# Overview

This document describes the Orbit and Coverage (O&C) code adapted from GMAT for the TAT-C project. It comprises two main sections, the first being a description of the interfaces and a high level description of system behavior. The second main section documents the system structure in more detail, defining class responsibilities, class dependencies, key data structures and key functions. The system structure in turn is divided into two sections, one covering the Propagator and Spacecraft, the other the CoverageChecker.

In addition to the descriptive documentation, Doxygen output for all the source code is included as an appendix.

# Interface Description

This section provides information needed to incorporate the Orbit and Coverage (O&C) code into a larger system such as TAT-C. It provides the interface to key routines used to access O&C capabilities. Including precise definitions for each argument.

It also provides high level descriptions of the O&C subsystem’s behavior. This is intended to provide a broad outline, the details are provided in the source code itself and in the System Structure section of this document.

## Interface to Key Routines

### Propagate

The Propagate function is defined in the Propagator class, and it has the following signature:

virtual Rvector6 Propagate(const AbsoluteDate &toDate);

Argument **&toDate** – this is an AbsoluteDate object; class AbsoluteDate provides the ability to represent dates as either Julian or Gregorian dates. Generally Gregorian dates are used for initialization and Julian dates used for computations. The propagator will propagate the spacecraft’s state to that time.

Return value **Rvector6** – this is a 6 element vector of real numbers representing the spacecraft state. The first three elements of this vector represent the spacecraft’s position in Earth-centered inertial coordinates, the next 3 represent the velocity in the same coordinate frames.

### AccumulateCoverageData

The AccumulateCoverageData function is defined in the CoverageChecker class, and it has two overloaded versions. The one with no arguments is used when propagating and checking for sensor visibility at the same time. The one with time as an argument is used when stepping the event locator multiple times within each orbit propagation step.

virtual IntegerArray AccumulateCoverageData();

virtual IntegerArray AccumulateCoverageData(Real atTime);

When the time is provided as the argument **atTime** it is a Julian date that is represented by a real number counting days from a reference date. When the time is not provided as an argument the time stored by the Spacecraft is used.

In both cases, the **return array of integers** contains indices of points from a PointGroup (see discussion in System Structure section of this document) that are visible at a given time.

## High Level System Behavior

This section gives a high level view of how pieces of the general Initialize-Propagate-Postprocess use case work. They are presented as descriptive text and snippets of actual code that show the key concepts of how this subsystem is intended to be used. This does show the highest level of processing and the functions that would be called by other TAT-C code, without showing too much of the internal functions and data structures. Much of the detail will be found by reading the code called by these high level functions, or by reading the “System Structure” section of this document.

### Initialization

In the system test driver, the classes initialize in the following order. Dependencies on predecessor classes are listed for each class.

* LagrangeInterpolator – none
* Earth – none
* AbsoluteDate – none
* OrbitState – none
* Sensor subclasses (ConicalSensor, RectangularSensor, CustomSensor) – none
* NadirPointingAttitude – none
* Spacecraft (Attitude, AbsoluteDate, OrbitState ,LaGrangeInterpolator)
* Propagator (Spacecraft)
* PointGroup
* CoverageChecker

Note that NadirPointingAttitude is a subclass of Attitude.

In addition to the constructor dependencies listed above sensors are associated with the Spacecraft via the AddSensor() operation provided by the Spacecraft class. CoverageChecker then accesses sensor(s) and their field of view via a Spacecraft object containing said sensor(s), providing a sensor ID to identify the correct sensor.

Finally, there is one other class of interest. Propagator and CoverageChecker each create a local copy of the Earth class; this class is primarily used to rotate vectors from an inertial frame (+X towards First Point of Aries) to an Earth-fixed frame (+X is 0 latitude, 0 longitude).

### Propagation & Coverage Without Interpolation

This shows the key processing loop in the case where coverage checks are done at the same rate as the spacecraft state is being propagated. The key steps are to

1. Propagate the spacecraft state up to the start time.
2. Loop until the end time; the date is a Julian date, which is expressed in ***days*** from a standard reference time.
   1. Check coverage. The function AccumulateCoverageData, in addition to returning data, loads several data structures within the CoverageChecker class which contain coverage information for each point of interest.
   2. Advance the time and propagate orbit to that time. The step size is measured in ***seconds***.
   3. Compute latitude longitude and height.

prop->Propagate(\*date);

while (date->GetJulianDate() < ((Real)startDate + 1.0))

{

// Compute points in view at time zero!

loopPoints = covChecker->AccumulateCoverageData();

// Propagate

date->Advance(stepSize);

prop->Propagate(\*date);

// Compute lat., lon., and height of s/c w/r/t the ellipsoid

Real jDate = sat1->GetJulianDate();

Rvector6 cartState = sat1->GetCartesianState();

Rvector3 inertialPosVec(cartState(0), cartState(1),cartState(2));

Rvector3 latLonHeight = earth->InertialToBodyFixed(inertialPosVec,

jDate, "Ellipsoid");

}

The loopPoints variable contains a list of point indexes for all the points of interest visible at the time that coverage is being checked.

### Propagation & Coverage With Interpolation

This option is used when the coverage checker’s event detection needs to take smaller steps than the orbit propagator. The Spacecraft sat1 provides a “time to interpolate” function that determines if interpolation is feasible. The inner loop will give the coverage checker the time to interpolate to when accumulating data, then advance the interpolation time.

prop->Propagate(\*date);

while (date->GetJulianDate() < ((Real) startDate + 1.0)) // 5.0))

{

date->Advance(stepSize);

prop->Propagate(\*date);

propTime = date->GetJulianDate();

// Interpolate when and if needed

if (sat1->TimeToInterpolate(propTime, midRange))

{

while (interpTime < (propTime - midRange))

{

loopPoints = covChecker->

AccumulateCoverageData(interpTime);

interpTime += interpolationStepSize/

GmatTimeConstants::SECS\_PER\_DAY;

}

}

}

In this scenario the interpolation step size is expected to be substantially smaller than the propagation step size. One second for interpolation and 1 minute for propagation is a plausible scenario.

### PostProcessing

Post processing is done in the CoverageChecker’s ProcessCoverage() operation. This function returns a vector of interval event reports, each of which defines a time interval when a given point of interest is in view. This function is invoked as follows:

std::vector<IntervalEventReport> coverageEvents;

coverageEvents = covChecker->ProcessCoverageData();

# System Structure

## Class Dependencies

The diagram below shows the key dependencies between components. The light shading shows the components that implement the main functions of modeling the spacecraft, propagating the spacecraft state, and identifying when points are within a sensor’s field of view. The dark shading indicates the models used by these major functions. Utilities such as vector and matrix arithmetic are not shown on this diagram.

Spacecraft

Coverage

Checker

Sensor

Attitude

Conical

Custom

Rectangular

Orbit

Interpolator

Propagator

Visibility reports (2 classes)

1,..n

1,..n

1,..n

Point Group

Orbit State

Absolute Date

Note that the class Sensor has three subclasses providing 3 different models of sensor field of views. The conical and rectangular fields of view are self-explanatory, a custom field of view allows the FOV perimeter to be defined by an arbitrary set of points.

The following sections provide tables detailing each class’ responsibilities and explanatory text for key data structures.

## Propagation & Spacecraft

### Class Responsibilities

| **Class** | **Responsibility** |
| --- | --- |
| Propagator | Propagates spacecraft state to a requested time. |
| Spacecraft | The Spacecraft class is a container for objects related to the spacecraft, including abstractions such as orbit and attitude, algorithms such as the LaGrange interpolator, or models of objects such as sensors. The spacecraft class provides operations to access the state of its contained objects, and to do computations based on that state.  For example, the CoverageChecker calls Spacecraft’s CheckTargetVisibility operator, which rotates the vector to the sensor frame and then calls the sensor to check whether it is in the field of view. A second example is the propagator accessing, propagating, and updating the Spacecraft’s time, position and velocity. |
| Sensor | The Sensor class defines a field of view, maintains knowledge of its orientation relative to the spacecraft body, and a function which determines if the a point is within the sensor field of view.  There are three subclasses of Sensor. A conical sensor’s FOV is defined by a constant cone angle, a rectangular sensor’s FOV is defined by angular width and angular height, both symmetrically around the boresight, and a custom sensor’s FOV is defined by an arbitrary set of points that are defined by cone and clock angle around the sensor frame’s +z axis. For nadir pointing instruments this is the same as the spacecraft z axis, and the body to sensor rotation is generally defined as the 3x3 identity matrix or an equivalent representation.  The Sensor class provides a CheckTargetVisibility() method which is implemented by each of the subclasses. This function determines if a vector (which must be rotated into the sensor frame to make this test valid) is inside the field of view or not. For cone and rectangular sensors these involve simple inequality tests, for the custom sensor a sophisticated line crossing algorithm is used. |
| Attitude | O-C uses the subclass NadirPointingAttitude, which orients the spacecraft to the center of the Earth. The main responsibility of this class is to store the rotation from an inertial frame to the nadir pointing reference frame, as computed from spacecraft position and velocity. |
| Orbit State | Orbit State contains the spacecraft position and velocity, which can be set and retrieved as either Keplerian or Cartesian elements. |
| Orbit Interpolator | tbs |
| Absolute Date | This class maintains a representation of date and time. The time can be set or retrieved as either a Gregorian date (year, month, day, hours, minutes and seconds) or a Julian date (days from a standard reference point), and it allows the date and time to be advanced by a number of seconds. This number may be negative to indicate movement backwards in time, however this is not being done for TAT-C. |

### Key Data Structures

There are no complex data structures in the Propagation/Spacecraft area.

### Key functions

The key functions for propagation and spacecraft are:

Propagator

* PropagateOrbitalElements() – this function propagates the Keplerian elements (a, e, i, RAAN, argP, MA), using the classic two-body problem with the addition of the J2 gravitational perturbation.
* Propagate() – this function calls PropagateOrbitalElements() and adds the option to model the effect of atmospheric drag
* ComputePeriapsisAltitude() – computes value needed in drag modeling

Spacecraft

* CheckTargetVisibility() – the implementation of this function is simple, it calls the CheckTargetVisibility() function in the Sensor class for a given sensor. The Sensor function in turn determines if a point is in its field of view.

## Coverage

### Class Responsibilities

|  |  |
| --- | --- |
| Class | Responsibility |
| CoverageChecker | CoverageChecker determines when points are in a sensor’s field of view and accumulates a database of which points are in the FOV at which times as the spacecraft continues to orbit. This class also provides functions needed to compute coverage statistics from this raw data. |
| PointGroup | PointGroup maintains a user defined or an automatically generated set of points on the surface of the central body. These points are accessed by an integer point ID and represented in terms of longitude and latitude or of a position vector expressed in the central body’s rotating coordinate frame (body-fixed coordinates). |
| VisiblePOIReport | The VisiblePOIReport is a container that for a given point contains:   * the observatory range * the observatory azimuth angle * the observatory zenith angle; and * the sun azimuth angle * the sun zenith angle   These points are stored and associated with time tags in the CoverageChecker data structures. |
| IntervalEventReport | The IntervalEventReport is a container that for a given point contains   * start time of interval that spacecraft is visible * end time of interval that spacecraft is visible * a vector of VisiblePOIReport data   This data structure is used by ProcessCoverageData() to generate a sequence of interval event reports, point by point. |

### Key data Structures

The key data structures for coverage checking all reside in the CoverageChecker class. They are supported by the class members in the PointGroup, VisiblePOIReport, and IntervalEventReport; all of which are containers with little or no processing beyond setting and getting data. These data structures are:

* pointGroup – is a pointer to the pointGroup being analyzed. The constructor sets this pointer from the input parameter ptGroup.
* pointArray – is an array of unit vectors representing the position of each point in pointGroup, represented in the body-fixed reference frame.
* dateData –is an array of Julian dates [represented as real numbers] that contains a time tag for each step of event location. CoverageChecker also has a member variable timeIdx that is used to index this array. The AccumulateCoverageData() functions store the current time in date data and increment timeIdx.
* timeSeriesData – is a vector of integer arrays. There is one vector element for each point of interest; this element is an integer array containing the indices into dateData for times in which the spacecraft is visible from the point of interest.
* discreteEventData – is an vector of visiblePOIReport vectors. Each point of interest has a single vector of POI reports, the containing vector is indexed by the POI number.
* numEventsPerPoint – is an IntegerArray (vector of integers) containing a counter of the number of times each point is in the sensor FOV.

### Key functions

The key functions for coverage checking are CheckPointCoverage(), which is called by both versions of AccumulateCoverageData(), and ProcessCoverageData(), which is called directly by TAT-C software using the O&C module. The behavior of these two functions is described below.

CheckPointCoverage()

For each POI in pointGroup loop

If (POI is above horizon) then

Check target visibility (call to spacecraft->CheckTargetVisibility)

If spacecraft is in view then

Store timeIdx in timeSeriesData for POI

Store POI index in result to be returned from function

Increment number of events for POI

If (option to compute target geometry)

Compute & store data in a visiblePOIreport

Store visiblePOIreport in discreteEventData for the point

End if

End if // spacecraft in view

End if // POI above horizon

End loop

Return list of points in field of view at current time.

Process Coverage Data()

for each POI in pointGroup loop

if (numEventsPerPoint[POI] >= 2)then

startTime = Julian date associated with 1st POI

for each time in timeSeriesData[POI] loop

if (time index not consecutive) then

set endTime // for interval

isEnd = True // for interval

else if (reached last event for point) then

set endTime // for interval

isEnd = True // for interval

else

noop;

// consecutive observations visible,

// keep looking for end of interva

end if // are points consecutive

if (isEnd) then

// construct intervalEventReport

add start and end times

add visiblePOIrecord for each time

between start and end times

reports.pushback(intervalEventReport)

// reset everything to search for next

// interval

end if // is end of interval

end loop // over time tags

end if // enough points to form an interval

end loop // over POIs

return reports

The final routines of interest in CoverageChecker are the two versions of AccumulateCoverageData(). In both cases the main function is to get the date and the spacecraft state, rotate the spacecraft state into body-fixed coordinates, increment the time index, and call CheckPointCoverage with the date and state. They are both less than 20 lines, and easy to understand lines at that. So read the source code directly to understand their role.

# Appendix: Doxygen Output