JSC-61777-models/dynamics/body_action

JSC Engineering Orbital Dynamics Body Action Model

Simulation and Graphics Branch (ER7)
Software, Robotics, and Simulation Division
Engineering Directorate

Package Release JEOD v5.1

Document Revision 1.3
July 2023

National Aeronautics and Space Administration
Lyndon B. Johnson Space Center
Houston, Texas

JSC-61777-models/dynamics/body_action

JSC Engineering Orbital Dynamics Body Action Model

Document Revision 1.3
July 2023

David Hammen

Simulation and Graphics Branch (ER7)
Software, Robotics, and Simulation Division
Engineering Directorate

National Aeronautics and Space Administration
Lyndon B. Johnson Space Center
Houston, Texas

Executive Summary

The Body Action Model forms a component of the dynamics suite of models within JEOD v5.1. It is located at models/dynamics/body_action.

The model comprises several classes that provide various mechanisms for setting aspects of instances of MassBody and DynBody classes as well as their derived classes. The capabilities provided by the model include:

- Setting the mass properties of a MassBody object,
- Attaching and detaching DynBody and MassBody objects to/from one another,
- Setting the state of a DynBody object, and
- Changing the frame in which a DynBody object's state is propagated.

Note that while the first function refers to a MassBody, the latter two capabilities pertain to DynBody. Meanwhile, attachment can involve interations between a DynBody and MassBody object. The subject can be specified by MassBody or DynBody, and the correct behavior implied (if possible).

Interations With Other Models

The Body Action Model interacts with several JEOD models. Chief among these are interactions with the *Mass Body Model* [14], which defines the MassBody class; the *Dynamic Body Model* [3], which defines the DynBody class; and the *Dynamics Manager Model* [4], which defines the DynManager class.

A JEOD simulation typically contains one or more instances of a class that derives from the DynBody class. Each DynBody object represents some vehicle or module in space. The DynBody-type vehicle may also have other DynBody or MassBody submodules attached. The purpose of this model is to alter some aspect of the subject body. Some of the model classes operate directly on MassBody objects via the use of public interfaces. Some operate directly on DynBody's using public interfaces. Others, such as attach actions, operate using either MassBody or DynBody methods depending on the type of subject.

In addition to the DynBody objects that represent vehicles, a JEOD simulation also contains a DynManager object that manages the dynamic objects in a simulation. The DynManager object uses the Body Action Model via a Standard Template Library list of pointers to queue Body Action Model objects. Simulation users can add elements to this list. As the list comprises pointers to objects, the pointed-to objects can be instances of different classes so long as they derive from the same base class. (All Body Action Model objects derive from a common base class.) The simulation DynManager object uses this list during simulation initialization time to initialize the simulation's MassBody/DynBody objects and uses this list during simulation run time to perform asynchronous actions.

In addition to the above three models, the Body Action Model interacts with several other JEOD models. A driving concern for the Body Action Model is to do as little as possible in the model. The model source files instead leverage capabilities that exist in other models. In addition to the models listed above, this model uses:

- The *Derived State Model* [16] to construct Local Vertical, Local Horizontal frames,
- The *Mathematical Functions Model* [15] to determine planetary coefficients,
- The *Planet Fixed Model* [6] to construct North-East-Down frames,
- The *Reference Frame Model* [13] to compute relative state.

Body Action Model Base Class

All of the instantiable Body Action Model classes ultimately derive from the abstract BodyAction base class. This abstract base class by itself does not change a single aspect of a MassBody or DynBody object. That is the responsibility of the various specialized classes that derive from the base BodyAction class. The base class provides a common virtual framework for describing actions to be performed on a body-type object. This framework includes:

- Descent from a common base class. This common basis enables forming a collection of body actions in the form of base class pointers. The DynManager class does exactly this. The Body Action Model was explicitly architected to enable this treatment.
- Base class member data. This member data includes a pointer to the subject of the Body-Action object in the form of either of DynBody subject or a MassBody subject. Populating both subject pointers is unnecessary, as the DynBody corresponding to a MassBody is implied (and vice versa).
- Base class member functions. The member functions specify how to operate on the BodyAction object. In particular, a trio of virtual methods described below form the standard basis by which derived classes can extend this base class.

The base class defines three virtual methods that together form the mechanism by which an external user of the Body Action Model functionally interacts with BodyAction objects:

initialize Each Body Action Model object must be initialized. This initialization is the task of the overridable `initialize()` method. This method initializes a BodyAction object, and should verify that the BodyAction is properly configured. It does not alter the subject object.

is_ready The `is_ready()` method indicates whether the BodyAction object is ready to perform its action on the subject. For example, a DynBodyInitLvlhState object sets parts of the subject DynBody object's state based on the Local Vertical, Local Horizontal (LVLH) frame defined by some reference DynBody object. The DynBodyInitLvlhState object remains in a not-ready state until the reference body's position and velocity vector have been set.

apply The `apply()` method performs the action on the subject. After the action is performed, it is removed from the DynManager's queue.

Body Action Model Derived Classes

The Body Action Model provides several classes that derive from the base BodyAction class. These are

MassBodyInit Derives from BodyAction.
Instances of this class set the subject MassBody object's mass properties and specify the object's mass points. In the case where the subject MassBody belongs to a DynBody, specifying the mass points also specifies the DynBody object's vehicle points.

BodyAttach Derives from BodyAction.
Classes derived from this class attach one body object to another. This abstract class does not accomplish this objective; it is a non-instantiable class. Attaching bodies is a task to be performed by derived classes specifying the attach method.

BodyAttachAligned Derives from BodyAttach.
Instances of this class attach the subject body to a parent body. The attachment is specified in terms of a pair of points, one on the subject body and the other on the parent body. The attachment makes the two points coincident and makes the transformation between the two points' reference frames be a 180 degree yaw.

BodyDetach and BodyDetachSpecific Derive from BodyAction.
Instances of these classes detach a body object from its immediate parent. The class BodyDetach detaches the subject from its immediate parent. The class BodyDetachSpecific searches the attachment tree for the specified other body (starting from the subject body), then detaches the linkage at the parent body. That is, if the subject is the progeny, the detach takes place at the other body; if the subject is the progenitor, the detach takes place at the subject. The found object is detached from the specified parent.

DynBodyInit Derives from BodyAction.
The purpose of this class is to initialize the state (or parts of the state) of a DynBody object. This specific class does not accomplish this goal; this is a non-instantiable class. The classes that derive from DynBodyInit provide multiple means of initializing aspects of a DynBody object's state.

DynBodyInitRotState Derives from DynBodyInit.
Instances of this class initialize a vehicle's attitude and/or attitude rate.

DynBodyInitTransState Derives from DynBodyInit.
Instances of this class initialize a vehicle's position and/or velocity.

DynBodyInitOrbit Derives from DynBodyInitTransState.
Instances of this class initialize a vehicle's position and velocity by means of some orbit specification.

DynBodyInitPlanetDerived Derives from DynBodyInitWrtPlanet, which in turn derives from DynBodyInit.
These classes form the basis for subsequent derived classes that initialize state with respect to a planetary frame.

DynBodyInitLvlhState Derives from DynBodyInitPlanetDerived.
Instances of this class initialize state with respect to the rectilinear LVLH frame defined by some reference vehicle's orbit about a specified planet. Two derived classes from this class, DynBodyInitLvlhRotState and DynBodyInitLvlhTransState, limit the initialization to the rotational and translational states of the subject DynBody.

DynBodyInitNedState Derives from DynBodyInitPlanetDerived.
Instances of this class initialize state with respect to the North-East-Down frame defined by some reference vehicle or reference point. Two derived classes from this class, DynBodyInitNedRotState and DynBodyInitNedTransState, limit the initialization to the rotational and translational states of the subject DynBody.

Integrating the Body Action Model in a Simulation

The following approaches present an obvious, but wrong, approach to using the Body Action Model in a Trick simulation:

- Defining data elements of the appropriate model classes for each vehicle simulation object in the simulation S_define file and
- Explicitly calling the `initialize()` and `apply()` methods for those model objects in the simulation S_define file.

Do not follow the above practice. This practice circumvents the flexibility built into the model and will almost certainly lead to difficulties in the future. It is best add the body action to the DynManager's queue, and allow DynManager to apply.

The BodyAction List

The recommended practice regarding the use of the Body Action Model in a Trick simulation is to use the model in conjunction with the simulation's DynManager object. The DynManager maintains a list of Body Action Model objects. Calling the DynManager's `add_body_action` adds an item to the list. The DynManager draws objects from this list at the appropriate time and only does so when the objects indicate that they are ready to be executed. This mechanism disconnects the execution order of the enqueued model objects from the order in which the simulation objects are declared and from the order in which the objects are added to the list.

This separation between initialization and declaration order was first addressed in JEOD 1.4/1.5, but the C-based solution involved some rather convoluted and very inflexible code. The C++ based solution introduced in JEOD 2.0 makes for a very extensible and simpler set of code. Users are strongly recommended to take advantage of this flexibility.

A Simple Simulation

A simple way to make use of the DynManager's BodyAction list limit the use of the Body Action Model to a pre-defined, but presumably well-vetted, set. A simulation S_define file that implements this option will:

- Declare a very specific set of model objects as simulation object elements for each vehicle in the simulation,
- Specifically invoke the DynManager `add_body_action()` method on each of these model objects as initialization class jobs,
- Invoke the DynManager `initialize_simulation()` method as an initialize class job that runs after the `add_body_action()` jobs, and
- Optionally invoke the DynManager `perform_actions()` as a scheduled job to enable asynchronous use of the model during the course of the simulation run.

This approach makes user input fairly simple. All the simulation user needs to do is populate the pre-defined set of model objects with the appropriate values.

The lack of flexibility is the key disadvantage of this option. To illustrate this, consider a mission involving a chaser and a target vehicle. Different aspects of the mission that need to be investigated include the following scenarios:

- The launch of the chaser vehicle. The chaser starts at rest with respect to the launch pad. The target vehicle does not even need to be involved in this scenario. Specifying the chaser's initial state in some inertial frame would be awkward at best. The latitude, longitude, and altitude of the launch pad are known quantities, as is the geometry of the chaser with respect to the pad. The most transparent option would be to specify the chaser initial state in terms of these known quantities.
- The transfer of the chaser from orbit insertion to far-field rendezvous. In this situation, the chaser needs to start in an orbit that is related to the target vehicle's orbit. The large initial separation between the two vehicles rules out a relative cartesian specification for the chaser.
- Various proximity operations between the chaser and target. Initializing the chaser's state relative to that of the target is exactly what is called for in this scenario.
- Chaser de-orbit and entry. As with the launch scenario, the target vehicle can be omitted from this scenario. The chaser vehicle needs to be specified in an orbit relative to the rotating Earth so that it will enter the atmosphere at the correct location with respect to the rotating Earth.

Multiple Initialization Capabilities

To handle phase transitions in multi-phase simulations (e.g., orbit to re-entry), patched approaches are recommended. In this methodology, synchronous BodyAction's re-initialize states to "tweak" as appropriate This imperfect approach allows the developer to implement multiple vehicle configurations as needed. The best way to initialize the chaser vehicle's state in these different scenarios is to use the Body Action Model state initialization classes best suited for the scenario. For example:

- The DynBodyInitNed class to initialization the chaser on the launch pad,
- The DynBodyInitTransState or DynBodyInitOrbit classes to initialize the chaser at orbit insertion and prior to entry, and
- The DynBodyInitLvlh class to initialize the chaser in close proximity to the target.

Each of these different initialization capabilities needs to be allocated in the S_define file, trick/C++ source, or python input layer to enable their use in the simulation (NOTE: as of Trick 19.2.3 the DynManager queue for BodyActions is checkpointable/restartable with some exceptions). One common way to accomplish this end is to declare as simulation object elements all of the state initializers needed by the various scenarios.

This option continues that used in the previous example. The simulation S_define file specifically invokes the DynManager `add_body_action()` method for each initializer. This solves the issue of different scenarios requiring different initializers, but introduces a new issue. The set of state initializers now over-specifies the state. The input file for a particular scenario will need to deactivate those state initializers that are not to be used in that scenario and/or use contextual triggers to activate them at the correct time.

Deferring `add_body_action()` Calls

An alternative approach to solving this over-specification problem is to never let it happen. The previous option required registering in the S_define file a call to `add_body_action()` for each state initializers declared in the S_define file. This option only registers one such call in the S_define file. A simulation S_define file that implements this option will:

- Declare a potentially broad set of Body Action Model objects as simulation object elements for each vehicle in the simulation.
- Declare a pointer to a BodyAction object in the same sim object that contains the DynManager object.
- Define *as a zero-rate scheduled job* a call to the DynManager's `add_body_action()` method that adds the contents of this pointer to the DynManager's BodyAction list.

The task of calling `add_body_action()` to place an initializer in the DynManager's body action list is deferred to the input file. An input file that follows this paradigm will perform the following:

- Set the contents of a particular initializer,

- Point the BodyAction pointer declared in the S_define file point to this initializer,
- Issue a `call` to the zero-rate `add_body_action()` job to add this initializer, and
- Repeat the above for each initializer to be used in the simulation run.

Dynamic allocation of initializers

The S_define file may become bloated with multiple initializers of the same type in a simulation that involves several vehicles. Along with deferring the calls to `add_body_action()`, this final option defers the allocation of the initializers to the input file via the Trick input processor's `new` capability.

The key advantage of this final approach is that it maximizes flexibility. This is also a potential disadvantage; enhanced flexibility invites errors on the part of the user. Another disadvantage is that the allocated objects cannot (currently) be logged.

Using the Body Action Model in a Simulation

How the Body Action Model is to be used depends on which of the above options was employed by the simulation integrator. This discussion assumes a setup along the lines of the penultimate option described above ("Deferring `add_body_action()` calls"). The simulation user selects those capabilities from the S_define file that best represent the needs of the scenario to be simulated.

For example, consider the case of investigating the behavior of a chaser vehicle in close proximity to a target vehicle. The S_define file provides multiple mechanisms for initializing the chaser and target. The simulation user selects a set of initializers that enables specifying:

- The target vehicle's mass properties and points of interest,
- The target vehicle's translational state with some given inertial values,
- The target vehicle's rotational state relative to the target vehicle's LVLH frame,
- The chaser vehicle's mass properties and points of interest, and
- The chaser vehicle's state in terms of the relative state between a pair of points of interest on the chaser and target.

For each selected item, the user must place statements in the simulation input that:

- Populate the data items for the object.
- Point the simulation body action pointer to this model object.
- Call the `add_body_action()` simulation job to add the object to the Dynamic Manager's list of model objects.

Four mistakes that the user can make in this process are:

- Failing to specify the subject body for each model object. The JEOD methods do not know that the simulation integrator has made the subject body and the model object parts of the same simulation object.
- Attempting to reuse model objects. The `add_body_action()` method must not (and cannot) create a copy of the object passed to it.
- Creating a circular set of dependencies. For example, specifying the state of vehicle A with respect to vehicle B and specifying the state of vehicle B with respect to vehicle A.
- Failing to specify all needed states.

JEOD detects and reports these errors with messages.

Requirements on Derived Classes

The Body Action Model places several functional constraints on the classes that derive from the BodyAction base class. These constraints apply to the classes that are a part of the model and to any extensions of the class written by model extenders. These constraints are on the trio of virtual methods as overridden by the derived class:

1. The `initialize()` method, if overridden, must forward the `initialize()` call to the immediate parent object. The typical approach is to check for error conditions, forward the `initialize()` call upward, and finally perform class-specific initializations. This removes the need to write redundant routines in derived classes.
2. The `is_ready()` method, if overridden, must query the readiness of the parent class in addition to performing class-specific readiness checks. Claiming to be ready against the advice of the parent is forbidden.
3. The `apply()` method, if overridden, must forward the `apply()` call to the immediate parent object. The typical approach is to reformulate the user inputs in terms that will be understood by the parent class, forward the `apply()` call upward, and finally perform any cleanup actions.

Requirements on Functional Users of the Body Action Model

The Body Action Model places several functional constraints on the use of the model. These constraints apply to the Dynamics Manager Model and to any use of the model written by a user of JEOD. These constraints are

1. A BodyAction object must be initialized. The object's `initialize()` method must be invoked prior to invoking either the `is_ready()` or `apply()` methods for that object.
2. A BodyAction object must be initialized once only. Invoking a single object's `initialize()` method multiple times may result in undetected errors such as memory leaks.

3. A BodyAction object must be initialized at the appropriate time.
    (a) BodyAttach actions must be initialized after the subject MassBody objects have had their initial mass properties set and have had their mass points defined.
    (b) DynBodyInit actions furthermore must be initialized after the initialization-time attachments have been performed.
    (c) All other actions must be initialized after the states of the simulation's DynBody objects have been set.
4. A BodyAction object's `apply()` method must not be invoked before the object indicates that it is ready to be applied. In other words, the object's `is_ready()` method must return `true` before calling the object's `apply()` method.
5. A BodyAction object's `apply()` method must not be invoked multiple times. The Body Action Model follows the fire-and-forget paradigm.

Contents

Part I

Overview

Chapter 1

Introduction

1.1 Purpose and Objectives of the Body Action Model

The Body Action Model comprises several classes that provide various mechanisms for setting aspects of instances of the MassBody and DynBody class and their derived classes. The provided capabilities include:

- Setting the mass properties of a MassBody object,
- Attaching and detaching MassBody or DynBody objects to/from one another or a RefFrame,
- Setting the state of a DynBody object, and
- Changing the frame in which a DynBody object's state is propagated.

1.2 Context within JEOD

The following document is parent to this document:

- *JSC Engineering Orbital Dynamics* [7]

The Body Action Model forms a component of the dynamics suite of models within JEOD v5.1. It is located at models/dynamics/body_action.

1.3 Documentation History

Author	**Date**	**Revision**	**Description**
David Hammen	November, 2009	1.0	Initial Version
David Hammen	April, 2010	1.1	Tests & Traceability
Jeff Morris	April, 2016	1.2	Updated For Curviliner LVLH
Mitch Hollander	September, 2021	1.3	Updated for JEOD 4.0
Thomas Brain	February, 2023	1.4	Updated for BodyAction changes for JEOD 5.0

1.4 Documentation Organization

This document is formatted in accordance with the NASA Software Engineering Requirements Standard [9].

The document comprises several parts:

Overview This first part of the document provides an overview of the Body Action Model.

BodyAction Part II describes the class BodyAction, the base class for the Body Action Model. Each subsequent part describes a collection of related classes that derive from this base class.

MassBodyInit Part III describes the class MassBodyInit, which initializes the mass properties and mass points of a MassBody object.

Attach/Detach Part IV describes the classes that cause MassBody and DynBody objects to attach to and detach from one another or a RefFrame.

DynBodyInit Part V describes the class DynBodyInit and its derived classes, which initializes the state of a DynBody object.

FrameSwitch Part VI describes the class DynBodyFrameSwitch, which switches the frame in which a DynBody object's state is propagated.

Each part is organized into a similar structure, comprising the following chapters in order:

Introduction - This introduction describes the objective and purpose of the Body Action Model. The introductions for subsequent parts describe the objective and purpose of the classes that are the subject of that part. To avoid undo repetition, the introductory chapters of the subsequent parts are of an abbreviated nature.

Product Requirements - The requirements chapter in this overview part of the document describes the requirements on the Body Action Model as a whole. The requirements chapters of subsequent parts describe the requirements that pertain to the classes that are the subject of that specific part.

Product Specification - The specification chapter in this overview part of the document describes the architecture and design of the Body Action Model as whole. Where applicable, The specification chapter in the subsequent parts describes the underlying theory, architecture, and design of the subject classes of the part in detail. Where applicable, the product specification chapters are organized into sections as follows:

- Conceptual Design.
- Mathematical Formulations (where applicable).
- Detailed Design.

- Version Inventory (Part I - Overview only).

User Guide - Describes how to use the Body Action Model as a whole (this part) / sub-models (subsequent parts). Where applicable, the User Guide chapters are organized into sections that represent the following JEOD-defined user types:

- Analysts or users of simulations (Analysis).
- Integrators or developers of simulations (Integration).
- Model Extenders (Extension).

Inspection, Verification, and Validation - The inspection, verification, and validation (IV&V) chapter in this overview part of the document shows the traceability of the high-level requirements to the sub-model detailed requirements. The IV&V chapters in the subsequent parts describes the verification and validation procedures and results for the subject classes of the part.

Chapter 2

Product Requirements

Requirement BodyAction_1: Top-level requirement

Requirement:
This model shall meet the JEOD project-wide requirements specified in the JEOD v5.1 top-level document requirements[7].

Rationale:
This is a project-wide requirement.

Verification:
Inspection

Requirement BodyAction_2: Body Action Base Class

Requirement:
All action classes that modify aspects of a MassBody, DynBody or derived class instance shall derive from a common base class.

Rationale:
A common base class enables external functional users of the model to refer to any of the model object in terms of a base class pointer.

Remarks:
This requirement does not pertain to the Body Action Model message class, BodyAction-Messages. That class does not modify aspects of a subject body. It instead defines error and message IDs used for interaction with the JEOD Message Handler.

Verification:
Inspection

Requirement BodyAction_3: Subject Specification

Requirement:
Each instance of the classes defined by this model shall designate a MassBody or DynBody object as the primary subject of the model instance.

Rationale:
The goal of this model is to alter various aspects of an object, therefore the object must be known to the action instance.

Verification:
Inspection

Requirement BodyAction_4: Initialize Mass Properties

Requirement:
This model shall provide the ability to initialize the mass properties of a MassBody object.

Rationale:
A driving goal of this model is to provide a common interface for initializing key aspects of MassBody and DynBody objects.

Verification:
Inspection

Requirement BodyAction_5: Initialize Mass Points

Requirement:
This model shall provide the ability to specify the points of interest on MassBody and DynBody objects.

Rationale:
A driving goal of this model is to provide a common interface for initializing key aspects of MassBody and DynBody objects.

Verification:
Inspection, Test

Requirement BodyAction_6: Attach MassBody and DynBody Objects

Requirement:
This model shall provide the ability to attach one MassBody object to another, one DynBody to another, a MassBody as a child attachment to a DynBody, or a DynBody to a RefFrame. The model shall not provide the ability to attach a DynBody as a child attachment to a MassBody.

Rationale:

A driving goal of this model is to provide a common interface for initializing key aspects of and for performing asynchronous operations on MassBody and DynBody objects. The DynBody model specifies that it may not attach as a child to a MassBody, which is stateless.

Verification:

Inspection, Test

Requirement BodyAction_7: Detach MassBody and DynBody Objects

Requirement:

This model shall provide the ability to detach one MassBody object from another, detach one DynBody from another, detach a MassBody from a DynBody, and detach a DynBody from a RefFrame.

Rationale:

A driving goal of this model is to provide a common interface for performing asynchronous operations on MassBody and DynBody objects.

Verification:

Inspection, Test

Requirement BodyAction_8: Initialize DynBody State

Requirement:

This model shall provide the ability to initialize the dynamic state (position, velocity, attitude, and rate) of a DynBody object.

Rationale:

A driving goal of this model is to provide a common interface for initializing key aspects of MassBody and DynBody objects.

Verification:

Inspection, Test

Requirement BodyAction_9: Switch DynBody Integration Frame

Requirement:

This model shall provide the ability to switch the frame in which a DynBody object's state is represented and propagated.

Rationale:

A driving goal of this model is to provide a common interface for performing asynchronous operations on MassBody and DynBody objects.

Verification:

Inspection, Test

Requirement BodyAction_10: Error Detection and Handling

Requirement:
The Body Action Model shall detect and handle erroneous conditions.

Rationale:
All models, and particularly those that deal with user inputs, must address erroneous conditions.

Verification:
Inspection, Test

Requirement BodyAction_11: Trace Capability

Requirement:
The Body Action Model shall provide a (disable-able) ability to summarize actions performed on the subject objects.

Rationale:
This capability is needed for error handling and message generation.

Verification:
Inspection, Test

Chapter 3

Product Specification

3.1 Conceptual Design

3.1.1 Overview

The Body Action Model comprises several classes that provide various mechanisms for setting aspects of instances of the MassBody class, DynBody class, and their derived classes. The capabilities provided by the model include:

- Setting the mass properties of a MassBody object,
- Attaching and detaching MassBody and DynBody objects to/from one another or a RefFrame,
- Setting the state of a DynBody object, and
- Changing the frame in which a DynBody object's state is propagated.

Note that while the first function refers to a MassBody, the latter two capabilities pertain to DynBody. Meanwhile, attachment can involve interactions between a DynBody and MassBody object. The subject can be specified by MassBody or DynBody, and the correct behavior implied (if possible).

The first two capabilities pertain to body objects in general. The model does not specifically provide the ability to set the mass properties of a DynBody object. Because a DynBody contains a MassBody and because the mass properties are orthogonal to the additional capabilities provided by the DynBody class, the same capability used to set the mass properties of a MassBody object functions correct when applied to a DynBody object.

Attaching and detaching DynBody objects to/from one another involves considerably more computational effort than does attaching/detaching simple MassBody objects. The DynBody class adds state information to the MassBody class. How attachment and detachment affect state is not a part of the basic MassBody attachment and detachment calculations due to conservation of momentum calculations. Thanks to the construction of the attachment and detachment methods, there is no need for a special-purpose DynBody attachment or detachment body action. The basic method works correctly when applied to a DynBody object.

3.1.2 Base Classes

All of the instantiable Body Action Model classes ultimately derive from the BodyAction base class. This base class provides a common framework for describing actions to be performed on a MassBody (or derived class) object. The common basis for all Body Action Model objects enables forming a collection of body actions in the form of base class pointers. The DynManager class does exactly this. The Body Action Model was explicitly architected to enable this treatment.

The Body Action Model defines one other base class, the BodyActionMessages class. This static class is not instantiable and has no derived classes. It exists to conform with the JEOD scheme for generating messages.

See part II for a complete description of these base classes.

3.1.3 Derived Classes

The BodyAction base class by itself does not change a single aspect of a subject MassBody or DynBody type object. Making changes to a subject is the responsibility of the various classes that derive from the BodyAction base class. The derived classes include

- MassBodyInit.
  MassBody objects represent masses with structural and body reference frames. This class initializes a subject MassBody object's mass properties, specifies the object's structure-to-body transformation, and any defines user-defined points of interest ("mass points").

  See part III for a complete description of the MassBodyInit class.

- BodyAttach and BodyDetach.
  Connected vehicle objects form a mass tree topology. Classes that derive from BodyAttach attach a subject Body object to some other Body object or RefFrame. JEOD defines the BodyAttachAligned class to connect two bodies or a body and reference frame by aligning specified vehicle points or reference frames at coincident position with a 180 degree yaw relative orientataion. The BodyAttachMatrix class defines attachment by defining the relative position of a child body structure with respect to a parent body structure point or reference frame. The classes BodyDetach and BodyDetachSpecific undo these attachments.

  See part IV for a complete description of the attach/detach sub-model.

- DynBodyInit.
  The DynBody class includes several attributes absent from the MassBody class that are needed to model a vehicle in space. This includes the object's state—its position, velocity, attitude, and angular velocity with respect to some inertial (non-rotating) reference frame. Classes that derive from DynBodyInit initialize part or all of a subject DynBody object's state. JEOD defines several classes that derive from the DynBodyInit class.

  See part V for a complete description of the DynBodyInit sub-model.

- DynBodyFrameSwitch.
  The ability to switch the frame in which a DynBody object's state is represented and propagated was a driving requirement for JEOD. The DynBodyFrameSwitch accomplishes that goal.

See part VI for a complete description of the DynBodyFrameSwitch class.

3.1.4 Interactions With Other Models

The Body Action Model interacts with several JEOD models. Chief among these are interactions with the *Mass Body Model* [14], which defines the MassBody class; the *Dynamic Body Model* [3], which defines the DynBody class; and the *Dynamics Manager Model* [4], which defines the DynManager class.

A JEOD simulation typically contains one or more instances of a class that derives from the DynBody class. Each DynBody object represents some vehicle in space. The DynBody class contains an instance of the MassBody class, and it can have other MassBody objects or DynBody object attached to it.

The purpose of this model is to alter some aspect of an instance of the MassBody, DynBody, or derived class. Some of the model classes operate directly on MassBody objects, some on DynBody objects. In general, the user may specify either as the subject to the BodyAction, and the correct body type will be inferred. The Body Action Model classes that operate on MassBody and DynBody objects by utilizing their public interfaces methods. Other model classes specialize to altering some aspect of a DynBody object. These classes use the public interfaces of the DynBody class to achieve their desired end.

Along to the DynBody objects that represent vehicles, a JEOD simulation also contains a DynManager object that manages the dynamic objects in a simulation. The DynManager object uses the Body Action Model via a Standard Template Library list of pointers to Body Action Model objects. Simulation users can add elements to this list. As the list comprises pointers to objects, the pointed-to objects can be instances of different classes so long as they derive from the same base class. (All Body Action Model objects derive from a common base class.) The simulation DynManager object uses this list during simulation initialization time to initialize the simulation's MassBody/DynBody objects and uses this list during simulation run time to perform asynchronous actions.

In addition to the above three models, the Body Action Model interacts with several other JEOD models. A driving concern for the Body Action Model is to do as little as possible in the model. The model source files instead leverage capabilities that exist in other models. In addition to the models listed above, this model uses:

- The *Derived State Model* [16] to construct Local Vertical, Local Horizontal frames,
- The *Mathematical Functions Model* [15] for matrix and vector manipulations,
- The *Memory Management Model* [5] to allocate memory,
- The *Message Handler Model* [10] to report errors and generate messages,
- The *Named Item Model* [11] to construct names,
- The *Orbital Elements Model* [12] to compute orbits,
- The *Planet Model* [8] to determine planetary coefficients,

- The *Planet Fixed Model* [6] to construct North-East-Down frames, and
- The *Reference Frame Model* [13] to compute relative state.

3.2 Mathematical Formulations

N/A

3.3 Detailed Design

See the *Body Action Model API* [2] for a description of classes that comprise the model and for descriptions of the member data and member functions defined by these classes.

3.4 Inventory

All Body Action Model files are located in `${JEOD_HOME}/models/dynamics/body_action`. Relative to this directory,

- Header and source files are located in the model `include` and `src` subdirectories. Table **??** lists the configuration-managed files in these directories.
- Documentation files are located in the model `docs` subdirectory. See table **??** for a listing of the configuration-managed files in this directory.
- Verification files are located in the model `verif` subdirectory. See table **??** for a listing of the configuration-managed files in this directory.

Chapter 4

User Guide

This chapter provides a top-level description of the use of the Body Action Model. Details on how to use specific classes are described in the User Guides for the various sub-models.

4.1 Analysis

How the Body Action Model is to be used depends on which of the above options was employed by the simulation integrator. This discussion assumes a setup along the lines described in section 4.2.4. The simulation user selects those capabilities from in the S_define file that best represent the needs of the scenario to be simulated.

For example, consider the case of investigating the behavior of a chaser vehicle in close proximity to a target vehicle. The S_define file provides multiple mechanisms for initializing the chaser and target. The simulation user selects a set that enables specifying

- The target vehicle's mass properties and points of interest,
- The target vehicle's translational state with some given inertial values,
- The target vehicle's rotational state relative to the target vehicle's LVLH frame,
- The chaser vehicle's mass properties and points of interest, and
- The chaser vehicle such in terms of the relative state between a pair of points of interest on the chaser and target.

For each selected item, the user must place statements in the simulation input that

- Populate the data items for the object.
- Point the simulation body action pointer to this model object.

- Call the `add_body_action()` simulation job to add the object to the Dynamic Manager's list of model objects.

Four mistakes that the user can make in this process are

- Failing to specify the subject body for each model object. The JEOD methods do not know that the simulation integrator has made the subject body and the model object parts of the same simulation object.
- Attempting to reuse model objects. The `add_body_action()` method must not (and cannot) create a copy of the object passed to it.
- Creating a circular set of dependencies. For example, specifying the state of vehicle A with respect to vehicle B and specifying the state of vehicle B with respect to vehicle A.
- Failing to specify all needed states.

The JEOD detects and reports these errors.

4.2 Integration

The obvious, but wrong, approach to using the Body Action Model in a Trick simulation is to

- Define data elements of the appropriate model classes for each vehicle simulation object in the simulation S_define file and
- Explicitly call the `initialize()` and `apply()` methods for those model objects in the simulation S_define file.

Do not follow the above practice. This practice circumvents the flexibility built into the model and will almost certainly lead to difficulties in the future.

4.2.1 The BodyAction list

The recommended practice regarding the use of the Body Action Model in a Trick simulation is to use the model in conjunction with the simulation's DynManager object. The DynManager maintains a list of Body Action Model objects. Calling the DynManager's `add_body_action` adds an item to the list. The DynManager draws objects from this list at the appropriate time and only does so when the objects indicate that they are ready to be executed. This mechanism disconnects the execution order of the enqueued model objects from the order in which the simulation objects are declared and from the order in which the objects are added to the list.

This separation between initialization and declaration order was first addressed in JEOD 1.4/1.5, but the C-based solution involved some rather convoluted and very inflexible code. The C++ based JEOD 2.0 solution to this problem makes for an very extensible and simpler set of code. Users are strongly recommended to take advantage of this flexibility.

4.2.2 A simple simulation

A simple way to make use of the DynManager's BodyAction list limit the use of the Body Action Model to a pre-defined, but presumably well-vetted, set. A simulation S_define file that implements this option will

- Declare a very specific set of model objects as simulation object elements for each vehicle in the simulation.
- Specifically invoke the DynManager `add_body_action()` method on each of these model objects as initialization class jobs.
- Invoke the DynManager `initialize_simulation()` method as an initialize class job that runs after the `add_body_action()` jobs.
- Optionally invoke the DynManager `perform_actions()` as a scheduled job to enable asynchronous use of the model during the course of the simulation run.

This approach makes user input fairly simple. All the simulation user needs to do is populate the pre-defined set of model objects with the appropriate values.

The lack of flexibility is the key disadvantage of this option. To illustrate this, consider a simulation involving a chaser and a target. Different runs of the simulation are to investigate

- The launch the chaser vehicle,
- The transfer of the chaser from orbit insertion to far-field rendezvous,
- Various proximity operations between the chaser and target, and
- Chaser de-orbit and entry.

4.2.3 Multiple initialization capabilities

The best way to initialize the chaser vehicle's state in these different scenarios is to use the Body Action Model state initialization classes best suited for the scenario. For example,

- The DynBodyInitNed class to initialization the chaser on the launch pad,
- The DynBodyInitTransState or DynBodyInitOrbit classes to initialize the chaser at orbit insertion and prior to entry, and
- The DynBodyInitLvlh class to initialize the chaser in close proximity to the target.

Each of these different initialization capabilities needs to be declared in the S_define file to enable their use in the simulation. One way to accomplish this end is to declare as simulation object elements all of the state initializers needed by the various scenarios.

This option continues that used in the previous example. The simulation S_define file specifically invokes the DynManager `add_body_action()` method for each initializer. This solves the issue of different scenarios requiring different initializers, but introduces a new issue. The set of state initializers now over-specifies the state. The input file for a particular scenario will need to deactivate those state initializers that are not to be used in that scenario.

4.2.4 Deferring `add_body_action()` calls

An alternative approach to solving this over-specification problem is to never let it happen. The previous option required a registering in the S_define file a call to `add_body_action()` for each state initializers declared in the S_define file. This option only registers one such call in the S_define file. The S_define file

- Declares a potentially broad set of Body Action Model objects as simulation object elements for each vehicle in the simulation.
- Declares a pointer to a BodyAction object in the same sim object that contains the DynManager object.
- Defines *as a zero-rate scheduled job* a call to the DynManager's `add_body_action()` method that adds the contents of this pointer to the DynManager's BodyAction list.

The task of calling `add_body_action()` to place an initializer in the DynManager's body action list is deferred to the input file. The input file now looks like

- Setting the contents of a particular initializer,
- Pointing the BodyAction pointer declared in the S_define file point to this initializer,
- Issuing a `call` to the zero-rate `add_body_action()` job to add this initializer, and
- Repeating the above for each initializer to be used in the simulation run.

4.2.5 Dynamic allocation of BodyAction objects

The S_define file may become bloated with multiple initializers of the same type in a simulation that involves several vehicles. Along with deferring the calls to `add_body_action()`, this final option defers the allocation of the initializers to the input file via the Trick input processor.

The key advantage of this final approach is that it maximizes flexibility. This is also a potential disadvantage; enhanced flexibility invites errors on the part of the user. Another disadvantage is that the objects allocated in the input cannot be reliably checkpointed/restarted without due diligence on the part of the simulation developer. However, Body Action's generally do not require checkpointing/restarting due to their implementation generally being a static and/or single-use.

4.2.6 Functional Use of the Model

While the Body Action Model was designed primary for use at the S_define level, it most certainly can be used within some user-defined model. This section describes two approaches to using the model within some body of C++ code.

Before doing so, however, ask yourself why you want to do this. The primary reason for using the Body Action Model within a user-defined model is to take advantage of the fact that the application of body actions can be postponed until some later time. On the other hand, if some action is to be performed immediately, simply invoke the desired functionality directly. There is no need to use the Body Action Model for such direct and immediate actions.

Emulating S_define/Input Processing

One form of the functional use of this model will involve implementing, in code, the same operations performed in an S_define file and a simulation input file. The user code will

- Create a specific instance of a class that derives from BodyAction,
- Populate it with data,
- Add it to the Dynamic Manager's list of body actions by invoking the Dynamic Manager's `add_body_action()` method.

If the created object is to executed at initialization time, the job that performs the above must run prior to the DynManager `initialize_simulation()` job. If, on the other hand, the execution of the created object is to be deferred (e.g., attaching a pair of DynBody objects), the created action must initially be deactivated. Activating the object at the appropriate time will cause the DynManager `perform_actions()` job to perform the action.

Calling Model Methods

The Body Action Model places several functional constraints on the way in which the Dynamics Manager Model uses this model. These constraints apply to any externally-developed model that acts in lieu of the Dynamics Manager. These constraints are

1. A BodyAction object must be initialized. The object's `initialize()` method must be invoked prior to invoking either the `is_ready()` or `apply()` methods for that object.
2. A BodyAction object must be initialized once only. Invoking a single object's `initialize()` method multiple times may result in undetected errors (e.g., memory leaks).
3. A BodyAction object must be initialized at the appropriate time.
   (a) BodyAttach actions must be initialized after the `mass_subject` MassBody or `dyn_subject` DynBody object has had their initial mass properties set and have had their mass points defined. Same goes for the `mass_parent` MassBody or `dyn_parent` DynBody objects.
   (b) DynBodyInit actions furthermore must be initialized after the initialization-time attachments have been performed.
   (c) All other actions must be initialized after the states of the simulation's DynBody objects have been set.

4. A BodyAction object's `apply()` method must not be invoked before the object indicates that it is ready to be applied. In other words, the object's `is_ready()` method must return `true` before calling the object's `apply()` method.

5. A BodyAction object's `apply()` method must not be invoked multiple times. The Body Action Model follows the fire-and-forget paradigm.

4.3 Extension

4.3.1 The Body Action Model Was Designed For Extensibility

The set of model classes provided with JEOD doesn't do what you want done? Fine! Design a new class that does what you want.

The Body Action Model places several functional constraints on the classes that derive from the BodyAction base class. These constraints apply to the classes that are a part of the model and to any extensions of the class written by model extenders. These constraints are on the trio of virtual methods as overridden by the derived class.

1. The class must of course ultimately derive from the BodyAction base class.

2. The `initialize()` method, if overridden, must forward the `initialize()` call to the immediate parent object that handles the method. The typical approach is to check for error conditions, forward the `initialize()` call upward, and finally perform class-specific initializations.

3. The `is_ready()` method, if overridden, must query the readiness of the parent class in addition to peforming class-specific readiness checks. In particular, claiming to be ready against the advice of the parent is forbidden.

4. The `apply()` method, if overridden, must forward the `apply()` call to the immediate parent object. The typical approach is to reformulate the user inputs in terms that will be understood by the parent class, forward the `apply()` call upward, and finally perform any cleanup actions.

Chapter 5

Inspection, Verification, and Validation

5.1 Inspection

All of the high-level requirements levied on the Body Action Model are satisfied by inspection.

Inspection BodyAction_1: Top-level Requirements

By inspection, the Body Action Model satisfies the requirements specified in requirement BodyAction_1.

Inspection BodyAction_2: Derived Requirements

The requirements for the Body Action Model are implemented as derived requirements in the subsequent document parts. The inspections, verifications, and validations used to satisfy the derived requirements in turn satisfy these top-level requirements. The requirements flow-down for the model is depicted in table 5.1.

By inspection, the Body Action Model satisfies the requirements specified in requirements BodyAction_2 to BodyAction_9.

Table 5.1: Requirements Flowdown

Requirement	**Artifact**
BodyAction_2 Base class	Reqt. BodyAction_12 Base class Reqt. BodyAction_15 Activation Reqt. BodyAction_16 Virtual methods Reqt. BodyAction_17 Base class mandate
BodyAction_3 Subject specification	Reqt. BodyAction_14 Subject body
BodyAction_4 Initialize mass properties	Reqt. BodyAction_19 Initialize mass properties
BodyAction_5 Initialize mass points	Reqt. BodyAction_20 Initialize mass points
BodyAction_6 Attach MassBody/DynBody objects	Reqt. BodyAction_22 Attach MassBody/DynBody objects Reqt. BodyAction_25 Kinematic Attach DynBody to RefF
BodyAction_7 Detach MassBody/DynBody objects	Reqt. BodyAction_23 Detach from parent Reqt. BodyAction_24 Specific detach
BodyAction_8 State initialization	Reqt. BodyAction_26 Translational state Reqt. BodyAction_27 Rotational state Reqt. BodyAction_29 LVLH frame Reqt. BodyAction_30 NED frame Reqt. BodyAction_31 Readiness detection
BodyAction_9 Frame switch	Reqt. BodyAction_32 Frame switch

Inspection BodyAction_3: Messages

The Body Action Model detects the following classes of errors and warnings:

- Fatal error, `BodyActionMessages::fatal_error`.
  Issued when a non-fatal error is detected and the user has requested that such errors be treated as terminal errors.
- Null pointer, `BodyActionMessages::null_pointer`.
  Issued when a pointer that should have been set is null. Null pointer errors are always fatal.
- Invalid name, `BodyActionMessages::invalid_name`.
  Issued when some name is invalid. The error is not fatal if the action can proceed, but the results should always be treated as suspect when this happens. Invalid names that preclude performing the action are fatal errors.
- Invalid object, `BodyActionMessages::invalid_object`.
  Issued when some object is not of the expected type. Invalid object errors are treated similarly to invalid name errors.
- Illegal value, `BodyActionMessages::illegal_value`.
  Issued when some value is not valid given the context. A warning is issued if the value is correctable. Uncorrectable illegal values are fatal errors.
- Unperformed action, `BodyActionMessages::not_performed`.
  These are serious but non-fatal errors, issued under two circumstances:
  - An object never indicates that it is ready to be performed when it should have indicated readiness at some point in time. This will occur, for example, when the user has created circular dependencies amongst the queued DynBodyInit objects.
  - An object indicates that it is ready to be performed but fails to perform the action as requested. This will occur, for example, when the user has attempts to create an illegal attachment.

The above are reported to the user via the *Message Handler Model* [10].

The Body Action Model also uses the Message Handler Model to indicate successful application of a body action. These success messages use the `MessageHandler::debug()` method with message class `BodyActionMessages::trace`. These debug messages are not generated by default; users will not see these messages unless they explicitly enable their generation. The tests BodyAction_1, BodyAction_3, and BodyAction_4 do exactly that.

By inspection, the Body Action Model satisfies requirements BodyAction_10 and BodyAction_11.

5.2 Metrics

Table 5.2 presents coarse metrics on the source files that comprise the model.

Table 5.2: Coarse Metrics

	Number of Lines			
File Name	**Blank**	**Comment**	**Code**	**Total**
Total	0	0	0	0

Table 5.3 presents the extended cyclomatic complexity (ECC) of the methods defined in the model.

Table 5.3: Cyclomatic Complexity

Method	**File**	**Line**	**ECC**
jeod::std::get_identifier (void)	include/body_action.hh	260	1
jeod::BodyAttach::~Body Attach (void)	include/body_attach.hh	176	1
jeod::BodyAttachAligned::~ BodyAttachAligned (void)	include/body_attach_ aligned.hh	136	1
jeod::BodyAttachMatrix::~ BodyAttachMatrix (void)	include/body_attach_ matrix.hh	126	1
jeod::BodyDetach::~Body Detach (void)	include/body_detach.hh	119	1
jeod::BodyDetachSpecific::~ BodyDetachSpecific (void)	include/body_detach_ specific.hh	159	1
jeod::BodyReattach::~Body Reattach (void)	include/body_reattach.hh	130	1
jeod::DynBodyInitRotState::~ DynBodyInitRotState (void)	include/dyn_body_init_rot_ state.hh	146	1
jeod::DynBodyInitTrans State::~DynBodyInitTrans State (void)	include/dyn_body_init_trans_ state.hh	143	1
jeod::MassBodyInit::~Mass BodyInit (void)	include/mass_body_init.hh	139	1
jeod::BodyAction::Body Action (void)	src/body_action.cc	55	1
jeod::BodyAction::~Body Action (void)	src/body_action.cc	72	1
jeod::BodyAction::shutdown (void)	src/body_action.cc	82	1
jeod::BodyAction::initialize (DynManager & dyn_ manager JEOD_UNUSED)	src/body_action.cc	93	2

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	**File**	**Line**	**ECC**
jeod::BodyAction::apply (Dyn Manager & dyn_manager J EOD_UNUSED)	src/body_action.cc	118	1
jeod::BodyAction::is_ready (void)	src/body_action.cc	133	1
jeod::BodyAction::set_ subject_body (MassBody &mass_body_in)	src/body_action.cc	146	1
jeod::BodyAction::set_ subject_body (DynBody &dyn_body_in)	src/body_action.cc	152	1
jeod::BodyAction::validate_ body_inputs (DynBody *& dyn_body_in, MassBody *& mass_body_in, const std:: string & body_base_name, bool allow_failure)	src/body_action.cc	158	10
jeod::BodyAction::is_same_ subject_body (MassBody &mass_body_in)	src/body_action.cc	201	2
jeod::BodyAction::is_subject_ dyn_body ()	src/body_action.cc	213	3
jeod::BodyAction::get_ subject_dyn_body ()	src/body_action.cc	229	3
jeod::BodyAction::validate_ name (const std::string & variable_value, const std:: string & variable_name, const std::string & variable_ type)	src/body_action.cc	245	2
jeod::BodyAttach::Body Attach (void)	src/body_attach.cc	49	1
jeod::BodyAttach::initialize (DynManager & dyn_ manager)	src/body_attach.cc	68	3
jeod::BodyAttach::set_parent_ body (MassBody &mass_ body_in)	src/body_attach.cc	97	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::BodyAttach::set_parent_body (DynBody &dyn_body_in)	src/body_attach.cc	104	1
jeod::BodyAttach::set_parent_frame (RefFrame & ref_parent_in)	src/body_attach.cc	111	1
jeod::BodyAttach::apply (DynManager & dyn_manager)	src/body_attach.cc	118	8
jeod::BodyAttachAligned::BodyAttachAligned (void)	src/body_attach_aligned.cc	51	1
jeod::BodyAttachAligned::initialize (DynManager & dyn_manager)	src/body_attach_aligned.cc	64	3
jeod::BodyAttachAligned::apply (DynManager & dyn_manager)	src/body_attach_aligned.cc	99	7
jeod::BodyAttachMatrix::BodyAttachMatrix (void)	src/body_attach_matrix.cc	48	1
jeod::BodyAttachMatrix::apply (DynManager & dyn_manager)	src/body_attach_matrix.cc	60	7
jeod::BodyDetach::Body Detach (void)	src/body_detach.cc	47	1
jeod::BodyDetach::apply (DynManager & dyn_manager)	src/body_detach.cc	60	4
jeod::BodyDetach::is_ready (void)	src/body_detach.cc	114	1
jeod::BodyDetachSpecific::BodyDetachSpecific (void)	src/body_detach_specific.cc	49	1
jeod::BodyDetachSpecific::initialize (DynManager & dyn_manager)	src/body_detach_specific.cc	64	1
jeod::BodyDetachSpecific::apply (DynManager & dyn_manager)	src/body_detach_specific.cc	82	9
jeod::BodyDetachSpecific::set_detach_from_body (MassBody &mass_body_in)	src/body_detach_specific.cc	154	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::BodyDetachSpecific:: set_detach_from_body (Dyn Body &dyn_body_in)	src/body_detach_specific.cc	160	1
jeod::BodyDetachSpecific::is_ ready (void)	src/body_detach_specific.cc	166	1
jeod::BodyReattach::Body Reattach (void)	src/body_reattach.cc	47	1
jeod::BodyReattach::apply (DynManager & dyn_ manager)	src/body_reattach.cc	60	4
jeod::DynBodyFrameSwitch:: DynBodyFrameSwitch (void)	src/dyn_body_frame_switch.cc	54	1
jeod::DynBodyFrameSwitch:: ~DynBodyFrameSwitch (void)	src/dyn_body_frame_switch.cc	70	1
jeod::DynBodyFrameSwitch:: initialize (DynManager & dyn_manager)	src/dyn_body_frame_switch.cc	80	4
jeod::DynBodyFrameSwitch:: apply (DynManager & dyn_ manager)	src/dyn_body_frame_switch.cc	140	4
jeod::DynBodyFrameSwitch:: is_ready (void)	src/dyn_body_frame_switch.cc	188	3
jeod::DynBodyInit::DynBody Init (void)	src/dyn_body_init.cc	57	1
jeod::DynBodyInit::~Dyn BodyInit (void)	src/dyn_body_init.cc	82	1
jeod::DynBodyInit::initialize (DynManager & dyn_ manager)	src/dyn_body_init.cc	92	6
jeod::RefFrameItems:: initializes_what (void)	src/dyn_body_init.cc	166	1
jeod::DynBodyInit::is_ready (void)	src/dyn_body_init.cc	180	8
jeod::DynBodyInit::apply (DynManager & dyn_ manager)	src/dyn_body_init.cc	244	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::DynBodyInit::report_failure (void)	src/dyn_body_init.cc	288	2
jeod::DynBodyInit::apply_user_inputs (void)	src/dyn_body_init.cc	312	7
jeod::DynBodyInit::compute_rotational_state (void)	src/dyn_body_init.cc	374	2
jeod::DynBodyInit::compute_translational_state (void)	src/dyn_body_init.cc	397	2
jeod::DynBodyInit::find_planet (const DynManager & dyn_manager, const std::string & planet_name, const std::string & variable_name)	src/dyn_body_init.cc	421	2
jeod::DynBodyInit::find_dyn_body (const DynManager & dyn_manager, const std::string & dyn_body_name, const std::string & variable_name)	src/dyn_body_init.cc	458	2
jeod::DynBodyInit::find_ref_frame (const DynManager & dyn_manager, const std::string & ref_frame_name, const std::string & variable_name)	src/dyn_body_init.cc	495	2
jeod::DynBodyInit::find_body_frame (DynBody & frame_container, const std::string & body_frame_identifier, const std::string & variable_name)	src/dyn_body_init.cc	532	2
jeod::DynBodyInitLvlhRotState::DynBodyInitLvlhRotState (void)	src/dyn_body_init_lvlh_rot_state.cc	56	1
jeod::DynBodyInitLvlhRotState::~DynBodyInitLvlhRotState (void)	src/dyn_body_init_lvlh_rot_state.cc	70	1
jeod::DynBodyInitLvlhRotState::initialize (DynManager & dyn_manager)	src/dyn_body_init_lvlh_rot_state.cc	81	6

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::DynBodyInitLvlhState:: DynBodyInitLvlhState (void)	src/dyn_body_init_lvlh_ state.cc	46	1
jeod::DynBodyInitLvlhState:: ~DynBodyInitLvlhState (void)	src/dyn_body_init_lvlh_ state.cc	61	1
jeod::DynBodyInitLvlhState:: set_lvlh_frame_object (Lvlh Frame & lvlh_frame_object)	src/dyn_body_init_lvlh_ state.cc	70	1
jeod::DynBodyInitLvlhState:: initialize (DynManager & dyn_manager)	src/dyn_body_init_lvlh_ state.cc	82	2
jeod::DynBodyInitLvlhState:: apply (DynManager & dyn_ manager)	src/dyn_body_init_lvlh_ state.cc	107	5
jeod::DynBodyInitLvlhTrans State::DynBodyInitLvlh TransState (void)	src/dyn_body_init_lvlh_trans_ state.cc	50	1
jeod::DynBodyInitLvlhTrans State::~DynBodyInitLvlh TransState (void)	src/dyn_body_init_lvlh_trans_ state.cc	63	1
jeod::DynBodyInitLvlhTrans State::initialize (Dyn Manager & dyn_manager)	src/dyn_body_init_lvlh_trans_ state.cc	73	3
jeod::DynBodyInitNedRot State::DynBodyInitNedRot State (void)	src/dyn_body_init_ned_rot_ state.cc	51	1
jeod::DynBodyInitNedRot State::~DynBodyInitNed RotState (void)	src/dyn_body_init_ned_rot_ state.cc	65	1
jeod::DynBodyInitNedRot State::initialize (Dyn Manager & dyn_manager)	src/dyn_body_init_ned_rot_ state.cc	76	3
jeod::DynBodyInitNedState:: DynBodyInitNedState (void)	src/dyn_body_init_ned_ state.cc	57	1
jeod::DynBodyInitNedState:: ~DynBodyInitNedState (void)	src/dyn_body_init_ned_ state.cc	76	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::DynBodyInitNedState:: set_use_alt_pfix (const bool use_alt_pfix_in)	src/dyn_body_init_ned_ state.cc	86	1
jeod::DynBodyInitNedState:: initialize (DynManager & dyn_manager)	src/dyn_body_init_ned_ state.cc	96	3
jeod::DynBodyInitNedState:: apply (DynManager & dyn_ manager)	src/dyn_body_init_ned_ state.cc	123	5
jeod::DynBodyInitNedTrans State::DynBodyInitNed TransState (void)	src/dyn_body_init_ned_trans_ state.cc	49	1
jeod::DynBodyInitNedTrans State::~DynBodyInitNed TransState (void)	src/dyn_body_init_ned_trans_ state.cc	62	1
jeod::DynBodyInitNedTrans State::initialize (Dyn Manager & dyn_manager)	src/dyn_body_init_ned_trans_ state.cc	72	3
jeod::DynBodyInitOrbit::Dyn BodyInitOrbit (void)	src/dyn_body_init_orbit.cc	56	1
jeod::DynBodyInitOrbit::~ DynBodyInitOrbit (void)	src/dyn_body_init_orbit.cc	84	1
jeod::DynBodyInitOrbit:: initialize (DynManager & dyn_manager)	src/dyn_body_init_orbit.cc	93	13
jeod::DynBodyInitOrbit:: apply (DynManager & dyn_ manager)	src/dyn_body_init_orbit.cc	187	16
jeod::DynBodyInitPlanet Derived::DynBodyInit PlanetDerived (void)	src/dyn_body_init_planet_ derived.cc	48	1
jeod::DynBodyInitPlanet Derived::~DynBodyInit PlanetDerived (void)	src/dyn_body_init_planet_ derived.cc	64	1
jeod::DynBodyInitPlanet Derived::initialize (Dyn Manager & dyn_manager)	src/dyn_body_init_planet_ derived.cc	74	3
jeod::DynBodyInitPlanet Derived::is_ready (void)	src/dyn_body_init_planet_ derived.cc	99	3

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	**File**	**Line**	**ECC**
jeod::DynBodyInitPlanet Derived::apply (Dyn Manager & dyn_manager)	src/dyn_body_init_planet_ derived.cc	123	1
jeod::DynBodyInitRotState:: DynBodyInitRotState (void)	src/dyn_body_init_rot_state.cc	51	1
jeod::RefFrameItems:: initializes_what (void)	src/dyn_body_init_rot_state.cc	63	4
jeod::DynBodyInitRotState:: is_ready (void)	src/dyn_body_init_rot_state.cc	91	5
jeod::DynBodyInitRotState:: initialize (DynManager & dyn_manager)	src/dyn_body_init_rot_state.cc	140	4
jeod::DynBodyInitRotState:: apply (DynManager & dyn_ manager)	src/dyn_body_init_rot_state.cc	166	1
jeod::DynBodyInitTrans State::DynBodyInitTrans State (void)	src/dyn_body_init_trans_ state.cc	50	1
jeod::RefFrameItems:: initializes_what (void)	src/dyn_body_init_trans_ state.cc	63	4
jeod::DynBodyInitTrans State::is_ready (void)	src/dyn_body_init_trans_ state.cc	91	6
jeod::DynBodyInitTrans State::initialize (Dyn Manager & dyn_manager)	src/dyn_body_init_trans_ state.cc	145	4
jeod::DynBodyInitTrans State::apply (DynManager & dyn_manager)	src/dyn_body_init_trans_ state.cc	172	1
jeod::DynBodyInitWrt Planet::DynBodyInitWrt Planet (void)	src/dyn_body_init_wrt_ planet.cc	45	1
jeod::DynBodyInitWrt Planet::~DynBodyInitWrt Planet (void)	src/dyn_body_init_wrt_ planet.cc	61	1
jeod::DynBodyInitWrt Planet::initialize (Dyn Manager & dyn_manager)	src/dyn_body_init_wrt_ planet.cc	72	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::RefFrameItems:: initializes_what (void)	src/dyn_body_init_wrt_ planet.cc	96	1
jeod::DynBodyInitWrt Planet::is_ready (void)	src/dyn_body_init_wrt_ planet.cc	109	1
jeod::DynBodyInitWrt Planet::apply (DynManager & dyn_manager)	src/dyn_body_init_wrt_ planet.cc	124	1
jeod::MassBodyInit::Mass BodyInit (void)	src/mass_body_init.cc	52	1
jeod::MassBodyInit::apply (DynManager & dyn_ manager)	src/mass_body_init.cc	66	1

5.3 Requirements Traceability

The IV&V artifacts that demonstrate the satisfaction of the requirements in chapter 2 are depicted in table 5.4.

Table 5.4: Requirements Traceability

Requirement		Artifact
BodyAction_1	Top-level requirement	Insp. BodyAction_1 Top-level requirements
BodyAction_2	Base class	Insp. BodyAction_2 Derived requirements
BodyAction_3	Subject specification	Insp. BodyAction_2 Derived requirements
BodyAction_4	Initialize mass properties	Insp. BodyAction_2 Derived requirements
BodyAction_5	Initialize mass points	Insp. BodyAction_2 Derived requirements
BodyAction_6	Attach MassBody objects	Insp. BodyAction_2 Derived requirements
BodyAction_7	Detach MassBody/DynBody objects	Insp. BodyAction_2 Derived requirements
BodyAction_8	State initialization	Insp. BodyAction_2 Derived requirements
BodyAction_9	Frame switch	Insp. BodyAction_2 Derived requirements
BodyAction_10	Errors	Insp. BodyAction_3 Messages
BodyAction_11	Trace	Insp. BodyAction_3 Messages

Part II

Body Action Base Classes Sub-Model

Chapter 6

Introduction

6.1 Purpose and Objectives of the Body Action Model Base Classes

The Body Action Model defines two base classes: BodyAction and BodyActionMessages. The latter class encapsulates identifiers for use with the MessageHandler. All of the instantiable Body Action Model classes ultimately derive from the BodyAction base class. This part of the document describes these base classes.

The BodyAction base class by itself does not change a single aspect of a MassBody or DynBody object. Making changes to a MassBody, DynBody, or derived class object is the responsibility of the various classes that derive from the base BodyAction class. What the base class does provide is a common framework for describing actions to be performed on bodies.

6.2 Part Organization

This part of the Body Action Model document is organized along the lines described in section 1.4. It comprises the following chapters in order:

Introduction - This introduction describes the objective and purpose of the base classes.

Product Requirements - The Body Action Model Base Classes Product Requirements chapter describes the requirements on the model base classes and the requirements levied on classes that derive from the BodyAction class.

Product Specification - The Body Action Model Base Classes Product Specification chapter describes the underlying theory, architecture, and design of the BodyAction class.

User Guide - Describes the use of the BodyAction class. The BodyAction User Guide chapter is organized in the following sections:

- Analysts or users of simulations (Analysis).
- Integrators or developers of simulations (Integration).

- Model Extenders (Extension).

Inspection, Verification, and Validation - The BodyAction class is not an instantiable class. As a result, all of the BodyAction class requirements involve inspection only.

Chapter 7

Product Requirements

Requirement BodyAction_12: Body Action Base Class Definition

Requirement:
The Body Action Model shall define a class capable of forming the basis for all model classes that operate on a linked MassBody or DynBody object.

Rationale:
This requirement derives from requirement BodyAction_2.

Verification:
Inspection

Requirement BodyAction_13: Object Identification

Requirement:
Each instance of the Body Action Model base class shall provide the ability to identify itself.

Rationale:
This capability is needed for error handling and message generation.

Verification:
Inspection

Requirement BodyAction_14: Subject Specification

Requirement:
Each instance of the Body Action Model base class shall identify the subject body object on which the model instance is to operate.

Rationale:
This requirement derives from requirement BodyAction_3.

Verification:

Inspection

Requirement BodyAction_15: Activation

Requirement:

The Body Action Model base class shall provide an activation capability.

Rationale:

This derived requirement requirement derives from requirement BodyAction_2.

Verification:

Inspection

Requirement BodyAction_16: Virtual Methods

Requirement:

The BodyAction class shall define the functional interfaces that generically describe the operations on model objects. These functional interfaces are

16.1 Initialization. The BodyAction class shall define a generic interface for initializing BodyAction objects.

16.2 Query readiness. The BodyAction class shall define a generic interface for querying the execution readiness of a BodyAction object.

16.3 Execute. The BodyAction class shall define a generic interface for querying the execution readiness of a BodyAction object. A BodyAction instance shall provide the ability to perform some action on a MassBody/DynBody object.

Rationale:

This requirement derives from requirement BodyAction_2.

Verification:

Inspection

Requirement BodyAction_17: Body Action Base Class Mandate

Requirement:

All Body Action Model classes that operate on a MassBody/DynBody object shall derive from the BodyAction base class.

Rationale:

This requirement derives from requirement BodyAction_2.

Note: This is a requirement levied by the Body Action Base Classes Sub-Model on the subsequent sub-models.

Verification:

Inspection

Requirement BodyAction_18: Virtual Method Overrides

Requirement:

All Body Action Model classes that override the virtual methods described in requirement Body-Action_16 shall forward the method invocation to the most immediate parent class that handles this method.

18.1 Initialization. Derived classes that override the `initialize()` shall not return from `initialize()` without having invoked the overridden implementation of the method.

18.2 Query readiness. Derived classes that override the `is_ready()` must not indicate that the instance is ready to execute if the overridden `is_ready()` method would indicated otherwise.

18.3 Execute. Derived classes that override the `apply()` shall not return from `apply()` without having invoked the overridden implementation of the method.

Rationale:

This is an internal requirement levied on derived classes. The intent is to reduce replication of code and to ensure that the parent classes are safe to operate.

Note: This requirement does not pertain in the exceptional cases where the overriding method does not return. In particular, the method MessageHandler::Fail() does not return.

Verification:

Inspection

Chapter 8

Product Specification

8.1 Conceptual Design

All of the instantiable Body Action Model classes ultimately derive from the BodyAction base class. This base class by itself does not change a single aspect of a subject MassBody, DynBody, or derived class object. Making changes to a body is the responsibility of the various classes that derive from the base BodyAction class. The base class provides a common framework for describing actions to be performed. This framework includes

- Descent from a common base class. This common basis enables forming a collection of body actions in the form of base class pointers. The DynManager class does exactly this. The Body Action Model was explicitly architected to enable this treatment.
- Base class member data, including a pointer to the `mass_subject` MassBody object or `dyn_subject` DynBody object that is the subject of the BodyAction object.
- Base class member functions that specify how to operate on the BodyAction object. In particular, a trio of virtual methods described below form a standard the basis by which derived classes can extend this base class.

8.2 Mathematical Formulations

N/A

8.3 Detailed Design

The base class defines three virtual methods that together form the function interface by which an external user of the Body Action Model functionally interacts with BodyAction objects. These methods are

initialize Each Body Action Model object must be initialized. This initialization is the task of the overridable `initialize()` method. This method initializes a BodyAction object. It does not alter the subject object.

is_ready The `is_ready()` method indicates whether the BodyAction object is ready to perform its action on the subject object, but does not alter the object. For example, a DynBodyInitLvlhState object sets parts of the subject DynBody object's state based on the Local Vertical, Local Horizontal (LVLH) frame defined by some reference DynBody object. The DynBodyInitLvlhState object remains in a not-ready state until the reference body's position and velocity vector have been set.

apply The `apply()` method performs the action on the subject object.

The BodyActionMessages class simply encapsulates labels for use with the JEOD Message Handler. The model reports the following types of messages:

fatal_error Issued when performing an action results in an error return from the method performing the action.

illegal_value Issued when a simple type (e.g. an enum) has an illegal value.

invalid_name Issued when a name is invalid (NULL, empty, or does not name an object of the specified type).

invalid_object Issued when a pointer points to an object of the wrong type.

null_pointer Error issued when a pointer is required but was not provided.

not_performed Issued when a BodyAction cannot be run.

trace Debug message issued to trace BodyAction actions.

Chapter 9

User Guide

The BodyAction class is not instantiable. The use of the model as a whole is described in section 4. The uses of specific parts of the model are described in the User Guide chapters for those specific parts.

Chapter 10

Inspection, Verification, and Validation

10.1 Inspection

Inspection BodyAction_4: Body Action Base Class

The Body Action Model defines the BodyAction as a base class for operating on MassBody-derived and DynBody-derived objects.

The BodyAction class defines

- Member data that identifies the subject object,
- Member data that specifies whether the action is active,
- Member data and functions that identify the BodyAction instance, and
- A trio of virtual methods that generically define the operations on the classes that derive from the BodyAction class.

By inspection, the Body Action Model satisfies the following requirements:

- Requirement BodyAction_12
- Requirement BodyAction_13
- Requirement BodyAction_14
- Requirement BodyAction_15
- Requirement BodyAction_16

Inspection BodyAction_5: Derived Classes

The Body Action Model defines two base classes, BodyActionMessages and BodyAction, and several derived classes. The class BodyActionMessages is used for generating messages, is not instantiated, and has no derived classes. All derived model classes ultimately inherit from the BodyAction base class as required.

All model derived classes that override the `initialize()`, `is_ready()`, and `apply()` methods pass these invocations to the relevant parent class and appropriately use the return value (if any) from that parent invocation before returning from the overriding method.

By inspection, the Body Action Model satisfies the following requirements:

- Requirement BodyAction_17
- Requirement BodyAction_18

10.2 Validation

The BodyAction class is an abstract class and thus cannot be tested directly.

10.3 Requirements Traceability

Table 10.1 summarizes the inspections and tests that demonstrate the satisfaction of the requirements levied on the Body Action Base Classes Sub-Model.

Table 10.1: BodyAction Requirements Traceability

Requirement	**Artifact**
BodyAction_12 Base class	Insp. BodyAction_4 Base class
BodyAction_13 Object identification	Insp. BodyAction_4 Base class
BodyAction_14 Subject body	Insp. BodyAction_4 Base class
BodyAction_15 Activation	Insp. BodyAction_4 Base class
BodyAction_16 Virtual methods	Insp. BodyAction_4 Base class Insp. BodyAction_6 MassBodyInit design Insp. BodyAction_7 Attach/detach design Insp. BodyAction_8 DynBodyInit design Insp. BodyAction_9 Frame switch design
BodyAction_17 Base class mandate	Insp. BodyAction_5 Derived classes Insp. BodyAction_6 MassBodyInit design Insp. BodyAction_7 Attach/detach design Insp. BodyAction_8 DynBodyInit design Insp. BodyAction_9 Frame switch design
BodyAction_18 Virtual method overrides	Insp. BodyAction_5 Derived classes

Part III

MassBodyInit Class Sub-Model

Chapter 11

Introduction

11.1 Purpose and Objectives of the MassBodyInit Class

The MassBodyInit Class is responsible for initializing a MassBody object's mass properties and mass points.

11.2 Part Organization

This part of the Body Action Model document is organized along the lines described in section 1.4. It comprises the following chapters in order:

Introduction - This introduction describes the objective and purpose of the MassBodyInit Class.

Product Requirements - The MassBodyInit Class Product Requirements chapter describes the requirements on the MassBodyInit class.

Product Specification - The MassBodyInit Class Product Specification chapter describes the underlying theory, architecture, and design of this class.

User Guide - Describes the use of this class.

Inspection, Verification, and Validation - The DMassBodyInit Class IV&V chapter describes the techniques used to ascertain that this class satisfies the requirements levied upon it and the summarizes the results of the sub-model verification and validation tests.

Chapter 12

Product Requirements

Requirement BodyAction_19: Initialize Mass Properties

Requirement:

This class shall provide the ability to initialize the mass properties of a subject MassBody object and specify the orientation of the subject object's body reference frame with respect to its structural reference frame.

Rationale:

This is the primary purpose of this class. This requirement derives directly from requirement BodyAction_4.

Verification:

Inspection, Test

Requirement BodyAction_20: Initialize Mass Points

Requirement:

This class shall provide the ability to specify the points of interest ('mass points' and 'vehicle points') on MassBody and DynBody objects.

Rationale:

This requirement derives directly from requirement BodyAction_5.

Verification:

Inspection, Test

Chapter 13

Product Specification

13.1 Conceptual Design

The MassBodyInit class initializes the subject body's mass properties and establishes the subject body's mass points. The class derives directly from the model BodyAction base class.

13.2 Mathematical Formulations

All of the underlying mathematics is performed by the Mass Model. Refer to the Mathematical Formulations section of the *Mass Body Model* [14].

13.3 Detailed Design

This is a very simple sub-model. It adds three data elements to the base class and overrides the base class `apply()` method. The data elements are

properties type = `MassPropertiesInit`
: Specifications for the subject mass body's core mass properties.

points type = `MassPointInit *`
: Specifications for the subject mass body's mass points.

num_points type = `unsigned int`
: Size of the points array.

The classes MassPropertiesInit and MassPointInit are part of the Mass Model. See the *Mass Body Model* [14] for details.

The `apply()` method calls the subject body's `initialize_mass()` method, which initializes the body's mass properties given the supplied mass properties initialization data and establishes the body's auxiliary mass points given the supplied mass point initialization data.

The MassBody class initializes its mass points by calling the `add_mass_point()` for each supplied mass point. The DynBody class overrides the `add_mass_point()` method. The effect of the override is to add the point of interest to the underlying MassBody object's list of MassPoint objects and to the DynBody object's list of VehiclePoint objects. These vehicle points can later be used for attaching bodies and initializing body states.

Chapter 14

User Guide

The following assumes the scheme outlined in section 4.2.4.

14.1 Analysis

Assume a simulation S_define file declares simulation objects named ‘vehicle’ and ‘dynamics’. The vehicle object contains a DynBody object name ‘dyn_body’ and a MassBodyInit object named ‘mass_init’. The dynamics object contains a DynManager object named ‘manager’ and a BodyAction pointer named ‘body_action_ptr’.

The following will cause the vehicle’s mass properties to be initialized. In this example, the body and structural frames are a related by a 180 degree pitch. The vehicle is initialized with one point of interest.

```
vehicle.mass_init.set_subject_body(vehicle.dyn_body.mass);
vehicle.mass_init.action_name = "vehicle.mass";

vehicle.mass_init.properties.mass {kg} = 10000.0;
vehicle.mass_init.properties.position[0] {M} = 27.856, 0.003, 9.600;

vehicle.mass_init.properties.inertia_spec = MassPropertiesInit::Body;
vehicle.mass_init.properties.inertia[0][0] {kg*M2} =  7e11,  0.0,   0.0;
vehicle.mass_init.properties.inertia[1][0] {kg*M2} =  0.0,  12e11,  0.0;
vehicle.mass_init.properties.inertia[2][0] {kg*M2} =  0.0,   0.0,  10e11;

vehicle.mass_init.properties.pt_orientation.data_source =
   Orientation::InputMatrix;
vehicle.mass_init.properties.pt_frame_spec =
   MassPointInit::StructToBody;
vehicle.mass_init.properties.pt_orientation.trans[0][0] =  -1.0,  0.0,  0.0;
vehicle.mass_init.properties.pt_orientation.trans[1][0] =   0.0,  1.0,  0.0;
vehicle.mass_init.properties.pt_orientation.trans[2][0] =   0.0,  0.0, -1.0;
vehicle.mass_init.num_points = 1;
vehicle.mass_init.points = alloc(1);
vehicle.mass_init.points[0].set_name( "attach_point" );
vehicle.mass_init.points[0].position[0] {M} = 3.937, 0.003, 9.600;
vehicle.mass_init.points[0].pt_orientation.trans[0][0] = -1.0,  0.0,  0.0;
vehicle.mass_init.points[0].pt_orientation.trans[1][0] =  0.0,  1.0,  0.0;
vehicle.mass_init.points[0].pt_orientation.trans[2][0] =  0.0,  0.0, -1.0;
vehicle.mass_init.points[0].pt_orientation.data_source =
   Orientation::InputMatrix;
vehicle.mass_init.points[0].pt_frame_spec =
   MassPointInit::StructToPoint;

dynamics.body_action_ptr = &vehicle.mass_init;
call dynamics.dynamics.manager.add_body_action (dynamics.body_action_ptr);
```

14.2 Integration

The simulation integrator must make the MassBodyInit visible to the simulation user. The above example assumes the integrator declared a MassBodyInit object named 'mass_init' as part of the vehicle simulation object.

14.3 Extension

This class is not recommended for user extensions.

Chapter 15

Inspection, Verification, and Validation

15.1 Inspection

Inspection BodyAction_6: Design Inspection

The MassBodyInit Class Sub-Model comprises a single class, class MassBodyInit, which derives directly from the BodyAction class. This class overrides the `apply()` method. The implementation of that method invokes the overridden parent class implementation.

By inspection, the MassBodyInit Class Sub-Model satisfies requirements BodyAction_16 and BodyAction_17.

15.2 Validation

This sub-model is validated as a part of the MassBody Attach/Detach Sub-Model validation. See section 20.2.

15.3 Requirements Traceability

Table 15.1 summarizes the inspections and tests that demonstrate the satisfaction of the requirements levied on the MassBodyInit Class Sub-Model.

Table 15.1: MassBodyInit Requirements Traceability

Requirement	**Artifact**
BodyAction_19 Initialize mass properties	Test BodyAction_1 Attach/detach
BodyAction_20 Initialize mass points	Test BodyAction_1 Attach/detach

Part IV

MassBody Attach/Detach Sub-Model

Chapter 16

Introduction

16.1 Purpose and Objectives of the MassBody Attach/Detach Sub-Model

The MassBody Attach/Detach Sub-Model is responsible for attaching and detaching a subject MassBody or DynBody object to and from another Body object or RefFrame object.

16.2 Part Organization

This part of the Body Action Model document is organized along the lines described in section 1.4. It comprises the following chapters in order:

Introduction - This introduction describes the objective and purpose of the Attach/Detach Sub-Model.

Product Requirements - The Body Attach/Detach Sub-Model Product Requirements chapter describes the requirements on the sub-model and the requirements levied on user-defined classes that derive from this sub-model.

Product Specification - The Body Attach/Detach Sub-Model Product Specification chapter describes the underlying theory, architecture, and design of this sub-model.

User Guide - Describes the use of this sub-model. The Body Attach/Detach Sub-Model User Guide chapter the following sections:

- Analysts or users of simulations (Analysis).
- Integrators or developers of simulations (Integration).
- Model Extenders (Extension).

Inspection, Verification, and Validation - The Body Attach/Detach Sub-Model IV&V chapter describes the techniques used to ascertain that this sub-model satisfies the requirements levied upon it and the summarizes the results of the sub-model verification and validation tests.

Chapter 17

Product Requirements

Requirement BodyAction_21: Extensible Attachment Framework

Requirement:

This sub-model shall provide an extensible framework for attaching the subject MassBody to some other MassBody object.

Rationale:

The MassBody class defines multiple `attach()` interfaces.

Verification:

Inspection

Requirement BodyAction_22: Attach MassBody/DynBody Objects

Requirement:

This sub-model shall provide the ability to cause the subject MassBody object to be attached to another MassBody object

22.1 Point-to-point attach Given a pair of mass points on the two objects to be attached with zero offset between the mass point origins and a 180° yaw between the mass point axes.

22.2 Matrix/offset attach Given an offset from the target body's structural origin to the subject body's structural origin and a transformation matrix from the target body's structural axes to the subject body's structural axes.

Rationale:

This requirement derives directly from requirement BodyAction_6.

Verification:

Inspection, Test

Requirement BodyAction_23: Detach MassBody/DynBody Object From Immediate Parent

Requirement:

This sub-model shall provide the ability to cause the subject MassBody object to be detached from its mass tree parent object via the subject MassBody object's immediate `detach()` method.

Rationale:

This requirement derives directly from requirement BodyAction_7.

Verification:

Inspection, Test

Requirement BodyAction_24: Detach MassBody/DynBody Objects Below Specified Parent

Requirement:

This sub-model shall provide the ability to cause the mass subtree that contains the subject MassBody object to be detached from a specified parent.

Rationale:

This requirement derives directly from requirement BodyAction_7. The MassBody model provides two `detach()` methods. This method uses the indirect detach version of that method.

Verification:

Inspection, Test

Requirement BodyAction_25: Kinematic Attach DynBody to RefFrame Objects

Requirement:

This sub-model shall provide the ability to cause the subject DynBody object to be attached to a RefFrame object kinematically.

25.1 Point-to-point attach Given a mass point on the DynBody and the reference frame to be attached with zero offset between the two point origins and a 180° yaw between the mass point and RefFrame frames.

25.2 Matrix/offset attach Given an offset from the parent reference frame origin to the subject body's structural origin and a transformation matrix from the parent reference frame axes to the subject body's structural axes.

Rationale:

This requirement derives directly from requirement BodyAction_6.

Verification:

Inspection, Test

Chapter 18

Product Specification

18.1 Conceptual Design

The MassBody Attach/Detach Sub-Model comprises six classes.

- BodyAttach, which derives directly from the base BodyAction class. This class provides the conceptual basis for attaching two MassBody objects, a MassBody object to a DynBody, two DynBody objects or a DynBody to a RefFrame, exclusively. The subject of the action is to be attached as a child of a parent object. The geometric details of the attachment (and the actual attachment) is a task left to derived classes.

- BodyAttachAligned, which derives from BodyAttach. A BodyAttachAligned object attaches two objects using a point-to-point attachment mechanism.

- BodyAttachMatrix, which also derives from BodyAttach. A BodyAttachMatrix object attaches two objects using a geometric attachment mechanism.

- BodyReattach, which derives directly from BodyAction. A BodyReattach object alters the geometry of an existing attachment.

- BodyDetach, which derives directly from BodyAction. A BodyDetach object detaches a MassBody object from its immediate parent (which is not specified).

- BodyDetachSpecific, which derives directly from BodyDetach. A BodyDetachSpecific object detaches a MassBody subtree from a specific parent.

18.2 Mathematical Formulations

All of the underlying mathematics is performed by the Mass Model and the Dynamic Body Model, depending on what kinds of bodies are being attached, reattached, or detached. Refer to the Mathematical Formulations section of the *Mass Body Model* [14] and the *Dynamic Body Model* [3].

18.3 Detailed Design

18.3.1 Attachment

The Body Action Model BodyAttach class provides the basis for performing attachments. This class contains a member data element that identifies the parent MassBody to which the subject MassBody is to be attached. However, this class itself does not perform the attachment. That is the task of classes that derive from BodyAttach. Two concepts drove this design.

- The MassBody class provides multiple ways to accomplish the basic task of attaching MassBody objects in the form of multiple `attach_to/attach_child()` methods. Distinct classes that derive from BodyAttach use distinct `attach_to/attach_child()` mechanisms.
- The DynBody class provides multiple ways to accomplish the basic task of attaching DynBody objects in the form of multiple `attach_to/attach_child()` methods. Distinct classes that derive from BodyAttach use distinct `attach_to/attach_child()` mechanisms.
- The DynBody class provides multiple ways to accomplish the basic task of kinematically attaching a DynBody object to a RefFrame in the form of multiple `attach_to_frame()` methods. Distinct classes that derive from BodyAttach use distinct `attach_to_frame()` mechanisms.
- The Dynamics Manager takes advantage of this inheritance to attach MassBody objects at initialization time. Any enqueued BodyAction objects that derive from BodyAttach and are ready to run at initialization time will be run as a part of the overall initialization processing. Attachments are performed after having performed all of the actions that initialize the mass properties and mass points of the simulation's MassBody objects but before performing any state initializations.

JEOD supplies two classes that derive from BodyAttach. BodyAttachAligned uses the point-to-point form of `attach_to/attach_child()/attach_to_frame()` while BodyAttachMatrix uses the offset+matrix form. The BodyAttachAligned class contains data members that name the attach points on the two vehicles. The point-to-point `apply()` method attaches the subject MassBody as a child of the parent MassBody object at the named attach points. The BodyAttachMatrix contains data members that specify the location and orientation of the subject body's structural frame with respect to the parent body's structural frame. The offset+matrix `apply()` method attaches the subject MassBody as a child of the parent MassBody object using the supplied offset and orientation.

18.3.2 Reattachment

18.3.3 Detachment

The MassBody class provides two versions of the `detach()` method. Each of the two sub-model classes that perform detachments use one of these methods. The BodyDetach class uses the basic `detach()` method to detach the subject MassBody or DynBody object from its immediate parent. The class BodyDetachSpecific uses the form of `detach()` that specifies the parent object. The body that is actually detached is the child of the specified parent that lies along the chain of mass bodies that link the subject body to the parent body in terms of parent-child relationships.

18.3.4 MassBody versus DynBody Attachment and Detachment

Properly attaching and detaching DynBody objects is a considerably more involved process that the corresponding tasks for MassBody objects. DynBody objects have momentum that must be conserved; energy must also be conserved on detachment. This is not a concern for simple MassBody objects. Fortunately for this model. all that detail is hidden in the guts of the `attach_to/attach_child()` and `detach()` methods. This model does not need to concern itself with the different behaviors; it merely needs to call the appropriate `attach_to/attach_child()` or `detach()` method.

Chapter 19

User Guide

The following assumes the scheme outlined in section 4.2.5.

19.1 Analysis

The Trick input file code below illustrates the use of the MassBody Attach/Detach Sub-Model. The objects involved in this example are

Vehicle A A DynBody object with two points of interest labeled as ”point_C” and ”point_D”.

Vehicle B A DynBody object with one point of interest labeled as ”point_C”.

Vehicle C A DynBody object with two points of interest labeled as ”point_B” and ”point_A”

Vehicle D A DynBody object with two points of interest labeled as ”point_A” and ”point_E”.

Mass E This is a simple MassBody object with one point of interest labeled as ”point_D”.

This example does not depict the mass and state initializations. The mass initializations will endow the objects with mass properties and create the points of interest identified above. The state initializations will provide the independent vehicles with state.

The events in this example are

- At initialization time, vehicle C is to be attached to vehicle B such that point B on vehicle C is connected to point C on vehicle B.
- Also at initialization time, mass E is to be attached to vehicle D such that point D on mass E is connected to point E on vehicle D.
- At t=10, vehicle C is to be attached to vehicle A such that point A on vehicle C is connected to point C on vehicle A. Note that because vehicle C is already attached to vehicle B, it is vehicle B that is attach to vehicle A rather than the C-to-A attachment that the user requested.

- At t=20, vehicle D is to be attached to vehicle A such that point A on vehicle D is connected to point D on vehicle A.
- At t=30, vehicle C is to detach from vehicle A. Note that the real attachment is B attached to A, and it is vehicle B that detaches from vehicle A rather than vehicle C.
- At t=40, vehicle D is to detach from vehicle A.

```
BodyAttachAligned * C_to_B = new BodyAttachAligned[1];
C_to_B->action_name        = "C_to_B";
C_to_B->set_subject_body(vehicle_C.dyn_body.mass);
C_to_B->subject_point_name = "point_B";
C_to_B->set_parent_body(vehicle_B.dyn_body.mass);
C_to_B->parent_point_name  = "point_C";
dynamics.body_action_ptr   = C_to_B;
call dynamics.dynamics.manager.add_body_action (dynamics.body_action_ptr);

BodyAttachAligned * E_to_D = new BodyAttachAligned[1];
E_to_D->action_name        = "E_to_D";
E_to_D->set_subject_body(vehicle_D.mass_E);
E_to_D->subject_point_name = "point_D";
E_to_D->set_parent_body(vehicle_D.dyn_body);
E_to_D->parent_point_name  = "point_E";
dynamics.body_action_ptr   = E_to_D;
call dynamics.dynamics.manager.add_body_action (dynamics.body_action_ptr);

BodyAttachAligned * D_to_A = new BodyAttachAligned[1];
D_to_A->action_name        = "D_to_A";
D_to_A->set_subject_body(vehicle_D.dyn_body);
D_to_A->subject_point_name = "point_A";
D_to_A->set_parent_body(vehicle_A.dyn_body);
D_to_A->parent_point_name  = "point_D";
D_to_A->active             = false;
dynamics.body_action_ptr   = D_to_A;
call dynamics.dynamics.manager.add_body_action (dynamics.body_action_ptr);

BodyDetachSpecific * C_from_A = new BodyDetachSpecific[1];
C_from_A->action_name      = "D_from_A";
C_from_A->set_subject_body(vehicle_C.dyn_body.mass);
C_from_A->set_detach_from_body(vehicle_A.dyn_body.mass);
dynamics.body_action_ptr   = C_from_A;
call dynamics.dynamics.manager.add_body_action (dynamics.body_action_ptr);


// Mass and state initializations not shown.
```

```
stop = 10;
BodyAttachAligned * C_to_A = new BodyAttachAligned[1];
C_to_A->action_name        = "C_to_A";
C_to_A->set_subject_body(vehicle_C.dyn_body);
C_to_A->subject_point_name = "point_A";
// OK to use MassBody instance within DynBody (checked via MassBody::dyn_owner)
C_to_A->set_parent_body(vehicle_A.dyn_body.mass);
C_to_A->parent_point_name  = "point_C";
C_to_A->active             = true;
dynamics.body_action_ptr   = C_to_A;
call dynamics.dynamics.manager.add_body_action (dynamics.body_action_ptr);

stop = 20;
D_to_A->active           = true;

stop = 30;
C_from_A->active         = true;

stop = 40;
BodyDetach * D_from_A = new BodyDetach[1];
D_from_A->action_name      = "D_from_A";
D_from_A->set_subject_body(vehicle_D.dyn_body);
D_from_A->active           = true;
dynamics.body_action_ptr   = D_from_A;
call dynamics.dynamics.manager.add_body_action (dynamics.body_action_ptr);

stop = 50;
```

19.2 Integration

The simulation integrator must make the MassBody Attach/Detach Sub-Model classes needed simulation user available for use. Unlike other examples in this document, the above only requires the integrator to declare a pool of objects.

```
sim_object {
   BodyAttachAligned * body_attach;
   BodyDetach * detach;
   BodyDetachSpecific * detach_specific;
} body_actions;
```

Because this example involves asynchronous operations, the simulation integrator also needs to ensure that the Dynamics Manager `perform_actions()` method is called as a scheduled job.

19.3 Extension

The DynBody and MassBody classes provides multiple ways to attach objects. Each MassBody Attach/Detach Sub-Model subclass provides only one of those techniques. One possible extension to the sub-model is to create a class that derives from BodyAttach that uses some other attachment mechanism.

Chapter 20

Inspection, Verification, and Validation

20.1 Inspection

Inspection BodyAction_7: Design Inspection

Table 20.1 portrays the class hierarchy for this sub-model and identifies which classes provide implementations of the `initialize()`, `is_ready()`, and `apply()` methods. From that table, all MassBody Attach/Detach Sub-Model classes derive from BodyAction, some directly, others indirectly. Each implementation of the specified methods invokes the overridden parent class implementation. The two `is_ready()` methods properly incorporate the result from the invocation of the overridden `is_ready()` into the readiness assessment.

By inspection, the MassBody Attach/Detach Sub-Model satisfies requirements BodyAction_16 and BodyAction_17.

Table 20.1: Mass Body Attach/Detach Sub-Model Virtual Methods

Class	**Parent Class**	**Class Defines ···**		
Name [a]	**Name** [a]	**initialize()**	**is_ready()**	**apply()**
··· Attach	BodyAction	✓		✓
··· AttachAligned	··· Attach	✓		✓
··· AttachMatrix	··· Attach			✓
··· Detach	BodyAction	✓	✓	✓
··· DetachSpecific	BodyAction	✓	✓	✓
··· Reattach	BodyAction			✓

[a] To reduce the table width, the leading MassBody on the class names is replaced with center dots. For example, the class BodyAttachAligned is abbreviated as ··· AttachAligned.

20.2 Validation

This section describes various tests conducted to verify and validate that the MassBody Attach/Detach Sub-Model satisfies the requirements levied against it. All verification and validation test source code, simulations and procedures for this sub-model are in the `SIM_verif_attach_mass`, or `SIM_ref_attach`, subdirectory of the JEOD directory `models/dynamics/body_action/verif`.

Test BodyAction_1: MassBody Attach/Detach

Background This test exercises the capabilities for this sub-model and for the MassBodyInit Class Sub-Model. The MassBodyInit class initializes the mass properties and mass points of a MassBody object. The classes defined in this sub-model provide the ability to attach and detach MassBody objects to/from one another.

Test description This test defines 21 run directories. Test RUN_05 tests mass properties initialization only; it does not exercise the requirements of this particular sub-model. Tests RUN_01 to RUN_04 and RUN_06 to RUN_11 use the matrix/offset attachment mechanism while RUN_101 to RUN_104 and RUN_106 to RUN_111 invoke identical behavior using the point-to-point attachment mechanism.

Test results These tests are described in detail in the *Mass Body Model* [14]. Per the criteria in that document, all test cases pass.

Applicable requirements This series of tests collectively exercises the capabilities of this sub-model and the MassBodyInit Class Sub-Model. These tests demonstrate requirements BodyAction_19 to BodyAction_20 and BodyAction_21 to BodyAction_24.

Test BodyAction_2: Kinematic Attach

Background This test exercises the capabilities for this sub-model's kinematic RefFrame attach capability. For the RefFrame attachments, we must demonstrate that the resulting attachment zeroes any previous velocities w.r.t. the parent frame and that mass tree is not impacted by this type of attachment.

Test description Test RUN_ref_attach_pt2pt attaches a Dynbody to a planet reference frame using the point-to-point attachment. Test RUN_ref_attach_matrix attaches a DynBody to a planet reference frame using the matrix/offset attachment mechanism.

Test results All test cases pass.

Applicable requirements This series of tests collectively exercises the capabilities of this sub-model. These tests demonstrate requirements BodyAction_25

20.3 Requirements Traceability

Table 20.2 summarizes the inspections and tests that demonstrate the satisfaction of the requirements levied on the MassBody Attach/Detach Sub-Model.

Table 20.2: BodyAttach_Detach Requirements Traceability

Requirement	**Artifact**
BodyAction_21 Attach framework	Test BodyAction_1 Attach/detach
BodyAction_22 Attach MassBody/DynBody objects	Test BodyAction_1 Attach/detach
BodyAction_23 Detach from parent	Test BodyAction_1 Attach/detach
BodyAction_24 Specific detach	Test BodyAction_1 Attach/detach
BodyAction_25 Specific detach	Test BodyAction_1 Attach/detach

Part V

DynBody State Initialization Sub-Model

Chapter 21

Introduction

21.1 Purpose and Objectives of the DynBody State Initialization Sub-Model

The DynBody State Initialization Sub-Model is responsible for initializing the state (position, velocity, attitude, and angular velocity) of a DynBody object.

21.2 Part Organization

This part of the Body Action Model document is organized along the lines described in section 1.4. It comprises the following chapters in order:

Introduction - This introduction describes the objective and purpose of the DynBody State Initialization Sub-Model.

Product Requirements - The DynBody State Initialization Sub-Model Product Requirements chapter describes the requirements on the sub-model and the requirements levied on user-defined classes that derive from this sub-model.

Product Specification - The DynBody State Initialization Sub-Model Product Specification chapter describes the underlying theory, architecture, and design of this sub-model.

User Guide - Describes the use of this sub-model. The DynBody State Initialization Sub-Model User Guide chapter the following sections:

- Analysts or users of simulations (Analysis).
- Integrators or developers of simulations (Integration).
- Model Extenders (Extension).

Inspection, Verification, and Validation - The DynBody State Initialization Sub-Model IV&V chapter describes the techniques used to ascertain that this sub-model satisfies the requirements levied upon it and the summarizes the results of the sub-model verification and validation tests.

Chapter 22

Product Requirements

Requirement BodyAction_26: Translational State Specification in Terms of a Known Frame

Requirement:
This model shall provide the ability to initialize the position and/or velocity vectors of a subject DynBody object given the subject body's position and/or velocity vectors with respect to a reference frame whose state with respect to the vehicle's integration frame is known.

Rationale:
This requirement derives from requirement BodyAction_8.

Verification:
Inspection, Test

Requirement BodyAction_27: Rotational State Specification in Terms of a Known Frame

Requirement:
This model shall provide the ability to initialize the attitude and/or attitude rate of a subject DynBody object given the object's attitude and/or attitude rate with respect to a reference frame whose state with respect to the vehicle's integration frame is known.

Rationale:
This requirement derives from requirement BodyAction_8.

Verification:
Inspection, Test

Requirement BodyAction_28: Orbital Elements

Requirement:
This model shall provide the ability to initialize the position and velocity of a subject DynBody object given a set of osculating orbital elements as represented in some non-rotating reference

frame about some planet. Table 22.1 lists the orbital elements sets that pertain to this requirement.

Rationale:

The multiplicity of options enables the use of known vehicle translational states as provided by external sources.

Verification:

Inspection, Test

Table 22.1: Orbital Elements

	Orbital Elements [a]														
Subrequirement	**Inclination**	**Longitude of ascending node**	**Semi-major axis**	**Semi-latus rectum**	**Eccentricity**	**Altitude at periapsis**	**Altitude at apoapsis**	**Argument of periapsis**	**Argument of latitude**	**Time since periapsis passage**	**Mean anomaly**	**True anomaly**	**Orbital radius**	**Radial velocity**	**Index Number** [b]
28.1	√	√	√		√			√		√					1
28.2	√	√	√		√			√			√				2
28.3	√	√		√	√			√				√			3
28.4	√	√				√	√	√				√			4
28.5	√	√				√	√	√		√					5
28.6	√	√	√						√				√	√	6
28.7	√	√	√		√			√				√			10
28.8	√	√				√	√	√				√			11

[a] Six orbital elements are needed to uniquely specify the orbital state. Six orbital elements are checked in each line of the table. The Body Action Model shall provide the ability to transform the indicated sets of six orbital elements to position and velocity vectors.

[b] The options are indexed in this fashion to replicate the numbering scheme used in JEOD 1.4/1.5.

Requirement BodyAction_29: State Specification in Terms of an LVLH Frame

Requirement:

If the position and velocity of a reference DynBody object's body frame with respect to a subject DynBody object's integration frame are known, this model shall provide the ability to initialize aspects of the state of the subject body given elements of the subject body's state with respect to the reference body's rectilinear or curvilinear Local Vertical, Local Horizontal frame with respect to some planet.

Rationale:

This requirement derives from requirement BodyAction_8.

Verification:

Inspection, Test

Requirement BodyAction_30: State Specification in Terms of a North-East-Down Frame

Requirement:

30.1 Point-relative. This model shall provide the ability to initialize aspects of the state of a subject body given elements of the subject body's state with respect to a planet-based reference point.

30.2 Vehicle-relative. If the position and velocity of a reference DynBody object's body frame with respect to a subject DynBody object's integration frame are known, this model shall provide the ability to initialize aspects of the state of the subject body given elements of the subject body's state with respect to the reference body's North-East-Down frame with respect to some planet.

Rationale:

This requirement derives from requirement BodyAction_8.

Verification:

Inspection, Test

Requirement BodyAction_31: Readiness Detection

Requirement:

This model shall provide the ability to detect when a state initializer is ready to be executed (and when it is not ready) based on the data needed by the initialization object.

Rationale:

This requirement derives from requirement BodyAction_8.

One of the key goals of this model is to remove dependencies upon the order in which objects are declared in the S_define file or the order in which objects are added to the Dynamic Manager's list of BodyAction objects.

Verification:

Inspection, Test

Chapter 23

Product Specification

23.1 Conceptual Design

The DynBody State Initialization Sub-Model comprises several classes that provide various mechanisms for initializing the state (position, velocity, attitude, and angular velocity) of a DynBody object. All of the classes in this sub-model derive from a single base class. The base class for initializing the state of a DynBody object is the DynBodyInit class. The class inheritance for the sub-model is depicted below.

```
BodyAction
|
+-DynBodyInit
  |
  +-DynBodyInitRotState
  |
  +-DynBodyInitTransState
  | |
  | +-DynBodyInitOrbit
  |
  +-DynBodyInitWrtPlanet
    |
    +-DynBodyInitPlanetDerived
      |
      +-DynBodyInitLvlhState
      | |
      | +-DynBodyInitLvlhRotState
      | |
      | +-DynBodyInitLvlhTransState
      |
      +-DynBodyInitNedState
        |
        +-DynBodyInitNedRotState
```

```
|
+-DynBodyInitNedTransState
```

23.2 Mathematical Formulations

Most of the classes defined in the DynBody State Initialization Sub-Model rely on some other model to do all of the mathematical calculations. The one exception is the DynBodyInitOrbit class. This class uses the Orbital Elements Model to do most of the work in converting orbital elements to position and velocity vectors. The Orbital Elements Model accepts but one set of orbital elements. The DynBodyInitOrbit accepts seven sets. Translating the data in those seven sets to the set accepted by the Orbital Elements Model requires a bit of work[17]:

$$\begin{aligned}
e\cos E &= \frac{a-r}{a} && (23.1)\\
e\sin E &= \frac{r\dot{r}}{\sqrt{\mu a}} && (23.2)\\
e^2 &= (e\cos E)^2 + (e\cos E)^2 && (23.3)\\
k\cos\nu &= e\cos E - e^2 && (23.4)\\
k\sin\nu &= \sqrt{1-e^2}e\sin E && (23.5)\\
\tan\nu &= \frac{k\sin\nu}{k\cos\nu} && (23.6)\\
\omega &= u-\nu && (23.7)\\
a &= R_{\text{planet}} + (h_{\text{apo}} + h_{\text{peri}})/2 && (23.8)\\
e &= \frac{(h_{\text{apo}} - h_{\text{peri}}}{2a} && (23.9)\\
p &= a(1-e^2) && (23.10)\\
M &= \frac{T_{\text{peri}}}{a}\sqrt{\frac{\mu}{a}} && (23.11)
\end{aligned}$$

23.3 Detailed Design

23.3.1 Class DynBodyInit

The base class for initializing the state of a DynBody object is DynBodyInit.

A DynBody has a minimum of three distinct reference frames:

- The structural frame, with origin at a fixed point on the vehicle and reference frame axes defined with respect to the vehicle;
- The core body frame, with origin at the core center of mass and and reference frame axes a fixed rotation from the structural axes; and

- The composite body frame, with origin at the composite center of mass and and reference frame axes collinear with the core body frame axes.

Any given instance of a DynBodyInit object can specifically initialize elements of one of those frames.

A DynBodyInit object contains the position, velocity, attitude, and angular velocity of the subject reference frame with respect to some reference reference frame. These state data form the primary user inputs to the DynBodyInit object. (The DynBodyInitOrbit class ignores these inputs.)

The DynBodyInit class overrides the trio of virtual methods defined by the the class BodyAction. It adds one new virtual method to this trio, the `initializes_what()` method. This method specifies which of the elements of the subject body frame are to be initialized. The default implementation indicates that no elements of the subject body frame are to be initialized. All derived classes must override this default, either directly or by inheritance from some intermediate derived class.

23.3.2 Derived Classes

DynBodyInitRotState Derives from DynBodyInit.
Instances of this class initialize a vehicle's attitude and/or attitude rate.

DynBodyInitTransState Derives from DynBodyInit Instances of this class initialize a vehicle's position and/or velocity.

DynBodyInitOrbit Derives from DynBodyInitTransState.
Instances of this class initialize a vehicle's position and velocity by means of some orbit specification.

DynBodyInitPlanetDerived Derives from DynBodyInitWrtPlanet, which in turn derives from DynBodyInit.
These classes form the basis for subsequent derived classes that initialize state with respect to a planetary frame.

DynBodyInitLvlhState Derives from DynBodyInitPlanetDerived.
Instances of this class initialize state with respect to the rectilinear or curvilinear LVLH frame defined by some reference vehicle's orbit about a specified planet. Two derived classes from this class, DynBodyInitLvlhRotState and DynBodyInitLvlhTransState, limit the initialization to the rotational and translational states of the subject DynBody.

DynBodyInitNedState Derives from DynBodyInitPlanetDerived.
Instances of this class initialize state with respect to the North-East-Down frame defined by some reference vehicle or reference point. Two derived classes from this class, DynBodyInitNedRotState and DynBodyInitNedTransState, limit the initialization to the rotational and translational states of the subject DynBody.

Chapter 24

User Guide

The following assumes the scheme outlined in section 4.2.4.

24.1 Analysis

The following examples illustrate various uses of the sub-model to initialize one of the vehicles. The examples assume a simulation S_define file that declares simulation objects named ‘launchpad’, ‘target’, ‘chaser’, and ‘dynamics’. The launchpad, target, and chaser objects each contain a DynBody object name ‘dyn_body’ and a number of objects of a type derived from DynBodyInit. The dynamics object contains a DynManager object named ‘manager’ and a BodyAction pointer named ‘body_action_ptr’. See the Integration section below for a detailed description of the S_define file.

24.1.1 DynBodyInitOrbit

The DynBodyInitOrbit class initializes translational state using a variety of orbital elements sets. The following specifies the orbit in terms of semi-major axis, eccentricity, inclination, right ascension of ascending node, argument of periapsis, and the time since periapsis passage.

```
target.orb_init.set_subject_body(target.dyn_body);
target.orb_init.action_name = "target.orb_init";

target.orb_init.reference_ref_frame_name = "Earth.inertial";
target.orb_init.body_frame_id = "composite_body";

target.orb_init.set = DynBodyInitOrbit::SmaEccIncAscnodeArgperTimeperi;
target.orb_init.orbit_frame_name = "Earth.inertial";
target.orb_init.planet_name      = "Earth";

target.orb_init.arg_periapsis {d}       =  100.582445989;
target.orb_init.eccentricity            =    0.00129073350;
target.orb_init.inclination {d}         =   51.670450765;
target.orb_init.ascending_node {d}      =   49.708417385;
target.orb_init.semi_major_axis {km}    = 6732.90120152;
target.orb_init.time_periapsis {s}      = 4581.96167293;

dynamics.body_action_ptr = &target.orb_init;
call dynamics.dynamics.manager.add_body_action (dynamics.body_action_ptr);
```

24.1.2 DynBodyInitTransState

The DynBodyInitTransState class initializes translational state given the state relative to some known reference frame. The following initializes the chaser relative to the target using a point-to-point state specification and initializes the target given an Earth-centered, Earth-fixed state specification.

Note the order in which these two initializations are specified. The chaser cannot be initialized until the target is initialized. There is nothing wrong with adding initializers in an out-of-order sequence as is done in this example. The Dynamics Manager will sort things out.

```
chaser.trans_init.set_subject_body(chaser.dyn_body.mass);
chaser.trans_init.action_name = "chaser.trans_state_tpoint_cpoint";

chaser.trans_init.reference_ref_frame_name = "target.attach_point";
chaser.trans_init.body_frame_id = "attach_point";

chaser.trans_init.position[0] {M}   = 100.0, 0.0, 5.0;
chaser.trans_init.velocity[0] {M/s} = -1.0, 0.0, 0.0;

dynamics.body_action_ptr = &chaser.trans_init;
call dynamics.dynamics.manager.add_body_action (dynamics.body_action_ptr);


target.trans_init.set_subject_body(target.dyn_body);
target.trans_init.action_name = "target.trans_state_pfix_body";

target.trans_init.reference_ref_frame_name = "Earth.pfix";
target.trans_init.body_frame_id = "composite_body";

target.trans_init.position[0] {M}   =
    5406298.5700, -2074684.5600,  3426540.0300;
target.trans_init.velocity[0] {M/s} =
    -706.0655810,  5764.3071620,  4587.2461630;

dynamics.body_action_ptr = &target.trans_init;
call dynamics.dynamics.manager.add_body_action (dynamics.body_action_ptr);
```

24.1.3 DynBodyInitRotState

The DynBodyInitRotState class initializes rotational state given the state relative to some known reference frame. The following initializes the target attitude rate given an Earth-centered inertial state specification.

```
target.rot_init.set_subject_body(target.dyn_body.mass);
target.rot_init.action_name = "target.rate_state_inertial_body";

target.rot_init.reference_ref_frame_name = "Earth.inertial";
target.rot_init.body_frame_id = "composite_body";

target.rot_init.state_items = DynBodyInitRotState::Rate;
target.rot_init.ang_velocity[0] {d/s}  =
    0.002,
    0.006 - 0.06556131568278,
   -0.003;

dynamics.body_action_ptr = &target.rot_init;
call dynamics.dynamics.manager.add_body_action (dynamics.body_action_ptr);
```

24.1.4 DynBodyInitLvlhState

The DynBodyInitLvlhState class initializes state given the state relative to some rectilinear or curvilinear LVLH reference frame. The classes DynBodyInitLvlhTransState and DynBodyInitLvlhRotState derive from DynBodyInitLvlhState and only set translational and rotational aspects of the state; DynBodyInitLvlhState can set the full state.

The DynBodyInitTransState and DynBodyInitRotState classes cannot be used with these LVLH frames because derived frames are not known at initialization time. If supplied, the DynBodyInitLvlhState class uses an existing user-specified LVLH reference frame. Otherwise, it constructs a temporary one to be used only for initialization.

The following initializes the chaser's full state with respect to the target LVLH frame.

```
target.lvlh.subject_name    = "target.composite_body"
target.lvlh.planet_name     = "Earth"

# Choose Rectilinear or CircularCurvilinear
chaser.lvlh_init.lvlh_type = trick.LvlhType.Rectilinear;
# chaser.lvlh_init.lvlh_type = trick.LvlhType.CircularCurvilinear;
chaser.lvlh.subject_name    = "chaser.composite_body"
chaser.lvlh.planet_name     = "Earth"

chaser.lvlh_init.set_subject_body(chaser.dyn_body.mass);
chaser.lvlh_init.action_name = "chaser_lvlh_init"
chaser.lvlh_init.planet_name = "Earth"
chaser.lvlh_init.set_lvlh_frame_object (target.lvlh)
chaser.lvlh_init.position  = trick.attach_units( "m",[ -100.0, 25.0, 7.5])
chaser.lvlh_init.velocity  = trick.attach_units( "m/s",[ 0.0, 0.0, 1.0])

chaser.lvlh_init.orientation.data_source    = trick.Orientation.InputEulerRotation
chaser.lvlh_init.orientation.euler_sequence = trick.Orientation.Yaw_Pitch_Roll
chaser.lvlh_init.orientation.euler_angles   = trick.attach_units( "degree",[ 0.0, 0.0, 0.0])
chaser.lvlh_init.ang_velocity               = trick.attach_units( "degree/s",[ 0.0, 0.0, 4.5])

dynamics.dyn_manager.add_body_action (chaser.lvlh_init);
```

24.1.5 DynBodyInitNedState

The DynBodyInitNedState class initializes state given the state relative to North-East-Down reference frame. The classes DynBodyInitNedTransState and DynBodyInitNedRotState derive from DynBodyInitNedState and only set translational and rotational aspects of the state; DynBodyInitNedState can set the full state.

Like the LVLH frames described above, North-East-Down frames are derived frames. The DynBodyInitNedState class constructs the North-East-Down reference frame so that it can be used for initialization.

The following initializes the full state of the launchpad with respect to a specified point on the Earth and initializes the chaser's translational state relative to the launchpad.

```
launchpad.ned_init.set_subject_body(launchpad.dyn_body);
launchpad.ned_init.action_name = "launchpad.ned_state_ref_struct";

launchpad.ned_init.reference_ref_frame_name = "Earth.inertial";
launchpad.ned_init.body_frame_id = "structure";
launchpad.ned_init.planet_name = "Earth";

launchpad.ned_init.set_use_alt_pfix(False);
launchpad.ned_init.ref_point.altitude {M}  =   3.0;
launchpad.ned_init.ref_point.latitude {d}  =  28.6082;
launchpad.ned_init.ref_point.longitude {d} = -80.6040;
launchpad.ned_init.altlatlong_type = NorthEastDown::elliptical;

launchpad.ned_init.set_items = RefFrameItems::Pos_Vel_Att_Rate;
launchpad.ned_init.position[0] {M}   =  0.0,  0.0, 10.0;
launchpad.ned_init.velocity[0] {M/s} =  0.0,  0.0,  0.0;
launchpad.ned_init.orientation.data_source = Orientation::InputEulerRotation;
launchpad.ned_init.orientation.euler_sequence = Pitch_Yaw_Roll;
launchpad.ned_init.orientation.euler_angles[0] {d} = 0.0, 0.0, 0.0;
launchpad.ned_init.ang_velocity[0] {d/s} = 0.0, 0.0, 0.0;

dynamics.body_action_ptr = &launchpad.ned_init;
```

```
call dynamics.dynamics.manager.add_body_action (dynamics.body_action_ptr);


chaser.ned_trans_init.set_subject_body(chaser.dyn_body);
chaser.ned_trans_init.action_name = "chaser.trans_ned_launch_pad_struct";

chaser.ned_trans_init.reference_ref_frame_name = "Earth.inertial";
chaser.ned_trans_init.body_frame_id = "structure";
chaser.ned_trans_init.planet_name = "Earth";
chaser.ned_trans_init.ref_body_name = "launchpad";
chaser.ned_trans_init.altlatlong_type = NorthEastDown::elliptical;

chaser.ned_trans_init.position[0] {M}   = 0.0, 0.0,  -40.0;
chaser.ned_trans_init.velocity[0] {M/s} = 0.0, 0.0, -100.0;

dynamics.body_action_ptr = &chaser.ned_trans_init;
call dynamics.dynamics.manager.add_body_action (dynamics.body_action_ptr);
```

Internally, a planet-centered, planet-fixed frame is used in the initialization process. If the planet has an alternate planet-fixed frame, the user may choose to use it for these internal calculations. In that case, the user may `set_use_alt_pfix()` to true, as is shown in the example below.

```
vehicle.ned_init.set_use_alt_pfix(True);
```

24.2 Integration

The main concern of the simulation integrator with respect to this model is making enough initializers available to the simulation users so that the users can easily initialize the vehicles in the simulation. Segments of the S_define file used in the above examples are shown below. The launchpad and chaser simulation objects are not shown. The launchpad is a very pared-down version of the target (it only needs a DynBodyInitNedState object) while the chaser has the same set of initializers as does the target. This is most likely overkill. Most simulations will not need the plethora of initializers used here.

```
/**
 * sim object target
 */
sim_object {

   //
   // Data structures
   //
   dynamics/dyn_body:         DynBody dyn_body;
   dynamics/body_action:      MassBodyInit mass_init;
   dynamics/body_action:      DynBodyInitOrbit orb_init;
   dynamics/body_action:      DynBodyInitTransState trans_init;
   dynamics/body_action:      DynBodyInitRotState rot_init;
   dynamics/body_action:      DynBodyInitLvlhState lvlh_init;
   dynamics/body_action:      DynBodyInitLvlhTransState lvlh_trans_init;
   dynamics/body_action:      DynBodyInitLvlhRotState lvlh_rot_init;
   dynamics/body_action:      DynBodyInitNedState ned_init;
   dynamics/body_action:      DynBodyInitNedTransState ned_trans_init;
   dynamics/body_action:      DynBodyInitNedRotState ned_rot_init;

   ...

} target;


/**
 * sim object chaser -- not shown; it is a replicate of the target
 */

/**
 * sim object dynamics
 */
sim_object {

   //
   // Data structures
   //
   dynamics/dyn_manager: DynManager          manager;
   dynamics/dyn_manager: DynManagerInit      manager_init;
   dynamics/body_action: BodyAction        * body_action_ptr;
   sim_services/include: INTEGRATOR          integ;
   utils/message:        TrickMessageHandler msg_handler;

   //
   // Jobs registered for input file activation.
   //
   (0.0, environment) dynamics/dyn_manager:
   dynamics.manager.add_body_action (
      Inout BodyAction * body_action = dynamics.body_action_ptr);

   //
   // Initialization jobs
   //
   P_MNGR (initialization) dynamics/dyn_manager:
   dynamics.manager.initialize_model (
      Inout INTEGRATOR *     integ = &dynamics.integ,
      In    DynManagerInit & init  =  dynamics.manager_init);

   P_BODY (initialization) dynamics/dyn_manager:
   dynamics.manager.initialize_simulation ();

   ...

} dynamics;
```

24.3 Extension

Extensibility was one of the key drivers in the development of this sub-model.

Suppose you as a simulation integrator find that your users insist on having the ability to initialize state with respect to the inertial frame that is instantaneously colinear and co-moving with some reference vehicle reference frame. JEOD does not supply this capability, so how to supply it?

The answer is to create a class that derives from DynBodyInit. This new class will need data member(s) that specify the reference vehicle, the target reference frame on that vehicle, and possibly the states the user wants to initialize. The new class will also need to override four virtual member functions.

The `initialize()` method should check for errors in the specification of the reference vehicle and the reference reference frame and then should call the `DynBodyInit::initialize()` method.

The default `initializes_what()` method says that the initializer initializes nothing. You will certainly want to override that. Exactly what state elements are to be initialized by this new class is between you and your users. The overridden `initializes_what()` method needs to reflect this decision.

This new class needs to wait until the reference position and velocity are known. The default `is_ready()` method does not know this. You will need to override the `is_ready()` method to make it wait. To conform with the model requirements, the overridden method needs to query `DynBodyInit::is_ready()` before it returns `true`.

Finally, you will need to override the `apply()` method. One approach is to

1. Create a new reference frame that represents this collinear and co-moving inertial frame. A simply way to do this is to declare a RefFrame object as a local variable. Note that a reference frame initializes to having a null translation and rotational with respect to its parent.
2. Give this frame a name such as "vehicle.inertial", with the vehicle name substituted for vehicle. Various JEOD methods do not like reference frames that don't have a name.
3. (Temporarily) attach this new reference frame to the reference frame tree. In this case, attaching the frame to the reference body's integration frame will make that collinear and co-moving requirement easy to satisfy.
4. Copy the position and velocity from the reference reference frame to this new frame. Voila! This is now an inertial frame that is collinear and co-moving with the reference vehicle.

5. Point the inherited `reference_ref_frame` data member to this newly created frame.

6. Call the inherited `apply_user_inputs()` method to interpret the user inputs in terms of this frame.

7. Call the overridden `DynBodyInit::apply()` method to initialize the subject vehicle state.

8. Reset the inherited `reference_ref_frame` data member to NULL. The item to which it points is about to go out of scope.

9. Return. The reference frame created by declaring a RefFrame variable will go out of scope and be destroyed. The RefFrame destructor automatically removes the frame from the reference frame tree.

Chapter 25

Inspection, Verification, and Validation

25.1 Inspection

Inspection BodyAction_8: Design Inspection

Table 25.1 portrays the class hierarchy for this sub-model and identifies which classes provide implementations of the `initialize()`, `is_ready()`, and `apply()` methods. As shown in that table,

- The DynBodyInit class derives directly from the BodyAction class.
- All other DynBody State Initialization Sub-Model classes derive from DynBodyInit, some directly, others indirectly.
- For each provided implementation of the key virtual methods an implementation of that method also exists in the parent class.

Code inspection reveals that each implementation of the key virtual methods invokes the overridden parent class implementation and that all of the `is_ready()` methods properly incorporate the result from the invocation of the overridden `is_ready()` into the readiness assessment.

By inspection, the DynBody State Initialization Sub-Model satisfies requirements BodyAction_16 and BodyAction_17.

25.2 Validation

This section describes various tests conducted to verify and validate that the DynBody State Initialization Sub-Model satisfies the requirements levied against it. All verification and validation

Table 25.1: DynBody State Initialization Sub-Model Virtual Methods

Class Name [a]	**Parent Class Name** [a]	**Class Defines ···**		
		initialize()	**is_ready()**	**apply()**
DynBodyInit	BodyAction	✓	✓	✓
···WrtPlanet	DynBodyInit	✓	✓	✓
···PlanetDerived	···WrtPlanet	✓	✓	✓
···LvlhState	···PlanetDerived	✓		✓
···LvlhRotState	···LvlhState	✓		
···LvlhTransState	···LvlhState	✓		
···NedState	···PlanetDerived	✓		✓
···NedRotState	···NedState	✓		
···NedTransState	···NedState	✓		
···TransState	DynBodyInit	✓	✓	✓
···Orbit	···TransState	✓		✓
···RotState	DynBodyInit	✓	✓	✓

[a]To reduce the table width, the leading DynBodyUnit on classes that derive from DynBodyInit is replaced with center dots. For example, the class DynBodyInitWrtPlanet is abbreviated as ···WrtPlanet.

test source code, simulations and procedures for this sub-model are in either the `SIM_orbinit`, or `SIM_lvlh_init`, subdirectory of the JEOD directory `models/dynamics/body_action/verif`.

Each of the simulation directories comprise

- A Trick S_define file. This file defines two dynamic bodies, each having a variety of state initializers; a dynamics manager, which is needed to run the initializers, and a minimal set of other objects needed by the dynamics manager.
- Run directories in each of the `SET_test` and `SET_test_val` directories, The run directories within the `SET_test` directory contain simulation input files which the correspondingly-named run directories in the `SET_test_val` directory contain log files used for regression testing.
- Several `Modified_data` files and one `Log_data` file, which are used by the run input files to configure a simulation run.
- In `SIM_orbinit`, one test script, `scripts/run_tests.pl`. The output of the shell script is organized to correspond with the validation tests described in this section. Numbers that pass the various test criteria are marked in green while failing values are marked in red.

Test BodyAction_3: State Initializations

Background The purpose of the DynBody State Initialization Sub-Model is to initialize reference frame states contained in a DynBody object. A reference frame state comprises two main parts, the translational and rotational aspects of the frame. A reference frame's translational state describes the position and velocity of the origin of the frame with respect to the parent frame; the rotational state describes the orientation and angular velocity of the frame's axes with respect to the parent frame.

The DynBodyInit subclasses set some but not necessarily all aspects of some reference frame associated with a DynBody object. The set reference frame might or might not be the DynBody object's integrated frame, and the reference reference frame for the DynBodyInit object might or might not be the DynBody object's integration frame.

Test description These tests verify the errors that result from initializing state via the various DynBodyInit subclasses fall within bounds.

Test cases Table 25.2 lists the simulation run directories associated with this test and the results of the test.

Success criteria These tests pertain to a vehicle initialized in low Earth orbit using either specifications relative to the Earth or to another spacecraft in low Earth orbit. The errors should be very small, and thus the success criteria are quite stringent for this test. Each component of the position vector must have an error no larger than 1 millimeter and each component of the velocity vector must have an error no larger than 0.01 millimeters/second.

Test results All tests pass.

Applicable requirements This test collectively demonstrates the satisfaction of requirements BodyAction_26 to BodyAction_31.

Table 25.2: Cartesian Position/Velocity Test Cases

Run Directory	**Class Name** [a]	**Source Frame**	**Subj. Frame**	**Subj. Vehicle**	**Pos. Err.** (m)	**Vel. Err.** (m/s)	**Status**
RUN_0400	Trans	Inertial	Body	ISS	0.0	0.0	Passed
RUN_0401	Trans	Inertial	Body	STS 114	0.0	0.0	Passed
RUN_0410	Trans	Pfix	Body	ISS	3.0×10^{-9}	1.9×10^{-12}	Passed
RUN_0411	Trans	Pfix	Body	STS 114	3.0×10^{-9}	1.8×10^{-12}	Passed
RUN_0441	Trans	Tgt. Body	Body	STS 114	1.3×10^{-10}	3.5×10^{-13}	Passed
RUN_0571	LvlhTrans	Tgt. LVLH	Body	STS 114	1.3×10^{-10}	3.3×10^{-13}	Passed
RUN_0681	NedTrans	Tgt. NED	Body	STS 114	4.1×10^{-9}	2.2×10^{-12}	Passed
RUN_3771	Lvlh	Tgt. LVLH	Body	STS 114	1.3×10^{-10}	3.3×10^{-13}	Passed
RUN_3822	Ned	Ref. NED	Struct	Pad 39A	2.1×10^{-9}	7.1×10^{-14}	Passed
RUN_4451	Trans	Tgt. Struct	Struct	STS 114	3.3×10^{-10}	2.8×10^{-13}	Passed
RUN_4681	NedTrans	Tgt. NED	Struct	STS 114	1.3×10^{-9}	1.3×10^{-13}	Passed
RUN_5461	Trans	Tgt. Point	Point	STS 114	7.7×10^{-10}	1.0×10^{-12}	Passed

[a]The class names in the table are listed in abbreviated form. The true class name is the listed class name prefixed with "DynBodyInit" and suffixed with "State". For example, Trans is short for DynBodyInitTransState.

Table 25.3: Orbital Elements Test Cases

Run Directory	**Source Frame**	**Subject Vehicle**	**Set #**	**Pos. Err** (m)	**Vel. Err.** (m/s)	**Status**
RUN_0001	Inertial	ISS	1	4.9×10^{-5}	6.6×10^{-8}	Passed
RUN_0002	Inertial	ISS	2	2.7×10^{-5}	5.4×10^{-8}	Passed
RUN_0003	Inertial	ISS	3	1.9×10^{-5}	5.7×10^{-8}	Passed
RUN_0004	Inertial	ISS	4	1.7×10^{-5}	5.3×10^{-8}	Passed
RUN_0005	Inertial	ISS	5	2.9×10^{-5}	5.9×10^{-8}	Passed
RUN_0006	Inertial	ISS	6	1.7×10^{-5}	5.3×10^{-8}	Passed
RUN_0010	Inertial	ISS	10	1.7×10^{-5}	5.7×10^{-8}	Passed
RUN_0011	Inertial	ISS	11	1.7×10^{-5}	5.3×10^{-8}	Passed
RUN_0101	Inertial	STS 114	1	1.7×10^{-5}	3.2×10^{-8}	Passed
RUN_0102	Inertial	STS 114	2	4.2×10^{-5}	4.4×10^{-8}	Passed
RUN_0103	Inertial	STS 114	3	8.3×10^{-5}	1.1×10^{-7}	Passed
RUN_0104	Inertial	STS 114	4	8.3×10^{-5}	9.6×10^{-8}	Passed
RUN_0105	Inertial	STS 114	5	2.0×10^{-5}	3.4×10^{-8}	Passed
RUN_0106	Inertial	STS 114	6	3.9×10^{-5}	5.0×10^{-8}	Passed
RUN_0110	Inertial	STS 114	10	8.4×10^{-5}	1.1×10^{-7}	Passed
RUN_0111	Inertial	STS 114	11	8.3×10^{-5}	9.6×10^{-8}	Passed
RUN_0201	Pfix	ISS	1	8.0×10^{-5}	6.7×10^{-8}	Passed
RUN_0202	Pfix	ISS	2	7.1×10^{-5}	3.8×10^{-8}	Passed
RUN_0203	Pfix	ISS	3	6.4×10^{-5}	3.7×10^{-8}	Passed
RUN_0204	Pfix	ISS	4	6.3×10^{-5}	6.4×10^{-8}	Passed
RUN_0205	Pfix	ISS	5	7.0×10^{-5}	5.1×10^{-9}	Passed
RUN_0206	Pfix	ISS	6	6.3×10^{-5}	6.2×10^{-8}	Passed
RUN_0210	Pfix	ISS	10	6.4×10^{-5}	3.7×10^{-8}	Passed
RUN_0211	Pfix	ISS	11	6.3×10^{-5}	6.4×10^{-8}	Passed
RUN_0301	Pfix	STS 114	1	4.2×10^{-5}	6.2×10^{-8}	Passed
RUN_0302	Pfix	STS 114	2	8.9×10^{-5}	8.9×10^{-8}	Passed
RUN_0303	Pfix	STS 114	3	2.3×10^{-5}	6.6×10^{-8}	Passed
RUN_0304	Pfix	STS 114	4	1.9×10^{-5}	5.7×10^{-8}	Passed
RUN_0305	Pfix	STS 114	5	2.4×10^{-5}	6.0×10^{-8}	Passed
RUN_0306	Pfix	STS 114	6	1.9×10^{-5}	5.6×10^{-8}	Passed
RUN_0310	Pfix	STS 114	10	2.5×10^{-5}	6.7×10^{-8}	Passed
RUN_0311	Pfix	STS 114	11	1.9×10^{-5}	5.7×10^{-8}	Passed

Table 25.4: Attitude Test Cases

Run Directory	**Class Name** [a]	**Source Frame**	**Subj. Frame**	**Subj. Vehicle**	**Att. Err.** (d)	**Status**
RUN_1230	LvlhRot	Tgt. LVLH	Body	ISS	0.0	Passed
RUN_2100	Rot	Inertial	Body	ISS	3.0×10^{-14}	Passed
RUN_3771	Lvlh	Tgt. LVLH	Body	STS 114	2.1×10^{-14}	Passed
RUN_3822	Ned	Ref. NED	Struct	Pad 39A	1.8×10^{-14}	Passed
RUN_4451	Rot	Tgt. Struct	Struct	STS 114	2.7×10^{-14}	Passed
RUN_4681	NedRot	Tgt. NED	Struct	STS 114	1.4×10^{-14}	Passed
RUN_5461	Rot	Tgt. Point	Point	STS 114	1.3×10^{-14}	Passed

[a]See table 25.2.

Table 25.5: Body Rate Test Cases

Run Directory	**Class Name** [a]	**Source Frame**	**Subj. Frame**	**Subj. Vehicle**	**Rate Err.** (d/s)	**Status**
RUN_1230	LvlhRot	Tgt. LVLH	Body	ISS	7.8×10^{-19}	Passed
RUN_2100	Rot	Inertial	Body	ISS	0.0	Passed
RUN_3771	Lvlh	Tgt. LVLH	Body	STS 114	1.2×10^{-17}	Passed
RUN_3822	Ned	Ref. NED	Struct	Pad 39A	9.8×10^{-19}	Passed
RUN_4451	Rot	Tgt. Struct	Struct	STS 114	2.2×10^{-17}	Passed
RUN_4681	NedRot	Tgt. NED	Struct	STS 114	4.5×10^{-19}	Passed
RUN_5461	LvlhRot	Tgt. LVLH	Body	STS 114	0.0	Passed

[a]See table 25.2.

25.3 Requirements Traceability

Table 25.6 summarizes the inspections and tests that demonstrate the satisfaction of the requirements levied on the DynBody State Initialization Sub-Model.

Table 25.6: DynBodyInit Requirements Traceability

Requirement	Artifact
BodyAction_26 Translational state	Test BodyAction_3 State initializations
BodyAction_27 Rotational state	Test BodyAction_3 State initializations
BodyAction_28 Orbital elements	Test BodyAction_3 State initializations
BodyAction_29 LVLH frame	Test BodyAction_3 State initializations
BodyAction_30 NED frame	Test BodyAction_3 State initializations
BodyAction_31 Readiness detection	Test BodyAction_3 State initializations

Part VI

DynBodyFrameSwitch Class Sub-Model

Chapter 26

Introduction

26.1 Purpose and Objectives of the DynBodyFrameSwitch Class

The DynBodyFrameSwitch Class is responsible for smoothly switching the frame in which a DynBody object's state is represented and propagated.

26.2 Part Organization

This part of the Body Action Model document is organized along the lines described in section 1.4. It comprises the following chapters in order:

Introduction - This introduction describes the objective and purpose of the DynBodyFrameSwitch Class.

Product Requirements - The DynBodyFrameSwitch Class Product Requirements chapter describes the requirements on the DynBodyFrameSwitch class.

Product Specification - The DynBodyFrameSwitch Class Product Specification chapter describes the underlying theory, architecture, and design of this class.

User Guide - Describes the use of this class.

Inspection, Verification, and Validation - The DynBodyFrameSwitch Class IV&V chapter describes the techniques used to ascertain that this class satisfies the requirements levied upon it and the summarizes the results of the sub-model verification and validation tests.

Chapter 27

Product Requirements

Requirement BodyAction_32: Frame Switch

Requirement:
This class shall provide the ability to change the frame in which a DynBody object's state is represented and propagated.

Rationale:
This is the primary purpose of this class. This requirement derives directly from requirement BodyAction_9.

Verification:
Inspection, Test

Requirement BodyAction_33: Frame Switch Continuity

Requirement:
To within numerical precision, this class shall maintain a continuous state for the vehicle with respect to the closest common ancestor of to the two integration frames when the frame switch is performed.

Rationale:
This class defers the mathematical details of switching integration frames to the DynBody class. This requirement is also levied on the Dynamics Body model.

Verification:
Inspection, Test

Requirement BodyAction_34: Frame Switch on Entering the Sphere of Influence

Requirement:
This class shall provide the ability to trigger the change from one integration frame to another when the vehicle enters the sphere of influence of the new integration frame.

Rationale:

Refinement of requirement BodyAction_32.

Verification:

Inspection, Test

Requirement BodyAction_35: Frame Switch on Exiting the Sphere of Influence

Requirement:

This class shall provide the ability to trigger the change from one integration frame to another when the vehicle exits the sphere of influence of the current integration frame.

Rationale:

Refinement of requirement BodyAction_32.

Verification:

Inspection, Test

Chapter 28

Product Specification

28.1 Conceptual Design

The DynBodyFrameSwitch class switches the frame in which the subject DynBody object's state is represented and propagated. This class extends the base BodyAction class. Frame switching occurs when the subject DynBody transitions between spheres of influence.

The class provides two modes of operation:

SwitchOnApproach The subject body's integration frame is switched when the body enters the sphere of influence of the new integration frame.

SwitchOnDeparture The subject body's integration frame is switched when the body leaves the sphere of influence of the current integration frame.

The class uses a user-provided switch distance to determine whether the body has entered the sphere of influence of the new integration frame or left the sphere of influence of the current integration frame.

28.2 Mathematical Formulations

Two commonly used concepts to describe the gravitational sphere of influence of a smaller gravitational body orbiting a larger gravitational body are the Hill sphere[18] and the Laplace sphere of influence[1].

The Hill sphere is

$$r_{\text{Hill}} \approx a \left(\frac{m}{3M} \right)^{1/3} \tag{28.1}$$

The Laplace sphere of influence is

$$r_{\text{SOI}} \approx a \left(\frac{m}{M} \right)^{2/5} \tag{28.2}$$

Table 28.1 lists the radii of the Hill and Lagrange spheres of influence for a few of the pairs of solar system bodies. These values are supplied for the use by the user. The DynBodyFrameSwitch class uses a user-defined value to test transitioning between spheres of influence.

Table 28.1: Spheres of Influence

Bodies	**Hill**		**Laplace**	
Earth/Moon	61,500	km	66,200	km
Sun/Earth	1,496,700	km	924,700	km
Sun/Mars	1,084,110	km	577,300	km

28.3 Detailed Design

This is a very simple sub-model. The `initialize()` method simply checks for errors and subscribes to the target integration frame. The `is_ready()` method checks for the body entered the sphere of influence of the new integration frame or leaving the sphere of influence of the current integration frame based on the user-provided mode and switch distance. The `apply()` method invokes the DynBody `switch_integration_frames()` method to perform the frame switch.

Chapter 29

User Guide

The following assumes the scheme outlined in section 4.2.4.

29.1 Analysis

Assume a simulation S_define file declares simulation objects named ‘chaser’ and ‘dynamics’. The chaser object contains a DynBody object name ‘dyn_body’ and a DynBodyFrameSwitch object named ‘frame_switch’. The dynamics object contains a DynManager object named ‘manager’ and a BodyAction pointer named ‘action_ptr’.

The following will cause the vehicle to switch from the initial Earth-centered inertial integration frame to a Moon-centered inertial integration frame upon entering the Moon’s sphere of influence.

```
chaser.frame_switch.set_subject_body(chaser.dyn_body);
chaser.frame_switch.action_name         = "frame_switch_enter";
chaser.frame_switch.integ_frame_name    = "Moon.inertial";
chaser.frame_switch.switch_sense        = DynBodyFrameSwitch::SwitchOnApproach;
chaser.frame_switch.switch_distance {M} = 66.2e6;
dynamics.body_action_ptr = &chaser.frame_switch;
call dynamics.dynamics.manager.add_body_action (dynamics.body_action_ptr);
```

29.2 Integration

The simulation integrator must make the DynBodyFrameSwitch visible to the simulation user. The above example assumes the integrator declared a DynBodyFrameSwitch object named ‘frame_switch’ as part of the chaser simulation object.

Because this is an asynchronous operation, the simulation integrator needs to ensure that the Dynamics Manager `perform_actions()` method is called as a scheduled job.

29.3 Extension

This model relies on the user to specify the radius of the sphere of influence. An obvious extension to this model is to calculate that radius.

Chapter 30

Inspection, Verification, and Validation

30.1 Inspection

Inspection BodyAction_9: Design Inspection

The DynBodyFrameSwitch Class Sub-Model comprises a single class, class DynBodyFrameSwitch, which derives directly from the BodyAction class. This class overrides each of the `initialize()`, `is_ready()`, and `apply()` methods. Each method invokes the overridden parent class implementation; the `is_ready()` method properly incorporates the results from the invocation of the overridden `is_ready()` method into the readiness assessment.

By inspection, the DynBodyFrameSwitch Class Sub-Model satisfies requirements BodyAction_16 and BodyAction_17.

30.2 Validation

This section describes the test used to verify and validate that the DynBodyFrameSwitch Class Sub-Model satisfies the requirements levied against it. The verification and validation test infrastructure for this sub-model is in the `SIM_verif_frame_switch` subdirectory of the JEOD directory `models/dynamics/body_action/verif`.

Test BodyAction_4: Frame Switch

Background A driving requirement for JEOD was to provide the ability to switch the reference frame in which a vehicle's state is integrated during the course of a simulation run. An Apollo-style mission was chosen as the reference mission for this capability. The vehicle's integration frame must be switched from Earth-centered to Moon-centered inertial to avoid an otherwise inevitable loss of accuracy in the integrated state somewhere along the translunar trajectory.

Test description This test represents a 100 second interval of the translunar trajectory of the Apollo 8 mission near the point of the transition to the Moon's sphere of influence. The test verifies that the frame switching mechanisms maintain state to within numerical precision.

The simulation S_define file defines a vehicle and three gravitational bodies, the Sun, Earth and Moon. Two input files are provided that have identical initial conditions. One causes the vehicle to switch to Moon-centered inertial when the vehicle enters the Moon's gravitational sphere of influence; the other leaves the vehicle in Earth-centered inertial throughout.

Success criteria The frame switch is to maintain a continuous state with respect to the common parent of the former and new integration frames. In the case of a switch from the Earth-centered inertial to Moon centered-inertial frames, this common parent is the Earth-Moon barycenter frame. The difference between the Earth-Moon barycenter relative states in the two runs must be attributable to numerical differences for the test to pass.

Test results The transition from Earth-centered inertial to Moon-centered inertial occurs 45.5 seconds into the simulation. At this time, the position and velocity vectors of the vehicle relative to the Earth-Moon barycenter are $[298463.448, -116959.258, -55769.040]$ km and $[0.934601239, -0.199088249, -0.297865412]$ km/s. The switched and unswitched runs show an exact match in position; the velocities differ in y and z by 5.7×10^{-17} km/s. These small differences are clearly numerical.

The simulation ends 54.5 seconds later. At this time, the position and velocity vectors of the vehicle relative to the Earth-Moon barycenter are $[298514.379, -116970.107, -55785.274]$ km and $[0.934431436, -0.199064620, -0.297866911]$ km/s. The differences between the switched and unswitched runs at this point in time are $[1.1 \times 10^{-8}, 4.3 \times 10^{-9}, 1.1 \times 10^{-9}]$ km and $[1.8 \times 10^{-10}, -1.2 \times 10^{-10}, -1. \times 10^{-10}]$ km/s. These differences are no longer purely numerical but are still quite small. Some of these small differences result from the way the simulation is constructed and used, but some result from the fact that at this stage a Moon-centered inertial frame is a better choice than an Earth-centered inertial frame.

The test passes.

Applicable requirements This test demonstrates the satisfaction of requirements BodyAction_32 to BodyAction_34.

30.3 Requirements Traceability

Table 30.1 summarizes the inspections and tests that demonstrate the satisfaction of the requirements levied on the DynBodyFrameSwitch Class Sub-Model. Note that requirement BodyAction_35 has not yet been tested.

Table 30.1: DynBodyFrameSwitch Requirements Traceability

Requirement	**Artifact**
BodyAction_32 Frame switch	Test BodyAction_4 Frame switch
BodyAction_33 Continuity	Test BodyAction_4 Frame switch
BodyAction_34 Switch on entry	Test BodyAction_4 Frame switch

Bibliography

[1] Charles D. Brown. *Spacecraft Mission Design, Second Edition.* AIAA, Reston, Virginia, 1998.

[2] (generated by doxygen). *Body Action Reference Manual*. National Aeronautics and Space Administration, Johnson Space Center, Software, Robotics & Simulation Division, Simulation and Graphics Branch, 2101 NASA Parkway, Houston, Texas, 77058, July 2023.

[3] Hammen, D. Dynamic Body Model. Technical Report JSC-61777-dynamics/dyn_body, NASA, Johnson Space Center, Houston, Texas, July 2023.

[4] Hammen, D. Dynamics Manager Model. Technical Report JSC-61777-dynamics/dyn_manager, NASA, Johnson Space Center, Houston, Texas, July 2023.

[5] Hammen, D. Memory Management Model. Technical Report JSC-61777-utils/memory, NASA, Johnson Space Center, Houston, Texas, July 2023.

[6] Jackson, A. Planet Fixed Model. Technical Report JSC-61777-utils/planet_fixed, NASA, Johnson Space Center, Houston, Texas, July 2023.

[7] Jackson, A., Thebeau, C. JSC Engineering Orbital Dynamics. Technical Report JSC-61777-docs, NASA, Johnson Space Center, Houston, Texas, July 2023.

[8] Morris, J. Planet Model. Technical Report JSC-61777-environment/planet, NASA, Johnson Space Center, Houston, Texas, July 2023.

[9] NASA. NASA Software Engineering Requirements. Technical Report NPR-7150.2, NASA, NASA Headquarters, Washington, D.C., September 2004.

[10] Shelton, R. Message Handler Model. Technical Report JSC-61777-utils/message, NASA, Johnson Space Center, Houston, Texas, July 2023.

[11] Shelton, R. Named Item Model. Technical Report JSC-61777-utils/named_item, NASA, Johnson Space Center, Houston, Texas, July 2023.

[12] Shelton, R. Orbital Elements Model. Technical Report JSC-61777-utils/orbital_elements, NASA, Johnson Space Center, Houston, Texas, July 2023.

[13] Spencer, A. Reference Frame Model. Technical Report JSC-61777-utils/ref_frames, NASA, Johnson Space Center, Houston, Texas, July 2023.

[14] Thebeau, C. Mass Body Model. Technical Report JSC-61777-dynamics/mass, NASA, Johnson Space Center, Houston, Texas, July 2023.

[15] Thompson, B. Mathematical Functions Model. Technical Report JSC-61777-utils/math, NASA, Johnson Space Center, Houston, Texas, July 2023.

[16] Turner, G. Derived State Model. Technical Report JSC-61777-dynamics/derived_state, NASA, Johnson Space Center, Houston, Texas, July 2023.

[17] Vallado, D.A. *Fundamentals of Astrodynamics and Applications.* McGraw-Hill, New York, 1997.

[18] Mauri J. Valtonen and Hannu Karttunen. *The Three-Body Problem.* Cambridge University Press, Cambridge, UK, 2006.