { ... }
```

A typical implementation of a method looks like

```
inline void initialSolve() { OsiClpSolverInterface::initialSolve() ; }
```

The only feature requiring nontrivial adaptation is the `ApplyCutsReturnCode` structure returned by `applyCuts`. This is handled with an adapter method as:

```
inline Osi1API::ApplyCutsReturnCode applyCutsPrivate(const OsiCuts &cuts,
 double eff = 0.0)
{
  const OsiSolverInterface::ApplyCutsReturnCode &tmp =
      OsiClpSolverInterface::applyCuts(cuts,eff) ;
  const Osi1API::ApplyCutsReturnCode
      retval(tmp.getNumInconsistent(),
             tmp.getNumInconsistentWrtIntegerModel(),
             tmp.getNumInfeasible(),
             tmp.getNumIneffective(),
             tmp.getNumApplied()) ;
  return (retval) ;
}
```

The Frontier

Further work on OSI2 requires some fundamental decisions about the type of usage that should be supported. In particular, how will the plugin library support interactions between API objects. As a running example, assume that OSI2 defines an Osi2::ModelAPI API for model construction and modification and an Osi2::SolveAPI API for 'solving' the model by finding an optimal solution. Consider the following scenario: The client starts out constructing a model using an object which supports the Osi2::ModelAPI API. Once the model is constructed, the client would like an object which supports the Osi2::SolveAPI API which can be used to solve the model just constructed. There may be multiple iterations of modify and solve, and the user will expect that changes to the model are seen by the solver.

There are any number of questions to ask at this point. Here are two:

- Should the client be expected to work with two distinct (but coordinated) objects, one supporting the Osi2::ModelAPI API and one supporting the Osi2::SolveAPI API? If not, whose responsibility is it to define a single class that supports both APIs?
- Independent of the client's view, should the implementation of this composite capability be two interacting objects or a single object providing both APIs?

The first question seems easier to answer: If a single object supporting multiple APIs is what is desired, it's the client's responsibility to derive the necessary class fron Osi2::API. The client knows the exact set of combinations that is needed. OSI2 could guess at the set of desireable combinations, but combinatorial explosion will defeat any attempt to exhaustively enumerate the possible combinations.

The second question seems much more difficult. Here are two possible models:

- Equal interacting objects can be supported within the OSI2 APIs. One can divide the APIs into two broad classes: data objects and analytic (action) objects. In the running example, we have a single data object (Osi2::ModelAPI) and a single analytic object (Osi2::SolveAPI). Clearly the general case is a many-to-many interaction. An analytic object can process multiple data objects. A data object can be processed by multiple analytic objects.

  As a general algorithm for synchronisation, data objects could keep sequence numbers that track modifications to the data object. Each analytic object could record the most recent sequence number for its associated data objects. At the point where an analytic object is asked to process its associated data objects, it queries each data object to determine its current sequence number and updates itself as necessary. Updates are of most concern where the analytic object maintains an internal copy of the

data object. (True, for instance, for most solvers, which load in a problem and maintain an internal representation that is suited to the needs of the solver's algorithms.)

Data objects could also maintain a list of modifications. This would allow for incremental update of analytic objects, where that was more efficient than bulk update. It also provides a mechanism for an undo/redo capability.

To interface to the single-object client-facing model, the client-facing class would simply maintain pointers to the underlying objects.

❋ Single (monolithic) objects can be supported within the plugin library. Coordination is trivial in the sense that it's the responsibility of the object. No additional support is required within the OSI2 API hierarchy.

As with the multiple interacting object model, the client-facing class need only maintain a (single) pointer to the appropriate object implemented by the plugin library.

A limitation of this model, as opposed to the multiple interacting object model, is that arbitrary composition is not possible (because of the same combinatorial explosion that prevents OSI2 from supporting arbitrary collections of APIs). Each additional API is layered on top of previous APIs.

The advantage of this model is that it captures the great majority of existing solvers. Referring to the running example, most solvers will implement both the Osi2::ModelAPI API and the Osi2::SolveAPI API (a consequence of maintaining a tailored internal representation). Augmenting the Osi2::ModelAPI API with the Osi2::SolveAPI API is a natural (nearly trivial) operation.

A related question is whether the client prefers a new, independent object after an API change or prefers the same object with augmented capabilities. In the context of the running example, should the object that supports the Osi2::ModelAPI API be independent from the object that also supports the Osi2::SolveAPI API. Our design decision is that the default should be to augment the existing object whenever possible. The user will need to explicitly request an independent object.

Another issue is the interface between the client-facing object and the implementing object provided by the plugin library. The client-facing interface is defined by the OSI2 APIs, but the plugin-facing interface need not be the same. Arguably, the plugin-facing interface should be parsimonious, particularly in the monolithic plugin object model, where each additional capability is layered on top of previous capabilities. The danger here is that the work required to implement a minimally functional plugin will become an insurmountable bar (one of the criticisms labelled at the existing `OsiSolverInterface`). This must be balanced against efficiency: if the set of methods is too small and/or primitive, composing complex operations can be made less efficient.

Plugin Manager Architecture

The plugin manager architecture used in OSI2 is broadly adopted from an architecture originally described by G. Sayfan [1].

There is one (static) instance of the plugin manager, returned by a call to `getInstance`. The plugin manager provides certain services and information to plugins, specified in the `PlatformServices` class. Currently, a `PlatformServices` object specifies a registration method that allows a plugin to register with the plugin manager, and it provides a general-purpose service method that the plugin can invoke to request other services from the manager[1].

A deficiency in the original architecture is that it assumes a very structured interaction between plugin libraries and the client program.

[1] No general-purpose services are implemented at present.

❋ There is no provision for maintaining state to manage a library or individual APIs supported by the library.

The plugin libraries used for OSI2 are in general not stateless, hence the interface between the plugin manager and the library has been expanded to allow a reference to a management object to be returned to the plugin manager during library initialisation and passed back to the plugin library whenever the library is asked to create an object to implement an API.

❋ There is no provision to associate registered APIs with a particular plugin library, hence there is no way to unload a single plugin library.

The OSI2 architecture supports loading and unloading of individual plugin libraries throughout execution. It also allows a client to specify that a particular plugin library should be used to satisfy a request to create an object implementing some API. To support these capabilities, the plugin manager maintains bookkeeping information to associate registered APIs with a specific library.

The working sequence to load a plugin library, create an object, destroy an object, and unload the library, is described below. The first two steps (loading the shared library, initialising the plugin and registering APIs) are encapsulated in `PluginManager::loadOneLib`, so that load and initialisation of a plugin library is an atomic action from the point of view of a client.

1. Load the plugin library.

   Currently, this is accomplished by a call to the static `load` method in the `DynamicLibrary` class, which will invoke the system `dlopen` method[2]. The plugin manager supports specification of a default plugin directory[3].

2. Initialise the plugin library and register APIs implemented by the library.

   Given a `DynamicLibrary` object, `getSymbol` is used to retrieve the plugin library's initialisation (`InitFunc`) method, which must be named `initPlugin`. This method is then invoked, passing a `PlatformServices` structure as a parameter. The `InitFunc` is responsible for getting the plugin library ready to work and registering its capabilities (APIs).

   For each API supported by the plugin library, a `RegisterParams` object is created specifying the identifying string for the API, create (`CreateFunc`) and destroy (`DestroyFunc`) methods, descriptive information (version and language), and an associated object to hold any state information the library wishes to maintain for objects supporting this API. The `RegisterParams` object is handed back to the plugin manager as a parameter to the registration (`RegisterFunc`) method supplied in the `PlatformServices` object. The `RegisterFunc` method adds the `RegisterParams` object to an appropriate temporary structure in the plugin manager according to the API identity string (exact match or wildcard).

   Once all APIs are registered, the `InitFunc` method stores a pointer to the library's control object in the `PlatformServices` structure and returns to the plugin manager. The return value is a pointer to the library's cleanup (`ExitFunc`) method.

   At this point it is no longer possible for initialisation of the plugin library to fail. The plugin manager copies the API registration information from the temporary structures to permanent structures to complete the load of the library.

[2]Support for Windows will be provided by has not been tested.

[3]We need to add infrastructure to check the environment. We might also want to allow for a search list of directories. A design decision is whether this capability should move to Osi2::ControlAPI or be replicated in Osi2::ControlAPI.

3. Ask the plugin manager for an object that implements an API.

   Using the plugin manager's `createObject` method, the client asks for an object that implements a particular API by specifying the identity string registered for that API. The `createObject` method searches the maps held by the plugin manager looking for a suitable match. If a match is found, an `ObjectParams` object is created, including the API requested, a `PlatformServices` object, and a pointer to the control object for the API. (A pointer to the control object for the library is passed in the `PlatformServices` object.) The `ObjectParams` object is passed as a parameter to the `CreateFunc` that the plugin registered for this type of API. The `CreateFunc` returns a pointer to an object that implements the requested API.

   The client can specify that the API must be produced by a specific plugin library. This allows full client control when more than one library can produce an object that implements an API.

4. Ask the plugin manager to destroy an object.

   It is possible to simply invoke `delete` on an object, but use of the plugin manager's `destroyObject` method allows some additional functionality. The `destroyObject` method invokes the plugin library's `DestroyFunc` for the object, passing the same `ObjectParams` parameter block used for `createObject`. This allows the plugin library to perform additional management at the API and library levels, if necessary.

   The plugin manager requires that the request to destroy the object specify the plugin library that produced the object. This is necessary to ensure that the request to destroy an object goes to the plugin library that created it.

5. Ask the plugin manager to unload a library.

   The `unloadOneLib` method will scan the plugin manager's API registration maps and remove all entries associated with the library to be removed. It will then invoke the library's `ExitFunc`. Finally, the destructor for the dynamic library object will invoke `dlclose` to unload the library.

A given plugin library can supply multiple APIs, and a client can ask for an object that implements some combination of those APIs by specifying multiple API identity strings in a request to create an object. Any given request to a plugin library can return only one object. It's the business of the plugin library to keep track of what particular APIs or combination of APIs it supports, and what objects it's given out, to the extent that this is necessary.

There is no distinct 'capability inquiry' method for the plugin manager[4]. Given a string specifying one or more APIs to be supported by an object, the plugin manager first looks to see if some plugin library has registered exactly that API string. If not, the plugin manager looks to see if any libraries have registered wildcard capability. For each such plugin library, the plugin manager sends the API string to the library. If the library returns an object, the plugin manager registers the API string so that future requests will be satisfied with an exact match.

Shims

The OSI2 term for an implementation of an OSI2 APIs is a 'shim'. Two shim architectures have been tried, called 'heavy' and 'light'.

A 'heavy' shim includes any underlying libraries in the link that creates the shared library for the shim. A 'light' shim dynamically loads additional libraries as part of its initialisation.

As an example, there are light and heavy shims for the clp solver. The heavy shim links includes `libClp` as part of the link. The light shim dynamically loads `libClp` as part of its initialisation[5]. The primary

[4] At the time of writing, it's looking like this would be a useful extension.

advantage to the heavy shim is that C++ classes can be used as normal. The disadvantage is that the shim cannot be distributed independently of the underlying support libraries.

The primary advantage to the light shim is that the shim can be distributed independently of the underlying libraries. The disadvantage is that all library methods must be explicitly loaded with `getSymbol` (`dlsym`) and are subject to all the restrictions of 'C' linkage.

The Frontier

What is the correct model for providing services to a plugin library? Here are three models that appear to be useful:

- Use the existing service function hook, `InvokeServiceFunc`, provided by the plugin manager. This seems the appropriate route for stateless services, particularly services where efficiency isn't a dominant concern. Logging would be an example of such a service.
- A plugin library could request from the plugin manager an object that implements a particular API. These might be supplied by some other plugin library, or they could be 'static' plugins compiled into the plugin manager. This would be a good model for services that require state, or where there will be frequent use and efficiency is a concern. Better to hand out an object that can be used repeatedly without further interaction with the plugin manager.
- A service can be implemented in an API. For some services, it will be necessary to insert functionality into calls from the client to the plugin. This can be handled by placing implementation code in the API definition class, where it can execute before or after invoking methods provided by the plugin library.

  Caching can serve as an example. A caching service that maintains a data structure embedded in the API could minimise the number of times that it is necessary to call the underlying plugin library. This requires code in the API to manage the embedded data structure. A ModelAPI, for example, could maintain an embedded representation of the model which it could modify incrementally in response to calls from the client, completely avoiding calls to the underlying library. At some point, the client would request that the underlying library process the model. The caching code could push the revised model to the underlying library prior to invoking the library's processing method. (To couch this in terms of specific calls, consider a solver plugin. An API could maintain a cached copy of a constraint system, trapping modifications within the API. When the client invokes an optimisation method, the API would note that the model was modified and push the updated model to the solver before invoking the solver's optimisation method.)

References

[1] G. Sayfan. Building your own plugin framework, parts 1 – 5. *Dr. Dobbs*, November 2007. Online at http://drdobbs.com/cpp/204202899 (first of five parts).

[5] At present, this is accomplished with direct calls to methods in the `DynamicLibrary` class. It's not yet clear whether this is the appropriate choice. The alternative is to use the plugin manager, but one of the major services provided by the plugin manager (registration of APIs) may not be needed here. If API registration is used, the relevant code in the plugin manager will need to be made reentrant. Currently, initialising a library from within the initialisation of another library will pollute the temporary structures used to hold API registration.