Writing a GMAT Plug-in

Darrel J. Conway
Thinking Systems, Inc.

July 24, 2008

Abstract

GMAT is designed to accept new user modules through shared libraries imported at run time. This document describes the process of building and using a GMAT plug-in. An example is provided that implements a modified solar radiation pressure force appropriate for use as a starting point in building a solar sailing model.

Contents

1 Overview

The General Mission Analysis Tool (GMAT) contains many classes used to model spacecraft missions. While the system contains most of the components anticipated by the project team, we also recognized from the start that GMAT could not contain everything that every user would need. Therefore, part of the design philosophy behind GMAT is the development of user classes that generalize functionality in base classes that can be extended to meet future needs of the user community.

The user configurable components of GMAT are all derived from a general purpose base class, GmatBase, which defines interfaces used throughout the system to access common properties of the user classes. This base class defines a framework used by GMAT's engine to manage the objects used when modeling a mission.

Classes derived from GmatBase are specialized to model different aspects of the mission. Classes located deeper in the GmatBase class hierarchy are, in general, more specialized than those at the higher levels. Details of GMAT's class structure can be found in the GMAT Architectural Specification[1], or by running GMAT's source code through the Doxygen source code documentation generator[2]. The classes derived from GmatBase are called GMAT's user classes. GMAT also contains classes that do not share this ancestor; those components make up the elements of the engine, user interfaces, and other elements of GMAT's framework that provides the infrastructure for the system.

Builds of GMAT made using development source code after June 25, 2008 [1] have the ability to load shared libraries at run time and retrieve new user objects from these libraries. The approach taken for this capability was built on a prototype extension implemented at Thinking Systems in April, 2008 to meet some specific mission needs and documented as an extension to GMAT[3]. This document explains how to use the plug-in extensions to add new capabilities to GMAT. A specific example – the addition of a new force for GMAT's force model – is described in some detail, with emphasis on the features necessary for incorporation into GMAT at run time.

We'll begin by looking at the steps needed to construct a GMAT plug-in. Once these steps have been described, the design of the example plug-in code – a basic solar sail model for GMAT's force model – is described, along with descriptions of the pieces needed to incorporate the new model through the plug-in interface. Finally, the steps needed to tell GMAT about the new plug-in are presented. The document closes with appendices that describe details of the development process for Eclipse users, and that contain full source code listings for the six files that comprise the plug-in.

2 Plug-in Programmatic Requirements

GMAT's Plug-in capabilities apply to the classes derived from the GmatBase class. Instances of these classes are constructed using GMAT's Factory subsystem. Plug-in authors capitalize on this design by creating custom factories designed to support the new components that they are adding to the system. A GMAT plug-in is a shared library, linked against a shared library build of GMAT's base code, that contains the class code for the new capability, one or more supporting Factory for the new components, and a set of three C-style interface functions that are accessed by GMAT to load the plug-in. Each of these plug-in elements is introduced in this section, starting from the build requirements for GMAT, proceeding through the interface functions and factory requirements, and finishing with the actual new component that is being added. The next section of this document describes a sample plug-in of to illustrate the process.

2.1 Building GMAT for Plug-in Use

GMAT plug-ins create classes that are derived from classes in GMAT's base code. The plug-in needs to be linked against that code in order to use the capabilities of the base classes, and to build the complete derived objects. In order to do this, the plug-in library needs to be linked against the base code that will be run when GMAT runs. There are two ways this requirement can be accomplished. The code can be built with

[1]The development code for this capability will be merged into the trunk code when the next merge occurs.

all of the required classes as part of the plug-in library. That approach makes the plug-in much larger than necessary; in general, GmatBase subclasses need the code from the utility and foundation folders, the folders related to the type of subclass that is being built, and all of the referenced classes used by the members of its hierarchy. In addition, these elements need to stay synchronized with GMAT's base code as new features are implemented and as anomalies are corrected in the base code.

The preferred approach to plug-in development is to build GMAT's base code as one or more shared libraries. GMAT's build control file, BuildEnv.mk, has a setting for this option. If you add this line to the file:

```
SHARED_BASE = 1
```

and then clean and build GMAT, the make files will build the base code as a shared library – named libGmatGui – that can be used for plug-in development. Once you have built GMAT this way, you are ready to start coding your plug-in.

2.2 The Plug-in Interface Functions

GMAT accesses new user classes contained in plug-in libraries by calling three methods in the plug-in: GetFactoryCount(), GetFactoryPointer(), and SetMessageReceiver(). These functions are used as the entry point into the plug-in components. They are defined as follows:

- `Integer GetFactoryCount()`: This function reports the number of Factory classes that are contained in the plug-in. The current implementation of GMAT requires that factories only support a single core type because of an implementation limitation in the FactoryManager, so larger plug-in libraries may need more than one supporting factory.
- `Factory* GetFactoryPointer(Integer index)`: This function retrieves Factory pointers from the plug-in. Once GMAT know the number of factories in the library, it calls this function to retrieve the contained factories one at a time.
- `void SetMessageReceiver(MessageReceiver* mr)`: Messages posted in GMAT are all sent to a MessageReceiver. This function is used to set the MessageReceiver for a plug-in.

  Note: The design that required this method incorporated base code in the plug-in library. Now that the base code can – and should be – built as a shared library, this function may no longer be needed, and may be removed at a later date.

The code in Section B.1 shows one implementation of these methods.

2.3 The Custom Factory

GMAT's Factory subsystem is described in some detail in the Architectural Specification[1]. GMAT uses this subsystem to create user objects that are needed to run a mission. The class diagram for the subsystem is shown in Figure 1.

The Factory base class defines the interface used to create user objects. It includes subclass specific interfaces for the core user class types, as can be seen from this portion of the class definition:

```
class GMAT_API Factory
{
public:
   // method to return objects as generic type
   virtual GmatBase*        CreateObject(const std::string &ofType,
                                         const std::string &withName = "");
```

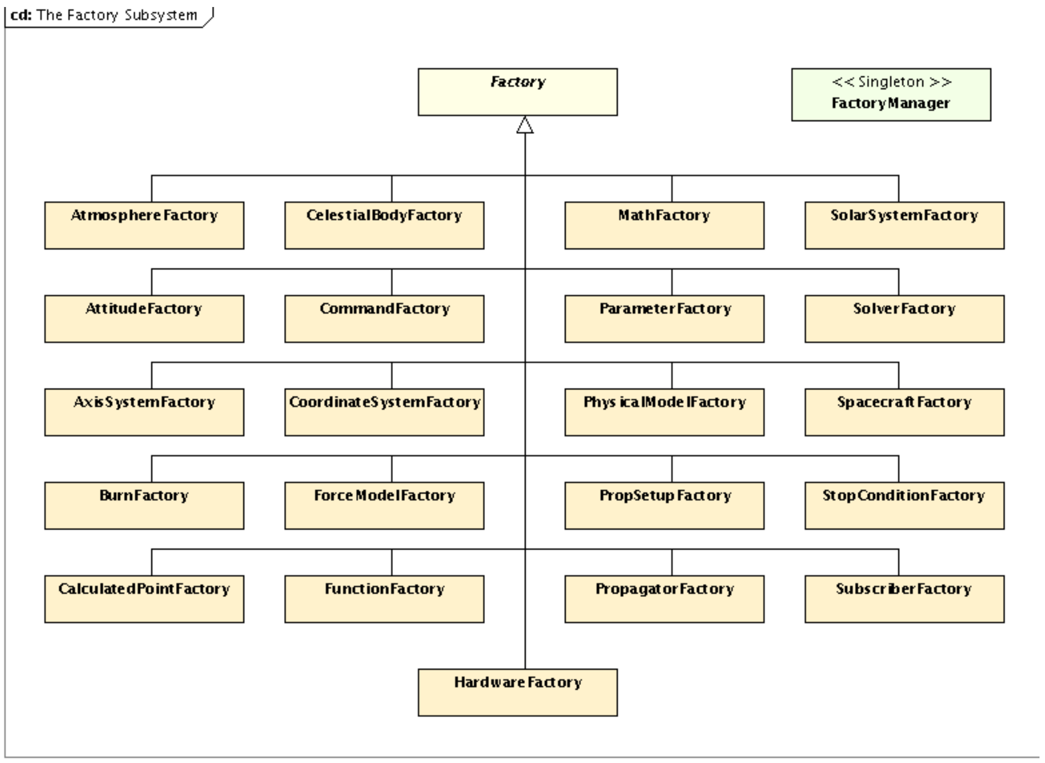

Figure 1: The Factory Subsystem

```
// methods to return objects of specified types
virtual SpaceObject*     CreateSpacecraft(const std::string &ofType,
                                          const std::string &withName = "");
virtual Propagator*      CreatePropagator(const std::string &ofType,
                                          const std::string &withName = "");
virtual ForceModel*      CreateForceModel(const std::string &ofType,
                                          const std::string &withName = "");
virtual PhysicalModel*   CreatePhysicalModel(const std::string &ofType,
                                             const std::string &withName = "");
virtual PropSetup*       CreatePropSetup(const std::string &ofType,
                                         const std::string &withName = "");
virtual Parameter*       CreateParameter(const std::string &ofType,
                                         const std::string &withName = "");
virtual Burn*            CreateBurn(const std::string &ofType,
                                    const std::string &withName = "");
virtual StopCondition*   CreateStopCondition(const std::string &ofType,
                                             const std::string &withName = "");
virtual CalculatedPoint* CreateCalculatedPoint(const std::string &ofType,
                                               const std::string &withName = "");
virtual CelestialBody*   CreateCelestialBody(const std::string &ofType,
                                             const std::string &withName = "");
virtual SolarSystem*     CreateSolarSystem(const std::string &ofType,
                                           const std::string &withName = "");
virtual Solver*          CreateSolver(const std::string &ofType,
                                          const std::string &withName = "");
virtual Subscriber*      CreateSubscriber(const std::string &ofType,
                                          const std::string &withName = "",
                                          const std::string &fileName = "");
virtual GmatCommand*     CreateCommand(const std::string &ofType,
                                       const std::string &withName = "");
virtual AtmosphereModel* CreateAtmosphereModel(const std::string &ofType,
                                               const std::string &withName = "",
                                               const std::string &forBody = "Earth");
virtual Function*        CreateFunction(const std::string &ofType,
                                        const std::string &withName = "");
virtual Hardware*        CreateHardware(const std::string &ofType,
                                        const std::string &withName = "");
virtual AxisSystem*      CreateAxisSystem(const std::string &ofType,
                                          const std::string &withName = "");
virtual CoordinateSystem* CreateCoordinateSystem(const std::string &ofType,
                                                 const std::string &withName = "");
virtual MathNode*        CreateMathNode(const std::string &ofType,
                                        const std::string &withName = "");
virtual Attitude*        CreateAttitude(const std::string &ofType,
                                        const std::string &withName = "");
...
```

When you have decided what type of new component you will be implementing, select the appropriate factory method from this group and implement it to call your component's constructor. Each of the factory classes shown in Figure 1 are available for browsing in the src/base/factory folder of GMAT's source tree, so you should be able to select an appropriate Factory as a starting point for your custom Factory. Section B.2 shows sample code for a Factory supporting a new PhysicalModel class.

2.4 The New Feature

The purpose of all of the support code described above is, of course, the implementation of a new user component. GMAT provides a rich set of classes that can be used as a starting point for your new component. If you use an existing class as your base class for the new feature, you'll find that there is support for most or all of the plug-in component in the core GMAT engine already, significantly reducing your integration efforts[2]. The next section describes the design and implementation of one such component – a custom force used in GMAT's force model.

3 An Example

This section presents the design for a complete GMAT plug-in library. The example shown here is a new force for the force model. The new force used for this example is a directed solar radiation pressure force, appropriate for solar sailing. Appendix B contains the complete source code listing for this plug-in library.

For the purposes of this example, the spacecraft attitude will be used to calculate the direction of the normal to the reflecting surface, and thus the direction of of the force vector. More specifically, for this example the spacecraft's x-axis, as specified by its attitude, will be treated as the normal, $\hat{\mathbf{n}}$, to the surface that the light hits. The spacecraft's coefficient of reflectivity, $1 <= C_r <= 2$, determines the amount of light that reflects off of the spacecraft; $C_r = 1$ means that all of the incident light is absorbed, while $C_r = 2$ means that the light is all reflected.

[2]The GMAT developers have endeavored to keep the internal interfaces into GMAT as generic as possible. However, it may be that we have overlooked a feature that is critical yo your new component. If you think that may have happened, please send additional interface requests to the GMAT team so that we can provide assistance and, if needed help to update GMAT's interfaces to address your needs.

The following sections define the new force, describe the class used to model the force, and then present the code needed to add the new force to GMAT using the plug-in architecture.

3.1 The Physics of the SolarSail Model

We'll begin by describing the model implemented in the code. The vector from the spacecraft to the Sun, e_s, makes an angle θ with the surface normal $\hat{\mathbf{n}}$. The absorbed radiation applies a force f_a directed opposite to the sun vector. The reflected radiation applies a force directed anti-parallel to the normal vector, $\hat{\mathbf{n}}$.

The magnitude of each of these forces is equal to the incident radiation pressure P_r multiplied by the incident surface area, A, and then adjusted to take into account the amount of light reflected or absorbed. The force for absorbed light is given by

$$\mathbf{F}_{abs} = -(1-\varepsilon)P_r A\cos(\theta)\hat{\mathbf{e}}_s \tag{1}$$

while that of the reflected light is given by

$$\mathbf{F}_{ref} = -2\varepsilon P_r A\cos^2(\theta)\hat{\mathbf{n}} \tag{2}$$

The constant ε in these equations is the percentage of the incident light that is reflected from the surface, and is related to the coefficient of reflectivity through the equation

$$\mathbf{C}_r = 1+\varepsilon \tag{3}$$

Finally, the factor of 2 in equation 2 accounts for the reflectance effect of Newton's third law. The cosine term in this equation is squared because the reflected light applies its force exclusively in the anti-normal direction; the force components parallel to the reflecting surface from the incoming and outgoing light cancel out.

The incident radiation pressure, P_r, is a function of the distance from the Sun to the spacecraft. Spacecraft closer to the Sun experience a larger incident radiation pressure than those further away. This effect follows an inverse square relationship; if the solar radiation pressure at one astronomical unit from the Sun, R_{AU}, is written as P_{AU}, the radiation pressure at an arbitrary distance r_s is given by

$$P_r = P_{AU}\left(\frac{R_{AU}}{r_s}\right)^2 \tag{4}$$

Putting all of these pieces together, the force implemented in this plug-in is given by

$$\mathbf{F}_{sail} = -P_{AU}\left(\frac{R_{AU}}{r_s}\right)^2 A\cos\theta\{(1-\varepsilon)\hat{\mathbf{e}}_s + 2\varepsilon\cos\theta\hat{\mathbf{n}}\} \tag{5}$$

GMAT's equations of motion are expressed in terms of derivatives of the position vectors. That means that the function that models a force in GMAT, `GetDerivatives()`, needs to express the effect of the force in terms of an acceleration. The Spacecraft model contains a reflectivity coefficient, C_r, which matches the coefficient in equation 3. Using equation 3 and the relationship $\mathbf{F} = m\mathbf{a}$, the resulting acceleration is

$$\mathbf{a}_{sail} = -P_{AU}\left(\frac{R_{AU}}{r_s}\right)^2 \frac{A}{m}\cos\theta\{(2-C_r)\hat{\mathbf{e}}_s + 2(C_r-1)\cos\theta\hat{\mathbf{n}}\} \tag{6}$$

This equation is encapsulated in the class, SolarSail, described below.

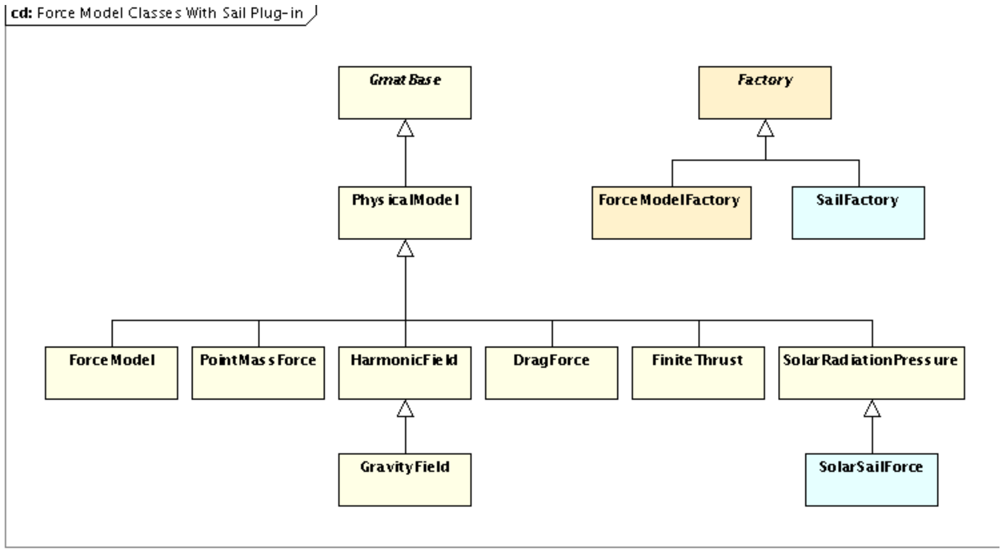

Figure 2: The Solar Sail Model Components. The plug-in components are shown in blue.

3.2 The SolarSail Class

GMAT's force model classes are all implementations of a base PhysicalModel class. The SolarSail plug-in uses many of the features and structures already implemented in the SolarRadiationPressure class, one of the members of the force model subsystem. The SolarSail class uses its own factory, implemented as the Factory component of the plug-in library. These additions are shown in the ForceModel class hierarchy, shown in Figure 2.

The solar sail force uses many of the same calculations as are performed for GMAT's solar radiation pressure model. For that reason, the SolarSailForce class is derived from the SolarRadiationPressure class. The new class does need to implement a different acceleration model, so it overrides the GetDerivatives() method to provide accelerations as described above. It also provides implementations for the four C++ default methods: the constructor, copy constructor, destructor, and assignment operator. The new force has data structures that need to be initialized, so the Initialize() method is overridden (and calls the SolarRadiationPressure::Initialize() method internally). GMAT's ForceModel class contains a method, IsUserForce(), which is called to determine how to handle scripting for forces added by users. This method is overridden to report the new force as a user force. Finally, the Clone() methods is overridden so that GMAT can make copies of the new force from a GmatBase pointer.

The full source code for the SolarSailForce class is included in Appendix B, Section B.3.

3.3 The SailFactory and Interface code

The SailFactory is used to create new instances of the SolarSailForce. The code – shown in Section B.2 – is identical to many of the core factories found in GMAT's src/base/factory file folder. There are three sections specific to the SolarSailForce: the CreatePhysicalModel() method:

```
PhysicalModel* SailFactory::CreatePhysicalModel(const std::string &ofType,
                                                const std::string &withName)
{
   if (ofType == "SailForce")
```

```
         return new SolarSailForce(withName);

      return NULL;
   }
```

and the code in the constructor and copy constructor that populates the list of creatable object names. That code has this form:

```
      if (creatables.empty())
      {
         creatables.push_back("SailForce");
      }
```

The rest of the factory code fills out the required elements: the constructors, assignment operator, and destructor, as required in GMAT's coding standards[4].

The code in the interface functions is nearly as transparent. There are three C-style functions that are used in the plug-in implementation, shown in Listing B.1. The GetFactoryCount() method returns the number of factories in the plug-in – one (1) for this example. GetFactoryPointer() creates an instance of the SailFactory and returns it to GMAT when it is called with an input index of 0 (indicating the first factory in the plug-in). SetMessageReceiver() is used by GMAT to ensure that the MessageReceiver pointer in the plug-in is set to the active MessageReceiver for the executable[3]

Once the code described above is in place, it can be compiled into a shared library that meets GMAT's plug-in requirements. The steps needed for the compilation process for Eclipse users are described in Appendix A.

3.4 Adding the Plug-in to GMAT

Once you have built the plug-in library described above, place the resulting code in the folder that contains your GMAT executable. The plug-in will become available in GMAT if you add a line to your GMAT startup file identifying the library as a plug-in. The required line looks like this for a plug-in named libSolarSail:

```
   PLUGIN               = libSolarSail
```

(The actual plug-in file name depends on your operating system – on Windows, the file name would be "libSolarSail.dll"; on Linux, it would be "libSolarSail.so", and on Mac, "libSolarSail.dylib".)

Once this line is in place in your startup file, GMAT will attempt to load the plug-in when it is started.

4 Summary

This document described the code necessary to construct a GMAT plug-in. It described the three core elements of a plug-in: the new functionality, the supporting factory, and the interface code used to load the plug-in features. An example, implemented in the code in the appendices, was described that illustrated the addition of a force for GMAT's force model. Finally, the steps needed to tell GMAT about the new functionality were provided.

Please feel free to contact the author of this document with questions or requests for clarification by sending an e-mail to djc@thinksysinc.com.

[3]SetMessageReceiver() may be removed in future builds. Early versions of the plug-in code required independent builds of the MessageInterface code, and needed to set the MessageReceiver pointer for that code for each plug-in. When plug-in libraries are built that use the shared library base code for GMAT, that step is not necessary.

A Eclipse Configuration for the Solar Sail Plug-in

This appendix outlines the steps used to build the plug-in using Eclipse.

A.1 Create the Project

1. Open Eclipse
2. Right-click in the Project Explorer, and add a New -> C++ Project.
3. On the new project dialog, set the project name to "SolarSail" and the project type to "Shared Library."
4. Press the "Next" button.
5. Set the desired configurations. (I use the defaults.)
6. Press the "Finish" button.

This gives you a project to use to build the plug-in library. Next you need to configure the project to use the GMAT shared base library.

A.2 Configure the project

1. Right click on the SolarSail node, select "New -> Folder," and add a directory named "src"
2. Add three folders under the src folder, named "plugin," "sail," and "factory." These are the folders that will contain the six source files needed for the plug-in.
3. Place the source files in the corresponding folders. The source files are available on GMAT's Wiki pages.

The next step is configuring Eclipse so that the C++ compiler and linker know where to find the header files and libraries needed to build the plug-in.

1. Right-click on the project node in the Project Explorer, and select the "Properties" option.
2. On the Properties dialog, select the "C/C++ Build — Settings" option from the panel on the left.
3. On the right panel, select "GCC C++ Compiler — Directories."
4. Press the "Add" button on the Include paths toolbar.
5. Press the "Workspace..." button on the "Add directory path" pop-up window, and navigate to the SolarSail/src/factory folder on the "Folder selection" pop-up dialog. Press the OK button to accept this path. Then press the OK button on the "Add directory path" dialog to send the path to the Eclipse settings panel.
6. Repeat the previous step to add the plugin and sail directory folders to your configuration.
7. Use this procedure to add the following folders from your GMAT build to the include directories:
    - src/base/include
    - src/base/util
    - src/base/foundation
    - src/base/factory
    - src/base/forcemodel

- src/base/solarsys
- src/base/spacecraft
- src/base/coordsys
- src/base/attitude

<Add Windows specifics – linking against the dll, and anything else uncovered when testing there.>

This completes the steps needed to build the plug-in. Right-click on the project node and select the "Build Project" option. Eclipse will build a new shared library for you, and place it in the Debug folder. Move that library into the GMAT executable folder and edit the startup file to use the new code.

B Source Code for the Plug-in

This section contains the SolarSail plug-in code in its entirety. The plug-in is built using six files, grouped as a header file and implementation. The generic interface functions are implemented in the plugin/GmatPluginFunctions code. The Factory supporting the new plug-in class in implemented in the factory/SailFactory code. The sail/SolarSailForce code implements the new force.

B.1 The Plug-in interface Code

B.1.1 src/plugin/GmatPluginFunctions.hpp

```
//$Id: GmatPluginFunctions.hpp,v 1.1 2008/07/03 19:15:33 djc Exp $
//------------------------------------------------------------------------------
//                             GmatPluginFunctions
//------------------------------------------------------------------------------
// GMAT: General Mission Analysis Tool
//
// **Legal**
//
// Developed jointly by NASA/GSFC and Thinking Systems, Inc. under contract
// number NNG06CA54C
//
// Author: Darrel J. Conway, Thinking Systems, Inc.
// Created: 2008/07/03
//
/**
 * Definition for library code interfaces.
 *
 * This is sample code demonstrating GMAT's plug-in capabilities.
 */
//------------------------------------------------------------------------------

#ifndef GMATPLUGINFUNCTIONS_H_
#define GMATPLUGINFUNCTIONS_H_

#include "Factory.hpp"

class MessageReceiver;
```

```
/**
 * This code defines the C interfaces GMAT uses to load a plug-in library.
 */
extern "C"
{
   Integer     GetFactoryCount();
   Factory*    GetFactoryPointer(Integer index);
   void        SetMessageReceiver(MessageReceiver* mr);
};


#endif /*GMATPLUGINFUNCTIONS_H_*/
```

B.1.2 src/plugin/GmatPluginFunctions.cpp

```
//$Id: GmatPluginFunctions.cpp,v 1.1 2008/07/03 19:15:33 djc Exp $
//------------------------------------------------------------------------------
//                             GmatPluginFunctions
//------------------------------------------------------------------------------
// GMAT: General Mission Analysis Tool
//
// **Legal**
//
// Developed jointly by NASA/GSFC and Thinking Systems, Inc. under contract
// number NNG06CA54C
//
// Author: Darrel J. Conway, Thinking Systems, Inc.
// Created: 2008/07/03
//
/**
 * Implementation for library code interfaces.
 *
 * This is sample code demonstrating GMAT's plug-in capabilities.
 */
//------------------------------------------------------------------------------

#include "GmatPluginFunctions.hpp"
#include "MessageInterface.hpp"

#include "SailFactory.hpp"

extern "C"
{
   //---------------------------------------------------------------------------
   // Integer GetFactoryCount()
   //---------------------------------------------------------------------------
   /**
    * Method used to determine how many factories the plug-in library supports.
    *
    * @return The number of factories in the plug-in library.
    */
```

```
//------------------------------------------------------------------------------
Integer GetFactoryCount()
{
   return 1;
}

//------------------------------------------------------------------------------
// Factory* GetFactoryPointer(Integer index)
//------------------------------------------------------------------------------
/**
 * Retrieves an instance of the factory that corresponds to the input integer
 *
 * This function is used to build all of the Factory instances needed by the
 * plug-in library.
 *
 * @param index Zero-based index identifying which factory is requested
 *
 * @return A pointer to the factory, or NULL if there is no factory with the
 *         specified index
 */
//------------------------------------------------------------------------------
Factory* GetFactoryPointer(Integer index)
{
   Factory* factory = NULL;

   switch (index)
   {
      case 0:
         factory = new SailFactory;
         break;

      default:
         break;
   }

   return factory;
}

//------------------------------------------------------------------------------
// void SetMessageReceiver(MessageReceiver* mr)
//------------------------------------------------------------------------------
/**
 * Method used to pass the active MessageReceiver to the plug-in.
 *
 * If the code in your plug-in writes to GMAT's message interfaces, implement
 * this method so that GMAT can set the output streams so that your messages
 * display correctly.
 *
 * @param mr The MessageReceiver.
 *
 * @note This method is marked to be deprecated, and may be removed in future
```

```
    * builds.
    */
   //------------------------------------------------------------------------------
   void SetMessageReceiver(MessageReceiver* mr)
   {
      MessageInterface::SetMessageReceiver(mr);
   }
};
```

B.2 The Factory Code

B.2.1 src/factory/SailFactory.hpp

```
//$Id: SailFactory.hpp,v 1.1 2008/07/03 19:15:33 djc Exp $
//------------------------------------------------------------------------------
//                              SailFactory
//------------------------------------------------------------------------------
// GMAT: General Mission Analysis Tool
//
// **Legal**
//
// Developed jointly by NASA/GSFC and Thinking Systems, Inc. under contract
// number NNG06CA54C
//
// Author: Darrel Conway
// Created: 2008/07/03
//
/**
 * Declaration code for the SailFactory class, which creates objects that extend
 * the SRP model for solar sailing.
 *
 * This is sample code demonstrating GMAT's plug-in capabilities.
 */
//------------------------------------------------------------------------------


#ifndef SailFactory_hpp
#define SailFactory_hpp


#include "Factory.hpp"


/**
 * SailFactory is a factory plug-in that creates SolarSailForce objects for use
 * in GMAT's force model.
 */
class SailFactory : public Factory
{
public:
   virtual PhysicalModel* CreatePhysicalModel(const std::string &ofType,
                                const std::string &withName /* = "" */);
   // default constructor
   SailFactory();
   // constructor
   SailFactory(StringArray createList);
   // copy constructor
   SailFactory(const SailFactory& fact);
   // assignment operator
   SailFactory& operator=(const SailFactory& fact);

   virtual ~SailFactory();

};

#endif // SailFactory_hpp
```

B.2.2 src/factory/SailFactory.cpp

```
//$Id: SailFactory.cpp,v 1.1 2008/07/03 19:15:33 djc Exp $
//------------------------------------------------------------------------------
//                            SailFactory
//------------------------------------------------------------------------------
// GMAT: General Mission Analysis Tool
//
// **Legal**
//
// Developed jointly by NASA/GSFC and Thinking Systems, Inc. under contract
// number NNG06CA54C
//
// Author: Darrel Conway
// Created: 2008/07/03
//
/**
 * Implementation code for the SailFactory class, which creates objects that
 * extend the SRP model for solar sailing.
 *
 * This is sample code demonstrating GMAT's plug-in capabilities.
 */
//------------------------------------------------------------------------------

#include "gmatdefs.hpp"
#include "SailFactory.hpp"
#include "SolarSailForce.hpp"

#include "MessageInterface.hpp"


//---------------------------------
//  public methods
//---------------------------------
```

```
//------------------------------------------------------------------------------
//  PhysicalModel* SailFactory::(const std::string &ofType,
//                               const std::string &withName)
//------------------------------------------------------------------------------
/**
 * This method creates and returns a PhysicalModel object.
 *
 * @param <ofType> type of PhysicalModel object to create and return.
 * @param <withName> the name for the newly-created PhysicalModel object.
 *
 * @return A pointer to the created object.
 */
//------------------------------------------------------------------------------
PhysicalModel* SailFactory::CreatePhysicalModel(const std::string &ofType,
                                    const std::string &withName)
{
   if (ofType == "SailForce")
      return new SolarSailForce(withName);

   return NULL;
}


//------------------------------------------------------------------------------
//  SailFactory()
//------------------------------------------------------------------------------
/**
 * This method creates an object of the class SailFactory.
 * (default constructor)
 */
//------------------------------------------------------------------------------
SailFactory::SailFactory() :
   Factory     (Gmat::PHYSICAL_MODEL)
{
   if (creatables.empty())
   {
      creatables.push_back("SailForce");
   }
}

//------------------------------------------------------------------------------
//  SailFactory(StringArray createList)
//------------------------------------------------------------------------------
/**
 * This method creates an object of the class SailFactory.
 *
 * @param <createList> list of creatable solver objects
 *
 */
//------------------------------------------------------------------------------
SailFactory::SailFactory(StringArray createList) :
   Factory(createList, Gmat::PHYSICAL_MODEL)
{
}


//------------------------------------------------------------------------------
//  SailFactory(const SailFactory& fact)
//------------------------------------------------------------------------------
/**
 * This method creates an object of the class SailFactory.  (copy constructor)
 *
 * @param <fact> the factory object to copy to "this" factory.
 */
//------------------------------------------------------------------------------
SailFactory::SailFactory(const SailFactory& fact) :
    Factory     (fact)
{
   if (creatables.empty())
   {
      creatables.push_back("SailForce");
   }
}


//------------------------------------------------------------------------------
//  CommandFactory& operator= (const CommandFactory& fact)
//------------------------------------------------------------------------------
/**
 * SailFactory operator for the SailFactory base class.
 *
 * @param <fact> the SailFactory object that is copied.
 *
 * @return "this" SailFactory with data set to match the input factory (fact).
 */
//------------------------------------------------------------------------------
SailFactory& SailFactory::operator=(const SailFactory& fact)
{
   Factory::operator=(fact);
   return *this;
}


//------------------------------------------------------------------------------
// ~SailFactory()
//------------------------------------------------------------------------------
/**
 * Destructor for the SailFactory base class.
 */
//------------------------------------------------------------------------------
SailFactory::~SailFactory()
{
}

//---------------------------------
//  protected methods
//---------------------------------

//---------------------------------
//  private methods
//---------------------------------
```

B.3 The Solar Sail Force Code

B.3.1 src/sail/SolarSailForce.hpp

```
//$Id: SolarSailForce.hpp,v 1.1 2008/07/03 19:15:33 djc Exp $
//------------------------------------------------------------------------------
//                            SolarSailForce
//------------------------------------------------------------------------------
// GMAT: General Mission Analysis Tool
//
// **Legal**
//
// Developed jointly by NASA/GSFC and Thinking Systems, Inc. under contract
// number NNG06CA54C
//
// Author: Darrel Conway
// Created: 2008/07/03
//
/**
 * Declaration code for the SolarSailForce class, which extends the SRP model
 * for solar sailing.
 *
 * This is sample code demonstrating GMAT's plug-in capabilities.
 */
//------------------------------------------------------------------------------


#ifndef SolarSailForce_hpp
#define SolarSailForce_hpp

#include "SolarRadiationPressure.hpp"

class SolarSailForce : public SolarRadiationPressure
{
public:
SolarSailForce(const std::string &name = "");
virtual ~SolarSailForce();
SolarSailForce(const SolarSailForce& ssf);
SolarSailForce& operator=(const SolarSailForce& ssf);

virtual bool IsUserForce();
virtual bool Initialize();
   virtual bool GetDerivatives(Real * state, Real dt = 0.0, Integer order = 1);

   // inherited from GmatBase
   virtual GmatBase* Clone() const;


protected:
   /// Flag used when completing initialization at first call to GetDerivatives
   bool firedOnce;
   /// The number of satellites in the propagation
   Real satCount;
   /// Vector normal to the surface in sunlight
   Real* norm;

   void CheckParameters();
};

#endif /* SolarSailForce_hpp*/
```

B.3.2 src/sail/SolarSailForce.cpp

```
//$Id: SolarSailForce.cpp,v 1.1 2008/07/03 19:15:33 djc Exp $
//------------------------------------------------------------------------------
//                              SolarSailForce
//------------------------------------------------------------------------------
// GMAT: General Mission Analysis Tool
//
// **Legal**
//
// Developed jointly by NASA/GSFC and Thinking Systems, Inc. under contract
// number NNG06CA54C
//
// Author: Darrel Conway
// Created: 2008/07/03
//
/**
 * Implementation code for the SolarSailForce class, which extends the SRP model
 * for solar sailing.
 *
 * This is sample code demonstrating GMAT's plug-in capabilities.
 */
//------------------------------------------------------------------------------


#include "SolarSailForce.hpp"
#include "MessageInterface.hpp"


//------------------------------------------------------------------------------
```

```
// SolarSailForce(const std::string &name)
//------------------------------------------------------------------------------
/**
 * Default constructor.
 *
 * Creates a solar sail force instance with a given name.
 *
 * @param name The name used for the force.  This is usually the empty string.
 */
//------------------------------------------------------------------------------
SolarSailForce::SolarSailForce(const std::string &name) :
   SolarRadiationPressure  (name),
   firedOnce               (false),
   satCount                (1),
   norm                    (NULL)
{
   MessageInterface::ShowMessage("Constructed SailForce\n");
   typeName = "SailForce";
}


//------------------------------------------------------------------------------
// ~SolarSailForce()
//------------------------------------------------------------------------------
/**
 * Destructor.
 */
//------------------------------------------------------------------------------
SolarSailForce::~SolarSailForce()
{
   if (norm)
      delete [] norm;

}


//------------------------------------------------------------------------------
// SolarSailForce(const SolarSailForce& ssf)
//------------------------------------------------------------------------------
/**
 * Creates a solar sail force with settings matching the input force.
 *
 * @param ssf The solar sail force that this one needs to match.
 */
//------------------------------------------------------------------------------
SolarSailForce::SolarSailForce(const SolarSailForce& ssf) :
   SolarRadiationPressure  (ssf),
   firedOnce               (false),
   satCount                (ssf.satCount),
   norm                    (NULL)
{
}


//------------------------------------------------------------------------------
// SolarSailForce& operator=(const SolarSailForce& ssf)
//------------------------------------------------------------------------------
/**
 * Assignment operator
 *
 * Sets this force to match the settings for the input force.
 *
 * @param ssf The solar sail force that this one needs to match.
 *
 * @return Reference to this instance.
 */
//------------------------------------------------------------------------------
SolarSailForce& SolarSailForce::operator=(const SolarSailForce& ssf)
{
   if (&ssf != this)
   {
      SolarRadiationPressure::operator=(ssf);

      firedOnce   = false;
      satCount    = ssf.satCount;
      norm        = NULL;
      initialized = false;
   }

   return *this;
}


//------------------------------------------------------------------------------
// bool IsUserForce()
//------------------------------------------------------------------------------
/**
 * Checks to see if this force is a user defined (aka plug-in) force.
 *
 * The SolarSailForce is user defined, so this method returns true.
 *
 * @return true for user defined forces.
 */
//------------------------------------------------------------------------------
bool SolarSailForce::IsUserForce()
{
   return true;
}


//------------------------------------------------------------------------------
// bool SolarRadiationPressure::Initialize(void)
```

```
//------------------------------------------------------------------------------
/**
 * Prepares the force model data structures for use.
 *
 * @return true if initialization was successful, false is not.
 */
//------------------------------------------------------------------------------
bool SolarSailForce::Initialize()
{
   bool retval = SolarRadiationPressure::Initialize();

   if (norm)
      delete [] norm;
   norm = new Real[dimension];

   return retval;
}

//------------------------------------------------------------------------------
// bool GetDerivatives(Real * state, Real dt, Integer order)
//------------------------------------------------------------------------------
/**
 * Method invoked to calculate derivatives
 *
 * This method is invoked to fill the deriv array with derivative information
 * for the system that is being integrated.  It uses the state and elapsedTime
 * parameters, along with the time interval dt passed in as a parameter, to
 * calculate the derivative information at time \f$t=t_0+t_{elapsed}+dt\f$.
 *
 * @param dt         Additional time increment for the derivatitive
 *                      calculation; defaults to 0.
 * @param state      Pointer to the current state data.  This can differ
 *                      from the PhysicalModel state if the subscribing
 *                      integrator samples other state values during
 *                      propagation.  (For example, the Runge-Kutta integrators
 *                      do this during the stage calculations.)
 * @param order      The order of the derivative to be taken (first
 *                      derivative, second derivative, etc)
 *
 * @return           true if the call succeeds, false on failure.  This default
 *                      implementation always returns false.
 */
//------------------------------------------------------------------------------
bool SolarSailForce::GetDerivatives(Real * state, Real dt, Integer order)
{
   if (!initialized)
       return false;

   if (order > 2)
       return false;
```

```
if (!firedOnce)
   CheckParameters();

Integer i6;
Real distancefactor = 1.0, absmag, refmag, common, absFactor, refFactor,
     cosTheta;
bool inSunlight = true, inShadow = false;

Real ep = epoch + dt / 86400.0;
sunrv = theSun->GetState(ep);

// Rvector6 is initialized to all 0.0's; only change it if the body is not
// the Sun
if (!bodyIsTheSun)
   cbrv = body->GetState(ep);
cbSunVector[0] = sunrv[0] - cbrv[0];
cbSunVector[1] = sunrv[1] - cbrv[1];
cbSunVector[2] = sunrv[2] - cbrv[2];

Real sunSat[3];

for (Integer i = 0; i < satCount; ++i)
{
   i6 = i*6;

   // Build vector from the Sun to the current spacecraft
   sunSat[0] = state[i6] - cbSunVector[0];
   sunSat[1] = state[i6+1] - cbSunVector[1];
   sunSat[2] = state[i6+2] - cbSunVector[2];
   sunDistance = sqrt(sunSat[0]*sunSat[0] + sunSat[1]*sunSat[1] +
                      sunSat[2]*sunSat[2]);
   if (sunDistance == 0.0)
      sunDistance = 1.0;
   // Make a unit vector for the force direction
   forceVector[0] = sunSat[0] / sunDistance;
   forceVector[1] = sunSat[1] / sunDistance;
   forceVector[2] = sunSat[2] / sunDistance;

   distancefactor = nominalSun / sunDistance;
   // Factor of 0.001 converts m/s^2 to km/s^2
   distancefactor *= distancefactor * 0.001;

   #ifdef DEBUG_SRP_ORIGIN
      if (shadowModel == 0)
         shadowModel = CONICAL_MODEL;
   #endif

   // Test shadow condition for current spacecraft (only if body isn't Sol)
   if (!bodyIsTheSun)
   {
      psunrad = asin(sunRadius / sunDistance);
      FindShadowState(inSunlight, inShadow, &state[i6]);
   }

   if (!inShadow)
   {
      // Need to use attitude here; for now, point norm parallel to the
      // vector from the Sun to the spacecraft (so it's really opposite to
      // the normal vector)
      norm[0] = forceVector[0];
      norm[1] = forceVector[1];
      norm[2] = forceVector[2];

      // cosTheta takes into account the angle between the reflecting
      // surface and the vector to the Sun
      cosTheta = norm[0] * forceVector[0] +
                 norm[1] * forceVector[1] +
                 norm[2] * forceVector[2];

      absFactor = 2.0 - cr[i];
      refFactor = 2.0 * (cr[i] - 1.0) * cosTheta;

      // Reflection and absorption both use this prefactor:
      common = percentSun * fluxPressure * area[i] / mass[i] *
               distancefactor * cosTheta;

      absmag = common * absFactor;
      refmag = common * refFactor;

      // Finally, calculate the acceleration.  Note that the minus sighn is
      // missing from the spec, because the forceVector and norm both are
      // set in the code to point away from the Sun rather than towards it.
      if (order == 1)
      {
         deriv[i6] = deriv[i6 + 1] = deriv[i6 + 2] = 0.0;
         deriv[i6 + 3] = absmag * forceVector[0] + refmag * norm[0];
         deriv[i6 + 4] = absmag * forceVector[1] + refmag * norm[1];
         deriv[i6 + 5] = absmag * forceVector[2] + refmag * norm[2];
      }
      else
      {
         deriv[ i6 ] = absmag * forceVector[0] + refmag * norm[0];
         deriv[i6+1] = absmag * forceVector[1] + refmag * norm[1];
         deriv[i6+2] = absmag * forceVector[2] + refmag * norm[2];
         deriv[i6 + 3] = deriv[i6 + 4] = deriv[i6 + 5] = 0.0;
      }
   }
   else
   {
      deriv[i6] = deriv[i6 + 1] = deriv[i6 + 2] =
      deriv[i6 + 3] = deriv[i6 + 4] = deriv[i6 + 5] = 0.0;
   }
```

```
   }

   return true;
}


//------------------------------------------------------------------------------
// GmatBase* Clone() const
//------------------------------------------------------------------------------
/**
 * Makes a copy of this object.
 *
 * @return A GmatBase pointer to the copy.
 */
//------------------------------------------------------------------------------
GmatBase* SolarSailForce::Clone() const
{
   return new SolarSailForce(*this);
}

//---------------------------------------
// Protected Methods
//---------------------------------------


//------------------------------------------------------------------------------
// void CheckParameters()
//------------------------------------------------------------------------------
/**
 * Checks to be sure that everything needed for calculation is set.
 *
 * Throws an exception if a problem is detected.
 */
//------------------------------------------------------------------------------
void SolarSailForce::CheckParameters()
{
    satCount = dimension / 6;

    if (!theSun)
       throw ForceModelException("The Sun is not set in SRP::GetDerivatives");

    if (!body)
       throw ForceModelException(
          "The central body is not set in SRP::GetDerivatives");

    if (!cbSunVector)
       throw ForceModelException(
          "The sun vector is not initialized in SRP::GetDerivatives");

    if (cr.size() != satCount)
    {
       std::stringstream msg;
       msg << "Mismatch between satellite count (" << satCount
           << ") and radiation coefficient count (" << cr.size() << ")";
       throw ForceModelException(msg.str());
    }

    if (area.size() != satCount)
    {
       std::stringstream msg;
       msg << "Mismatch between satellite count (" << satCount
           << ") and area count (" << area.size() << ")";
       throw ForceModelException(msg.str());
    }

    if (mass.size() != satCount)
    {
       std::stringstream msg;
       msg << "Mismatch between satellite count (" << satCount
           << ") and mass count (" << mass.size() << ")";
       throw ForceModelException(msg.str());
    }

    firedOnce = true;
}
```

References

[1] The GMAT Development Team, *The GMAT Architectural Specification*, (March 2008).

[2] Dimitri van Heesch, *Doxygen*, Available for free download at http://www.stack.nl/ dimitri/doxygen/.

[3] Darrel J. Conway, *An Approach to Plug-in Coding in GMAT*, (May 2008).

[4] Wendy Shoan and Linda Jun, *C++ Coding Standards and Style Guide*, as modified for the GMAT project (October 2008).