Function Call Samples and Discussion

Darrel J Conway

November 1, 2007

Abstract

In the Mission Control Sequence, the GMAT command classes all use a local object map during initialization that contains the Sandbox level clones of the configured objects used when running a mission. GMAT functions act like mini-Sandboxes, in that they contain a Function Control Sequence (FCS) structured identically to the the Mission Control Sequence (MCS) and a local object store that plays the role of the Sandbox's local object map. One design issue under discussion is the contents and scope of objects in the Sandbox's local object map and the GmatFunction's local object store. This document provides several options for the function object management, and discusses these objects in the context of actual scripted examples.

1 Definitions

Following the Humpty Dumpty rule ("That word means what I say it means"), here are a few basic definitions I'll try to use consistently:

Configuration The resource database managed by the Configuration Manager.

Local Object Map The database of cloned objects in the Sandbox, used during a run.

LOM A Local Object Map.

Local Object Store The object map used in a GmatFunction to find resources.

LOS A Local Object Store.

Global Object Store The mapping identifying the (Sandbox-level) global objects. This Sandbox feature is a new feature, added for object management when global objects are allowed.

GOS A Global Object Store.

Mission Control Sequence The sequence of commands built from the main script.

MCS A Mission Control Sequence.

Function Control Sequence The sequence of commands built for a function.

FCS A Function Control Sequence.

2 Features and Assumptions

Here are a few notes on the assumptions and system features I'll use in this document:

1. The core structures used in GMAT will not be rearchitected because of incorporation of Functions. What I mean here is that Functions will be designed and implemented to fit into the existing GMAT architecture – certainly through extensions to the architecture where needed, but not through extensive reworking of core GMAT components.

2. Object usage during a run has to follow a consistent set of rules, easily understood by users of the system.

3. When GMAT runs a mission, it performs two steps: First, all of the elements, including the functions, are built and initialized, and then the MSC and any contained FCS's are executed. At teh end of the build portion of this sequence, all of the objects used in the mission have been created.

4. GMAT manages objects – in the Configuration, as well as in the Sandbox – through object names and base class (aka GmatBase) pointers. When an object is needed – usually during initialization in the Sandbox – it is accessed through a map, defined as

   std::map <std::string, GmatBase*>

   (This feature of the design means that all of the objects accessed in the system must have unique names in the relevant scope. We cannot have 2 objects with the same name in the Configuration, or in the Sandbox's local object map, or in a function's local object store.)

5. As much as possible, the Function model will behave the same way the Sandbox's local object map / Mission Control Sequence works.

6.

First I'll attempt to describe four possible approaches to object management that we could use in GMAT as we implement functions. I think these are the ones most likely to make the implementation smooth, but if anyone can think of another, feel free to suggest it.

3 Object Management Options Considered in this Document

There are several options that I'll considered for the GmatFunction local object store.

3.1 Local Objects

In this approach, all of the resources that are used in a function must either be created inside the function or passed into it as function call parameters. This approach does not allow visibility of any Sandbox level objects (except for the Solar System) inside of the function unless they are passed in. It places a pretty heavy burden on the user – coordinate systems, propagators, Subscribers, and other resources must all be recreated in the function or passed in – and has therefore been removed from consideration. This approach does not need a Global Object Store.

3.2 Local Objects with Select Global Objects

With this option, several categories of objects would automatically be treated as global[1] in scope – Coordinate Systems, Subscribers, and possibly Propagators – while the other resources remain local. Specific objects can be identified as global using a new keyword, "Global," which must be stated in both the FCS or MCS using the overridden scope and in the FCS or MCS thaqt creates the object. This approach uses a Global Object Store to track the global objects.

For the purposes of the discussion that follows, I'll assume that all coordinate systems, propagators, functions, and subscribers are automatically included in the GOS.

3.3 Semi-Local Objects

This case treats GMAT objects in a strictly hierarchical fashion. In this model, when a command needs an object, it first looks at the function's local object store, then at the local object store for the object that called the function, and so on, working its way up the tree of calling objects until it reaches the CallFunction in the Mission Control Sequence that initiated the function call. When the named object is found, the search stops. If it is not found, an exception is thrown. This approach does not need a Global Object Store, because all of the used obects must exist somewhere in the calling hierarchy.

3.4 Global Objects

In this model, all of the created objects are treated as globals. Every object – inside or outside of a function – must be uniquely named. Objects created in a function are added to the Sandbox's local object map, regardless of type. This approach does not need a Global Object Store because all of the crated objects are global.

4 Sample Functions

This section contains some GmatFunctions used for the ensuing discussion.

4.1 CalculatePropTime

This function is designed propagate an input spacecraft to the the nth periapsis, plotting RMag vs epoch.

```
1  function [dt] = CalculatePropTime(sat, n)
2  %
3  % This function calculates the time required to propagate to
4  % the nth periapsis point.  The following objects must exist
5  % in scope when the function is called:
6  %
7  %     prop       A propagator
8  %
9  % The input parameters are:
10 %
11 %     sat        A Spacecraft that gets propagated
12 %     n          The number of periapses that are passed
13 %                before stopping
14 %
15 % The return value is
```

[1]Note that these objects are not truly global in scope. Access is restricted for most of these objects to the resources in the Sandbox. Subscribers also provide some restricted access outside of the Sandbox through the Publisher. This slightly larger scoping rule for Subscribers will be defined more completely when multiple Sandbox capabilities are added to GMAT.

```
%
%     dt         Number of seconds required to reach the
%                nth periapse

Create Variable startEpoch endEpoch j;

Create XYPlot   plot;
plot.IndVar = sat.A1ModJulian;
plot.Add    = sat.RMag;

%---------------------------------
% The Function Control Sequence
%---------------------------------
startEpoch = sat.A1ModJulian;

For j = 1 : n
   Propagate prop(sat) {sat.Periapsis};
EndFor

endEpoch = sat.A1ModJulian;

dt = (endEpoch - startEpoch) * 86400.0;
```

Listing 1: The CalculatePropTime Function

4.2 PropPlot

This function plots an input spacecraft's trajectory as it propagates for an input duration.

```
function [sc] = PropPlot(sat, dt)
%
% This function propagates a spacecraft for a given timespan,
% drawing the trajectory over that span.  The following
% object must exist to use the function:
%
%     prop       A propagator
%
% The input parameters are:
%
%     sat        A Spacecraft that gets propagated
%     dt         The duration of the propagation
%
% The return value is
%
%     sc         A spacecraft, configured to match sat at its
%                end state

Create OpenGLPlot   plot;
plot.Add    = sat;

%---------------------------------
```

```
% The Function Control Sequence
%--------------------------------
Propagate prop(sat) {sat.ElapsedSeconds = dt};
sc = sat;
```

Listing 2: The PropPlot Function

4.3 PropToPeriapsis

This function propagates a spacecraft to periapse, plotting RMag vs epoch.

```
function PropToPeriapsis(sat)
%
% This function propagates a Spacecraft to periapsis, plotting
% RMag vs epoch as it goes.  The following object must be in
% scope when the function is called:
%
%     prop      A propagator
%
% The input parameters are:
%
%     sat       A Spacecraft that gets propagated

Create XYPlot   plot;
plot.IndVar = sat.A1ModJulian;
plot.Add    = sat.RMag;

%--------------------------------
% The Function Control Sequence
%--------------------------------
Propagate prop(sat) {sat.Periapsis};
```

Listing 3: The PropToPeriapsis Function

4.4 PlotTrajectory

This function plots a spacecraft's trajectory as it propagates to periapsis. The actual propagation is performed using the PropToPeriapsis function.

```
function PlotTrajectory(sat, dt)
%
% This function builds an OpenGL plot to so a trajectory, then
% calls PropToPeriapsis to generate the plot.  The following
% objects must be in scope:
%
%     prop               A propagator
%     PropToPeriapsis    A GmatFunction that propagates a
%                        spacecraft.
%
% The input parameters are:
%
%     sat       A Spacecraft that gets propagated and plotted.
```

```
14
15 Create OpenGLPlot   plot;
16 plot.Add    = sat;
17
18 %---------------------------------
19 % The Function Control Sequence
20 %---------------------------------
21 PropToPeriapsis(sat)
```

Listing 4: The PlotTrajectory Function

4.5 PlotTrajectoryReworked

This function is a rework of the previos function, written to avoid plot name conflicts. It plots a spacecraft's trajectory as it propagates to periapsis. The actual propagation is performed using the PropToPeriapsis function.

```
1  function PlotTrajectoryReworked(sat, dt)
2  %
3  % This function builds an OpenGL plot to show a trajectory, then
4  % calls PropToPeriapsis to generate the plot.  The following
5  % objects must be in scope:
6  %
7  %     prop               A propagator
8  %     PropToPeriapsis    A GmatFunction that propagates a
9  %                        spacecraft.
10 %
11 % The input parameters are:
12 %
13 %     sat        A Spacecraft that gets propagated and plotted.
14
15 Create OpenGLPlot   glplot;
16 glplot.Add    = sat;
17
18 %---------------------------------
19 % The Function Control Sequence
20 %---------------------------------
21 PropToPeriapsis(sat)
```

Listing 5: The PlotTrajectoryReworked Function

5 Sample Scripts

Now that we have some functions defined, we can start examining the trade-offs between different scoping strategies by writing some scripts that use these functions.

5.1 Sample Script, Case 1: Two Independent Functions

The following script uses a function to find out how long it takes a spacecraft to reach the 4th perigee, and then propagates the spacecraft to that perigee.

```
Create Spacecraft sat1;
sat1.SMA = 12000.0;
sat1.ECC = 0.40;

Create ForceModel fm;
fm.CentralBody = Earth;
fm.PointMasses = {Earth, Sun, Luna}

Create Propagator prop;
prop.FM = fm;

Create GmatFunction CalculatePropTime PropPlot;
Create Variable deltaT;

Create OpenGLPlot plot;
plot.Add = sat1;

%------------------------------
% The Mission Control Sequence
%------------------------------
Propagate prop(sat1) {sat1.Apoapsis};

% How long to the fourth perigee?
[deltaT] = CalculatePropTime(sat1, 4)

% Now go there!
sat1 = PropPlot(sat1, deltaT);
```

Listing 6: Sample Script with Functions

5.1.1 Results with Local + Select Global Objects

At the end of the build/initialize phase, the object maps contain the following objects:

- Configuration
  - Spacecraft: sat1
  - ForceModel: fm
  - Propagator: prop
  - GmatFunction: CalculatePropTime, PropPlot
  - Variable: deltaT
- Local Object Map
  - SolarSystem: SolarSystem
  - Spacecraft: sat1
  - ForceModel: fm
  - Propagator: prop
  - GmatFunction: CalculatePropTime, PropPlot

  - Variable: deltaT

- Global Object Store
  - SolarSystem: SolarSystem
  - Propagator: prop[2]
  - GmatFunction: CalculatePropTime, PropPlot
  - XYPlot: plot
  - OpenGLPlot: plot

- Local Object Store for CalculatePropTime
  - XYPlot: plot
  - Variable: startEpoch, endEpoch, j

- Local Object Store for PropPlot
  - OpenGLPlot: plot

This case throws an exception at during the build process because the GOS has 2 objects with the same name; an OpenGLPlot and an XYPlot, both named "plot." A better choice of names in the functions would resolve the conflict, and the mission could run.

5.1.2 Results with Semilocal Objects

At the end of the build/initialize phase, the object maps contain the following objects:

- Configuration
  - Spacecraft: sat1
  - ForceModel: fm
  - Propagator: prop
  - GmatFunction: CalculatePropTime, PropPlot
  - Variable: deltaT

- Local Object Map
  - SolarSystem: SolarSystem
  - Spacecraft: sat1
  - ForceModel: fm
  - Propagator: prop
  - GmatFunction: CalculatePropTime, PropPlot
  - Variable: deltaT

- Global Object Store: Not Used

- Local Object Store for CalculatePropTime
  - XYPlot: plot
  - Variable: startEpoch, endEpoch, j

[2] fm is included by internal reference

- Local Object Store for PropPlot
  - OpenGLPlot: plot

This case should build and run. The identically named objects do not see each other when chaining up through the call hierarchy. The plot title bars would be confusing, since both are named "plot." Is there a potential conflict in the GUI? (I'm guessing not – I think the window management doesn't use the names to find windows. Linda should be able to say if that is indeed the case.)

5.1.3 Results with Global Objects

- Configuration
  - Spacecraft: sat1
  - ForceModel: fm
  - Propagator: prop
  - GmatFunction: CalculatePropTime, PropPlot
  - Variable: deltaT
- Local Object Map
  - SolarSystem: SolarSystem
  - Spacecraft: sat1
  - ForceModel: fm
  - Propagator: prop
  - GmatFunction: CalculatePropTime, PropPlot
  - Variable: deltaT, startEpoch, endEpoch, j
  - XYPlot: plot
  - OpenGLPlot: plot
- Global Object Store: Not used. Everything is in the LOM.
- Local Object Store for CalculatePropTime: Not used. Everything is in the LOM.
- Local Object Store for PropPlot: Not used. Everything is in the LOM.

This case throws an exception during the build process because the LOM has 2 objects with the same name; an OpenGLPlot and an XYPlot, both named "plot." A better choice of names in the functions would resolve the conflict, and the mission could run.

5.2 Sample Script, Case 2: Nested Functions

(I'll build a second case if needed.)

5.3 Sample Script, Case 3

(I'll build a third case if needed.)

6 Summary and Implications

Feature	Local/Global	Semilocal	Global
Name scope	Globals have unique names	Names resolved hierarchically	All resources have unique names