Model-Based Design Approach for the COTS Guidance, Navigation and Targeting (GN&T) Algorithm and Software Development

Ian T Mitchell

Draper Laboratory

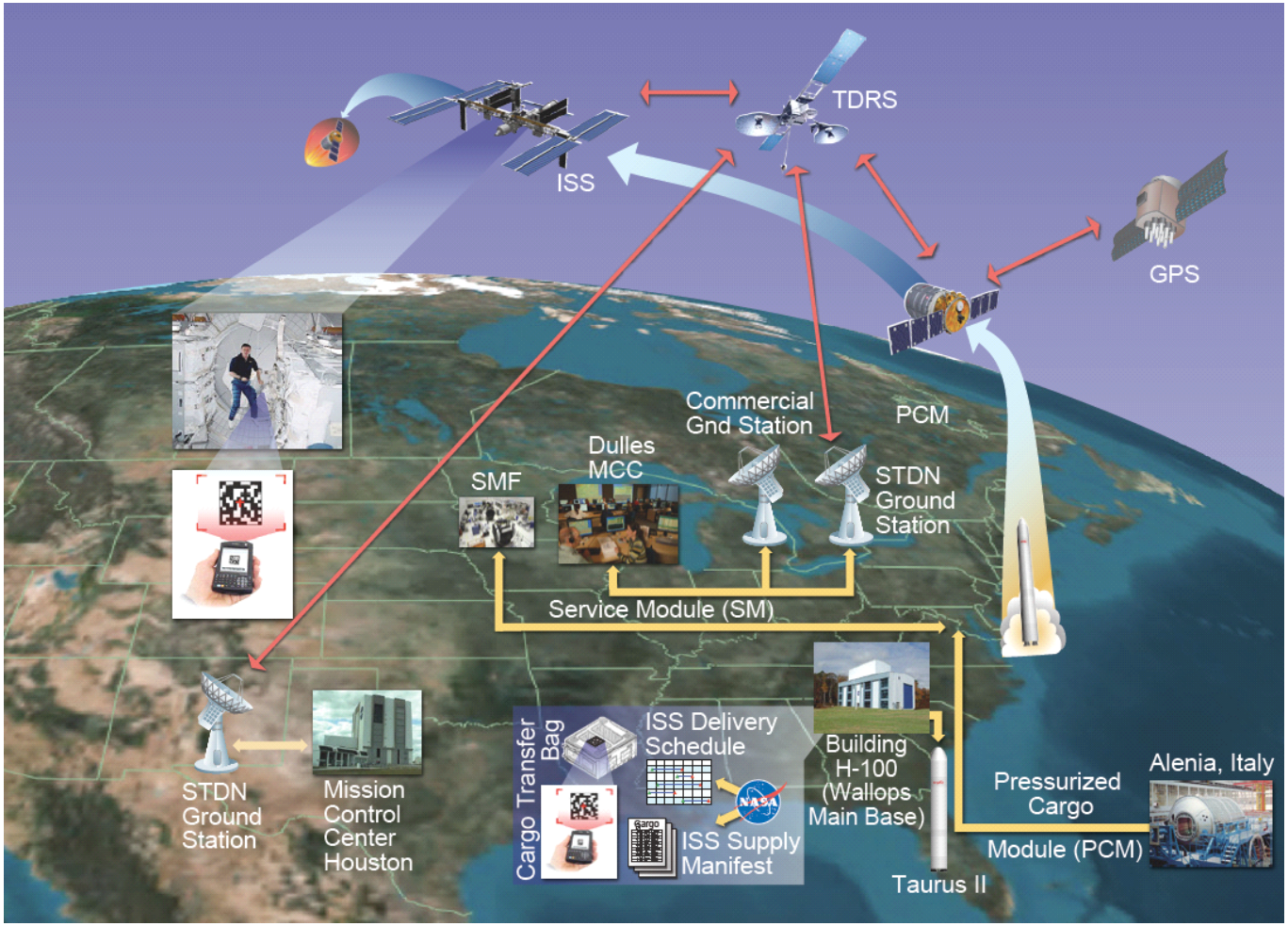
imitchell@draper.com

8th December, 2010

Agenda

- Overview
- Development Process
- Design and Implementation
- Unit Test Framework
- Monte-Carlo Simulation Framework
- Metrics
- Lessons Learned

Overview: Cygnus Mission and Team Members



http://www.orbital.com/NewsInfo/Publications/Cygnus_fact.pdf

Quick Facts

Mission Partners

Orbital Sciences Corporation
Prime contractor; engineering and development; Cygnus service module, mission and cargo operations

Thales Alenia Space
Pressurized cargo module

Draper Laboratory
Guidance, navigation and fault tolerant computer support

Mitsubishi Electric Corporation (MELCO)
Proximity location system

Odyssey Space Research
Visiting vehicle requirements support

SAS
Software Independent Verification and Validation (IV&V), engineering support

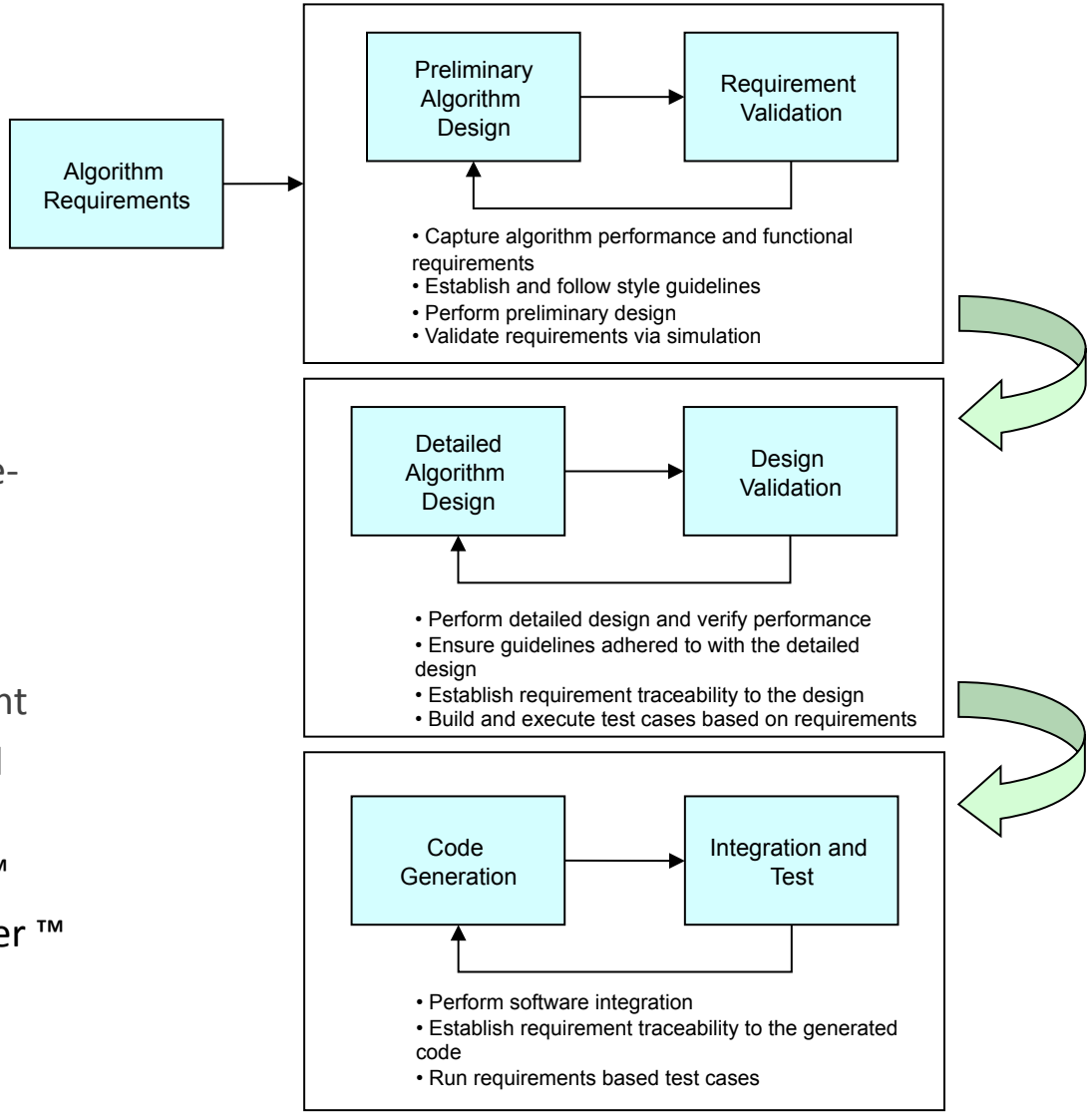
Cimarron
Mission Control Center - Houston (MCC-H) gateway developer

Overview: Draper GN&T Team

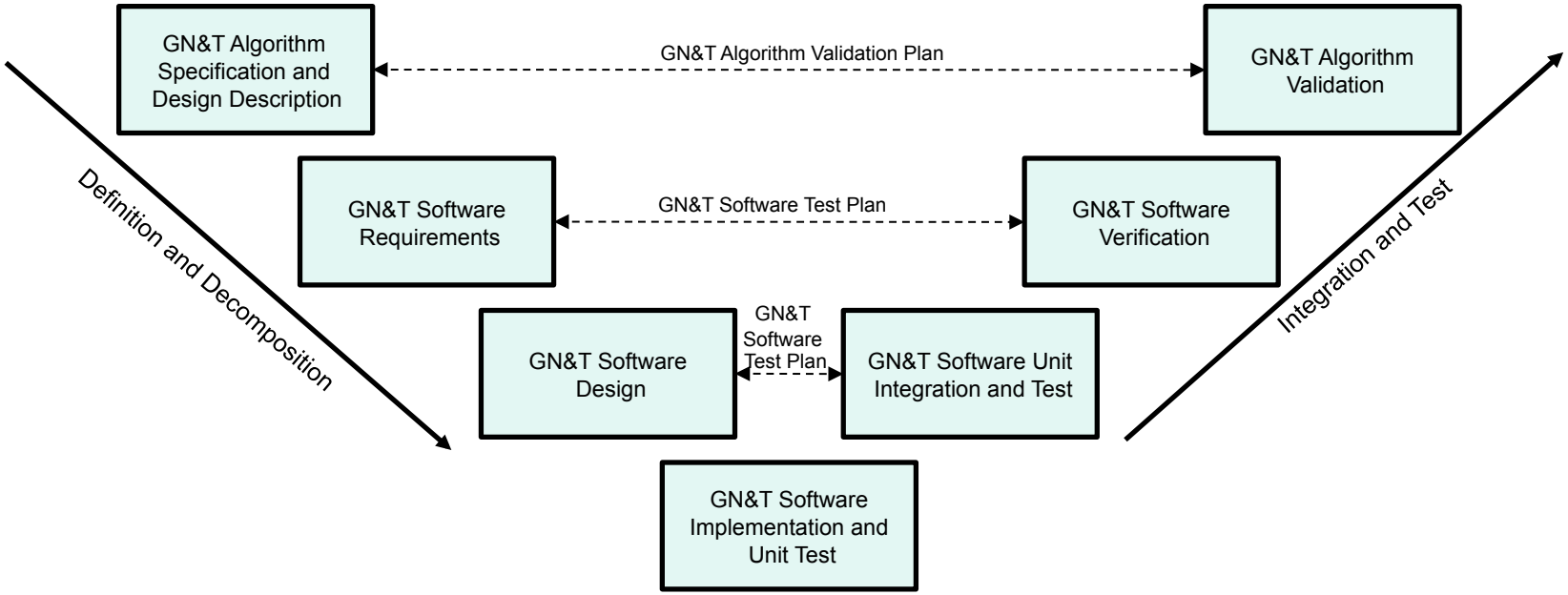
- David Benson (Simulation)
- Chris Bessette (FDI Algorithms/Software)
- Louis Breger (Proximity Guidance Algorithms/Software)
- Sungyung Lim (Proximity Guidance Algorithms)
- Ian Mitchell (Task Lead)
- Piero Miotto (Program Manager)
- Dick Phillips (FDI Algorithms)
- Russell Sargent (Targeting Algorithms/Software)
- Kurt Winikka (Software)
- Dave Woffinden (Algorithms)
- Renato Zanetti (Navigation Filter Algorithms/Software)

Overview: Model Based Design Workflow

- Process
  - Model-based design, graphical programming, Integrated Quality Management System (IQMS)
- Methods
  - Modeling, simulation, prototyping, automatic code generation, model testing and coverage, software-in-the-loop testing, processor-in-the-loop testing
- Tools
  - DOORS for requirements management
  - Simulink®, Stateflow® and Embedded MATLAB™
  - Simulink Verification and Validation ™
  - Real-Time Workshop Embedded Coder ™
  - C based tools (gcc, gcov and Perl) for source code verification

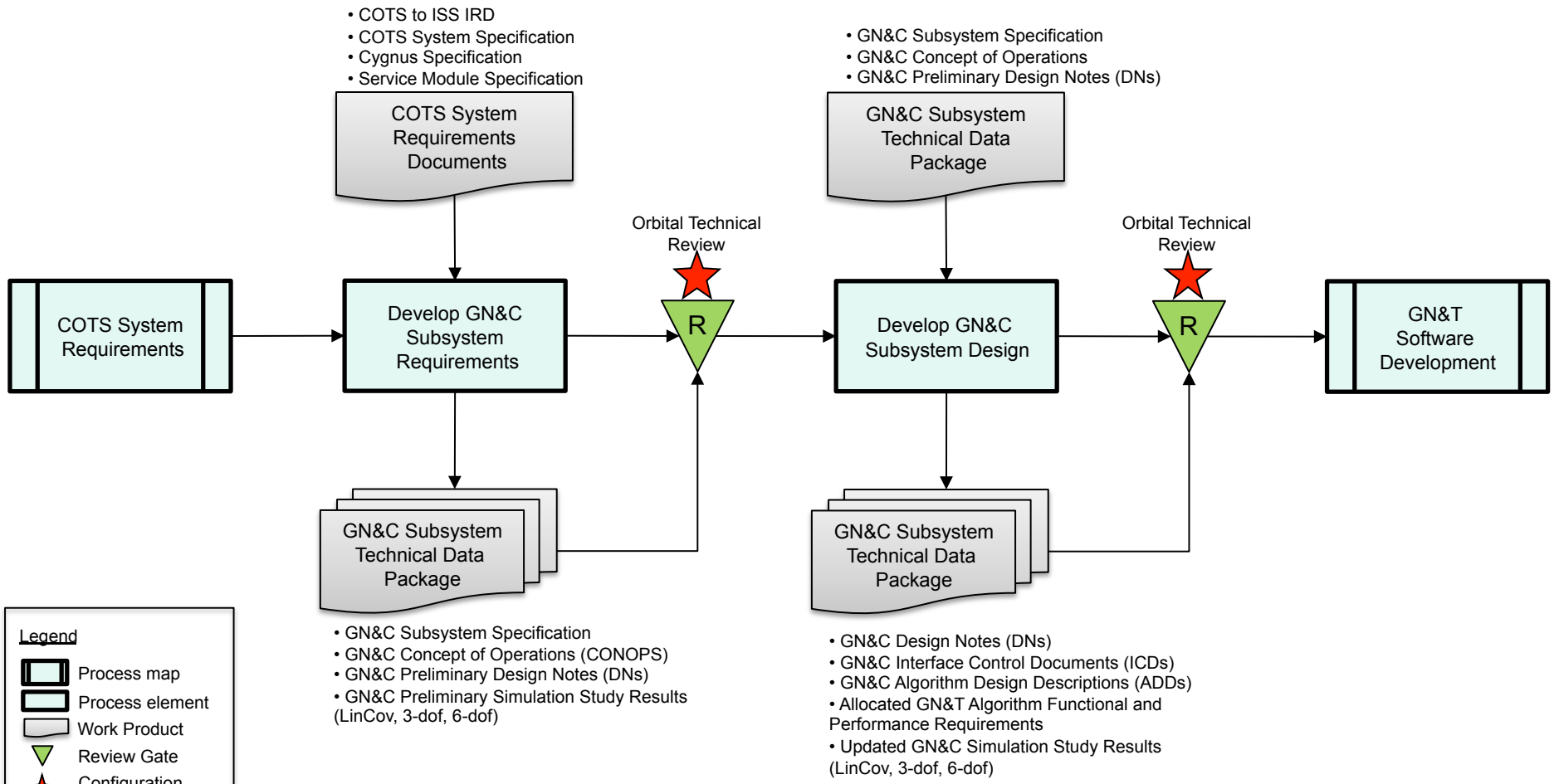


Overview: Development Process



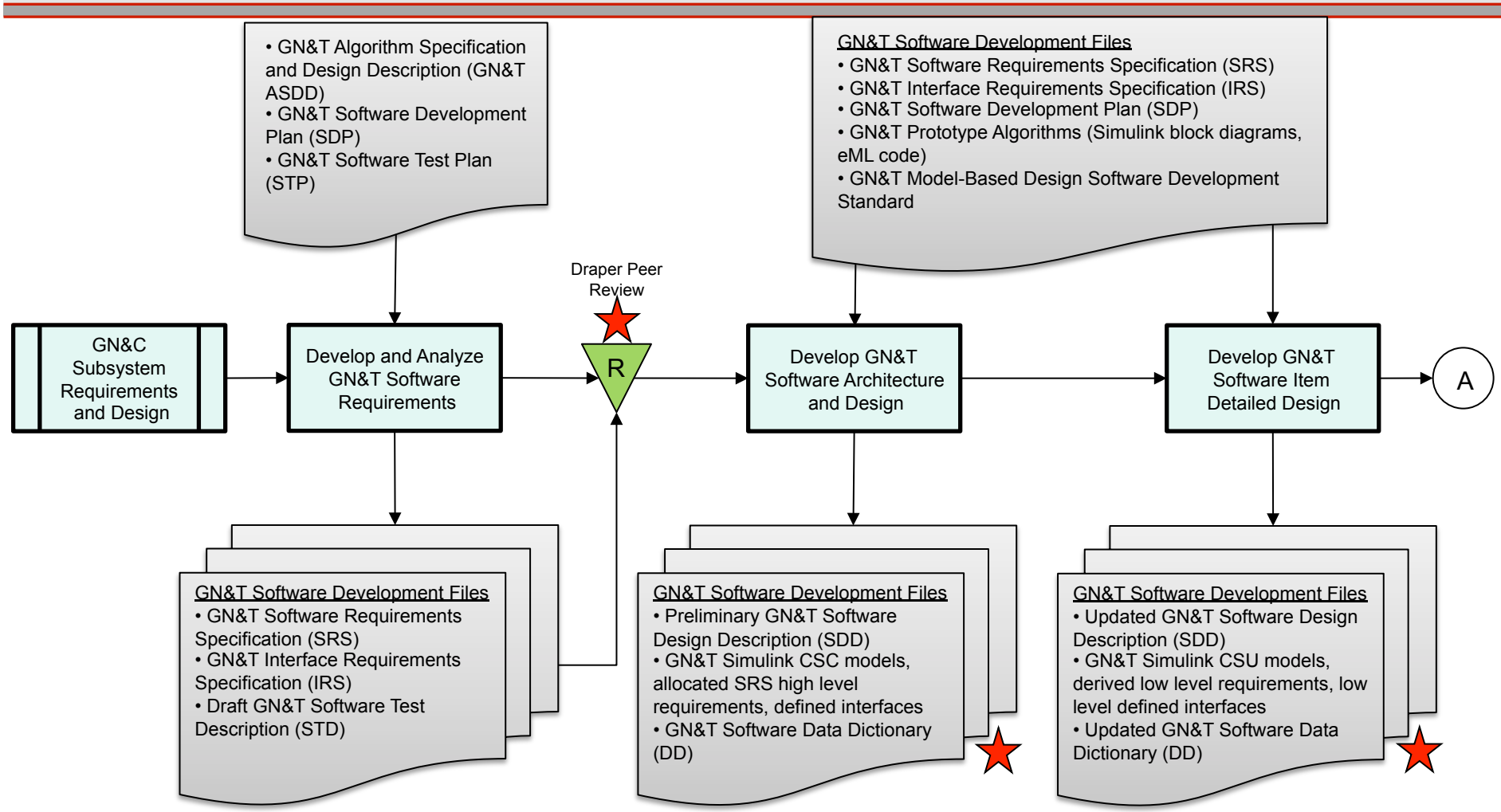
- V Model emphasizes requirements-driven design and testing
  - All GN&T high level design elements (CSCs) are traceable to one or more high level software requirement captured in the SRS/IRS
  - All GN&T low level design elements (CSUs) are traceable to one or more low level software requirement captured in the SDD
  - Every low level and high level requirement is addressed by one or more test cases captured in the STD
  - Bi-directional traceability ensures nothing is done unnecessarily and everything that is necessary is accomplished

Development Process: Subsystem Design



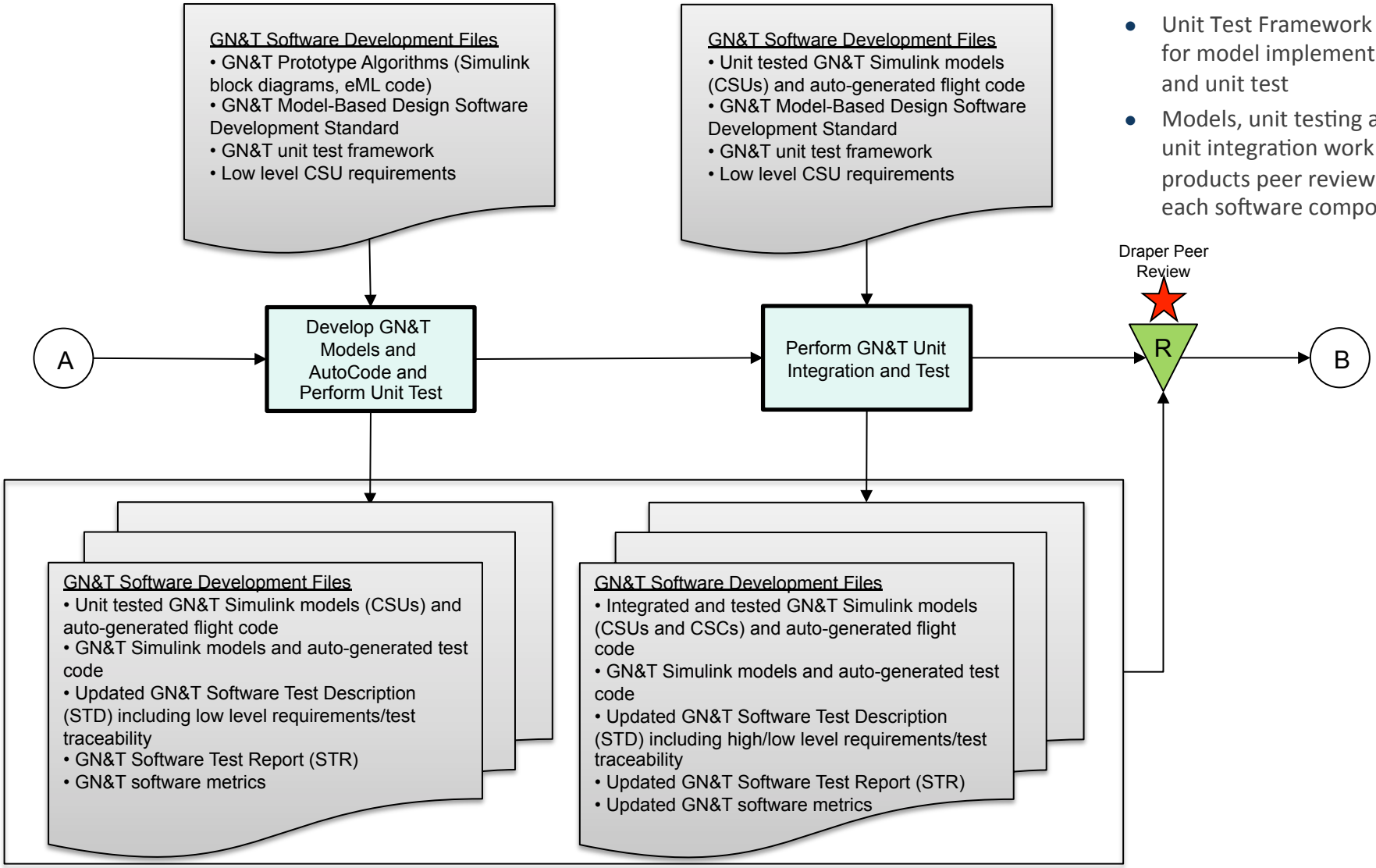
- System and GN&C subsystem performance requirements were derived from NASA requirements
- Prototype algorithms were developed in Simulink and analyzed and simulated in order to validate the subsystem performance requirements

Development Process: Software Design



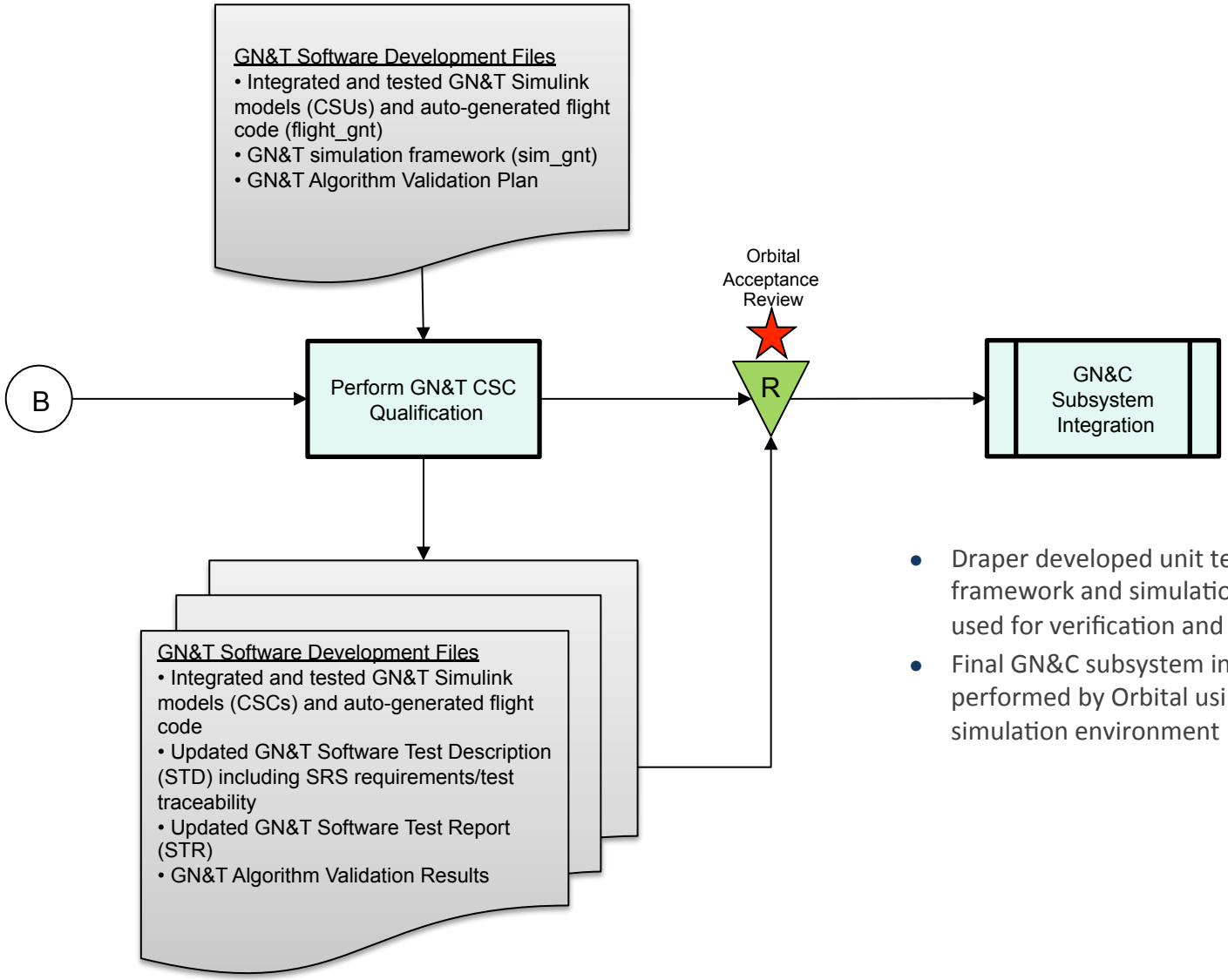
- Algorithm components were defined according to overall GN&C states and modes
- Functional or behavioral requirements for each of the algorithm components were derived
- Interfaces between Draper software components and Orbital software components were defined

Development Process: Implementation and Test



- Unit Test Framework used for model implementation and unit test
- Models, unit testing and unit integration work products peer reviewed for each software component

Development Process: Qualification



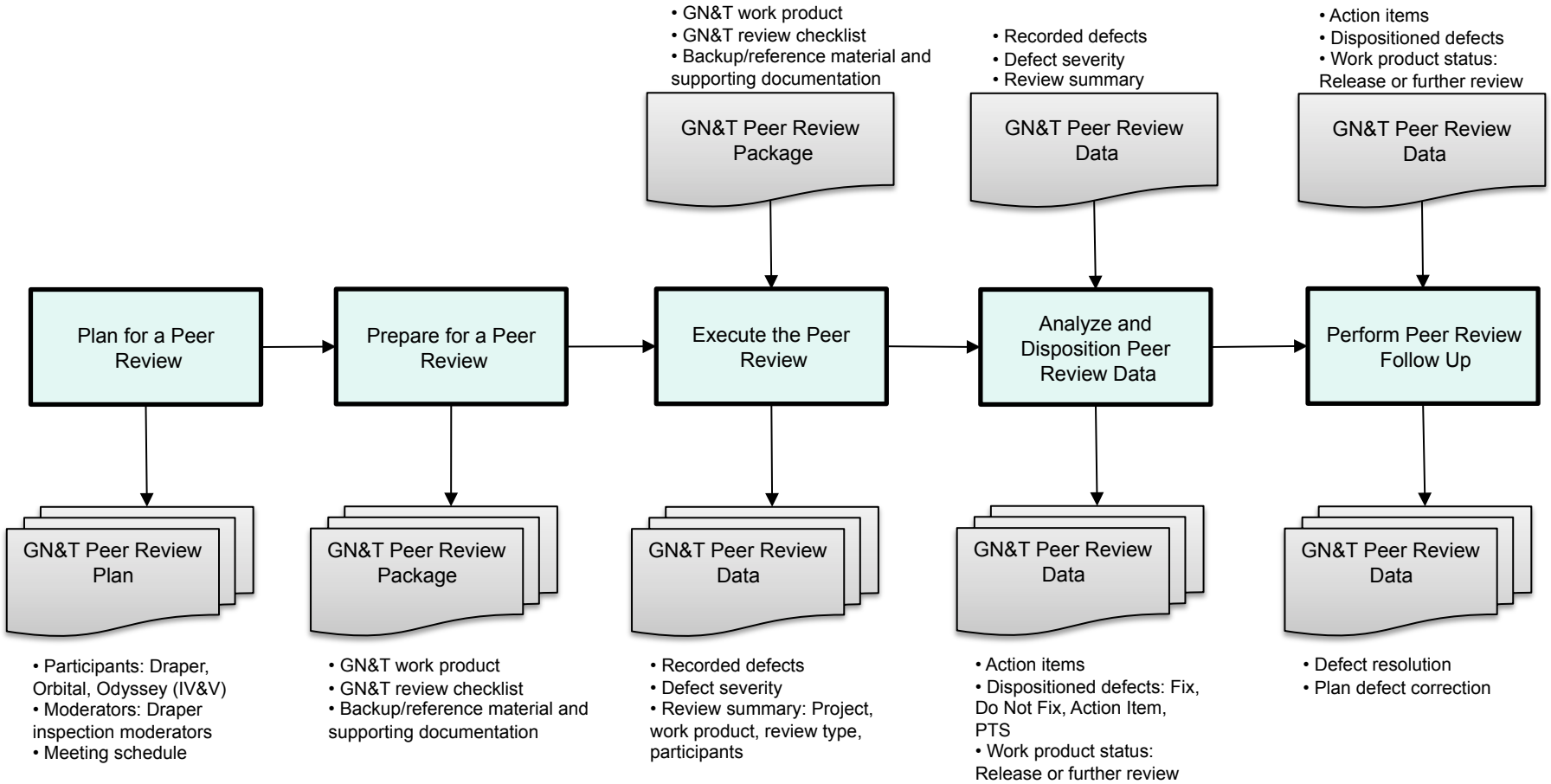
- Draper developed unit test framework and simulation framework used for verification and validation
- Final GN&C subsystem integration performed by Orbital using an Orbital simulation environment

Development Process: Build Planning

GN&T Software Development Activities	Build 1	Build 2	Build 3
GN&T Software Requirements	X	X	X
GN&T Software Design	X	X	X
GN&T Software Implementation and Unit Test	X	X	X
GN&T Software Unit Integration and Testing	X	X	X
GN&T Software Qualification			X
GN&T Software Delivery	X	X	X
GN&C/GN&T Software Integration	X	X	X
GN&C/GN&T Software/Hardware Integration and Testing			X
GN&C/GN&T Subsystem Qualification Testing			X
GN&T Software Configuration Management	X	X	X
GN&T Software Product Evaluation	X	X	X
GN&T Software Quality Assurance	X	X	X

- Incremental software life cycle model
  - Build 1: Interface build to mitigate risk associated with software integration
  - Build 2: Core guidance and targeting functionality to be used with perfect navigation
  - Build 3: Remaining navigation and FDI functionality

Development Process: Peer Review



- Peer review work products
  - All GN&T flight software documentation: SDP, STP, SRS, IRS, SDD, STD, STR, VDD
  - All GN&T Simulink modeling artifacts: Models, CSU/CSC test cases, test results, V&V results (coverage and complexity)

Development Process: Peer Review Checklist

- Model Peer Review Checklist
  - No peer review checklist existed for model-based design
  - GN&T team defined an initial peer review checklist to be used for all GN&T model peer reviews
- Automated Style Checking
  - Peer review defects can be substantially reduced with automated style checking
  - Simulink's style checker was not extensively used during the COTS program
  - Some custom scripts were developed to check specific items
  - Not sufficient time or resources to modify and/or develop own style checker

Software Model and Auto-Generated Code Review Checklist		**Review Date:**
Program:	**System:**	**Subsystem:**

Purpose
To aid reviewers and authors in the review of software models and auto-generated source code.

Instructions:
Add, delete, modify questions as appropriate to the program under consideration. Determine how feedback will be given (verbal, email, issues list). Review system engineering plans with the questions in mind.
Typical Supporting Documentation: Customer Requirements, System Requirements and Design, Software requirements and software design, Interface specifications

Is the model complete?

- ♣ Is the model complete, well commented and readable on a 19" monitor?
- ♣ Does the model contain all necessary information?
- ♣ Does the model comply with modeling standards for style, comments, code generation settings etc?
- ♣ Have inputs, including parameters and constants, into the model been clearly identified?
- ♣ Is wording and labeling understandable?
- ♣ Has line crossing been minimized without losing signal flow?
- ♣ Is timing inherited?
- ♣ Do inports and outports have the same names as the signals that pass through them?
- ♣ Are model interfaces clearly and correctly defined?
- ♣ Have all modeling standard exceptions been documented?

Is the model correct?

- ♣ Have the algorithmic designs been correctly modeled with the appropriate blocks?
- ♣ Is the modeling choice an acceptable translation of the detailed design?
- ♣ Are loops within eML entered and exited at the normal entry/exit points?
- ♣ Are error conditions handled in a consistent, non-destructive manner?
- ♣ Are increment/decrement counts properly initialized?
- ♣ Has the model generated code successfully?

Is the model clear?

- ♣ Is the model well decomposed and easy to follow?
- ♣ Do the comments fully describe what is happening in each section of the model?
- ♣ Do all eML functions have a return value?

Is the data handled correctly?

- ♣ Has all of the data been properly defined and initialized before usage?
- ♣ Is all defined data utilized?
- ♣ Are all constants defined?
- ♣ Are meaningful labels and variable names used?
- ♣ Have file and variable naming conventions been followe

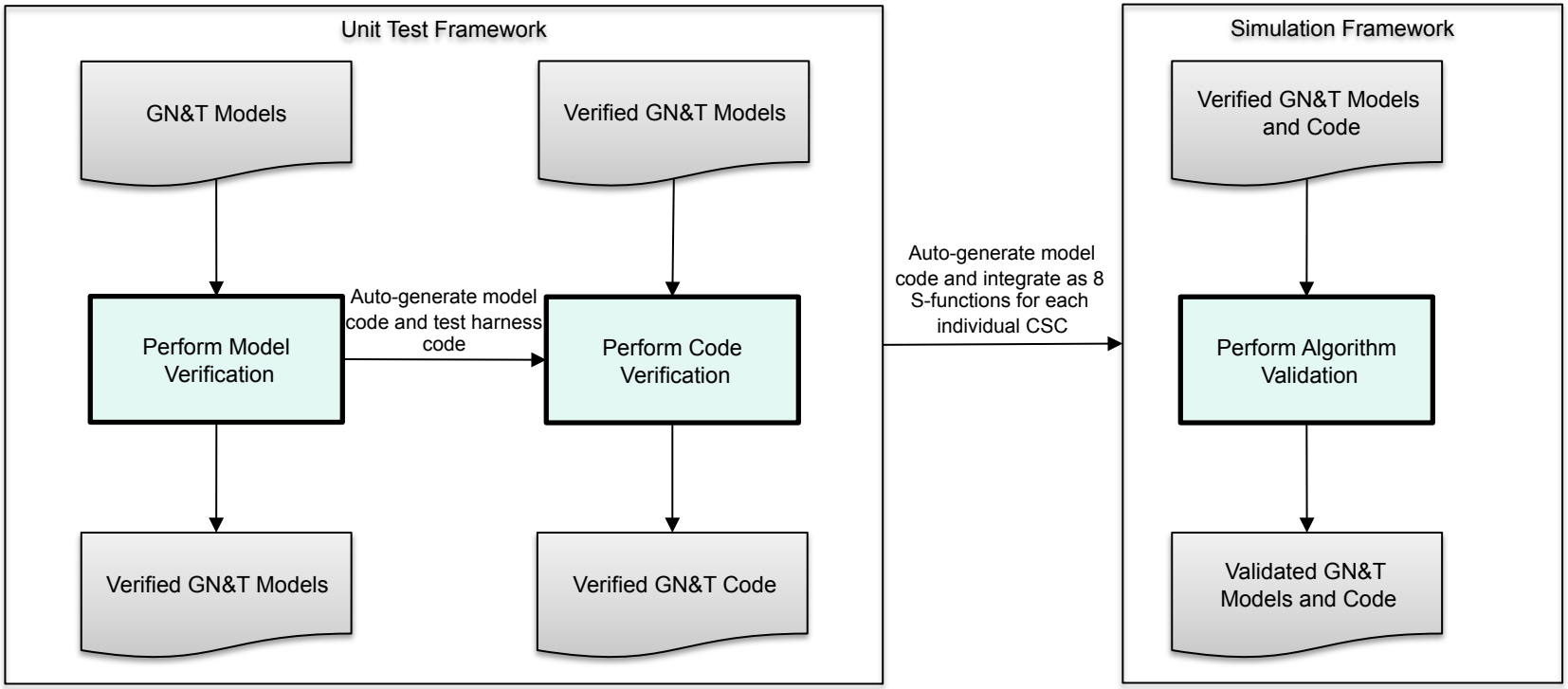
Is the model maintainable?

- ♣ Is the model easily extensible?
- ♣ Is the model testable in the existing environment?
- ♣ Does model complexity satisfy the goal of the program?

Is the auto-generated code satisfactory?

- ♣ Was the code auto-generated with the correct real-time workshop embedded coder settings?
- ♣ Are models, blocks, eML functions and eML code fully traceable to the generated code?
- ♣ Does code complexity satisfy the goal of the program?

Development Process: Test Approach



- Three Test Phases for GN&T Verification and Validation
  - GN&T Model Verification: Simulink environment for both analysis and test (Simulink®, Real-Time Workshop Embedded Coder ™, Simulink Verification and Validation ™) using unit test framework
  - GN&T Code Verification: C code environment for both analysis and test (gnu gcc, gnu gcov, Perl) using unit test framework
  - GN&T Algorithm Validation: Simulation environment (Simulink®, Real-Time Workshop Embedded Coder ™) using simulation framework

Development Process: Test Approach

Test Phase	Test Levels					Test Classes							
	Simulink Unit Verification	Code Unit Verification	CSC Integration Test	Regression Test	CSC Qualification Test	Functional	Monte-Carlo	Performance	Stress	Boundary	Error Conditions	Model Coverage & Analysis	Code Coverage & Analysis
GN&T Model Verification	1-3		2,3	2,3	3	1-3			3	2,3	3	2,3	
GN&T Code Verification		1-3	2,3	2,3	3	1-3		2,3	3	2,3	3		2,3
GN&T Algorithm Validation			2,3	2,3	3	2,3	2,3						

- Test Levels
  - Unit testing, integration testing, regression testing and qualification or acceptance testing
- Test Classes
  - Functional, Monte-Carlo, performance, stress, boundary, error conditions, coverage and analysis

Development Process: Test Approach

- Simulink Unit Verification
  - Uses Draper developed unit test framework and employs Simulink V&V toolbox for model coverage analysis and cyclomatic complexity metrics.
  - Test classes include functional, error conditions, boundary and structural testing
- Code Unit Verification
  - Uses Draper developed unit test framework and gnu tools for compiling auto-generated flight code and test code and generating code coverage, cyclomatic complexity, code timing and code stack usage metrics.
  - Test cases are all repeats of the Simulink unit verification test cases
- CSC Integration Test
  - Repeats Simulink Unit Verification and Code Unit Verification for integrated units. Top level CSC tests integrate Simulink S-function within unit test framework.
  - Uses simulation framework to validate algorithms and ensure performance requirements are met. CSCs are integrated as 8 S-functions for each individual CSC.
- Regression Test
  - Repeats test levels for previously tested CSUs and CSCs
- CSC Qualification Test
  - Formal execution of unit verification and CSC integration tests

Development Process: Test Approach

- Functional Tests
  - Functional test which verifies generated output in response to selected inputs and expected output
- Monte-Carlo Tests
  - Functional test which evaluates algorithm performance against validation criteria
- Performance Tests
  - Functional test which evaluates timing performance and stack usage of units, integrated units and CSCs
- Stress Tests
  - Functional tests which verifies software functionality under stress conditions such as extreme input values. Ensure that these conditions do not cause abnormal behavior (divide by zero exceptions for example) or that such faults are detected, isolated and reported.
- Boundary Tests
  - Structural test which verifies boundary values are tested. For example, loop boundaries at 0, 1, many, max-1, max, max+1 iterations
- Erroneous Input Tests
  - Functional test verifies software responds correctly to erroneous input values that may cause algorithm exceptions

Development Process: Test Approach

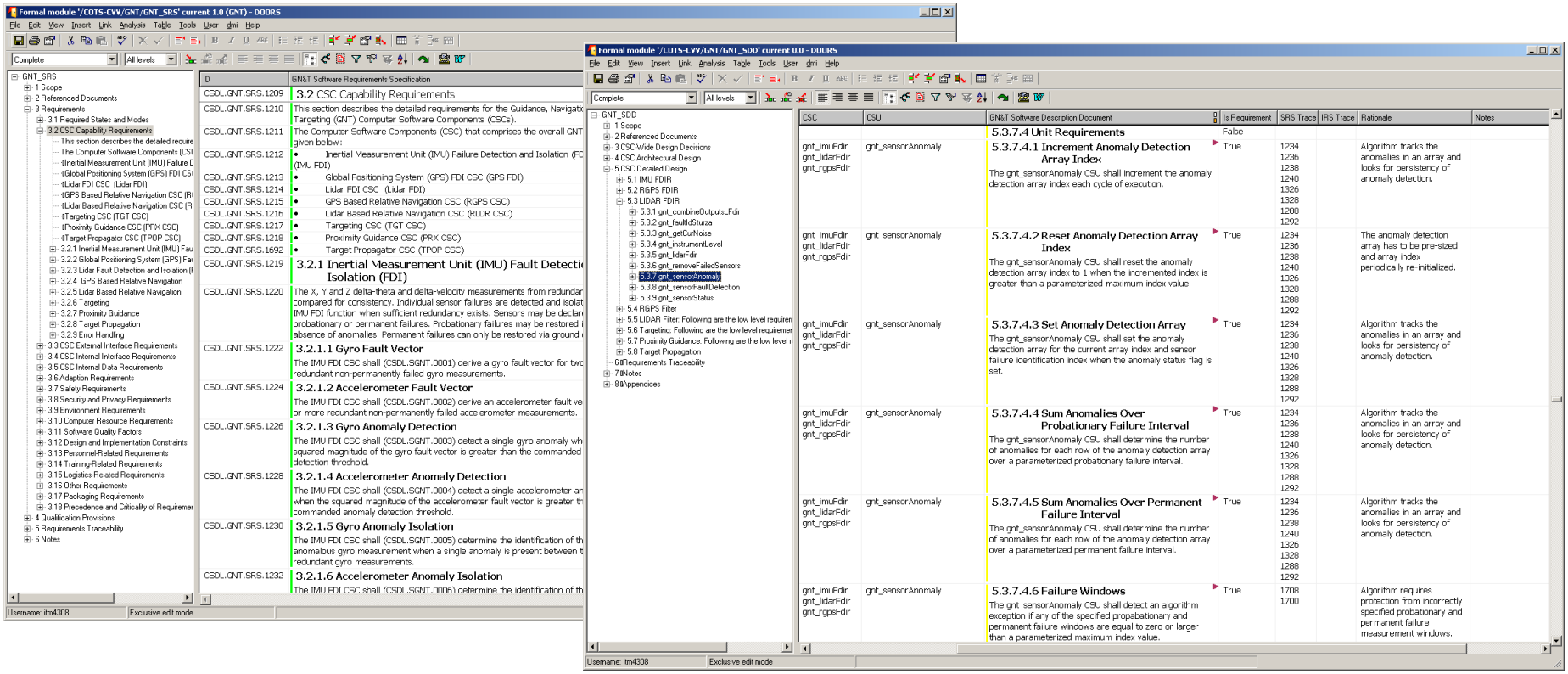
- Simulink Model Coverage and Analysis
  - Structural test which evaluates MC/DC model coverage by a particular test data set
  - Analysis performed to determine requirement traceability, test coverage, model complexity analysis, and model standards compliance
- Code Coverage and Analysis
  - Structural test which evaluates percentage of code exercised at least once by a particular test data set (the same test data set used in Simulink coverage testing)
  - Analysis performed to determine test coverage, code complexity, timing and memory usage

Design and Implementation: Requirements

- COTS GN&C Subsystem Specification
  - NASA requirements analyzed to determine system and GN&C subsystem requirements
  - Allocations to GN&T
- COTS GN&T Algorithm Specification and Design Description
  - Captured algorithm performance requirements and algorithm descriptions
- COTS GN&T Software Requirements Specification
  - Identified GN&T software components (CSCs) and high level functional requirements and interfaces
- COTS GN&T Software Design Description
  - Identified GN&T software units (CSUs) and low level functional requirements including error handling

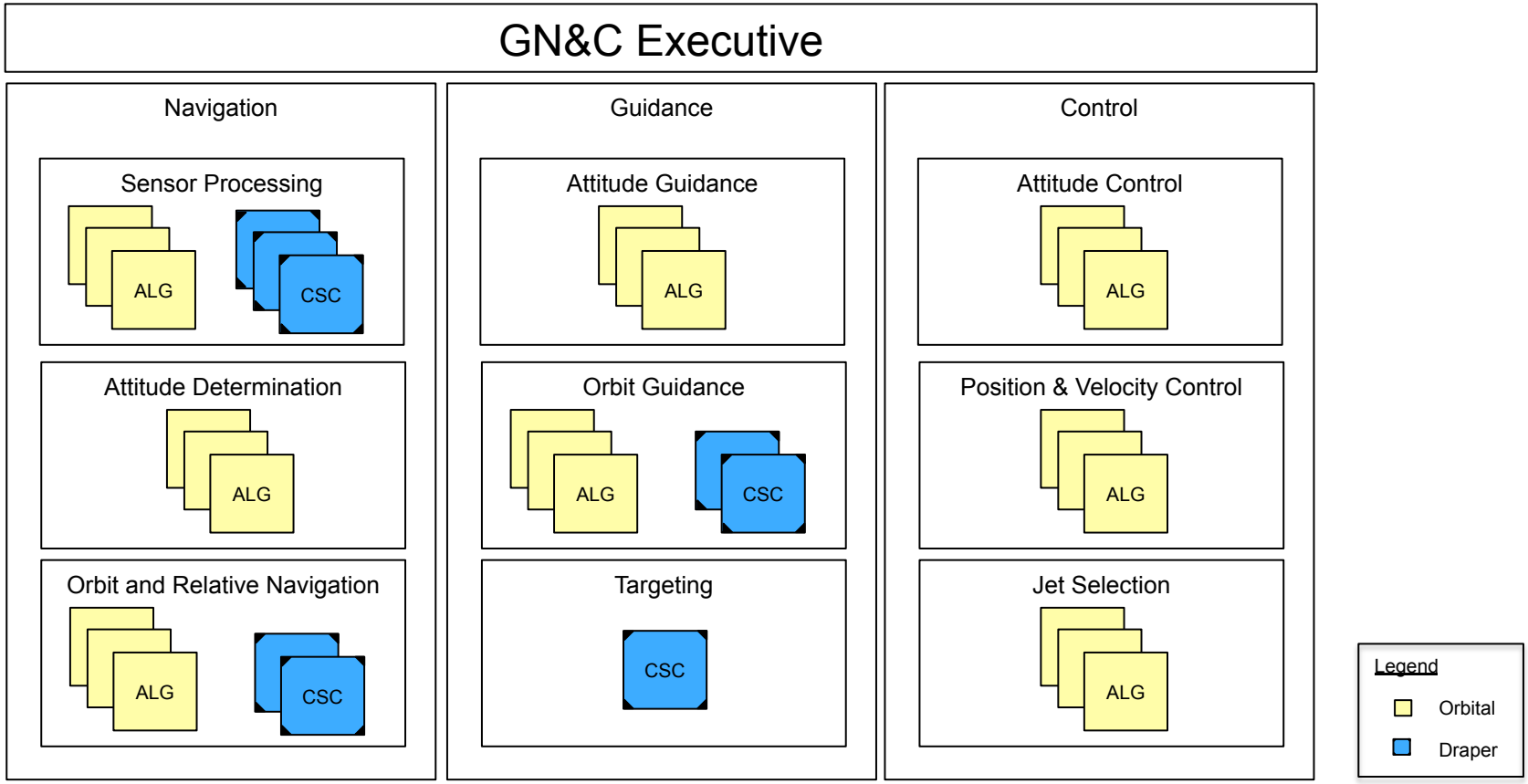
Specification Tree	Examples
COTS GN&C Subsystem Specification	Prior to Approach Initiation, the GN&C shall maintain nominal approach and planned contingency trajectories, including expected dispersions, to keep the COTS vehicle outside the Approach Ellipsoid.
COTS GN&T Algorithm Specification and Design Description	The GN&T shall autonomously compute proximity operations targeted maneuvers to acquire and maintain the 1.4 km co-elliptic such that the 3-sigma trajectory dispersions represented in the ISS LVLH frame after acquisition of the 1.4 km co-elliptic do not exceed +/- 150 m in the out-of-plane and radial directions.
COTS GN&T Software Requirements Specification	The gnt_targeting CSC shall provide targeting modes of operation for Idle, Initial Altitude Adjust, First Mid-Course Correction, Second Mid-Course Correction, Final Altitude Adjust, Final Mid-Course Correction and R-Bar Acquisition.
COTS GN&T Software Design Description	The gnt_chuggerBy CSU shall propagate the Cygnus J2000 ECI orbital state iteratively over N cycles defined by the delta-V burn duration and a parameterized maximum step size. The gnt_chuggerBy CSU shall detect an algorithm exception if the computed N cycles is greater than or equal to a parameterized threshold

Design and Implementation: Requirements



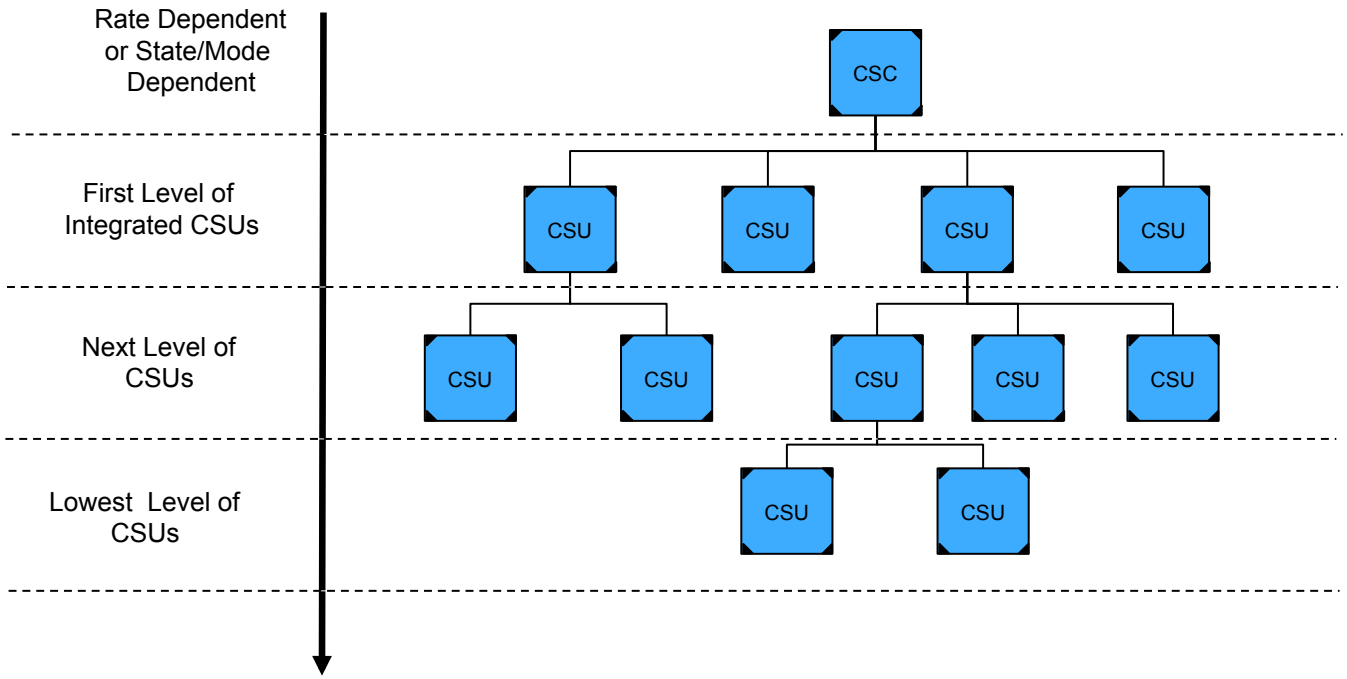
- DOORS used for managing requirements and maintaining bidirectional traceability
- Snapshot above is the COTS GN&T Software Requirements Specification (SRS) on the left and the COTS GN&T Software Design Description (SDD) on the right. Trace links to the COTS GN&T Algorithm Specification and Design Description (ASDD), the SDD and SRS are shown.
- The COTS GN&T Software Test Description (STD) not shown is also maintained within DOORS and provides trace links from test cases to requirements.
- CSC modules and CSU names are also shown for each of the CSU requirements.
- Chose not to include requirements in the Simulink models themselves (via Simulink V&V toolbox)

Design and Implementation: Architecture



- Orbital using Rhapsody® modeling tool for algorithm block development and state chart tool for GN&C states and modes sequencer development
- Orbital using an object oriented approach for the GN&C executive and algorithm blocks. There are 40 algorithm blocks including the Orbital wrappers for the Draper CSCs.
- Orbital GN&C executive calls algorithm blocks and CSCs at the appropriate rate (5 Hz, 1 Hz, 0.1 Hz) and in the appropriate GN&C state and mode

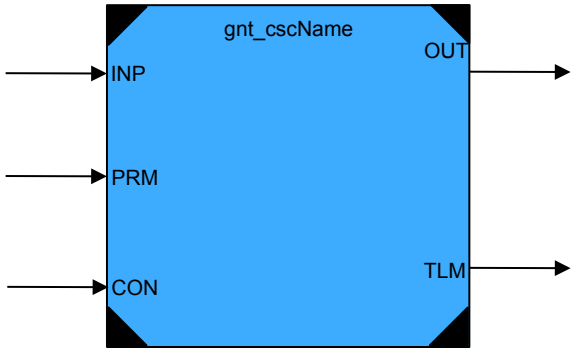
Design and Implementation: Architecture



- Total of 8 Draper CSCs: gnt_imuFdir, gnt_rgpsFdir, gnt_lidarFdir, gnt_rgpsFilt, gnt_lidarFilt, gnt_targeting, gnt_proxGuid and gnt_targetProp. Each CSC has a common interface (INP, PRM, CON, OUT and TLM buses) and common methods for initialization, calling and termination.
- All CSCs and CSUs are defined as Model Reference blocks. Model Reference blocks allow models to be reused and shared, facilitate parallel development and configuration management.
- CSUs were defined according to a functional decomposition and the general guideline to have a CSU cyclomatic complexity < 20
- Depth of decomposition is different for each CSC: gnt_imuFdir (2 deep), gnt_lidarFdir (2), gnt_rgpsFdir (3), gnt_rgpsFilt (5), gnt_lidarFilt (4), gnt_targeting (5), gnt_proxGuid (4) and gnt_targetProp (2)

Design and Implementation: CSC Definitions

- Model Components (8 CSCs)
  - Inertial Measurement Unit (IMU) Failure Detection and Isolation (FDI) CSC: gnt_imuFdir
  - Global Positioning System (GPS) FDI CSC: gnt_rgpsFdir
  - Lidar FDI CSC: gnt_lidarFdir
  - GPS Based Relative Navigation CSC: gnt_rgpsFilt
  - Lidar Based Relative Navigation CSC: gnt_lidarFilt
  - Targeting CSC: gnt_targeting
  - Proximity Guidance CSC: gnt_proxGuid
  - Back-up Target Propagation CSC: gnt_targetProp
- Model Units (96 CSUs)
  - Each CSC is comprised of unique and/or common CSUs
  - Allowable primitive blocks defined in SCLIB
- Model Utilities (51 Utilities)
  - Low level eML utilities for quaternion math, GPS time calculations, guarded functions, table lookup etc



CSC Model Reference

CSC	Total CSUs
gnt_imuFdir	7(1)
gnt_rgpsFdir	21(1)
gnt_lidarFdir	8 (1)
gnt_rgpsFilt	19 (2)
gnt_lidarFilt	15 (2)
gnt_targeting	23 (3)
gnt_proxGuid	20 (3)
gnt_targetProp	2 (4)

Notes:
(1) Includes 6 CSUs common to each Fdir CSU
(2) Includes 4 CSUs common to each Filt CSU
(3) Includes 1 CSU common to targeting and guidance CSUs
(4) Includes 2 CSUs common to each Filt CSU

Design and Implementation: CSC Definitions

- CSC Common Interface Definition
  - Each of the 8 CSCs use a common interface which is defined by a data structure that contains all the data associated with the CSC
  - Input data structure: contains all the dynamic inputs to the CSC
  - Parameter data structure: contains algorithm parameters that may change during the mission
  - Constant data structure: contains physical constants that change infrequently if at all
  - Output data structure: contains all the dynamic outputs from the CSC that are destined for other CSCs
  - Telemetry data structure: contains all the telemetry data that is transmitted to the ground

Object Data Structure

```
struct GNT_cscname_Strc
{
        INP_gnt_cscname inp;     /* Input      */
        PRM_gnt_cscname prm;     /* Parameters */
        CON_gnt_cscname con;     /* Constants  */
        OUT_gnt_cscname out;     /* Output     */
        TLM_gnt_cscname tlm;     /* Telemetry  */
};
typedef struct GNT_cscname_Strc GNT_cscname_StrcType;
```

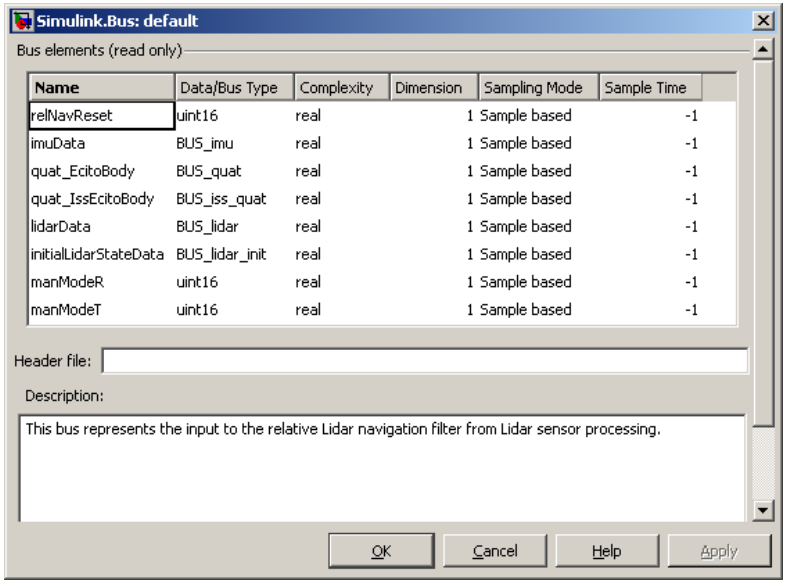
- CSC Common Method Definition
  - Each of the 8 CSCs is invoked using 3 methods
  - The initialization method is called before starting to use the call method
  - The call method is used to call the CSC and generate the output and is executed at the particular rate of the CSC
  - The terminate method may be called as part of a system shutdown. Currently, terminate functions are not utilized.

- Initialization method: `void GNT_cscname_Init(GNT_cscname_StrcType *intfc);`
- Call method: `void GNT_cscname_Call (GNT_cscname_StrcType *intfc);`
- Termination method: `void GNT_cscname_Term(GNT_cscname_StrcType *intfc);`

Design and Implementation: CSC Interfaces

Field Name	Component	Field Identifier	Field Description	Source	Destination	Frequency	Units	Min Value	Max Value	Default Value	Bits	Type	Bus Type
INP_gnt_lidarFilt	NAV	relNavReset	reset lidar filter to idle mode	Executive	Relative Navigation	5 Hz	N/A	0	1	0	16	uint	N/A
INP_gnt_lidarFilt	NAV	imuData	IMU data input.	Sensor Processing	Relative Navigation	5 Hz	N/A	N/A	N/A	N/A	528	bus	BUS_imu
INP_gnt_lidarFilt	NAV	quat_EcitoBody	Cygnus inertial to body attitude. This is assumed to be at the current time.	Inertial Attitude Navigation	Relative Navigation	5 Hz	N/A	N/A	N/A	N/A	256	bus	BUS_quat
INP_gnt_lidarFilt	NAV	quat_IssEcitoBody	ISS inertial to body attitude.	Inertial Attitude Navigation	Relative Navigation	5 Hz	N/A	N/A	N/A	N/A	400	bus	BUS_iss_quat
INP_gnt_lidarFilt	NAV	lidarData	Lidar data input from selected Lidar.	Sensor Processing	Relative Navigation	5 Hz	N/A	N/A	N/A	N/A	384	bus	BUS_lidar
INP_gnt_lidarFilt	NAV	initialLidarStateData	Filter initialization and re-initialization data.	Executive	Relative Navigation	5 Hz	N/A	N/A	N/A	N/A	912	bus	BUS_lidar_init
INP_gnt_lidarFilt	NAV	manModeR	Rotational maneuver mode. Set to 1 when vehicle is performing an attitude maneuver.	Executive	Relative Navigation	5 Hz	N/A	0	1	0	16	uint	N/A
INP_gnt_lidarFilt	NAV	manModeT	Translational maneuver mode. Set to 1 when vehicle is perforing a translational maneuver.	Executive	Relative Navigation	5 Hz	N/A	0	1	0	16	uint	N/A
INP_gnt_lidarFilt	*Bus*	*INP_gnt_lidarFilt*	*This bus represents the input to the relative Lidar navigation filter from Lidar sensor processing.*										

- A Data Dictionary, captured in Microsoft Office Excel, was created to manage CSC I/O. A representative bus definition for the gnt_lidarFilt CSC input bus is shown above and the bus itself, INP_gnt_lidarFilt, is shown at right.
- MATLAB™ scripts were developed to automatically create the Simulink bus I/O definitions for each CSC. Each CSC has a dedicated worksheet within the data dictionary.
- Orbital naming conventions (including drinking camel style) were adopted at the CSC interface
- A common worksheet contains all the common bus types that are used for input, parameters, constants, output and telemetry, BUS_imu and BUS_quat for example



Design and Implementation: Error Handling

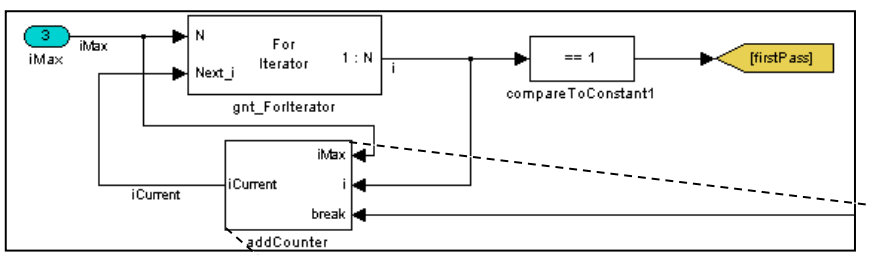
- Algorithm Exceptions
  - Each CSC detects and reports potential algorithm exceptions via the CSC interface
  - Algorithm developers define the types of algorithm exceptions that are unique to his/her algorithm
- Math Exceptions
  - Each CSC detects and reports potential math exceptions via the CSC interface
  - Guarded functions for division, square root, exponential, arc-sine, and arc-tangent are part of the model utilities and are used extensively within eML blocks.
- Common Exception Reporting
  - Algorithm exceptions or asserts are issued directly through a EXASSERT_EML function that is embedded within an eML block.
  - Math exceptions are issued indirectly by each of the utility functions. Each utility function includes a call to the EXASSERT_EML function if a potential math exception is detected.
  - Each exception, whether it is an algorithm exception or a math exception, has a unique ID associated with it, referred to as the assertion number.
  - The assertion number is reported as an output of each CSC

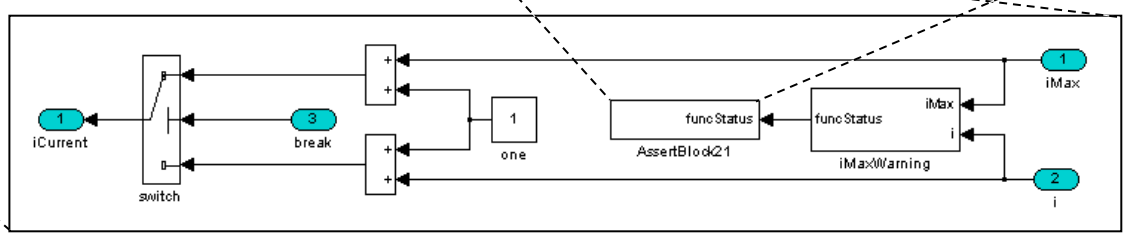
Design and Implementation: Error Handling

- Example: for loops within gnt_targeting
  - Each for loop instance determines whether the maximum number of iterations allowable is reached
  - An algorithm exception is reported if `i` reaches `iMax`
  - EXASSERT_EML is called when this condition is detected
  - EXASSERT_EML function performs different actions depending on calling environment

- When auto-coding, EXASSERT_EML is autocoded as a macro defined as the C function gnt_exAssert
- The C function gnt_getExAssert called within an eML blocks is used to access the global assert variable and report as an output

```
function assertOut = EXASSERT_EML( ...
  assertInt, funcStatus, errorThreshold, cscNum) %#eml
% EXASSERT_EML If needed, trigger assert in C code or Simulink

cscNumIn = cscNum;

assertOut = int32(assertInt);

if funcStatus >= errorThreshold

  if strcmp(eml.target,'rtw'),      % Follow if in RTW
    eml.ceval('ISSUE_ASSERT', assertOut, int32(cscNum) );
```

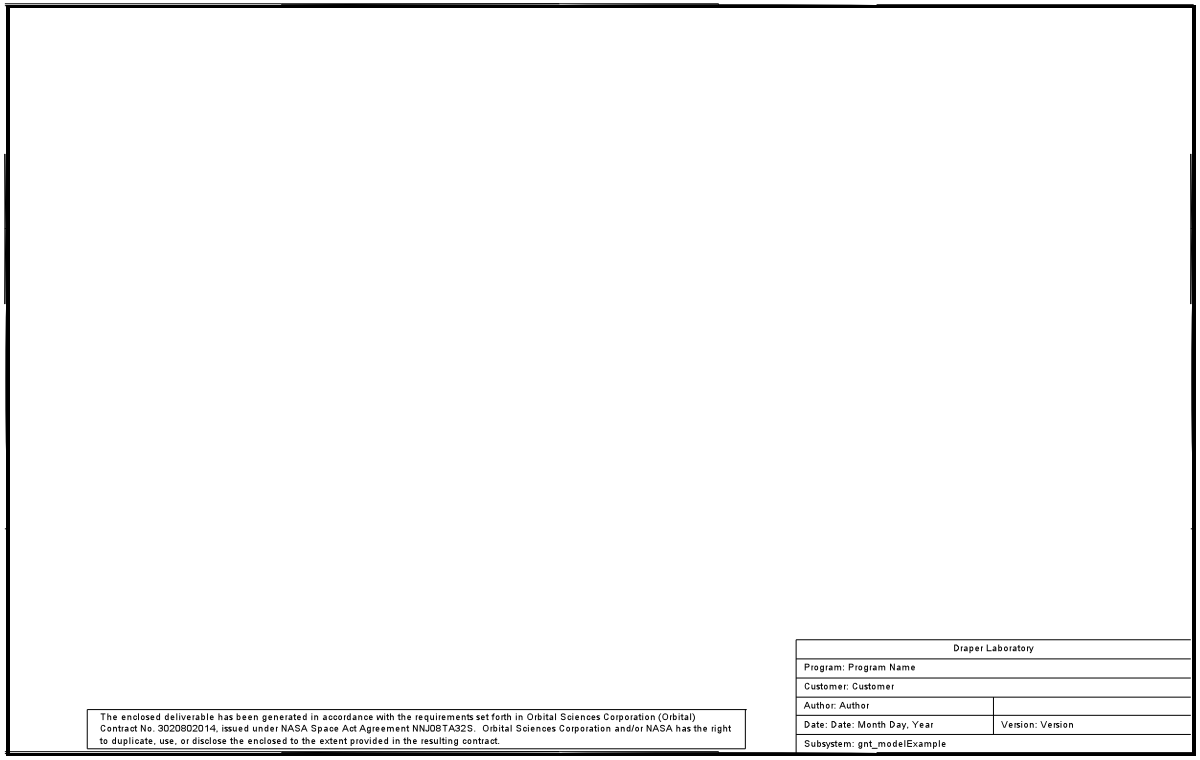




Design and Implementation: Standards

- GN&T Model-Based Design Software Development Standard
  - Defines development tools to be used
  - Defines standards for model architecture, use of Simulink and use of Embedded MATLAB™ (eML)
  - Similar to MAAB standards but with some differences: Stateflow is restricted to test harness and simulation components as needed

Design and Implementation: Standards



- All Simulink block diagrams include a banner block that contains the standard distribution statement required for deliverable items
- Banner block also includes: Program Name, Customer, Author, Date of Last Modification, Version Number and Model Name.
- Model revision number for each .mdl is managed by version control tool and callbacks within each model. Auto-code for each model also has its own revision number

Design and Implementation: Standards

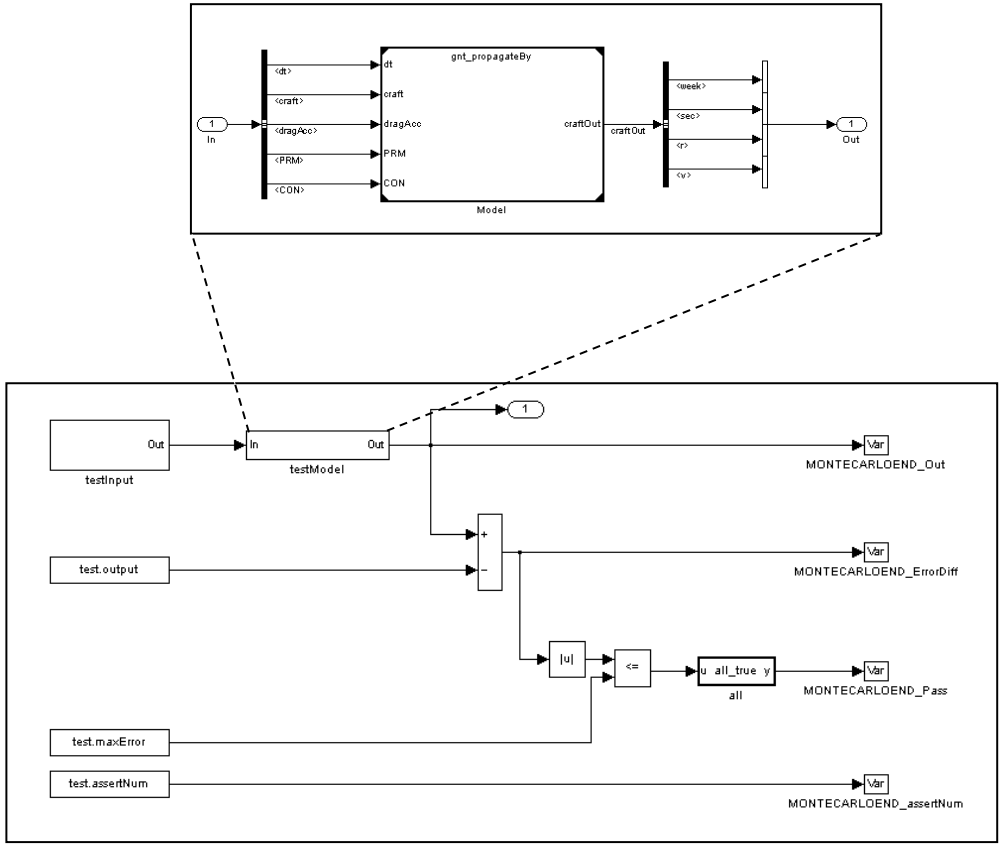
```
/*************************************************************************
*
 ** The Charles Stark Draper Laboratory, Inc.
 ** 555 Technology Square
 ** Cambridge, MA 02139-3563
 **
 ** File: gnt_targeting.c
 **
 ** Real-Time Workshop file version : 7.2  (R2008b)  04-Aug-2008
 ** TLC version                     : 7.2 (Aug  5 2008)
 **
 ** Model : Revision: 16141
 ** Code  : $Date: 2010-08-12 14:49:46 -0500 (Thu, 12 Aug 2010) $
 ** Code  : $Revision: 16218 $
 **
 **                           PROPRIETARY RESTRICTION
 **
 ** The enclosed deliverable has been generated in accordance with the
 ** requirements set forth in Orbital Sciences Corporation (Orbital) Contract
 ** No. 3020802014, issued under NASA Space Act Agreement NNJ08TA32S.
Orbital
 ** Sciences Corporation and/or NASA has the right to duplicate, use, or
 ** disclose the enclosed to the extent provided in the resulting contract.
 **

*************************************************************************
/
```

- Auto-generated code revision number for each .mdl is also managed by version control tool.
- The declarations at top of each .c and .h file show revision number for the auto-generated code as well as the revision number of the .mdl from whence it came

Unit Test Framework

- CSC/CSU Unit Testing
  - CSUs, integrated CSUs and top level CSCs each have an associated test harness (testBase.mdl)
  - Test script executes the Model Reference block with pre-defined inputs and compares actual output to pre-defined expected output
  - Test script auto-generates code for the Model Reference block, executes the auto-code and compares actual output with Simulink output
  - The whole test harness is auto-coded and the same test cases are run during Code Verification

- Model Reference is run in "Normal" mode for code coverage and "Accelerator" mode for code generation or when at the lower level in an integrated unit
- A CSC is auto-coded and re-integrated as an S-function



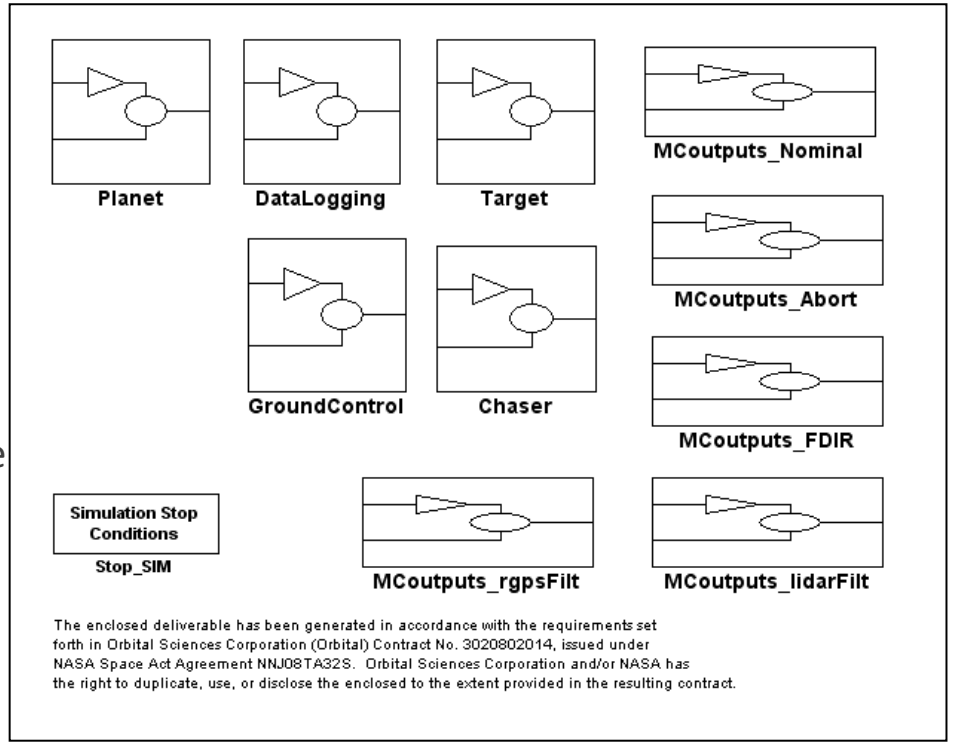
Unit Test Framework

- CSC/CSU Unit Testing
  - Testing is script intensive both for Model Verification and Code Verification
  - Each test script has a standard format for defining test inputs, expected test outputs, and any expected asserts during the execution of the test
  - Test script can execute both a Simulink test and auto-code test or Simulink test only
  - Test input and test output are stored as .mat files
  - Test log and coverage report is created for each executed test case

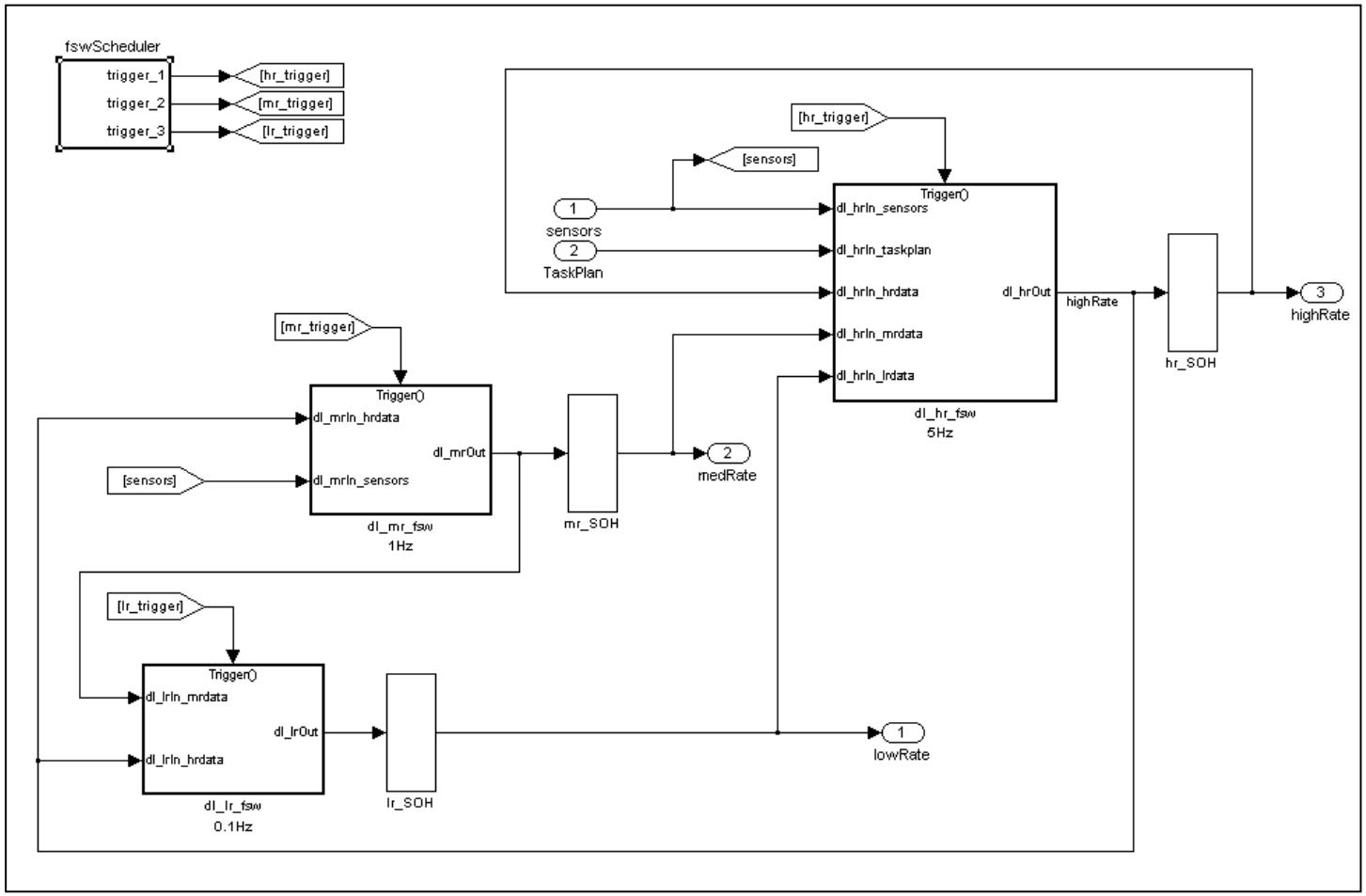
```
function SIM_P = gnt_propagateBy_UT_01(ExecutFlag)
%% Set Unit Name and Test ID
funcName = mfilename;

%% Set Nominal Iloads

SIM_P.test.input.PRM = gnt_targetingParametersConstr;
SIM_P.test.input.CON = gnt_targetingConstantsConstr;

%%%[START_DIARY]%%%

%% Change Iloads

%% Create Test Inputs

% Typical Craft State
SIM_P.test.input.craft.t.week = 2;
SIM_P.test.input.craft.t.sec = 43200;
SIM_P.test.input.craft.r = [5449738.3059999999;-3560213.6680000001;-1950161.4179999998];
SIM_P.test.input.craft.v = [1312.71;5061.3720000000003;-5601.2690000000002];

% Propagate forwards in Time
SIM_P.test.input.dt = 5000;

% No Drag
SIM_P.test.input.dragAcc = 0;

%% Define Error Tolerance

% Zero Tolerance
SIM_P.test.maxError = 1e-5;

%% Define Expected Output

% Expected Output Verified by .m Implementation
SIM_P.test.expectedOutput = [2;48200;3616520.7275637928;-5565050.9868103256;1436955.4908495285;4763.2042

% Expected Assert Triggered
SIM_P.test.assertNum = 0;

% Declare Ouput
SIM_P.test.output = [0;0;0;0;0;0;0;0];

%%%[END_DIARY]%%%
%% Test Execution, Test Results, and Data Storage
if nargin > 0
    ExecuteUnitTest(funcName, SIM_P, ExecutFlag);
end
```

Monte-Carlo Simulation Framework

- Overview
  - Modified Draper Simulink® simulation environment (ARAD sim aka GIDE) integrated with Monte-Carlo framework for both development and algorithm validation
  - Simulink® models for Target, Chaser, Planet, Ground. Data logging has been augmented with Monte-Carlo framework.
  - Model Reference blocks are used for multiple model instantiation, 3x Lidars, 3x GPS receivers, 3x IMUs. Also used for model version control, parallel development etc.
  - Flight algorithms are integrated into the simulation framework either as Model Reference blocks during development or as S-functions during algorithm validation
  - Flight algorithms are executed in their respective rate groups: 0.1 Hz, 1 Hz and 5 Hz.



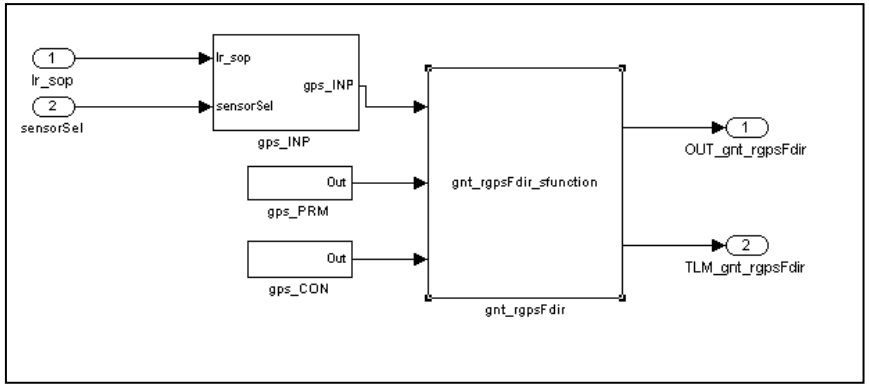
Monte-Carlo Simulation Framework



- Scheduler provides triggers for each of the 3 triggered subsystems: 0.1 Hz, 1 Hz and 5 Hz
- Scheduler triggers include time offsets to model dispatching of rate groups by a rate monotonic scheduler
- Sample and hold blocks are used to model rate group delays inherent in a rate monotonic scheduler

Monte-Carlo Simulation Framework

- Flight Algorithm Integration
  - Each CSC is integrated as a S-function for final algorithm validation
  - S-function wrapper calls the CSC call method previously described
  - Ensures actual CSC flight code is used when performing algorithm validation




```
/* Function: mdlOutputs ======================================================
 *
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    INP_gnt_rgpsFdir *INP = (INP_gnt_rgpsFdir *) ssGetInputPortSignal(S,0);
    PRM_gnt_rgpsFdir *PRM = (PRM_gnt_rgpsFdir *) ssGetInputPortSignal(S,1);
    CON_gnt_rgpsFdir *CON = (CON_gnt_rgpsFdir *) ssGetInputPortSignal(S,2);
    OUT_gnt_rgpsFdir *OUT = (OUT_gnt_rgpsFdir *) ssGetOutputPortSignal(S,0);
    TLM_gnt_rgpsFdir *TLM = (TLM_gnt_rgpsFdir *) ssGetOutputPortSignal(S,1);

    gnt_rgpsFdir_Outputs_wrapper(INP,PRM,CON,OUT,TLM);

}
```

```
// update function
void gnt_rgpsFdir_Outputs_wrapper(INP_gnt_rgpsFdir *INP,
                                  PRM_gnt_rgpsFdir *PRM,
                                  CON_gnt_rgpsFdir *CON,
                                  OUT_gnt_rgpsFdir *OUT,
                                  TLM_gnt_rgpsFdir *TLM)
{

    intfc.inp = *INP;
    intfc.prm = *PRM;
    intfc.con = *CON;

  GNT_RGPS_FDIR_Call(&intfc);

    (*OUT) = intfc.out;
    (*TLM) = intfc.tlm;

    //mexPrintf("OUT->rpopGuidStatus = %d\n", OUT->rpopGuidStatus);
    //mexPrintf("TLM->rpopGuidStatus = %d\n", TLM->rpopGuidStatus);

} // END FUNCTION gnt_rgpsFdir_Outputs_wrapper ()
```

Metrics

CSC	CSC Requirements	Total CSC Test Cases	Total CSUs	Total CSU Requirements	Total CSU Test Cases	Average Model McCabe Complexity	Average Code McCabe Complexity	Model Coverage	Code Coverage	Total SLOC: Comments	Total SLOC: Executable	CSC Timing Performance (ms)	CSC Fixed RAM Usage (Bytes)
gnt_targeting	17	15	23	85	186	17	22	100%	92%	9678	8719	209.94	694848
gnt_proxGuid	14	18	20	93	137	18	20	100%	92%	6571	6134	0.68	102432
gnt_imuFdir	22	8	7	32	31	9	14	100%	91%	1827	1144	1.44	92384
gnt_lidarFdir	11	13	8	37	45	11	18	100%	91%	1997	1126	0.94	65664
gnt_rgpsFdir	25	29	15	75	156	17	36	100%	93%	5531	3782	9.10	380816
gnt_lidarFilt	20	17	15	61	90	7	40	100%	92%	5777	5004	5.39	189008
gnt_rgpsFilt	28	9	15	70	118	13	50	100%	95%	5980	6031	47.25	317536
gnt_targetProp	3	6	2	6	7	6	10	100%	94%	982	538	0.34	40096
										Total	**Total**		
Notes: Two SRS requirements allocated to Error Handling										**38343**	**32478**		
Total of 41 algorithm requirements captured in the ASDD													

- Model Metrics
  - Complexity metrics were used to decompose CSCs into CSUs
  - Each CSC and CSU needed to have 100% model coverage for their test cases
- Code Metrics
  - Complexity and code coverage was analyzed for the auto-code
  - Several reasons for code complexity being greater than the model complexity. Draper guideline was to have the model complexity < 20 with less regard for the resulting auto-code complexity.
  - Structural differences between model and auto-code accounted for code coverage < model coverage. For example, utility functions were unit tested as models but these were in-lined in the auto-code.
- Peer Review Metrics
  - Number of defects, defect type, defect severity, defect density, defect rate

Lessons Learned: Overall Process

- Mandate Team-wide use of the same development platform and environment (desktop PC, Microsoft Visual C++, R2008b etc)
- Baseline Matlab/Simulink version and minimize dependencies on additional toolboxes
  - Avoid frequent version upgrades and new toolboxes.
- Develop and use customized libraries of blocks and models
  - Pre-built blocks and models ensure adherence to standards and efficient auto-generated code.
- Modeling standards must be clearly documented and maintained
  - Basic training and familiarization is needed for developers. Major part of peer review process.
- CSUs and CSCs should be defined as Model Reference blocks
  - Allows CSUs/CSCs to be developed, tested, maintained and configuration controlled as their own .mdl file
  - Each CSU/CSC should have a designated owner. Minimizes issues related to graphical merging and differencing.
- Data dictionary concept was essential to defining and controlling model interfaces
  - Model data managed and maintained in Microsoft Excel.
  - Model I/O automatically generated based on data dictionary.

Lessons Learned: Overall Process

- Software engineers are an essential part of the team
  - Auto-generating code is not just pushing a button!
  - Real-Time Workshop skills, process skills, standards development for efficient auto-generated code etc
  - Code verification is still necessary
- Typical software documentation requirements need to be reviewed
  - Draper IQMS required DIDs for MIL-STD-498 to be used.
  - Tailor DIDs for model-based development and use model content to the maximum extent to auto-generate design/test documentation and requirement/design/test bi-directional traceability.
- Model analysis and code analysis can yield different results
  - Model complexity versus code complexity
  - Code path coverage versus model path coverage
- Effort estimation
  - Need for non-SLOC based effort estimation methods
  - Techniques based on measures of model complexity are required.
  - COTS project is a good basis for the next Draper MBD project.