# THE GENERAL MISSION ANALYSIS TOOL (GMAT): CURRENT FEATURES AND ADDING CUSTOM FUNCTIONALITY

**Darrel J. Conway**[1] **and Steven P. Hughes**[2]

[1] *Thinking Systems, Inc., Tucson, AZ, USA*
[2] *NASA Goddard Space Flight Center, Greenbelt, MD, USA*

## ABSTRACT

The General Mission Analysis Tool (GMAT) is a software system for trajectory optimization, mission analysis, trajectory estimation, and prediction developed by NASA, the Air Force Research Lab, and private industry. GMAT's design and implementation are based on four basic principles: open source visibility for both the source code and design documentation; platform independence; modular design; and user extensibility. The system, released under the NASA Open Source Agreement, runs on Windows, Mac and Linux. User extensions, loaded at run time, have been built for optimization, trajectory visualization, force model extension, and estimation, by parties outside of GMAT's development group. The system has been used to optimize maneuvers for the Lunar Crater Observation and Sensing Satellite (LCROSS) and ARTEMIS missions and is being used for formation design and analysis for the Magnetospheric Multiscale Mission (MMS).

In this paper, we discuss two primary topics: GMAT's current feature set; and how to write plug-in libraries written outside of the main development code. The existing feature set is broken down into two principal categories, called resources and commands. GMAT's resources consist of models of the components used to build a mission timeline: celestial objects, spacecraft and ground stations, hardware components, propagators, numerical solvers, variables and arrays, and output components. The commands are used to tie these resources together in a time ordered sequence, and are used to describe how the resources interact.

We present several examples of extensions to GMAT that have been built to support mission specific goals using custom plug-in libraries. The key elements required for a GMAT plug-in are presented, along with an overview of the class structure for the system that makes these elements work.

Key words: GMAT; Astrodynamics; Open Source.

## 1. PROJECT AND SYSTEM OVERVIEW

### 1.1. Objectives and Goals

The goal of the GMAT project is to develop new astrodynamics technologies and provide software for operational mission design and navigation support by working inclusively with individuals, universities, businesses, and other government organizations. A second and important goal is to share that technology in an open and unhindered way. GMAT has been approved for release under the NASA Open Source Agreement (NOSA). You, your business, or your organization can get involved in the GMAT project in numerous ways. We use an open source model to encourage collaboration and to maximize technology transfer. Individuals, universities, industry, and other government organizations can contribute and collaborate in ways that meet their respective goals, needs, and interests.

### 1.2. Contributors

GMAT contributors include volunteers and those paid for services they provide. We welcome new contributors to the project, either as users providing feedback about the features of the system, or as developers interested in contributing to the implementation of the system. Current U.S. government participants include NASA and the Air Force Research Lab (AFRL). Past and present industry contributors to GMAT include Thinking Systems, Inc. (system architecture and all aspects of development), a.i.-solutions (testing), Boeing (algorithms and testing), The Schafer Corporation (all aspects of development), Honeywell Technology Solutions (testing), and the Computer Sciences Corporation (requirements). The NASA Jet Propulsion Laboratory (JPL) is providing funding for integration of their SPICE toolkit into GMAT. Additionally , the European Space Agency's (ESA) advanced concepts team has developed optimizer plug-ins for the Non-Linear Programming (NLP) solvers SNOPT (Sparse Nonlinear OPTimizer) and IPOPT (Interior Point OPTimizer) using the process described in the later half of this paper.

The Navigation and Mission Design Branch at NASA's Goddard Space Flight Center performs project management activities and is involved in most phases of the development process including requirements, algorithms, design, and testing. The Ground Software Systems Branch performs design, implementation, and integration testing. The Flight Software Branch contributes to design and implementation. AFRL is involved in all development areas but primarily in the areas of requirements specification, mathematical and algorithmic specifications, system prototyping, and testing.

## 1.3. Platforms

GMAT is written to run on Windows, Linux and Macintosh platforms, using the wxWidgets cross platform UI Framework, and can be built using either commercial development tools or the GNU Compiler Collection (GCC). The system is implemented in ANSI standard C++ (approximately 250,000 non-comment source lines of code) using an Object Oriented methodology, with a rich class structure designed to make new features simple to incorporate.

On Windows and Linux, GMAT does not call any operating system unique functions or methods. Calls to the operating system are standard calls for reading and writing data files and for writing data to the screen. On the Mac, GMAT makes a call to the operating system to open X11, which is required to run MATLAB on the Mac.

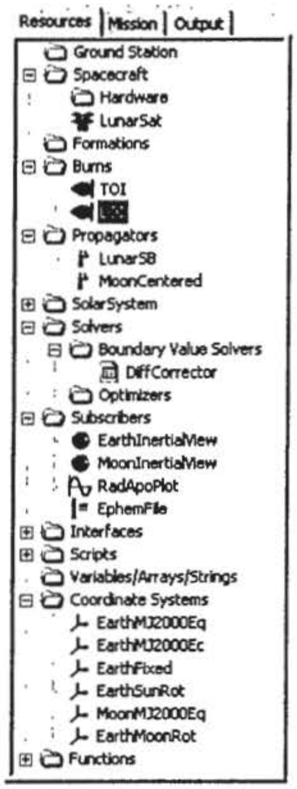
## 1.4. User Interfaces

GMAT has three user interfaces. The graphical user interface (GUI) illustrated in Figs. (1), (2), and (3) allows the user to set up and execute all aspects of GMAT. The script interface is textual and also allows the user to set up and execute all aspects of GMAT. The MATLAB interface is a secondary textual interface for running the system via calls from GMAT to MATLAB.

## 1.5. Status

While GMAT has undergone extensive testing and is mature software, at the present time we consider the software to be in Beta form. GMAT is not yet sufficiently verified to be used as a primary operational analysis sytem. GMAT has been used to optimize maneuvers for flight projects such as NASA's LCROSS and ARTEMIS missions, and for formation optimization and analysis for the MMS mission. However, for flight planning, we independently verify solutions generated in GMAT in the primary operational system.

The GMAT Team is currently working on several activities including maintenance, bug fixes, and the implementation of estimation components. The objective of the current development cycle is to provide a stable, non-beta release in the fall of 2010.



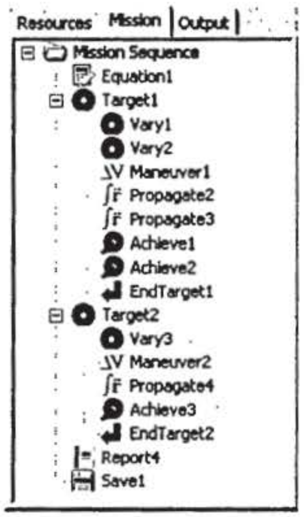
```
  Ground Station
⊟ Spacecraft
    Hardware
    LunarSat
  Formations
⊟ Burns
    TOI
    [illegible]
⊟ Propagators
    LunarSB
    MoonCentered
⊞ SolarSystem
⊟ Solvers
  ⊟ Boundary Value Solvers
      DiffCorrector
    Optimizers
⊟ Subscribers
    EarthInertialView
    MoonInertialView
    RadApoPlot
    EphemFile
⊞ Interfaces
⊞ Scripts
  Variables/Arrays/Strings
⊟ Coordinate Systems
    EarthMJ2000Eq
    EarthMJ2000Ec
    EarthFixed
    EarthSunRot
    MoonMJ2000Eq
    EarthMoonRot
⊞ Functions
```

*Figure 1. Screen Capture of Resource Tree*

## 2. FEATURES

GMAT is designed to model, optimize, and estimate spacecraft trajectories in flight regimes ranging from low Earth orbit to lunar applications, interplanetary trajectories, and other deep space missions.

Analysts model space missions in GMAT by first creating resources such as spacecraft, propagators, estimators, and optimizers. A figure of the resource tree for a lunar transfer application is shown in Fig. (1). Resources can be configured to meet the needs of specific applications and missions. GMAT contains an extensive set of available Resources that can be broken down into physical model Resources and analysis model Resources. Physical Resources include spacecraft, thruster, tank, transmitter*, transponder*, antenna*, receiver*, ground station, formation, impulsive burn, finite burn, planet, comet, asteroid, moon, barycenter, libration point, measurement model*, and measurement simulator*. Analysis model Resources include differential corrector, propagator, optimizer ,estimator*, 3-D graphic, x-y plot, report file, ephemeris file, user-defined variable, array, and string, coordinate system, custom subroutine, MATLAB function, and data



- Mission Sequence
  - Equation1
  - Target1
    - Vary1
    - Vary2
    - Maneuver1
    - Propagate2
    - Propagate3
    - Achieve1
    - Achieve2
    - EndTarget1
  - Target2
    - Vary3
    - Maneuver2
    - Propagate4
    - Achieve3
    - EndTarget2
  - Report4
  - Save1

*Figure 2. Screen Capture of the Mission Tree*

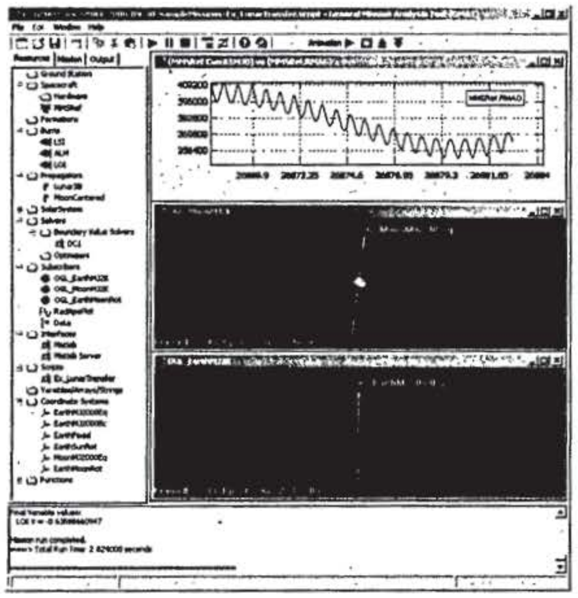


*Figure 3. Screen Capture of Lunar Transfer in GMAT*

file*. (Items with "*" are currently under development and not available in the public repository at the time of this writing as they have not been reviewed for ITAR and other release issues.)

After the resources are configured, they are used in the mission sequence, as shown in Fig. (2) to model spacecraft motion and simulate events in a mission's time evolution. Users employ built-in Commands that simulate trajectory dynamics or apply numerical methods such as estimators, optimizers, and boundary value solvers. The mission sequence supports the following commands: propagate, impulsive maneuver, finite maneuver, target, optimize, estimate, simulate measurements, non-linear constraint, minimize, call functions, inline math, vary parameter, achieve parameter, if/else, for, and while, and report.

The system can display trajectories in space, plot parameters against one another, and save parameters to files for later processing. The trajectory and plot capabilities are fully interactive, plotting data as a mission is run and allowing users to zoom into regions of interest.

Trajectories and data can be viewed in any coordinate system defined in GMAT, and GMAT allows users to rotate the view and set the focus to any object in the display. The trajectory view can be animated so users can watch the evolution of the trajectory over time. A screen capture of the graphics after computing a lunar transfer is shown in Fig. (3).

## 3. SYSTEM ARCHITECTURE

GMAT's System Architecture, described in the GMAT Architectural Specification[1], can be broken into three types of components: the Model, consisting of all of the elements required for simulating a spacecraft mission; the Engine, responsible for managing and connecting together the model elements; and interfaces, consisting of the scripting and graphical user interfaces, the MATLAB interface, and a simple console based interface. Users interact with GMAT through any of these interfaces to control the engine, and by controlling the engine, create a model that simulates specific spacecraft missions.

GMAT's architecture was designed so that the elements of the engine simulate spacecraft missions by operating on objects through fairly abstract classes. Specialization of these classes results in the ability to create objects that specialize these abstract interfaces into detailed elements of the spacecraft simulation. In other words, GMAT models a spacecraft mission by specializing high level classes into subclasses responsible for numerically simulating spacecraft and their environment, orbital computations, and the timeline defining the evolution of these elements. These specialized subclasses are the architectural components that a user sees as resources and commands, as described earlier in this paper.

While the system contains subclasses that simulate 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. The philosophy of starting from abstract classes defining interfaces that are specialized to meet mission needs provides a powerful mechanism for extending these classes through subclasses that meet the needs of the user community. The GMAT development team has used this approach repeatedly to add new capabilities to the system.

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].

These subclasses of GmatBase provide the framework for extension of GMAT's capabilities, either through work directly in GMAT's source code, or through plug-in modules loaded by GMAT at runtime. The remaining pages of this article explain the design of GMAT's plug-in architecture, and include an overview of a simple plug-in available in source form that illustrates the key elements of a GMAT plug-in extension.

## 4. GMAT PLUG-INS

Builds of GMAT made using development source code after June 25, 2008 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 specific needs of the LCROSS mission, and documented as an extension to GMAT[3]. The following paragraphs explain 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.

GMAT has been extended through the incorporation of new capabilities, loaded at run time using shared libraries, that incorporate a variety of features. These libraries, called GMAT plug-ins, have been used to add new optimizers, to develop estimation capabilities, to drive proprietary visualization elements, and to support ephemeris generation.

We'll begin this section 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, we present the steps needed to tell GMAT about the new plug-in.

### 4.1. The Plug-in Development Process

GMAT's Plug-in capabilities let developers extend 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 Factories for the new components, and a set of three C-style interface functions that are accessed by GMAT to load the plug-in. In the following paragraphs, we describe how to use each of these plug-in interfaces, 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 which illustrates the process.

### 4.2. Preparing 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.

One option for a plug-in developer when compiling is to build the plug-in using all of the required classes as part of the plug-in library. That approach makes the plug-in much larger than necessary, and makes the prospect of incompatibility between the plug-in and the evolving GMAT codebase likely. 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. A developer that is building a plug-in need only add this line to the file:

```
    SHARED_BASE = 1
```

and then clean and build GMAT. The build process will build the base code as a shared library – named libGmatBase or libGmatBaseNoMatlab, depending on the MATLAB build flags – that can be used for plug-in development. Once GMAT has been built this way, the plug-in developer is ready to start coding the plug-in.

### 4.3. The Plug-in Interface Functions

GMAT accesses new user classes contained in plug-in libraries by calling three methods in the plug-in library: GetFactoryCount(), GetFactoryPointer(), and SetMessageReceiver(). GMAT uses these functions 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 knows the number of factories in the library, it calls this function to retrieve the contained factories one at a time.

- **void SetMessageReceiver(MessageReceiver* m)**: Messages posted in GMAT are all sent to a MessageReceiver. This optional function is used to set the MessageReceiver for a plug-in if the developer incorporated GMAT's base code in the plug-in library, rather than linking against a shared library.

### 4.4. 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 4.

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:
   // Return objects as generic type
   virtual GmatBase* CreateObject(
         const std::string &ofType,
         const std::string &withName = "");

   // return objects as 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 = "");
   ...
```

When the programmer has decided what type of new component needs to be implemented, she creates a new factory that implements the corresponding factory method from this group and calls the new component's constructor. Each of the factory classes shown in Figure 4 is available for browsing in the src/base/factory folder of GMAT's source tree, so the developer should be able to select an appropriate Factory as a starting point for the custom Factory. The sample code includes code for a Factory supporting a new PhysicalModel class.

### 4.5. 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 the new component. Programmers use one of the existing classes as the base class for the new feature. That approach guarantees that GMAT has support for most or all of the plug-in component in the core GMAT engine, significantly reducing integration efforts. The next section describes the design and implementation of one such component: a custom force used in GMAT's force model.

## 5. 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, as described in Montenbruck and Gill[5]. The complete source code for this plug-in library, along with make files, is available from the Plug-in project on SourceForge[6].

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.

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.

### 5.1. 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, $\mathbf{e}_s$, makes an angle $\theta$ with the surface normal $\hat{\mathbf{n}}$. The absorbed radiation applies a force $\mathbf{f}_a$ 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

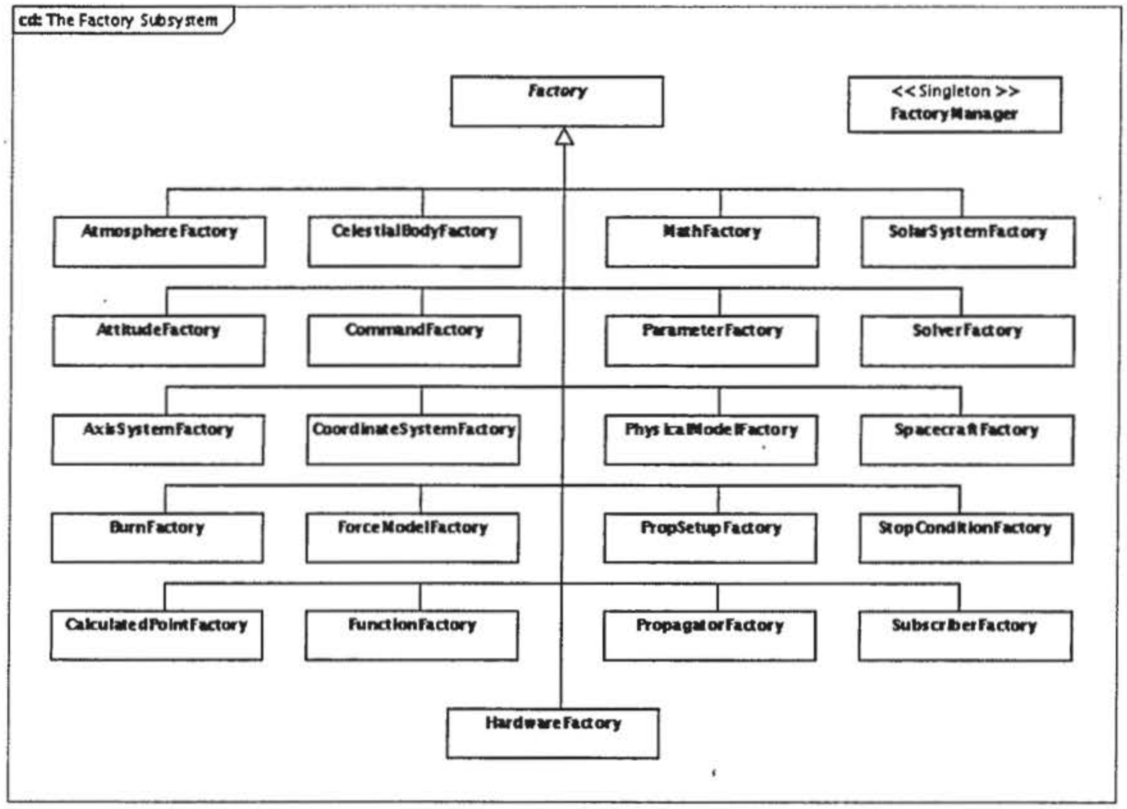


*Figure 4. The Factory Subsystem*

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

while that of the reflected light is given by

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

The constant $\varepsilon$ 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

$$C_r = 1+\varepsilon \qquad (3)$$

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

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

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

$$\mathbf{F}_{sail} = -P_r A\cos\theta\{(1-\varepsilon)\hat{\mathbf{e}}_s + 2\varepsilon\cos\theta\hat{\mathbf{n}}\} \qquad (5)$$

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

$$\mathbf{a}_{sail} = -P_r\frac{A}{m}\cos\theta\{(2-C_r)\hat{\mathbf{e}}_s + 2(C_r-1)\cos\theta\hat{\mathbf{n}}\} \qquad (6)$$

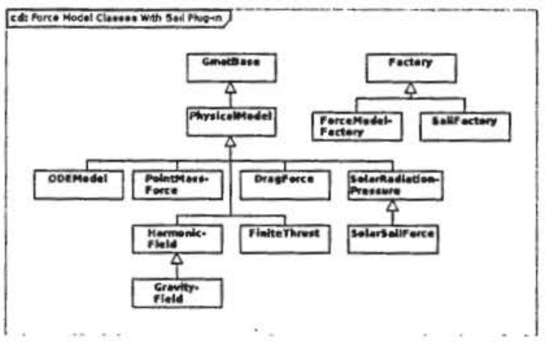


*Figure 5. The Solar Sail Model Components. The plug-in components are shown in blue.*

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

### 5.2. 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 5.

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() method 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 available from SourceForge[6] in the trunk/SolarSail folder of the project's Subversion repository.

### 5.3. The SailFactory and Interface code

The SailFactory is used to create new instances of the SolarSailForce. The code 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 two C-style functions that are used in the plug-in implementation: GetFactoryCount() and GetFactoryPointer(). 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), and returns NULL for calls with other factory indices. More complicated plug-in libraries use this feature to support multiple factories that are loaded by calling GetFactoryPointer for each factory in the library.

Once the code described above is in place, it can be compiled into a shared library that meets GMAT's plug-in requirements. GCC make files for the solar sail library plug-in are included in the SourceForge repository, along with a configuration file, SolarSailEnv.mk, for each of our supported platforms. The build process compiles the source files into object files, links those object files with references to the GMAT base code shared library (libGmatBase or libGmatBaseNoMatlab, described above), and produces a shared library compatible with your GMAT build.

### 5.4. 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". GMAT manages the file extension internally based on the operating system, so the line in the startup file does not explicitly specify the shared library extension.

Once this line is in place in your startup file, GMAT will attempt to load the plug-in when it is started. On success, the capabilities of the plug-in code – in this case, the new solar sail model – will be available for use from a GMAT script.

### 5.5. Upcoming Capabilities

The current plug-in capability provides new components to GMAT's model, along with the factory code that GMAT uses in its engine to make that capability available through GMAT's scripting language. At this writing, GMAT does not yet have the ability to generate complete user interface elements that extend the wxWidgets based graphical user interface. The plug-in interfaces for that capability are currently being designed, and should be ready for use in the second quarter of 2010.

## 6. SUMMARY

This paper provided an overview of the General Mission Analysis Tool, GMAT, and then provided some detail about how GMAT can be tailored through plug-in components loaded at run time. The discussion of the plug-in capabilities described the elements that GMAT expects for a plug-in module: the new functionality itself, the supporting factory or factories that make the module visible in GMAT's engine, and the interface code used to load the plug-in features. An example, implemented in code available for free download, 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.

Groups or individuals interested in GMAT can obtain a copy of the system from the GMAT website at NASA's Goddard Space Flight Center, http://gmat.gsfc.nasa.gov. The code for GMAT can be downloaded from SourceForge using the project address: http://sourceforge.net/projects/gmat. The GMAT plug-in example described in this paper can be downloaded from the GMAT plug-in project at SourceForge, https://sourceforge.net/projects/gmatplugins. Finally, GMAT's developer and user communities can be found at our forum and wiki sites, http://gmat.ed-pages.com/forum/index.php and http://gmat.ed-pages.com/wiki/tiki-index.php, respectively.

## ACKNOWLEDGMENTS

Numerous individuals have contributed to GMAT and the system would not exist without their contributions. The authors of this paper are only two among many. While a complete list is too lengthy, additional contributors include Russell Carpenter (GSFC), Edwin Dove (GSFC), Dunn Idle V (The Shafer Corp.), Moriba Jah (AFRL), Linda Jun (GSFC), Tom Kelecy (Boeing), LaMont Ruley(GSFC), Wendy Shoan (GSFC), Gene Stillman (The Shafer Corp.), Phil Silvia (The Shafer Corp.), Matt Wilkins (The Shafer Corp.),

## 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 (March 2010).

[5] Oliver Montenbruck and Eberhard Gill, *Satellite Orbits*, Springer-Verlag, 2000.

[6] Darrel J. Conway (Administrator), Publicly available GMAT plug-in code is available from https://sourceforge.net/projects/gmatplugins/