JSC-61777-models/dynamics/dyn_body

JSC Engineering Orbital Dynamics Dynamic Body Model

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

Package Release JEOD v5.1

Document Revision 1.1
July 2023

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

JSC-61777-models/dynamics/dyn_body

JSC Engineering Orbital Dynamics
Dynamic Body Model

Document Revision 1.1
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 Dynamic Body Model, a component of the dynamics suite of models within JEOD v5.1, is located at models/dynamics/dyn_body.

The primary purpose of the model is to represent dynamic aspects of spacecraft in a simulation. The model's DynBody class is the base class for such representations. A DynBody has properties such as mass that are intrinsic to the body itself as well as extrinsic properties such as the state of the body with respect to the outside world. External forces and torques act to change the body's state over time.

Key Concepts

Mass

Many core DynBody functions require interactions between mass properties. Each DynBody contains a MassBody member instance. However not all MassBody's belong to a DynBody. For this reason, every MassBody acting as a container for a DynBody's properties has an immutable pointer to the DynBody owner. MassBody objects have mass properties and can be attached to/detached from other MassBody objects. A basic MassBody object is stateless and lacks a connection to the dynamic world. The DynBody class adds that connection to a MassBody object.

State

The DynBody class makes the connection to the outside world in the form of 'state'. Multiple states, in fact. The Dynamic Body Model uses a derivative of the RefFrame class, the BodyRefFrame class, to represent state. All active reference frames in a simulation, including those associated with a DynBody object, are connected to one another to form the simulation's reference frame tree.

Integration Frame

The connection between a DynBody object's DynBodyFrame objects and the outside world is through the DynBody object's integration frame. The integration frame must be one of the simulation's inertial frames. All of the DynBodyFrame objects associated with a given DynBody object share the DynBody object's integration frame as a common parent.

Two driving requirements for JEOD were to provide (1) the ability for different vehicles in a multi-vehicle simulation to have their states represented in and integrated in different frames of reference and (2) the ability to switch integration frames for a given vehicle during the course of a simulation run. The Dynamic Body Model accomplishes both of these objectives by enabling each root DynBody object to have its own integration frame and providing tools to change this frame dynamically in a manner that is consistent with the laws of physics.

Attach/Detach

MassBody objects can be attached to/detached from one another. DynBody objects may also attach/detach to one another. MassBody subassemblies may also attach to a DynBody object, granted that they do not attach to another DynBody or its subassemblies. A DynBody may not attach as a subcomponent of a MassBody, because a DynBody has a state, while the parent MassBody does not. This model of capability is essential for modeling the docking and undocking of space vehicles and spacecraft component. The composite mass properties change when objects attach or detach. The composite mass properties aggregate recursively. For example, if A is attached to B is attached to C, then A's composite properties reflect only itself, B's composite properties reflect the combination of A+B,and C's composite properties reflect the combined A+B+C. Much more also happens when DynBody objects attach and detach, including conservation of momenta. Vehicle states change, and these changes must be performed in a manner consistent with the laws of physics.

Additionally, DynBody objects may be kinematically attached to any reference frame as a means of holding position relative to that frame. In this case, the mass tree is unaffected. One use case for this capability is when a vehicle is at a launchpad and should remain fixed positionally and rotationally w.r.t. the planet-fixed frame.

Initialization

Initializing a DynBody object is a rather complex process. The zeroth stage of the initialization process occurs when the object is constructed. The next stage of initialization involves preparing the object for the truly complex stages that follow. The body is registered with the dynamics manager, as are the standard body reference frames associated with the body. The body's integration frame is set and the standard body reference frames are linked to this integration frame. Note that while the links between the DynBody object are established in this phase, much work remains to be done. The body is unattached and its mass properties, mass points/vehicle points, and states remain uninitialized at this stage.

These yet-to-be initialized aspects of a DynBody must be initialized in the proper order. **The recommended approach is to first initialize the mass properties and mass points of all MassBody objects, then perform any initialization-time attachments, and finally initialize state.** This is the approach taken when the body action-based initialization scheme is used to initialize mass bodies and dynamic bodies. Deviating may result in improper initial states due to the aggregating nature of composite vehicle attachments.

State Initialization

Properly initializing the state of all of the vehicles in a simulation has been an ongoing and rather vexing issue. JEOD approaches this problem by spreading the responsibility for state initialization over several models.

Forces and Torques

In addition to the acceleration due to gravity, a spacecraft can be subject to many external forces and torques. For example a spacecraft's thrusters, atmospheric drag, and radiation pressure exert forces and torques on the vehicle. The source and nature of these forces and torques is more or less irrelevant to this model. Forces and torques are each categorized as effector, environmental, and non-transmitted. This categorization is external to this model; all this model sees is a set of three Standard Template Library (STL) vectors of CollectForces and a set of three STL vectors of CollectTorques. Each set refers to loads of alike nature: effector, environmental, and non-transmitted. A more in-depth explanation follows.

Collect Mechanism

The above discussion on forces and torques ignored how the STL collect vectors are formed. How these vectors are populated is not an issue of concern in the narrow view of the DynBody class taken by itself. From the perspective of the DynBody class, those vectors are populated by some external agent. This is a concern to the model taken as a whole, and even more importantly, it is a concern to the user.

JEOD and Trick developers worked together to create the *vcollect* mechanism. The mechanism takes advantage of capabilities provided with the C++ language, and is easy to implement outside of the Trick environment.

Gravity

The above discussion on forces intentionally skirted over the issue of gravitational acceleration. JEOD has always treated gravitation as a topic conceptually distinct from the other forces that act on a dynamic body. One reason for doing so is that acceleration (rather than force) is the natural output of any gravity model, including the JEOD Gravity Model. Acceleration (rather than force) is what is ultimately needed to formulate the equations of motion.

More importantly, gravitation truly is distinct from the external forces. Unlike any other force, the gravitational force acting on an object cannot be detected directly by any device. An accelerometer, for example, measures the acceleration due to the net non-gravitational force. Even though JEOD assumes Newtonian physics, it is a good idea to look at gravitation from the perspective of the more accurate general relativistic point of view. Gravitation is not a force in general relativity. It is something quite different from forces and is best treated as such.

Equations of Motion

The external forces and gravitational acceleration form the basis for the translational equations of motion for a dynamic body, while the external torques form the basis for the body's rotational equations of motion. Those equations of motion are second order differential equations; they must be integrated over time.

State Integration

This state integration must be performed numerically due to the varying and complex nature of the forces and torques. This model uses the Integration Model to perform the numerical integration.

The model only integrates the root body of a tree of attached DynBody objects and for that body, integrates only one reference frame. This is the body's integrated frame. Exactly which frame is integrated is a decision made by a class that derives from the DynBody class. The DynBody class integrates the composite body frame. The StructureIntegratedDynBody class (introduced with JEOD 3.4) integrates the root body's structural frame. Users can define extended classes that integrate the root body through alternative means.

State Propagation

A DynBody contains multiple reference frames. All of these frames must reflect the current state of the vehicle. Since only one frame is integrated, the other frames must be made consistent with this integrated frame. This is performed by propagating the results of the integration to all frames associated with the DynBody object, including the DynBody objects attached as children to the root DynBody object. This propagation is performed using the geometry information embodied in the assembly of DynBody's and attached MassBody's.

Interactions

JEOD Models Used by the Dynamic Body Model

The Dynamic Body Model uses the following JEOD models:

- *Dynamics Manager Model* [2]. The Dynamic Body Model uses the Dynamics Manager Model as (1) a name register of DynBody objects, (2) a name register and subscription manager of RefFrame objects, and (3) a name register and validator of integration frames.
- *Mass Body Model* [13]. The Dynamic Body Model uses the Mass Body Model through friendship. The DynBody class *has-a* MassBody. Friendship does not extended to derivations inheriting from DynBody, protecting its core functionality.
- *Gravity Model* [16]. The Dynamic Body Model relies on the Gravity Model to compute gravitational acceleration. To accomplish this end, each DynBody object contains an object that indicates which gravitational bodies in the simulation have an influence on the DynBody object.

- *Integration Model* [3]. The Dynamic Body Model uses the Integration Model to perform the state integration. This model merely sets things up for the Integration Model so that it can properly perform the integration. The Dynamic Body Model also uses er7_utils::IntegrableObject through inheritence. The DynBody class *is-a* er7_utils::IntegrableObject. In addition, Dynamic Body Model also provides the capability to associate other instances of IntegrableObjects to be integrated along with the state of the Dynamic Body Model. This capability is demonstrated by associating IntegrableObjects corresponding to thermal states of surface facets in verif/SIM_3_ORBIT_1st_ORDER_T10 of the radiation_pressure model.
- *Mathematical Functions* [14]. The Dynamic Body Model uses the Mathematical Functions to operate on vectors and matrices.
- *Memory Allocation Routines* [4]. The Dynamic Body Model uses the Memory Allocation Routines to allocate and deallocate memory.
- *Message Handling Class* [9]. The Dynamic Body Model uses the Message Handling Class to generate error and debug messages.
- *Named Item Routines* [10]. The Dynamic Body Model uses the Named Item Routines to construct names linked to the associated MassBody name.
- *Quaternion* [5]. The Dynamic Body Model uses the Quaternion to operate on the quaternions embedded in the RefFrame objects and to compute the quaternion time derivative.
- *Reference Frame Model* [12]. The Dynamic Body Model makes quite extensive use of the Reference Frame Model. A DynBody's integration frame is a RefFrame object, and each of the body-based reference frames contained in a DynBody object is a BodyRefFrame object. The BodyRefFrame *is-a* RefFrame. The Dynamic Body Model uses the Reference Frame Model functionality to compute relative states.
- *Simulation Engine Interface Model* [11]. All classes defined by the Dynamic Body Model use the
`JEOD_MAKE_SIM_INTERFACES` macro defined by the Simulation Engine Interface Model to provide the behind-the-scenes interfaces needed by a simulation engine such as Trick.

Use of the Dynamic Body Model in JEOD

The following JEOD models use the Dynamic Body Model:

- *Body Action Model* [6]. The Body Action Model provides several classes that operate on a DynBody object. The DynBodyInit class and its derivatives initialize a DynBody object's state. The DynBodyFrameSwitch class provides a convenient mechanism for switching a DynBody object's integration frame. The BodyAttach and Detach classes indirectly operate on a DynBody object through the Dynamic Body Model attach/detach mechanisms.
- *Derived State Model* [17]. The primary purpose of the Derived State Model is to express a DynBody state in some other form.
- *Dynamics Manager Model* [2]. The Dynamics Manager Model drives the integration of all DynBody objects registered with the dynamics manager.

- *Gravity Model* [16]. The Gravity Model computes the gravitational acceleration experienced by a DynBody object.
- *Gravity Gradient Torque Model* [15]. The Gravity Gradient Torque Model computes the gravity gradient torque exerted on a DynBody object, with the resultant torque expressed in the DynBody object's structural frame.

Contents

List of Tables

Chapter 1

Introduction

1.1 Purpose and Objectives of the Dynamic Body Model

The Dynamic Body Model comprises several classes that together provide the ability to represent dynamic aspects of spacecraft in a simulation.

1.2 Context within JEOD

The following document is parent to this document:

- *JSC Engineering Orbital Dynamics* [7]

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

1.3 Documentation History

Author	Date	Revision	Description
David Hammen	April, 2010	1.0	Initial Version
Mitch Hollander	September, 2021	1.1	JEOD 4.0 Revisions

1.4 Documentation Organization

This document is formatted in accordance with the NASA Software Engineering Requirements Standard [8].

The document comprises chapters organized as follows:

Chapter 1: Introduction - This introduction describes the objective and purpose of the Dynamic Body Model.

Chapter 2: Product Requirements - The requirements chapter describes the requirements on the Dynamic Body Model.

Chapter 3: Product Specification - The specification chapter describes the architecture and design of the Dynamic Body Model.

Chapter 4: User Guide - The user guide chapter describes how to use the Dynamic Body Model.

Chapter 5: Verification and Validation - The inspection, verification, and validation (IV&V) chapter describes the verification and validation procedures and results for the Dynamic Body Model.

Chapter 2

Product Requirements

Requirement DynBody_1: Project Requirements

Requirement:
This model shall meet the JEOD project requirements specified in the JEOD top-level document.

Rationale:
This is a project-wide requirement.

Verification:
Inspection

Requirement DynBody_2: Mass

Requirement:
The Dynamic Body Model shall satisfy all requirements levied on the *Mass Body Model* [13].

Rationale:
A dynamic body has the properties of a mass body, including mass, moment of inertia, and center of mass.

Verification:
Inspection, test

Requirement DynBody_3: Integration Frame

Requirement:
The Dynamic Body Model shall provide the ability to initially specify and to dynamically change the reference frame with respect to which the states associated with a dynamic body are referenced and are integrated.

Rationale:

The integration frame is the connection between a dynamic body and the outside world.

That each independent vehicle can have its own integration frame and that this frame can be changed during the course of a simulation run are driving requirements for JEOD.

Changing integration frames needs to be done within the confines of the laws of physics.

Verification:

Inspection, test

Requirement DynBody_4: State Representation

Requirement:

4.1 State Representation. The Dynamic Body Model shall provide the ability to represent the states (positions, velocities, orientations, and angular velocities) of local reference frames (origin and axes) associated with a dynamic body with respect to the body's integration frame.

4.2 Required Frames. The Dynamic Body Model shall represent the states of the object's

- Structural origin / structural axes,
- Core center of mass / body axes,
- Composite center of mass / body axes, and
- Mass point frames registered as object frames of interest.

Rationale:

This requirement gives a dynamic body a connection to the outside world.

Verification:

Inspection, test

Requirement DynBody_5: Staged Initialization

Requirement:

5.1 Separate Initializations. The Dynamic Body Model shall provide the ability to set one or more element of the state (position, velocity, orientation, and angular velocity) of one of the reference frames associated with a dynamic body.

5.2 Partial Propagation. The Dynamic Body Model shall provide the ability to propagate the partially set states for a given dynamic body to all reference frames associated with that body, subject to the mathematical limits regarding the ability to perform the propagation.

5.3 Initialized State Elements Query. The Dynamic Body Model shall provide the ability to query whether specific elements (position, velocity, orientation, and/or angular velocity) of the state of a reference frame associated with a dynamic body have been initialized.

Rationale:
Vehicle states must be initialized before they can be used.

Verification:
Inspection, test

Requirement DynBody_6: Equations of Motion

Requirement:

6.1 Forces and Torques. The Dynamic Body Model shall provide the ability to individually represent and collectively assess the forces and torques that act on a dynamic body, including forces and torques acting on dynamic bodies attached to the dynamic body in question.

6.2 Non-transmitted forces and torques. The model shall provide the ability to identify some forces and torques as being non-transmittable to the parent dynamic body.

6.3 Translational Acceleration. The model shall provide the ability to calculate the translational acceleration of a dynamic body that results from the collective forces acting on the dynamic body, including gravitation.

6.4 Rotational Acceleration. The model shall provide the ability to calculate the rotational acceleration of a dynamic body that results from the collective torques acting on the dynamic body, including the fictitious inertial torque.

Rationale:
Properly representing the equations of motion is a prerequisite to computing the time-varying nature of a dynamic body's state.

Verification:
Inspection, test

Requirement DynBody_7: State Integration and Propagation

Requirement:

7.1 State Integration. The Dynamic Body Model shall provide the ability to integrate at least one of a dynamic body's reference frames over time based on the dynamical equations of motion for that body.

7.2 State Propagation. The Dynamic Body Model shall provide the ability to propagate a dynamic body's non-integrated states such that the propagated states, local frames, and integrated states are consistent to within numerical precision.

Rationale:
Properly integrating vehicle state over time has been a driving requirement for JEOD since its inception.

Verification:
Inspection, test

Requirement DynBody_8: Vehicle Points

Requirement:

8.1 Vehicle Point Registration. The Dynamic Body Model shall provide the ability to register local reference frames associated with mass points as object frames of interest.

8.2 Vehicle Point Acceleration. The Dynamic Body Model shall provide the ability to compute the translational and rotational acceleration of a local reference frame associated with a mass point, including the non-gravitational translational acceleration.

Rationale:
This requirement derives from reqt 4.2.

Verification:
Inspection, test

Requirement DynBody_9: Attach/Detach

Requirement:

9.1 Dynamic Body Attach/Detach. The Dynamic Body Model shall provide the ability to attach and detach dynamic bodies in a manner consistent with the conservation laws.

9.2 Kinematic Body Attach/Detach The Dynamic Body Model shall provide the ability to attach and detach dynamic bodies to any reference frame kinematically.

Rationale:
The Mass Body Model attach/detach requirements do not mention the conservation laws, so this functionality must be introduced in Dynamic Body Model in order to provide capabilities for docking, berthing, jettision, etc. operations common to spaceflight missions.

Additionally, the ability to attach a dynamic body to a reference frame such that it is rigidly fixed to it is a valuable simulation capability. In this case, the conservation laws for attach or detach may be ignored because their effects are negligible or the attachment acts as a low-fidelity attachment to constrain the vehicle. This is a feature the user can decide to use and the Dynamic Body Model should provide this capability.

Verification:
Inspection, test

Chapter 3

Product Specification

3.1 Conceptual Design

The primary purpose of the Dynamic Body Model is to represent dynamic aspects of spacecraft in a simulation. The model's DynBody class is the base class for such representations. A DynBody has properties such as mass that are intrinsic to the body itself as well as extrinsic properties such as the state of the body with respect to the outside world. External forces and torques act to change the body's state over time.

The forces and torques that act on a DynBody are obviously *has-a* relationships.[1] Multiple forces and torques can be present and can vary in number and nature from body to body. State is also a *has-a* relationship as a DynBody has multiple states. The body's center of mass and structural origin, for example, have potentially distinct locations with respect to the outside world.

A spacecraft has mass properties such as mass, a center of mass, and moment of inertia. These intrinsic properties of a DynBody are represented by the *Mass Body Model* [13] MassBody class. The relationship between a DynBody and a MassBody can be either *has-a* or *is-a*. In JEOD 1.5, the relationship between an OrbitalBody and a MassBody was *has-a* by necessity. JEOD 1.5 was implemented in C, which does not offer *is-a* relationships. That limitation does not exist in JEOD 2.0. Making the relation *is-a* offered the advantage of overridden functions over a *has-a* relationship. The JEOD 2.0 DynBody class inherited from the JEOD 2.0 MassBody class. In JEOD 4.0, the relationship was reverted to a *has-a* relationship with friendship. This removed the need for constant checks to differentiate between a MassBody and a DynBody, and provides extension stability by preventing users from overriding protected JEOD functionality.

The enhanced flexibility introduced with recent changes to the *Integration Model* [3] Integration Model are enabled by abstracting the sorts of things which JEOD can integrate in the er7_utils::IntegrableObject class. Now a DynBody *is-a* er7_utils::IntegrableObject. In addition, the DynBody class now provides the capability to associate additional IntegrableObjects which are

[1] A *has-a* relationship entails containment while a *is-a* relationship entails specialization. A car *is-a* kind of vehicle but *has-a* carburetor and four tires. Sometimes the distinction between *is-a* and *has-a* is not as clear as in this simple example. For example, the very first design of JEOD 2.0 used a *has-a* relationship for the association between a DynBody object and a MassBody object, whereas JEOD 3.0 used a DynBody *is-a* MassBody approach. Ultimately, these design decisions should be made in order to balance code legibility, use cases, and user feedback.

integrated along with the DynBody state.

3.1.1 Key Concepts

This section describes the key concepts that form the conceptual foundation of the Dynamic Body Model.

3.1.1.1 Mass

As noted above, a DynBody *has-a* MassBody. MassBody objects have mass properties and can be attached to/detached from other MassBody objects. MassBody objects represent mass properties in the form of instances of the MassProperties class. The MassProperties class derives from the MassPoint class.

A MassBody object defines three mass properties objects, and hence three MassPoints, one for each of the object's (1) structural origin/structural axes, (2) core center of mass/body axes, and (3) composite center of mass/body axes. The MassPoints class defines a rudimentary kind of reference frame basis. These rudimentary reference frames include an origin and a set of orthogonal right-handed axes, but exclude the derivatives information found in the *Reference Frame Model* [12]. A basic MassBody object lacks a connection to the outside world. The DynBody class adds that connection to a MassBody object. DynBody mass attributes exist in the MassBody `DynBody::mass` member. MassBody's referring to the mass contents of a DynBody contain a pointer DynBody `MassBody::dyn_owner` member. Pure mass items, which do not refer to a dynamic body, should have a `NULL` value for this member.

3.1.1.2 State

The DynBody class makes the connection to the outside world in the form of 'state'. Multiple states, in fact. The Dynamic Body Model uses the BodyRefFrame class to represent state. A BodyRefFrame *is-a* RefFrame plus a pointer to a MassPoint object. The model maintains a BodyRefFrame for each of the MassPoint objects associated with a MassBody object.

A RefFrame object contains links and states. The links connect RefFrame objects to one another. The state describes the translational and rotational relation between the frame and its parent frame. All active reference frames in a simulation are connected to one another to form a tree structure. The problem at hand is to connect the reference frames associated with a DynBody object to this larger, simulation-wide reference frame tree.

3.1.1.3 Inertial Frames

The root node of the simulation reference frame tree is, by assumption, a true Newtonian inertial frame of reference. Other reference frames in the tree loosely qualify as being 'inertial' frames. These so-called inertial frames are not truly inertial frames; their origins are accelerating with respect to the root frame. What qualifies these frames as being 'inertial' frames is that they are not rotating with respect to the root frame, they are connected to the root frame either directly

or through other 'inertial' frames, and they are subject to gravitation forces only. Non-rotating planet-centered reference frames, for example, qualify as 'inertial' reference frames in JEOD. (And elsewhere. This is a very widespread and very convenient abuse of notation.)

Any non-inertial frame can be made to look like an inertial frame of reference (*i.e.*, $F = ma$) by means of fictional forces and torques. These fictional forces and torques are accounting tricks. The only such trick needed to make the JEOD inertial frames appear to be true inertial frames is to account for the inertial force due to frame acceleration, and that acceleration is only gravitational. The *Gravity Model* [16] performs these accounting tricks. From the perspective of this model, these pseudo-inertial frames of reference can be treated as inertial frames in which $F = ma$ is valid.

3.1.1.4 Integration Frame

The connection between a DynBody object's DynBodyFrame objects and the outside world is through the DynBody object's integration frame. The integration frame must be one of the simulation's inertial frames. All of the DynBodyFrame objects associated with a given DynBody object share the DynBody object's integration frame as a common parent.

A driving requirement for JEOD was to provide the ability for different vehicles in a multi-vehicle simulation to have their states represented in and integrated in different frames of reference. For example, using the Moon-centered inertial frame as the integration frame would best suit a vehicle orbiting the Moon, but the Earth-centered inertial frame would best suit a vehicle in low Earth orbit. The Dynamic Body Model accomplishes this goal by enabling each root DynBody object to have its own integration frame. This allows state to be represented in and integrated in the frame best-suited to the object in question.

Another driving requirement for JEOD was to provide the ability to switch integration frames for a given vehicle during the course of a simulation run. For example, a vehicle in transit from the Earth to the Moon should be able to switch the vehicle's integration frame from Earth-centered inertial to Moon-centered inertial at some point during the transit. The Dynamic Body Model accomplishes this goal as well. A DynBody object's integration frame can be changed dynamically in a manner that is consistent with the laws of physics.

3.1.1.5 Attach/Detach

MassBody objects can be attached to/detached from one another. So can DynBody objects. This capability is essential for modeling the docking and undocking of spacecraft. The composite mass properties change when MassBody objects attach or detach. That plus more happens when DynBody objects attach and detach. Vehicle states change, and these changes must be consistent with the laws of physics.

The Dynamic Body Model attach mechanism first queries both the child and parent regarding the validity of the proposed attachment. If both the child and parent assent to the attachment, the transform from the root body of the child to the desired attach point is calculated. If the child is already a root body, the transform is simply the mass point transform or the user-provided transform. The child is told to establish the linkages between child-and-parent/parent-and-child while the parent is told to update properties using the Mass Body Model attach mechanisms.

Dynbody objects may also be attached to a RefFrame object kinematically. This capability is primarily for sim operations or to assume the capabilities of a low-fidelity attachment model. As a result, momentum and energy conservation issues are ignored and the dynamic state is simply held rigid w.r.t. its parent RefFrame.

A detachment proceeds similarly. First there is a check for validity, and only then is the child told to sever linkages and the parent is told to update properties. The DynBody calls the Mass Body Model detach mechanisms.

3.1.1.6 Initialization

Initializing a DynBody object is a rather complex process. The zeroth stage of the initialization process occurs when the object is constructed. For example, the association between the core properties and core state is immutable and can be established at construction time.

The next stage of initialization involves preparing the object for the truly complex stages that follow. The body is registered with the dynamics manager, as are the standard body reference frames associated with the body. The body's integration frame is set and the standard body reference frames are linked to this integration frame. Note that while the links between the DynBody object are established in this phase, much work remains to be done. The body is unattached and its mass properties, mass points/vehicle points, and states remain uninitialized at this stage.

These yet-to-be initialized aspects of a DynBody must be initialized in the proper order. The recommended approach is to first initialize the mass properties and mass points of all MassBody objects, then perform any initialization-time attachments, and finally initialize state. This is the approach taken when the body action-based initialization scheme is used to initialize mass bodies and dynamic bodies. This approach involves a series of interactions between the *Dynamics Manager Model* [2] and the *Body Action Model* [6]. The Body Action Model in turn interacts with several other models, including this model, while performing its actions.

With this approach, the simulation's dynamics manager performs, in stages, the queued BodyAction objects that derive from the MassBodyInit class, then the queued BodyAction objects that derive from the BodyAttach class, and finally the queued BodyAction objects that derive from the DynBodyInit class. The first stage establishes the mass properties and the attach points needed to properly perform the next two stages. The attachments made in the second stage (potentially) reduce the number of independent states that need to be set. The final stage initializes states.

3.1.1.7 State Initialization

Properly initializing the state of all of the vehicles in a simulation has been an ongoing and rather vexing issue. The code that performed this initialization in pre-2.0 releases of JEOD was fragile, hard to extend, and overly complex. JEOD 2.0 solved this problem by spreading the responsibility for state initialization over several models.

The recommended approach is to use the Body Action Model to initialize states. That model provides several classes that derive from the DynBodyInit class. It is these DynBodyInit derivative classes that initialize DynBody states. Each DynBodyInit object sets one or more elements (position, velocity, orientation, and angular velocity) of a reference frame associated with the DynBody

object that is the subject of the DynBodyInit object. The to-be-set state elements are specified with respect to some other basis reference frame, which need not be the subject DynBody object's integration frame. For example, rendezvous simulations often initialize the state of one vehicle given the relative state with some other target vehicle.

Certain elements of the basis reference frame must already have been established before the values of the desired state elements can be determined. These required elements form a set of prerequisites for setting the desired state elements. Those prerequisites are sometimes satisfied immediately, as is the case when the basis is a planetary frame. Difficulties can arise when the basis is a vehicle frame. Suppose the reference object for a given DynBodyInit object is some vehicle frame for which the requisite state elements have not been set. One option would be to simply declare a user error when this happens. That is not the course taken by JEOD. Instead, the DynBodyInit object in question is put on hold until the reference vehicle frame is initialized to the extent needed by the DynBodyInit object.

Each BodyAction object, including the DynBodyInit objects, provides an indication of whether the object is ready to perform its action. The dynamics manager queries each enqueued object regarding its readiness before telling the object to perform its action. The readiness test for a DynBodyInit object includes a check of whether the prerequisite state elements have been set. The dynamics manager repeatedly loops over the queued DynBodyInit objects, stopping only when either all such objects have been applied or until no such object indicates that it is ready to be applied.

This model participates in the exchange as well. The DynBody class provides mechanisms to query which state elements have been set, to set specific state elements, and to propagate the set states to frames associated with the DynBody object, including those owned by DynBody objects attached to the subject DynBody object.

3.1.1.8 Forces and Torques

In addition to the acceleration due to gravity, a spacecraft can be subject to many external forces and torques. For example a spacecraft's thrusters, atmospheric drag, and radiation pressure exert forces and torques on the vehicle. The source and nature of these forces and torques is more or less irrelevant to this model. Forces and torques are each categorized as effector, environmental, and non-transmitted. This categorization is external to this model; all this model sees is a set of three Standard Template Library (STL) vectors of CollectForces and a set of three STL vectors of CollectTorques.

The distinction between the effector and environmental forces and torques is somewhat arbitrary and exists solely as an aid to the user.[2] On the other hand, the distinction between the non-transmitted forces and torques and the effector and environmental forces and torques is significant. For a non-root DynBody object, the effector and environmental forces and torques are transmitted to the DynBody object to which the non-root DynBody is attached. The non-transmitted forces and torques are not transmitted to the parent (hence the name).

The reason for having these non-transmitted forces and torques is again one of user convenience.

[2] From a JEOD perspective, things would be a bit easier had the effector and environmental forces been lumped together as transmitted forces, with the effector and environmental torques similarly lumped together as transmitted torques.

Consider the force due to atmospheric drag on a composite vehicle. While this composite drag force is the vector sum of the drag forces on the individual vehicles that comprise the composite, the close proximity of the components of the composite vehicle makes those individual drag forces different from what they would be in the absence of those other vehicles. Enabling users to mark certain forces and torques as non-transmittable simplifies the users' calculation of those forces and torques that are affected by the presence of or connection to other vehicles. Users can simply ignore those complicating factors and calculate those forces and torques as if other vehicles were not present.

The net force and torque on a root body form the basis for the equations of motion for that body. It is the net forces and torques that are of the greatest interest to JEOD. JEOD uses the superposition principle to compute the net force acting on a DynBody object. The net force on a root body is the vector sum of all of the forces acting directly on that body, including that body's non-transmitted forces, plus the sum of all transmittable forces acting on bodies attached to the root body. Calculating the net torque is a bit more complex. In addition to the torques specified through the collect vectors, torques due to non-centerline forces must be taken into account. The forces in the collect vectors are assumed to act through the center of mass of the DynBody object that contains those collect vectors. When a DynBody object is attached as a child to some other DynBody object, the centerline forces on the child body become non-centerline forces when transmitted to the parent body.

3.1.1.9 Wrenches

JEOD 3.4 introduces the concept of a wrench. A wrench comprises an external force and an external torque, with the the force applied at a specified point on the dynamic body. The torque induced by the force should not be included in the torque portion of a wrench; JEOD calculates this torque. For example, a typical chemical thruster model that uses a wrench will have the torque be zero. On the other hand, a model of an ion thruster will have a non-zero torque. The difference is that the exhaust from a chemical thruster typically does not have a curl while the exhaust from an ion thruster does. Even if an ion thruster's line of action passes directly through the vehicle's center of mass, the thruster imparts a torque on the vehicle due to the swirling exhaust.

3.1.1.10 Collect Mechanism

The above discussion on forces and torques ignored how the STL collect vectors are formed. How these vectors are populated is not an issue of concern in the narrow view of the DynBody class taken by itself. From the perspective of the DynBody class, those vectors are populated by some external agent. This is a concern to the model taken as a whole, and even more importantly, it is a concern to the user.

Prior to JEOD 2.0, Trick users used Trick's *collect* mechanism to identify the forces and torques in play. This approach was replaced for a number of reasons. Instead, JEOD and Trick developers worked together to create the Trick 07 *vcollect* mechanism. Compared to the *collect* mechanism, this new mechanism has the potential for greater safety (type, unit, and dimensionality), takes better advantage of capabilities provided with the C++ language, and is easier to implement outside of the Trick environment.

The wrenches introduced in JEOD 3.4 can also be collected via the vcollect mechanism. As of JEOD

3.4, only the StructureIntegratedDynBody provides the ability to collect and process wrenches. This capability may be migrated upward in future releases of JEOD.

3.1.1.11 Gravity

The above discussion on forces intentionally skirted over the issue of gravitational acceleration. JEOD has always treated gravitation as a topic conceptually distinct from the other forces that act on a dynamic body. One reason for doing so is that the *Gravity Model* [16] computes gravitational acceleration directly, and acceleration (rather than force) is what is ultimately needed to formulate the equations of motion.

More importantly, gravitation truly is distinct from the external forces. Unlike any other force, the gravitational force acting on an object cannot be detected directly by any device. An accelerometer, for example, measures the acceleration due to the net non-gravitational force. Even though JEOD assumes Newtonian physics, it is a good idea to look at gravitation from the perspective of the more accurate general relativistic point of view. Gravitation is not a force in general relativity. It is something quite different from forces and is best treated as such.

The earlier discussion on integration frames noted that the integration frame can be a not-quite inertial frame. The Gravity Model folds the acceleration of a DynBody's integration frame into the total gravitational acceleration presented to a DynBody object. As far as the DynBody object is concerned, the integration frame *is* an inertial frame of reference; Newton's second law can be employed to integrate a DynBody object's state.

3.1.1.12 Equations of Motion

The external forces and gravitational acceleration form the basis for the translational equations of motion for a dynamic body, while the external torques form the basis for the body's rotational equations of motion. Those equations of motion are second order differential equations; they must be integrated over time.

3.1.1.13 State Integration

This state integration must be performed numerically due to the varying and complex nature of the forces and torques. This model uses the Integration Model to perform the numerical integration.

The model only integrates the root body of a tree of attached DynBody objects and for that body, integrates only one reference frame. This is the body's integrated frame. Exactly which frame is integrated is a decision made by a class that derives from the DynBody class. The DynBody class provided integrates the root body's composite body frame. The StructureIntegratedDynBody class (introduced with JEOD 3.4) integrates the root body's structural frame.

3.1.1.14 State Propagation

A DynBody contains multiple reference frames. All of these frames must reflect the current state of the vehicle. Since only one frame is integrated, the other frames must be made consistent with

this integrated frame. This is performed by propagating the results of the integration to all frames associated with the DynBody object, including the DynBody objects attached as children to the root DynBody object. This propagation is performed using the geometry information embodied in the MassBody objects from which the DynBody objects are derived.

3.1.1.15 Point Acceleration

A reference frame state contains zeroth and first derivatives. Second derivatives are not a part of the state. Some vehicle sensors such as accelerometers need acceleration data, and they need the acceleration that would be sensed at the point where the accelerometer is located. An accelerometer senses acceleration due to non-gravitational forces. The model provides an acceleration propagation capability to meet this need.

3.1.2 Model Architecture

The Dynamic Body Model is implemented in the form of C++ classes. The model classes are described below.

DynBody The DynBody class is (or should be) an abstract class. This class provides all of the basic capabilities needed to represent a dynamic body except for the ability to integrate state over time. This exception is by design. The equations of motion vary with selection of which frame is to be integrated (composite body versus structural frame) and with assumptions regarding vehicle behavior dynamics.

BodyRefFrame The BodyRefFrame class extends the RefFrame class. Member data include a pointer to a MassBasicPoint object and an indicator of which state elements have been set.

BodyForceCollect The BodyForceCollect class contains (1) the six STL vectors that collect the effector, environmental, and non-transmitted forces and torques that act on a DynBody object, (2) the vector sums of the collected forces and torques, categorized by type, and (3) the total external force and torque acting on a DynBody object. The collected forces and torques are assumed to be represented in the structural frame.

CollectForce A CollectForce instance contains a pointer to a flag indicating whether the force is active and a pointer to a 3-vector that in turn contains the components of the force in the DynBody's structural frame. The CollectForce class forms part of the interface between JEOD and the Trick *vcollect* mechanism. The overloaded `CollectForce::create()` method allocates and constructs a CollectForce object; exactly how depends on the argument to `create()`. The Trick *vcollect* depends on this Factory Method architecture. The STL force vectors in a BodyForceCollect instance that contain the collected effector, environmental, and non-transmitted forces contain pointers to CollectForce instances, each of which is constructed via `create()`.

Force A Force contains a flag indicating whether the force is active and contains a 3-vector that in turn contains the components of the force as represented in a DynBody's structural frame. A CollectForce instance can be easily and safely constructed from a Force instance. The Force class is the recommended approach for representing forces in JEOD.

CInterfaceForce The class CInterfaceForce extends the CollectForce class. The intent is to provide the ability to create CollectForce instances from forces that are not represented using the Force class. This capability is essential for using JEOD with C-based force models. This capability is not particularly safe. A CInterfaceForce can be created from any double pointer. The burden falls solely upon the user to use this capability properly.

CollectTorque, Torque,and CInterfaceTorque These classes are the torque analogs of the CollectForce, Force, and CInterfaceForce classes.

FrameDerivs The FrameDerivs class contains the second derivatives needed for state integration.

DynBodyMessages The DynBodyMessages class defines the message types used in conjunction with the *Message Handling Class* [9]. The DynBodyMessages is not an instantiable class.

3.2 Interactions

3.2.1 JEOD Models Used by the Dynamic Body Model

The Dynamic Body Model uses the following JEOD models:

- *Dynamics Manager Model* [2]. The Dynamic Body Model uses the Dynamics Manager Model as (1) a name register of DynBody objects, (2) a name register and subscription manager of RefFrame objects, and (3) a name register and validator of integration frames.
- *Mass Body Model* [13]. The Dynamic Body Model uses the Mass Body Model through friendship to access protected functionality. The DynBody class *has-a* MassBody.
- *Gravity Model* [16]. The Dynamic Body Model relies on the Gravity Model to compute gravitational acceleration. To accomplish this end, each DynBody object contains a GravityInteraction object. That class is a part of the Gravity Model. A GravityInteraction indicates which gravitational bodies in the simulation have an influence on a DynBody object and contains the gravitational accelerations and gravity gradients as computed by the Gravity Model.
- *Integration Model* [3]. The Dynamic Body Model uses the Integration Model to perform the state integration. This model merely sets things up for the Integration Model so that it can properly perform the integration. With the migration of most integration functionality to the er7_utils package, the DynBody class now extends the er7_utils::IntegrableObject class. For a more detailed explanation, see the Integration Model documentation.
- *Mathematical Functions* [14]. The Dynamic Body Model uses the Mathematical Functions to operate on vectors and matrices.
- *Memory Allocation Routines* [4]. The Dynamic Body Model uses the Memory Allocation Routines to allocate and deallocate memory.
- *Message Handling Class* [9]. The Dynamic Body Model uses the Message Handling Class to generate error and debug messages.
- *Named Item Routines* [10]. The Dynamic Body Model uses the Named Item Routines to construct names.
- *Quaternion* [5]. The Dynamic Body Model uses the Quaternion to operate on the quaternions embedded in the RefFrame objects and to compute the quaternion time derivative.
- *Reference Frame Model* [12]. The Dynamic Body Model makes quite extensive use of the Reference Frame Model. A DynBody's integration frame is a RefFrame object, and each of the body-based reference frames contained in a DynBody object is a BodyRefFrame object. The BodyRefFrame *is-a* RefFrame. The Dynamic Body Model uses the Reference Frame Model functionality to compute relative states.
- *Simulation Engine Interface Model* [11]. All classes defined by the Dynamic Body Model use the
`JEOD_MAKE_SIM_INTERFACES` macro defined by the Simulation Engine Interface Model to provide the behind-the-scenes interfaces needed by a simulation engine such as Trick.

3.2.2 Use of the Dynamic Body Model in JEOD

The following JEOD models use the Dynamic Body Model:

- *Body Action Model* [6]. The Body Action Model provides several classes that operate on a DynBody object. The DynBodyInit class and its derivatives initialize a DynBody object's state. The DynBodyFrameSwitch class provides a convenient mechanism for switching a DynBody object's integration frame. The BodyAttach and BodyDetach classes indirectly operate on a DynBody object through the Dynamic Body Model attach/detach mechanisms.

- *Derived State Model* [17]. The primary purpose of the Derived State Model is to express a DynBody state in some other form.

- *Dynamics Manager Model* [2]. The Dynamics Manager Model drives the integration of all DynBody objects registered with the dynamics manager.

- *Gravity Model* [16]. The Gravity Model computes the gravitational acceleration experienced by a DynBody object.

- *Gravity Gradient Torque Model* [15]. The Gravity Gradient Torque Model computes the gravity gradient torque exerted on a DynBody object, with the resultant torque expressed in the DynBody object's structural frame.

3.2.3 Interactions with Trick

The Dynamic Body Model interacts with Trick in both an ordinary and extraordinary manner. Many models are designed to be used in the Trick environment in the form of data declarations and function calls placed in an S_define file. The Dynamic Body Model is no different from other models in this regard.

The model also interacts with Trick in an extraordinary manner. The *vcollect* mechanism was designed as an object-oriented replacement for the old-style *collect* mechanism. The syntax for the Trick *vcollect* statement is `vcollect container converter { [item [,item]...]};`

The `container` is an STL sequence container object (an STL deque, list, or vector) that implements the `push_back` method. The `converter` is a function, possibly overloaded, that converts each `item` in the item list into a form suitable for pushing onto the end of the `container`. A `vcollect` statement results in a series of calls to `container.push_back()`. For example, the following code is from the SIM_force_torque verification simulation S_define:

```
vcollect body2.body.collect.collect_environ_forc CollectForce::create {
   body2.environ_forc1,
   body2.environ_forc2
};
```

The generated code that corresponds to this `vcollect` statement is

```
trick->body2.body.collect.collect_environ_forc.push_back(
   CollectForce::create( trick->body2.environ_forc1 ) );
trick->body2.body.collect.collect_environ_forc.push_back(
   CollectForce::create( trick->body2.environ_forc2 ) );
```

3.2.4 Interaction Requirements on the Mass Body Model

The concepts discussed in section 3.1.1.1 conform with the requirements levied on the Mass Body Model. The DynBody class includes a public attribute MassBody class such that the MassBody class must declare friendship with the DynBody class to properly propagate states. The MassBody must also include a constant DynBody pointer populated at construction to ensure that restricted operations are not applied to a DynBody's attribute MassBody. These designs are particularly necessary for the attach and detach mechanisms as discussed in section 3.1.1.5. The attach and detach mechanisms must be, and were, designed with such that extension of the DynBody class will not interupt functionality.

3.2.5 Interaction Requirements on the Dynamics Manager Model

The Dynamic Body Model uses the Dynamics Manager Model as a (1) a name register of DynBody objects, (2) a name register and subscription manager of RefFrame objects, and (3) a name register and validator of integration frames. All of these dependencies place functional requirements on the Dynamics Manager Model. The initialization concepts described in sections 3.1.1.6 and 3.1.1.7 also place functional requirements on the Dynamics Manager Model.

The integration scheme implemented in the Dynamic Body Model implicitly assumes that a higher-level function controls the integration. The model's `integrate()` methods cannot be called as integration-class jobs from a Trick S_define file. The higher-level function that controls the integration is a part of the Dynamics Manager Model, and this functionality is addressed as requirements on that model.

3.2.6 Interaction Requirements on the Body Action Model

The initialization concepts described in sections 3.1.1.6 and 3.1.1.7 place functional requirements on the Body Action Model.

3.2.7 Interaction Requirements on the Integration Model

The Integration Model is required to, and was designed to, integrate state and time separately. This requirement was levied on the Integration Model precisely because of the way in which integration is performed within JEOD.

3.3 Mathematical Formulation

This section summarizes key equations used in the implementation of the Dynamic Body Model. The outline of this section is organized along the lines of the key concepts described in section 3.1.1. Only those concepts that involve mathematical details are described in this section.

3.3.1 Attach/Detach

The underlying MassBody class addresses the geometry of attachment and detachment, with BodyRefFrames constructed for MassPoints as necessary. The DynBody class needs to address state as well, and must do this in a manner consistent with the laws of physics (if possible). This is not always possible with attachment as the attach model allows physically impossible attachments to take place: The attaching bodies instantaneously snap into place upon attachment. For example, the instantaneous attachment of a MassBody or MassBody derived state (which does not have a dynamic state) cannot conserve momentum. The rationale for instantaneous snap attachment in general is that modeling the detailed contact forces and torques that take place during the docking of two vehicles is beyond the scope of many orbital simulations. The simulation instead brings the vehicles into very close proximity and alignment and then magic happens: The vehicles attach. Linear momentum can always be conserved, even in non-physical attachments. Angular momentum is more problematic. The model conserves angular momentum in the frame that is centered at and moving with the post-docking combined center of mass. This choice is consistent with the laws of physics when the attachment does not involve a step change in position and orientation.

Conservation of linear momentum determines the velocity of the combined vehicle. Equating the linear momentum before and after the attachment yields

$$(m_p + m_c)\boldsymbol{v}_{I+} = m_p\boldsymbol{v}_{Ip_-} + m_c\boldsymbol{v}_{Ic-} \tag{3.1}$$

where m_p and m_c are the pre-attachment masses of the parent and child bodies, $\boldsymbol{v}_{Ip_-}$ and $\boldsymbol{v}_{Ic_-}$ are the pre-attachment velocities of the parent and child bodies in the parent body's integration frame, and $\boldsymbol{v}_{I+}$ is the post-attachment velocity of the combined center of mass. Solving for the post-attachment velocity yields

$$\boldsymbol{v}_{I+} = \frac{m_p\boldsymbol{v}_{Ip_-} + m_c\boldsymbol{v}_{Ic-}}{m_p + m_c} \tag{3.2}$$

The pre-attachment angular momentum in the non-rotating frame with origin at the combined center of of mass $\boldsymbol{r}_{I+}$ and moving at velocity $\boldsymbol{v}_{I}+$ results from translation with respect to this origin and body rotations:

$$\begin{aligned}\boldsymbol{L}_{I-} = &m_p(\boldsymbol{r}_{Ip_-} - \boldsymbol{r}_{I+}) \times (\boldsymbol{v}_{Ip_-} - \boldsymbol{v}_{I+}) + \boldsymbol{T}_{I\to Bp}^{\top}(\boldsymbol{I}_p\,\boldsymbol{\omega}_{Bp_-}) + \\ &m_c(\boldsymbol{r}_{Ic_-} - \boldsymbol{r}_{I+}) \times (\boldsymbol{v}_{Ic_-} - \boldsymbol{v}_{I+}) + \boldsymbol{T}_{I\to Bc}^{\top}(\boldsymbol{I}_c\,\boldsymbol{\omega}_{Bc_-})\end{aligned} \tag{3.3}$$

Transforming to the parent body frame,

$$
\begin{aligned}
\boldsymbol{L}_{B_p-} = & \boldsymbol{T}_{I\to Bp}\left(m_p(\boldsymbol{r}_{Ip-} - \boldsymbol{r}_{I+}) \times (\boldsymbol{v}_{Ip-} - \boldsymbol{v}_{I+})\right) + \\
& \boldsymbol{I}_p\,\boldsymbol{\omega}_{Bp-} + \\
& \boldsymbol{T}_{I\to Bp}\left(m_c(\boldsymbol{r}_{Ic-} - \boldsymbol{r}_{I+}) \times (\boldsymbol{v}_{Ic-} - \boldsymbol{v}_{I+})\right) + \\
& \boldsymbol{T}_{I\to Bp}\boldsymbol{T}^{\top}_{I\to Bc}(\boldsymbol{I}_c\,\boldsymbol{\omega}_{Bc-})
\end{aligned}
\tag{3.4}
$$

The post-attachment orientation post-attachment angular momentum in this frame results from rotation only:

$$
\boldsymbol{L}_{B_p+} = \boldsymbol{I}_+\,\boldsymbol{\omega}_{Bp+} \tag{3.5}
$$

Equating the pre- and post-angular momenta to conserve angular momentum and solving for the post-attachment angular velocity yields

$$
\boldsymbol{\omega}_{Bp+} = {\boldsymbol{I}_+}^{-1}\,\boldsymbol{L}_{B_p-} \tag{3.6}
$$

Detachment is much simpler. The only state that needs to change is that of the parent composite body. The child body's state does not change with detachment; it simply flies away from the parent with no instantaneous change in any state element. The parent body's state is almost correct. The parent similarly flies away from the child with no instantaneous changes in any of the primitive states. The composite state is incorrect and needs to be updated. Any primitive state can be used as the basis for this update. The mathematics that underlies state update is described in section 3.3.6.

3.3.2 State Initialization

The model participates in the state initialization process by setting state elements and propagating the set states throughout the DynBody tree. The mathematics regarding this state propagation is described in section 3.3.6.

3.3.3 Force and Torque Collection

The total effector force acting on a parent body is calculated per the superposition principle:

$$
\boldsymbol{F}_{P_S\,\text{eff,tot}} = \sum_{\text{vector } i} \boldsymbol{F}_{P_S\,\text{eff,i}} + \sum_{\text{child } i} \boldsymbol{T}^{\top}_{P_S\to C_S}\,\boldsymbol{F}_{C_S\,\text{eff,tot}} \tag{3.7}
$$

where P_S is the parent body's structural frame, the first sum is over the collected effector forces for the parent body, and the second sum is over the child bodies attached to the parent body.

The total environmental force is calculated similarly while the total non-transmitted force omits the sum over the child bodies. The total force on a root body is the sum of the effector, environmental, and non-transmitted forces on that body.

$$
\boldsymbol{F}_{S\,\text{tot}} = \boldsymbol{F}_{S\,\text{eff,tot}} + \boldsymbol{F}_{S\,\text{env,tot}} + \boldsymbol{F}_{S\,\text{non-xmit,tot}} \tag{3.8}
$$

The equations of motion need the force in the inertial frame:

$$\boldsymbol{F}_{I\,\text{tot}} = \boldsymbol{T}_{I\to S}^{\top}\,\boldsymbol{F}_{S\,\text{tot}} \tag{3.9}$$

Torque calculations proceed similarly, but with the additional twist that non-centerline forces produce a torque. The total effector torque acting on a parent body is

$$\boldsymbol{\tau}_{P_S\,\text{eff,tot}} = \sum_{\text{vector } i} \boldsymbol{\tau}_{P_S\,\text{eff,i}} + \sum_{\text{child } i} \boldsymbol{T}_{P_S\to C_S}^{\top}\,\boldsymbol{\tau}_{C_S\,\text{eff,tot}} + \boldsymbol{r}_{P_S:P_{cm}\to C_{cm}} \times \left(\boldsymbol{T}_{P_S\to C_S}^{\top}\,\boldsymbol{F}_{C_S\,\text{eff,tot}}\right) \tag{3.10}$$

The total environmental torque is calculated similarly while the total non-transmitted torque omits the sum over the child bodies. The total torque on a root body is the sum of the effector, environmental, and non-transmitted torques on that body.

$$\boldsymbol{\tau}_{S\,\text{tot}} = \boldsymbol{\tau}_{S\,\text{eff,tot}} + \boldsymbol{\tau}_{S\,\text{env,tot}} + \boldsymbol{\tau}_{S\,\text{non-xmit,tot}} \tag{3.11}$$

The equations of motion need the torque in the body frame:

$$\boldsymbol{\tau}_{B\,\text{tot}} = \boldsymbol{T}_{S\to B}\,\boldsymbol{\tau}_{S\,\text{tot}} \tag{3.12}$$

3.3.4 Equations of Motion

Equation (3.9) combined with Newton's second law yields the acceleration due to non-gravitational forces:

$$\boldsymbol{a}_{I\,\text{non-grav,tot}} = \frac{1}{m}\boldsymbol{F}_{I\,\text{tot}} \tag{3.13}$$

The Gravity Model computes the apparent gravitational acceleration, the total gravitational acceleration acting on the body less the gravitational acceleration of the integration frame. Combining this with the non-gravitational acceleration yields the total acceleration of the body with respect to the integration frame.

$$\boldsymbol{a}_{I:I\to B} = \boldsymbol{a}_{I\,\text{non-grav,tot}} + \boldsymbol{a}_{I\,\text{grav,tot}} \tag{3.14}$$

The rotational equations of motion differ conceptually from the above translational equations of motion in two regards. That angular momentum is expressed in the body frame adds the complicating factor of computing derivatives in a rotating frame. That is offset by lack of gravitation as a specific concern, which simplifies the the rotational equations of motion a bit.

The rotational analog of Newton's second law states that the time derivative of the angular momentum vector as expressed in a non-rotating frame is simply the external torque:

$$\dot{\boldsymbol{L}}_I = \boldsymbol{\tau}_{I\,\text{tot}} \tag{3.15}$$

The angular momentum is readily computed in the body frame as the product of the inertia tensor and the body-referenced angular velocity:

$$\boldsymbol{L}_B = \boldsymbol{I}_B \, \boldsymbol{\omega}_B \tag{3.16}$$

The time derivative of the body-referenced angular momentum and the inertial-referenced angular momentum are related by the inertial torque:

$$\begin{aligned} \dot{\boldsymbol{L}}_B &= \boldsymbol{T}_{I \to B} \, \dot{\boldsymbol{L}}_I - \boldsymbol{\omega}_{B:I \to B} \times \boldsymbol{L}_B \\ &= \boldsymbol{\tau}_{B\text{tot}} - \boldsymbol{\omega}_{B:I \to B} \times \boldsymbol{L}_B \end{aligned} \tag{3.17}$$

Assuming the inertia tensor is constant (or nearly so) in the body frame,

$$\begin{aligned} \dot{\boldsymbol{L}}_B &= \frac{d}{dt} \left(\boldsymbol{I}_B \, \boldsymbol{\omega}_B \right) \\ &\approx \boldsymbol{I}_B \, \dot{\boldsymbol{\omega}}_B \end{aligned} \tag{3.18}$$

Combining the above yields the time derivative of the body-referenced angular momentum:

$$\dot{\boldsymbol{\omega}}_B = {\boldsymbol{I}_B}^{-1} \left(\boldsymbol{\tau}_{B\text{tot}} - \boldsymbol{\omega}_{B:I \to B} \times \boldsymbol{L}_B \right) \tag{3.19}$$

3.3.5 State Integration

This model uses the *Integration Model* [3] to perform the state integration. See the documentation for that model regarding the mathematics that underlies numerical integration.

3.3.6 State Propagation

The initialized state and the integrated state need to be propagated throughout a DynBody object, including to other DynBody objects attached to the object in question. The propagation from one frame to another must be consistent with the local reference frames embodied in the MassBasicPoints that are associated with the DynBodyFrames.

The model performs state propagation in terms of pairs of reference frames. The mathematics of the propagation of the state from frame B to frame C depends on whether the underlying MassBasicPoint is expressed as B to C or C to B. Different methods address these as forward and reverse propagations.

Forward propagation involves computing the state of frame C with respect to frame A (the integration frame) given the state of frame B with respect to frame A and the local transformation from frame B to frame C. Note that this local transformation is a part of the MassBasicPoint object and hence does not contain derivative data. This local transformation is assumed to be constant with respect to time.

$$\boldsymbol{T}_{A\to C} = \boldsymbol{T}_{B\to C}\,\boldsymbol{T}_{A\to B} \qquad \text{Orientation} \qquad (3.20)$$

$$\boldsymbol{\omega}_{C:A\to C} = \boldsymbol{T}_{B\to C}\,\boldsymbol{\omega}_{B:A\to B} \qquad \text{Angular velocity} \qquad (3.21)$$

$$\boldsymbol{r}_{A:A\to C} = \boldsymbol{r}_{A:A\to B} + \boldsymbol{T}^{\top}_{A\to B}\,\boldsymbol{r}_{B:B\to C} \qquad \text{Position} \qquad (3.22)$$

$$\boldsymbol{v}_{A:A\to C} = \boldsymbol{v}_{A:A\to B} + \boldsymbol{T}^{\top}_{A\to B}\,(\boldsymbol{\omega}_{B:A\to B} \times \boldsymbol{r}_{B:B\to C}) \qquad \text{Velocity} \qquad (3.23)$$

Reverse propagation involves computing the state of frame C with respect to frame A (the integration frame) given the state of frame B with respect to frame A and the local transformation from frame C to frame B. In other words, the local transformation is in the reverse direction of the desired propagation.

$$\boldsymbol{T}_{A\to C} = \boldsymbol{T}^{\top}_{C\to B}\,\boldsymbol{T}_{A\to B} \qquad \text{Orientation} \qquad (3.24)$$

$$\boldsymbol{\omega}_{C:A\to C} = \boldsymbol{T}^{\top}_{C\to B}\,\boldsymbol{\omega}_{B:A\to B} \qquad \text{Angular velocity} \qquad (3.25)$$

$$\boldsymbol{r}_{A:A\to C} = \boldsymbol{r}_{A:A\to B} - \boldsymbol{T}^{\top}_{A\to C}\,\boldsymbol{r}_{C:C\to B} \qquad \text{Position} \qquad (3.26)$$

$$\boldsymbol{v}_{A:A\to C} = \boldsymbol{v}_{A:A\to B} - \boldsymbol{T}^{\top}_{A\to C}\,(\boldsymbol{\omega}_{C:A\to C} \times \boldsymbol{r}_{C:C\to B}) \qquad \text{Velocity} \qquad (3.27)$$

3.3.7 Point Acceleration

The following assumes a rigid body connection between the composite center of mass and a vehicle point. Different mathematics are needed if this assumption is not valid.

With this assumption, the rotational acceleration at the vehicle point is equal to the rotational acceleration at the center of mass. The point however has its own reference frame, and the rotational acceleration needs to be expressed in this point frame.

$$\dot{\boldsymbol{\omega}}_{\mathrm{pt}:I\to\mathrm{pt}} = \boldsymbol{T}_{B\to\mathrm{pt}}\,\dot{\boldsymbol{\omega}}_{B:I\to B} \qquad (3.28)$$

With the rigid body assumption, the total translation acceleration at the point is the total translation at the center of mass plus the translational acceleration due to rotational motion. In the rotating composite body frame, this acceleration is

$$\Delta\boldsymbol{a}_{\mathrm{rot}\,B} = \boldsymbol{\omega}_{B:I\to B} \times (\boldsymbol{\omega}_{B:I\to B} \times \boldsymbol{r}_{B:B\to\mathrm{pt}}) + \boldsymbol{\omega}_{B:I\to B} \times \boldsymbol{r}_{B:B\to\mathrm{pt}} \qquad (3.29)$$

Transforming to inertial,

$$\Delta\boldsymbol{a}_{\mathrm{rot}\,I} = \boldsymbol{T}^{\top}_{I\to B}\,\Delta\boldsymbol{a}_{\mathrm{rot}\,B} \qquad (3.30)$$

The total acceleration at the point is

$$\boldsymbol{a}_{\mathrm{pt}\,I} = \boldsymbol{a}_I + \Delta\boldsymbol{a}_{\mathrm{rot}\,I} \qquad (3.31)$$

Calculating the non-gravitational acceleration is a bit more complex. The total acceleration at the point comprises gravitational and non-gravitational components, as does the total acceleration at

the composite center of mass.

$$\boldsymbol{a}_{\mathrm{pt}\,I} = \boldsymbol{a}_{\mathrm{pt,non\text{-}grav}\,I} + \boldsymbol{a}_{\mathrm{pt,grav}\,I} \tag{3.32}$$

$$\boldsymbol{a}_{\boldsymbol{B}\,I} = \boldsymbol{a}_{\boldsymbol{B},\mathrm{non\text{-}grav}\,I} + \boldsymbol{a}_{\boldsymbol{B},\mathrm{grav}\,I} \tag{3.33}$$

The gravitational acceleration at the point is that at the center of mass plus a delta due to the separation between the center of mass and the point. Assuming a small separation between the center of mass and the point, the gravity gradient yields an excellent estimate of this delta gravitational acceleration.

$$\boldsymbol{a}_{\mathrm{pt,grav}\,I} = \boldsymbol{a}_{\boldsymbol{B},\mathrm{grav}\,I} + \Delta\boldsymbol{a}_{\boldsymbol{pt},\mathrm{grav}\,I} \tag{3.34}$$

$$\Delta\boldsymbol{a}_{\mathrm{pt,grav}\,I} \approx \nabla\boldsymbol{a}_{\boldsymbol{B},\mathrm{grav}\,I}\left(\boldsymbol{T}^{\top}_{I\to B}\,\boldsymbol{r}_{B:B\to\mathrm{pt}}\right) \tag{3.35}$$

Combining equations 3.31 to 3.35 and solving for the non-gravitational acceleration at the point,

$$\boldsymbol{a}_{\mathrm{pt,non\text{-}grav}\,I} = \boldsymbol{a}_{\boldsymbol{B},\mathrm{non\text{-}grav}\,I} + \Delta\boldsymbol{a}_{\mathrm{rot}\,I} - \Delta\boldsymbol{a}_{\mathrm{pt,grav}\,I} \tag{3.36}$$

3.4 Detailed Design

The classes and methods of the Dynamic Body Model are described in detail in the *Dynamic Body Model API* [1].

The outline of this section is organized along the lines of the key concepts described in section 3.1.1. Only those concepts that impact the detailed design are described in this section.

3.4.1 Mass

A DynBody *has-a* public MassBody attribute with friendship access declared in MassBody.

In dyn_body.hh

```
class DynBody : virtual public RefFrameOwner,
    virtual public er7_utils::IntegrableObject {
...
public:
...
    MassBody mass;
...
};
```

In mass_body.hh

```
class MassBody {
...
    friend class DynBody;
...
};
```

Note that the DynBody class uses multiple inheritance. The RefFrameOwner class is merely an abstract class for defining interface taxonomy. The MassBody class is a full-fledged class. The public MassBody friendship functionality is passed to other classes and functions that use a DynBody object. The protected MassBody functionality is fully visible within a DynBody class but not within classes that derive from the DynBody class. Readers who wish to learn more about the MassBody functionality should refer to the *Mass Body Model* [13].

In previous versions of JEOD, a Simple6DofDynBody class inherited from the DynBody class to provide IntegrableObject functionality to DynBody. All of this functionality was collapsed into DynBody itself as of JEOD v4.0, because nearly all JEOD use cases implemented Simple6DofDynBody, making intermediate inheritance unnecessary.

3.4.2 State

The DynBody class contains three BodyRefFrame objects that correspond to the three MassProperties objects defined in the MassBody class. These are (1) `core_body`, which represents the state

of the MassBody's `core_properties` object, (2) `composite_body`, which represents the state of the MassBody's `composite_properties` object, and (3) `structure`, which represents the state of the MassBody's `structure_point` object. The connection between these three three BodyRefFrame objects and the corresponding MassProperties object is made by the DynBody constructor. Further BodyRefFrame's corresponding to the MassBody's MassPoints may be constructed through attachment.

3.4.3 Inertial Frames

The Dynamics Manager Model, with the help of other models, determines which frames are 'inertial' frames and which are not. The Dynamic Body Model uses the Dynamics Manager Model public interfaces that pertain to this concept.

3.4.4 Integration Frame

The user sets the integration frame *by name* at initialization time. The initialization-time method `DynBody::initialize_model()` sets the integration frame based on the provided name.

All attached DynBody objects share a common integration frame. The attach mechanisms described below ensure that this is the case.

The model provides public and protected methods to switch the integration frame. The public methods are overloaded versions of `switch_integration_frames()`. This is the interface that should be used to switch the integration frames. These methods ensure that the state is preserved when switching frames and propagate the change throughout the DynBody object. The protected methods are overloaded versions of `set_integ_frame()`. These methods do not preserve state and are for internal use only.

3.4.5 Attach/Detach

The DynBody attach/detach mechanism separately performs the actions encompassed by MassBody attach/mechanisms, with further extensions to verify validity and perform dynamics updates (e.g. update the composite BodyRefFrame). The mathematics of this is described in section 3.3.1. None of the attach methods are intended to be overridden by derived classes in order to preserve core functionality. Extensibility can be achieved by manipulating configuration before or after invoking these interface methods.

The top level methods used to attach and detach DynBody instances are:

- `DynBody::attach_child()` to specify a DynBody to attach as a child,
- `DynBody::attach_to()` to specify a DynBody to attach as a parent,
- `DynBody::attach_to_frame()` to specify a RefFrame to kinematically attach as a parent,
- `DynBody::add_mass_body()` to specify a MassBody or linkage of MassBodys to attach as child(ren),

- `DynBody::detach()` to order this body to detach from its parent or detach another DynBody from this body,
- `DynBody::remove_mass_body()` to order this body to detach child(ren) MassBodys.

Attachment orientation is specified in one of two ways: **explicitly** by relative position/orientation between the body structures or **implicitly** by names of body points to mate (with orientation defined as opposing in X and aligned in Y w.r.t. point frames). Momentum conservation is performed only if both the child and parent are two DynBody objects and have fully initialized states. Two other cases are possible:

- The parent is a RefFrame and the attachment is treated kinematically.
- Neither the parent nor child state has been initialized. This occurs during initialization-time attachments, which intentionally occur prior to state initialization. Only the MassBody properties are updated in this case.
- The parent has a fully initialized state but the child has no state information. This occurs when a simple MassBody is attached to a DynBody after initialization. The MassBody magically appears alongside the DynBody. The attachment is assumed to occur with no change to the parent body's core and structural states. State is propagated from the parent body's structural frame.

The attach mechanism invokes protected intermediate functions to organize the attachment process. Several of these functions belong to MassBody and are exploited by DynBody through friendship. These methods are

- `DynBody::attach_update_properties()` to perform dynamic attachment,
- `MassBody::attach_establish_links()` to update searchable tree linkage,
- `MassBody::detach_establish_links()` to remove tree linkage,
- `MassBody::attach_update_properties()` to combine the MassBody's internal properties,
- `MassBody::attach_update_properties()` to separate the MassBody's internal properties, and
- `MassPoint::compute_state_wrt_pred()` to perform relative geometry calculations.

These functions ensure that the attribute MassBody have correct composite properties and linkages. The DynBody attach/detach methods also invoke validation checks during the process to verify the commanded attachment is valid. These methods are

- `attach_validate_parent()`, which is sent to the child body with the parent body supplied as an argument;
- `attach_validate_child()`, which is sent to the parent body with the child body supplied as an argument;

- `add_mass_body_validate()`, which is called by the DynBody with the child MassBody supplied as an argument;

The validation methods ensure that the bodies are registered with JEOD, will not create a nodal attach tree such that any node is multiply or cyclically attached, and that the supplied arguments do not refer to invalid types (e.g., a DynBody may not attach as a child to a MassBody parent).

3.4.6 Initialization

Initialization of a DynBody object is a complex process that is spread amongst constructors, S_define-level initialization methods, and the body action-based initialization mechanism. All elements of a DynBody object are initialized by the constructors. Pointers (including essential ones) are set to NULL and states are set to a default value. A few known, non-null initializations occur during this zeroth phase of the initialization process.

The next step in the process is to perform model-level initializations. This is performed by the DynBody method `initialize_model()`. This method registers the DynBody object and the basic reference frames with the dynamics manager and sets the object's integration frame. This step leaves the mass properties and states uninitialized.

The final step of the initialization process is best performed using the body-action based initialization mechanism. This operates in three stages, with mass properties and mass points initialized first, initialization-time attachments performed next, and body states initialized last.

3.4.7 State Initialization

The model participates in the recommended body action-based state initialization process in two ways, readiness checking and action application. The body action class DynBodyInit and its derived classes are responsible for state initialization, and both the readiness checking and action application involve interactions with the Dynamic Body Model. Readiness checking and action application are integral parts of the body action-based scheme. The simulation's dynamic manager invokes a body action's `apply()` method only if the action's `is_ready()` method assents to the forthcoming application. In the case of an action that derives from the DynBodyInit class, the `is_ready()` mechanism includes checks of whether required state elements in various BodyRefFrame objects have been initialized. The action is ready to be applied when all prerequisites have been satisfied.

The Dynamic Body Model involvement in the readiness checking centers around the BodyRefFrame class. This class augments the RefFrame class with an `initialized_items` data member of type RefFrameItems. A RefFrameItems object contains an enumerated value that indicates which of the position, velocity, orientation, and angular velocity elements of a state have been set and provides methods to operate on that enumerated value. The Dynamic Body Model methods described in section 3.4.16 ensure that a BodyRefFrame object's `initialized_items` are updated to reflect which of the object's state elements have been set. This mechanism makes readiness checking quite easy; all the DynBodyInit object needs to do is query the BodyRefFrame object's `initialized_items` data member as to whether all required state elements for that object have been set.

The Dynamic Body Model involvement in the application of a DynBodyInit object centers around the DynBody class. The `DynBodyInit::apply()` method class invokes the subject DynBody ob-

ject's `set_state()` method to set the user-specified state elements and then invokes the subject DynBody object's `propagate_state()` method to propagate that newly computed state throughout the DynBody tree. The former sets the user-specified state elements in a user-specified body reference frame; the latter propagates known state elements throughout the tree. For details on the `set_state()` and `propagate_state()` methods see see section 3.4.16 below.

3.4.8 Forces and Torques

Forces and torques in JEOD have always been, and continue to be, represented in the structural frame of the DynBody object with which the forces and torques are directly associated. The association between forces/torques and a DynBody object is achieved via the Trick *vcollect* mechanism. The responsibility to properly represent forces and torques in an appropriate structural frame falls on the developer of a particular force/torque model while the responsibility to properly associate the forces and torques with the correct DynBody object falls on the simulation integrator. From the perspective of the Dynamic Body Model, the collected forces and torques are simply the elements of the six STL collect vectors data members contained in the DynBody object's `collect` data member.

The `collect` data member is a BodyForceCollect object. The BodyForceCollect member data are:

- Six STL vectors, three each for forces and torques, representing the effector, environmental, and non-transmitted forces and torques acting on a body;
- Six correspondingly named 3-vectors[3] that represent the category-specific sums of of the active collected forces and torques;
- Two more 3-vectors that contain the total force and torque as represented in the structural frame, and
- Another two 3-vectors that contain the total force and torque transformed to the frame needed by the equations of motion. This equations of motion frame is the inertial integration frame for force and the body frame for torque.

The three STL vectors that represent forces contain CollectForce pointers. A CollectForce object contains two pointers, active and force. The latter is a 3-vector that contains the components of the force in structural coordinates. The former is a pointer to a boolean that indicates whether the force 3-vector should be included in the totals. The three STL vectors that represent torques contain CollectTorque vectors. A CollectTorque is analogous to a CollectForce, with a torque data member replacing the force data member. See section 3.4.10 below for details on the population of the six STL vectors in a BodyForceCollect object and construction of the CollectForce and CollectTorque objects.

The method `DynBody::collect_forces_and_torques()` is responsible for interpreting and manipulating the contents of the DynBody `collect` data member. This method first sums the members

[3]The overloading of the term 'vector' is a bit unfortunate here. The objects into which the forces and torques are collected are STL vectors; that name was chosen by the C++ community. JEOD uses 3-vectors to represent forces and torques (and positions, velocities, ...); that use of the word 'vector' predates the computer science use of the word by a considerable amount of time.

of the six collect vectors, storing the sums in the correspondingly named sum. For example, the effector forces are collected at the S_define-level into the `collect_effector_forc` collect vector; the sum of the active elements of this STL vector is computed and stored in the `effector_forc` data member. The method is then recursively invoked on the DynBody objects attached as children to the DynBody object in question. The next stage of operation depends on whether the DynBody object is an attached or root body.

In the case of attached bodies, the transmittable forces and torques are converted to the parent body's structural frame and added to the parent's total. For example, a child body's total effector torque is converted to the parent body's structural frame and added to the parent body's `collect.effector_torq` data member. Forces are treated similarly, but with the added complexity that a child body force becomes a force and a torque when transmitted to the parent body.

In the case of a root body, the categorized totals for forces and torques are summed to form the grand total external force and external torque. These initial grand totals are expressed in structural components. These structurally-referenced grand totals are then transformed to the frame in which the equations of motion are represented—the inertial integration frame in the case of force but the body frame in the case of torque. The final step is to formulate the equations of motion for the composite body. The total inertial-referenced external force is combined with gravitational acceleration to form the translational equations of motion. The total body-referenced external torque is combined with the inertial torque to form the rotational equations of motion. The mathematical details of these operations are described in section 3.3.3 and 3.3.4.

3.4.9 Wrenches

Like forces and torques, wrenches in JEOD are represented in the structural frame of the DynBody object with which the wrenches are directly associated. Also like forces and torques, the association between wrenches and a DynBody object is achieved via the Trick *vcollect* mechanism. Unlike forces and torques, the collection mechanism for wrenches accounts for how forces induce torques if the line of action of the force does not pass through the vehicle's center of mass. This is what makes wrenches so appealing: They simplify the development of vehicle-specific models. A modeler no longer needs to calculate the torque induced by a force; the wrench mechanism does that for them.

From the perspective of the Dynamic Body Model, the collected wrenches add to the collected forces and torques (this older concept will not be abandoned), but with all of the individual wrenches displaced to the vehicle center of mass during the collection process. To keep the concepts somewhat distinct, JEOD 3.4 uses a separate object for collecting wrenches. This is the `effector_wrench_collection` data member of StructureIntegratedDynBody. As of JEOD 3.4, wrenches can only be used in the context of this class. The DynBody concepts of non-transmitted and environmental forces and torques have not yet been propagated to wrenches. This may change in future releases.

The class Wrench provides `operator+=` to simplify the wrench collection process. This function transforms the summand wrench to be added to the point of the application of the sum and then adds the force and transformed torque to the force and torque of the sum.

3.4.10 Collect Mechanism

The elements of the six collect vectors in a BodyForceCollect object are populated by the Trick *vcollect* mechanism. This mechanism uses a *converter* to convert the item list elements to a form suitable for pushing on the STL *container*; see section 3.2.3 for details on the Trick side of this mechanism. This mechanism is used in a JEOD-based simulation to add forces and torques to one of the collect vectors contained in a DynBody object's collect data member. The *container* is one of the six BodyForceCollect collect vectors; the *converter* is `CollectForce::create` for the force collect vectors but `CollectTorque::create` for the torque collect vectors.

The CollectForce and CollectTorque classes are Factory Method classes, with the `create()` methods forming overloaded named constructors that create CollectForce and CollectTorque objects. Each class provides five overloaded versions of `create()` distinguished by argument type. The `CollectForce::create()` methods are

- `static CollectForce * create (double * vec)`
  Creates a CInterfaceForce, a CollectForce derived class. The force pointer in the created CInterfaceForce object is the input `vec` pointer. The `active` pointer is a newly allocated boolean pointer with the contents set to true. (A CInterfaceForce is always active.) The CInterfaceForce destructor releases this allocated memory.

- `static CollectForce * create (Force & force)`
  Creates a CollectForce with the force and active pointers in the created CollectForce object pointing to the input Force object's force 3-vector and active flag.

- `static CollectForce * create (Force * force)`
  This behaves similarly to the previous method; the only distinction is that the argument is a pointer to rather than a reference to a Force object.

- `static CollectForce * create (CollectForce & force)`
  Creates a CollectForce via the CollectForce copy constructor.

- `static CollectForce * create (CollectForce * force)`
  This behaves similarly to the previous method; the only distinction is that the argument is a pointer to rather than a reference to a CollectForce object.

The `CollectTorque::create()` methods are analogous to those of CollectForce class.

The class StructureIntegratedDynBody augments the collected forces and torques with the forces and torques from the collected wrenches.

3.4.11 Gravity

The Dynamic Body Model does not compute the gravitation acceleration that acts on a DynBody object. The Dynamics Manager Model does this on the behalf of this model, and does so by invoking the Gravity Model. The S_define-level `DynManager::gravitation()` method invokes the `GravityManager::gravitation()` method for each root DynBody object. This architecture, while a bit duplicitous, eliminates a source of user errors, eliminates some rather ugly S_define code, and offers a significant speedup when bodies are attached (gravitation is computed for root bodies only).

The DynBody class contains a `GravityInteraction grav_interaction` data member. This data member is passed, along with the DynBody object's position vector, as arguments to the GravityManager `gravitation()` method. The GravityInteraction data members include an array of GravityControls objects that direct the gravitational computations, a specification of the DynBody object's integration frame, and placeholders for the outputs from the gravitational computations. These output products are the gravitational potential, the gravitational acceleration vector, and the gravity gradient tensor.

Each element of the GravityControls array in a GravityInteraction object specifies a gravitational body such as the Sun or Earth and specifies how the gravitational computations are to be performed for that gravitational body. Since each DynBody object has its own GravityInteraction object, different DynBody objects can specify different computation techniques for a given gravitational body.

For example, consider a multi-vehicle simulation with one vehicle in low Earth orbit and another orbiting the Moon. The vehicle in low Earth orbit can treat the Moon as a point mass but use a high order model for the Earth. The vehicle in lunar orbit can use the reverse setup, with a high order model used for lunar gravity but a low order model for Earth gravity.

The gravity computations can also be changed over the course of a simulation. For example, consider a simulation that starts with a vehicle in Earth orbit, performs a trans-Mars burn, and eventually reaches Mars. The simulation might start with Mars disabled, a point mass model for the Sun, and a high order model for the Earth. The Earth's gravitational influence diminishes as the vehicle starts moving away from the Earth, and eventually becomes negligible when the vehicle is well outside the Earth's Hill sphere. This suggests toning down the fidelity of the Earth's gravity model and eventually disabling the Earth completely. The reverse situation holds as the vehicle approaches Mars.

3.4.12 Equations of Motion

Formulating the equations of motion is a part (a very small part) of the operations performed by `DynBody::collect_forces_and_torques()`. See section 3.4.8 for a description of this method.

3.4.13 State Integration

The DynBody class declares three virtual methods that pertain to state integration, which may be overriden by classes such as StructureIntegratedDynBody. Users are free to write classes that derive from DynBody or that provide alternate implementations.

The three methods are

`create_integrators` This initialization-time method creates the state integrators that will be used during run time to integrate state. The method receives pointer to an instance of a class that derives from the IntegratorConstructor class. The `create_integrators()` method uses this to create state integrators that implement the user-selected integration algorithm. The DynBody implementation creates two state integrators, one for the translational state and one for the rotational state. The created translational state integrator uses 3-vectors for

generalized position and velocity. The created rotational state integrator uses a 4-vector for generalized position and a 3-vector for generalized velocity.

`reset_integrators` Integrators that use historic data need to discard that historic data on occasion. This method simply forwards the `reset` on to the state integrators.

`integrate` This method performs the state integration. The DynBody implementation accomplishes this by calling two internal methods. One of these internal methods integrates the translational state of the DynBody object's integrated frame using the object's translational state integrator (which was created by the call to `create_integrators()`). The other integrates the integrated frame's rotational state using the rotational state integrator.

Most of the work is performed by the state integrators. See the *Integration Model* [3] for details.

3.4.14 Center of Mass State Integration

The translational and rotational equations of motion decouple at a rigid body's center of mass. The translational state equations of motion are given by Newton's second law, $\boldsymbol{F} = m\boldsymbol{a}$. JEOD separates gravitational interactions from external forces, resulting in $\boldsymbol{a}_{\text{net}} = \boldsymbol{a}_{\text{grav}} + \boldsymbol{F}_{\text{ext}}/m$. The inertial state equations of motion are given by the rotational equivalent of Newton's second law, with an additional term that results from working in what inherently is a rotating frame, $\boldsymbol{\tau} = I\dot{\boldsymbol{\omega}} + (\mathbf{I}\boldsymbol{\omega}) \times \boldsymbol{\omega}$.

3.4.15 Structural State Integration

The translational and rotational equations of motion are coupled elsewhere on a rigid body. The translational state equations of motion at a point removed from the center of mass must account for centrifugal and Euler accelerations: $\boldsymbol{a}_{\text{net}} = \boldsymbol{a}_{\text{grav}} + \boldsymbol{F}_{\text{ext}}/m + \boldsymbol{\omega} \times (\boldsymbol{\omega} \times \boldsymbol{r}) + \dot{\boldsymbol{\omega}} \times \boldsymbol{r}$. Rotational state is even messier. However, the equations of motion for rotational state simplify to exactly the same equations used for rotation about the center of mass. This shouldn't be surprising; angular velocity and angular acceleration are "free vectors": They're the same for any point on a rigid body. The class StructureIntegratedDynBody takes advantage of this. It uses the same equations of motion for rotational state as does DynBody. It does however account for coupling in the translational equations of motion.

3.4.16 State Propagation

The term 'state propagation' as used in this document and in the model code means propagating the state from a reference frame associated with a DynBody object to other reference frames associated with that DynBody object, either directly or via attachment, using the geometry information embedded in the MassBody objects that underlie the DynBody objects. State propagation is a topic distinct from state integration. State needs to be propagated from frame to frame for several reasons:

- During initialization. The user can set state elements in a staged manner, and can set different state elements in different frames. For example, a user might set the position and velocity of

a subject vehicle in terms of the relative position and velocity between a VehiclePoint on the subject vehicle and another VehiclePoint on some reference vehicle, but set the orientation and angular velocity in terms of the relative orientation and angular velocity of the subject vehicle's body frame with respect to the reference vehicle's LVLH frame.

- After integration. The state integration mechanism integrates the state of one reference frame, the root DynBody object's integrated frame. The states of all of the other reference frames associated with a DynBody tree is propagated from this integrated frame.

- During attachment. The attach mechanism calculates and sets the state of one reference frame. The states of all of the other reference frames associated with a DynBody tree is propagated from this updated frame.

- During detachment. The states of all reference frames associated with a DynBody tree is propagated from this parent body's core body reference frame.

- During frame switch. The state of the root body's integrated frame is updated to reflect the switch to a new integration frame. The states of all of the other reference frames associated with a DynBody tree is propagated from this integrated frame.

- Per some user-defined model. For example, consider a JEOD-based federate in a distributed multi-vehicle simulation. The federate integrates vehicle states over time, even the states of vehicles that are not owned by the federate. When the owner of a state broadcasts new state elements, the federate must override the integrated state with the new official values. The state propagation mechanism can be employed to accomplish this goal.

The DynBody class provides several public methods related to initializing/overriding states and propagating the resulting settings throughout the DynBody tree. These methods are listed below. All but the last three methods in the list below take two arguments, the value of that element and the reference frame for which the appropriate state element is to be set. The last listed method is particularly important. Please read the entire list. These methods are

- `set_position()` sets the position element of the state of the input reference frame to the input position and denotes that the input frame is the definitive source for the DynBody object's position.

- `set_velocity()` sets the velocity element of the input frame's state and denotes that frame as the definitive source for the DynBody object's velocity.

- `set_attitude_left_quaternion()` sets the attitude element of the input frame's state and denotes that frame as the definitive source for the DynBody object's velocity.

- `set_attitude_rate()` sets the angular velocity element of the input frame's state and denotes that frame as the definitive source for the DynBody object's angular velocity.

- `set_attitude_right_quaternion()` provides an alternate means for setting attitude. JEOD uses left quaternions; others use right quaternions. This method converts the input right quaternion to a left quaternion before setting the attitude element.

- `set_attitude_matrix()` provides yet another alternate means for setting attitude. This method converts the input transformation matrix to a left quaternion before setting the attitude element.

- `set_state()` sets multiple state elements at once. This method takes three arguments, a bit flag that specifies which state elements are to be set, a RefFrameState that contains the desired state values, and the subject reference frame. It sets selected elements of the input frame's state based on the input state. The bit flag specifies which elements are to be set. The input frame is denoted as the definitive source for all set elements.

- `set_state_source()` denotes the given reference frame to be the definitive source for a set of specified state elements. This method takes two arguments, a bit flag that specifies the state elements and the subject reference frame. This method does not set state. It is used when some particular frame's state is known to be correct but the states of other frames is incorrect. This occurs, for example, with detachment. Detachment makes the state of the composite body frame of the parent body incorrect. The state of the core body frame is correct, so the detach mechanism sets the core body to be the (temporary) source of state information.

- `propagate_state()` propagates state settings throughout the DynBody tree. This method takes no arguments. Calling any of the above `set_` methods creates a problematic situation. The state of one frame has been set, but this setting has not been propagated throughout the DynBody tree. States are inconsistent with geometry. This final method rectifies that situation. If possible, it propagates state from the denoted definitive sources to the integrated frame, and from there to all frames associated with the DynBody tree. Upon completion, the integrated frame becomes the definitive source for all state elements.

The method `propagate_state()` can always propagate state as described above once all four state elements (position, velocity, attitude, and angular velocity) have been set. This propagation is not always immediately possible during initialization due to the staged initialization scheme employed in JEOD. Individual state elements can be propagated, provided the propagation is performed in the order listed in table 3.1. The first column of the table lists the element to be propagated; the second column specifies which state elements must have already been initialized in order to perform the propagation.

Table 3.1: Initialization Time State Propagation

Element to be Propagated	**Required Elements**
Orientation	No constraints
Angular velocity	Orientation
Position	Orientation
Velocity	Orientation and angular velocity

3.4.17 Point Acceleration

The DynBody method `compute_vehicle_point_derivatives()` computes the derivatives of the velocity and angular velocity vector for a specified BodyRefFrame. The method uses the mathematics described in section 3.3.7 to calculate these derivatives.

The envisioned use of this method is to provide the truth data needed by an accelerometer or inertial measurement unit (IMU). The specified BodyRefFrame will then be the case frame associated with that accelerometer or IMU.

One caveat: This is a generic DynBody method. As such it makes the simplifying assumption that changes in mass properties are small and do not measurably impact dynamics.

3.5 Inventory

All Dynamic Body Model files are located in `${JEOD_HOME}/models/dynamics/dyn_body`. 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 describes how to use the Dynamic Body Model from the perspective of a simulation user, a simulation developer, and a model developer.

4.1 Analysis

4.1.1 Initialization

Some (a rather small portion) of a DynBody object's data members need to be initialized directly in the simulation input file. At a minimum, the object itself must be given a name, the object's integration frame must be specified by name, and flags need to be set to enable vehicle dynamics. Assuming a sim object named `veh_object` contains a DynBody element named `body`, The following will prepare the body for a 6 DOF Earth orbit simulation. One caveat: Please do not copy the following verbatim into your input file. A better name for the vehicle than "vehicle" most likely exists, and using Earth-centered inertial for a vehicle that is not orbiting the Earth is not a good choice for integration frames.

```
veh_object.body.set_name( "vehicle" );
veh_object.integ_frame_name = "Earth.inertial";
veh_object.translational_dynamics = true;
veh_object.rotational_dynamics = true;
```

In a situation where multiple bodies are connected, the default is such that translational and rotational accelerations are not computed for any child bodies. However instances where a body is, for example, a sensor and requires these values in order to provide properly modeled data, the option exists to force the computation of these derivatives for any child bodies of interest.

```
veh_object.compute_point_derivative = true;
```

Gravity is active in most simulations. The gravity controls for the vehicle need to be specified in the input file. For example, the following will make the body subject to gravitational influences

from the Earth, Sun, and Moon. Earth gravity uses an 8×8 gravity field while the Sun and Moon are point sources.

```
veh_object.body.grav_interaction.n_grav_controls = 3;
veh_object.body.grav_interaction.grav_controls = alloc(3);
```

```
// Earth gravity
veh_object.body.grav_interaction.grav_controls[0].source_name = "Earth";
veh_object.body.grav_interaction.grav_controls[0].active = true;
veh_object.body.grav_interaction.grav_controls[0].spherical = false;
veh_object.body.grav_interaction.grav_controls[0].degree = 8;
veh_object.body.grav_interaction.grav_controls[0].order = 8;
veh_object.body.grav_interaction.grav_controls[0].gradient = true;

// Sun gravity
veh_object.body.grav_interaction.grav_controls[1].source_name = "Sun";
veh_object.body.grav_interaction.grav_controls[1].active = true;
veh_object.body.grav_interaction.grav_controls[1].spherical = true;
veh_object.body.grav_interaction.grav_controls[1].gradient = false;

// Moon gravity
veh_object.body.grav_interaction.grav_controls[2].source_name = "Moon";
veh_object.body.grav_interaction.grav_controls[2].active = true;
veh_object.body.grav_interaction.grav_controls[2].spherical = true;
veh_object.body.grav_interaction.grav_controls[2].gradient = false;
```

See the *Gravity Model* [16] for a detailed description of the GravityInteraction class.

4.1.2 Logging

Several of the data members in a DynBody object, either directly or indirectly through the attribute MassBody, are amenable to logging. For example, adding the following to a recording group's reference specification will log a vehicle's mass, the position of its composite center of mass with respect to structure, and the composite state:

```
veh_object.body.composite_properties.mass
veh_object.body.composite_properties.position[0-2]
veh_object.body.composite_body.state.trans.position[0-2]
veh_object.body.composite_body.state.trans.velocity[0-2]
veh_object.body.composite_body.state.rot.Q_parent_this.scalar
veh_object.body.composite_body.state.rot.Q_parent_this.vector[0-2]
veh_object.body.composite_body.state.rot.ang_vel_this[0-2]
```

That said, logging the quaternion often is not particularly insightful. Tools outside of the Dynamic Body Model such as the EulerDerivedState class provide insight into a DynBody object's state. That class is a part of the *Derived State Model* [17]. Users are strongly encouraged to make use of that model as an analysis tool.

4.2 Integration

4.2.1 S_define-Level Integration

Given the complexity of the Dynamic Body Model, integrating it into an S_define file is a relatively simple task.

The first step is to declare a DynBody derivative as an element of some simulation object. At the very minimum, the object needs to be prepared for initialization in order to be used during simulation:

```
##include "dynamics/dyn_body/include/dyn_body.hh"
##include "dynamics/dyn_manager/include/dyn_manager.hh"
...
class VehicleSimObject : public Trick::SimObject
{
  public:
    jeod::DynBody dyn_body;
    ...
    VehicleSimObject( jeod::DynManager& dm )
      : dyn_manager(dm)
    {
      P_ENV ("initialization") dyn_body.initialize_model( dyn_manager );
    };
    ...
};
```

Note well: The above `DynBody::initialize_model()` call must take place (in time) between the call to the dyn_manager's `DynManager::initialize_model()` method and `DynManager::initialize_simulation` method. This constraint is best satisfied using Trick's job prioritization scheme, as shown, using the "P_ENV" phase defined in "lib/jeod/JEOD_S_modules/default_priority_settings.sm."

Real vehicles change in mass as they burn fuel. To make the state integration reflect reality the vehicle's mass mass properties need to be updated, typically at the simulation's dynamic rate:

```
...
#define DYNAMICS 0.005
...
class VehicleSimObject : public Trick::SimObject
{
  public:
    jeod::DynBody dyn_body;
    ...
    VehicleSimObject( jeod::DynManager& dm )
      : dyn_manager(dm)
    {
      P_ENV ("initialization") dyn_body.initialize_model( dyn_manager );
```

```
      (DYNAMICS, scheduled) dyn_body.mass.update_mass_properties();
    };
    ...
};
```

Surprisingly, the above two calls will be the only S_define-level calls that directly invoke model functionality even in a complex simulation. Not so surprisingly, a lot of the functionality is invoked by other models. The above do not initialize state, compute state derivatives, integrate state, or analyze state. Initialization is performed by the *Dynamics Manager Model* [2] and the *Body Action Model* [6]. The Dynamics Manager Model guides derivative calculation, including calls to the *Gravity Model* [16] to compute gravitational acceleration and to this model's force/torque collection method. The Dynamics Manager Model similarly drives the state integration process. Finally, the *Derived State Model* [17] provides a number of analysis tools.

The vehicle as of yet does not have any forces or torques acting on it. The modeling of forces and torques is beyond the scope of the Dynamic Body Model. Using those modeled forces and torques is the bread and butter of this model. The object merely needs to be told that those forces and torques exist. This is accomplished by using the Trick *vcollect* mechanism.

```
vcollect veh_object.dyn_body.collect.collect_environ_forc jeod::CollectForce::create {
  veh_object.force_model.force,
  veh_object.another_force_model.force
};
```

Note that the above collects the listed forces into the `collect_environ_forc` data member. Two other options exist for forces: The `collect_effector_forc` and `collect_no_xmit_forc`. Torques are collected similarly into one of three groups; just use '_torq' instead of '_forc' for the data member name and CollectTorque::create instead of CollectForce::create.

There is little difference between the environmental and effector forces, but there is a huge distinction between these first two and the non-transmitted forces. When a DynBody object is attached as a child to a parent DynBody object, the environmental and effector forces acting on a child body are transmitted to the parent body. The non-transmitted forces are not transmitted to the parent (hence the name).

So what are these non-transmitted forces and torques? The concept does not make a lick of sense from the perspective of physics. The reason for classifying a force or torque as 'non-transmitted' is that the force or torque does not make a lick of sense when a body is attached to another body. A couple of examples are aerodynamic drag and gravity gradient torque.

A similar collect mechanism exists for wrenches:

```
vcollect veh_object.dyn_body.effector_wrench_collection.collect_wrench {
  &veh_object.wrench_model.wrench,
  &veh_object.another_wrench_model.wrench
};
```

As of this release, there are no environmental or non-transmitted wrenches, and wrenches can only be used with structure-integrated dynamic bodies.

When two bodies combine to form a composite, the nearby presence of one vehicle can significantly alter the aerodynamic drag on the other vehicle. The best thing to do is to use a different drag model that truly represents the combined vehicle. Lacking this best solution, a fairly good approximation can sometimes be attained by using the drag on the larger vehicle only. A simple way to accomplish this is to make the smaller vehicle attach to the larger as a child and denote the aerodynamic drag on the smaller vehicle as a non-transmitted force. Denoting aerodynamic drag and torque as non-transmitted forces and torques is a widespread practice. Be aware that this is a partial solution. The best thing to do is to mark these as non-transmitted *and* to switch the parent body to a different drag model (*e.g.*, a composite plate model).

Gravity gradient torque is calculated using the composite mass properties of a DynBody object. This means that the gravity gradient torque calculated for the root body of a composite body automatically includes the contributions of all attached bodies; the result is correct *as-is*. Adding the gravity gradient torque exerted on child bodies would invalidate this already correct result. Gravity gradient torque must be categorized as a non-transmitted torque.

4.2.2 Integration Frame Changes with Attach/Detach

When two bodies attach, the child body takes on the integration frame of the parent body. When the bodies detach, the child body retains the integration frame of the parent body, rather than reverting to its original integration frame. This is because JEOD cannot safely assume that the original integration frame of the child body is still the most appropriate frame. On the contrary, two vehicles that were just attached are in close enough proximity that they would probably be best served by continuing to share the same integration frame.

An example scenario demonstrating best practices is shown below:

- There are two vehicles (vehA and vehB).
- vehA's integration frame is Moon.inertial and vehB's integration frame is Earth.inertial.
- As vehB approaches vehA, its integration frame should switch to Moon.inertial via a call to `DynBody::switch_integration_frames()`. This is to avoid any potential inconsistencies between two interacting vehicles using different integration frames. The results should be close either way, but this provides less opportunity for numerical errors to be introduced.
- vehB attaches to vehA and its integration frame is still Moon.inertial because it is set to be the same as its parent body, vehA.
- vehB detaches from vehA and its integration frame remains Moon.inertial as the vehicles are in close proximity immediately after they detach.
- Assuming no more interaction is required between the two vehicles, vehB's integration frame may be set to Earth.inertial through a call to `DynBody::switch_integration_frames()`.
- Due to the multiple integration frames in use, a RelativeStates instance may be set up to continue to track vehB relative to Earth.inertial for other models, displays, or graphics for consistency even as vehB's integration frame changes over the course of the simulation.

4.2.3 Using the Model in a User Defined Model

All of the public member data and member functions of the DynBody and MassBody classes are accessible to user models that operate on a DynBody object. Fully explaining that functionality is beyond the scope of this section of the document. Users who wish to do so should consult section 3 of this document, the doxygen documentation supplied with JEOD (the HTML pages are vastly superior to the PDF documents), the header files that define the classes, and if worse comes to worse, the code itself.

A few methods are worthy of note and are described in table 4.1.

Table 4.1: Methods of Note

Method	Description

`DynBody::attach_to()`
`DynBody::attach_child()`
`DynBody::attach_to_frame()`
`DynBody::attach()`
`DynBody::add_mass_body()`
	If you as a model developer know that two bodies or a body and a RefFrame are to be attached, and know they are to be attached immediately, invoke the attach method directly in your model code. There is no reason to create and enqueue a body action to perform the attachment.

`MassBody::detach()`
`DynBody::detach()`
`DynBody::remove_mass_body()`
	Detach is similar to attach in the sense that if you as a model developer know that one body is to detach from another, immediately, simply invoke the detach method in your model code.

`DynBody::set_state()`	You need to override state in a distributed multi-vehicle simulation? This method, or one of its kin (e.g., `set_attitude_rate()`), is the method to call.
`DynBody::propagate_state()`	Always make sure to end a sequence of state overrides with a call to `propagate_state()` to pass the updated state along to child bodies. Failure to do so will result in states inconsistent with geometry.

4.3 Extension

The model was designed with extensibility in mind. The DynBody class makes simplifying assumptions that are not appropriate in all circumstances, particularly launch. One solution is to write a user-defined extension to the DynBody class that bypasses these assumptions by augmenting functionality.

Chapter 5

Verification and Validation

5.1 Inspection

Inspection DynBody_1: Top-level Inspection

This document structure, the code, and associated files have been inspected, and together satisfy requirement DynBody_1.

Inspection DynBody_2: Design Inspection

Table 5.1 summarizes the key elements of the implementation of the Dynamic Body Model that satisfy requirements levied on the model. By inspection, the Dynamic Body Model satisfies requirements DynBody_2 to DynBody_9.

Table 5.1: Design Inspection

Requirement	**Satisfaction**
DynBody_2 Mass	The class DynBody utilizes functionality from the MassBody class through friendship. The friendship is not inheritable, thereby granting DynBody objects full access MassBody functionality while conserving core behavior for user-derived classes.
DynBody_3 Integration Frame	The DynBody class data member `integ_frame` points to the integration frame for a particular DynBody object. This member is set at initialization time by name. Upon attachment as a child to another DynBody object, the integration frame for the child bodies are set to that of the root body. The integration frame can be changed dynamically via the `switch_integration_frames()` member function.

Continued on next page

Table 5.1: Design Inspection (continued from previous page)

Requirement	Satisfaction
DynBody_4 State Representation	The class BodyRefFrame derives from the RefFrame class. The inheritance is public, thereby granting external users of a DynBody object full access to the public RefFrame functionality of the BodyRefFrame objects contained in a DynBody object. A DynBody object contains three BodyRefFrame objects that represent the structure frame, core body frame, and the composite body frame. In addition to these three basic frames, a DynBody object contains an STL list of BodyRefFrame pointers that represent additional points of interest associated with the DynBody object.
DynBody_5 Staged Initialization	The BodyRefFrame `initialized_items` data member indicates which elements of a BodyRefFrame object's state have been set. The Dynamic Body Model methods described in section 3.4.16 ensure that a BodyRefFrame object's `initialized_items` properly reflect which of the BodyRefFrame object's state elements have been set.
DynBody_6 Equations of Motion	This requirement has four sub-requirements that specify the treatment of forces, torques, translational acceleration, and rotational acceleration. Sections 3.4.8 to 3.4.12 describe the treatment of forces and torques in the Dynamic Body Model and how these lead to the development of the equations of motion.
DynBody_7 State Integration and Propagation	As described in section 3.4.13, the DynBody class provides the ability to integrate the composite body frame associated with the root body of a composite body or of an isolated (unconnected) body. The integrated state is propagated throughout a DynBody tree as described in section 3.4.16.
DynBody_8 Vehicle Points	The DynBody class overrides the MassBody `add_mass_point()` method. This override creates a BodyRefFrame that corresponds to the MassPoint, thereby making these registered points of interest have a corresponding state. State is propagated to all reference frames associated with a DynBody object, including these vehicle points. The DynBody method `compute_vehicle_point_derivatives()` computes accelerations for a specific vehicle point as required.

Continued on next page

Table 5.1: Design Inspection (continued from previous page)

Requirement	Satisfaction
DynBody_9 Attach/Detach	As described in section 3.4.5, the DynBody class implements attach/detach member functions in addition to the Massbody attach/detach methods to provide the the required capability to attach and detach DynBody objects and to do so in a manner consistent with the laws of physics. The DynBody class also provides attachment methods for attaching to a RefFrame object

Inspection DynBody_3: Mathematical Formulation

The algorithmic implementations of the methods that provide the functionality of the Dynamic Body Model follow the mathematics described in section 3.3.

By inspection, the Dynamic Body Model satisfies requirements DynBody_6, DynBody_8 and DynBody_7.

5.2 Validation

This section describes various tests conducted to verify and validate that the Dynamic Body Model satisfies the requirements levied against it. The tests described in this section are archived in the JEOD directories `models/dynamics/dyn_body/verif` and `models/dynamics/body_action/verif`.

Test DynBody_1: State Initialization

Background 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 directory `models/dynamics/body_action/verif/SIM_orbinit`
This simulation's S_define file defines two dynamic bodies and several objects that derive from the BodyAction class that initialize aspects of these dynamic bodies. The simulation has no dynamics; it merely performs the initializations and stops at t=0. Several tests provided with the simulation exercise these initialization capabilities. The script `run_tests.pl` located in the simulation's script directory invokes the tests and reports the results.

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 Tables 5.2 to 5.4 summarize the results of this test.

All tests pass.

Applicable requirements This test demonstrates the partial or complete satisfaction of the following requirements:

- DynBody_2. Mass properties are initialized as expected.
- DynBody_3. The vehicle integration frame is initialized as expected.
- DynBody_4. All required states are represented and populated.
- DynBody_5. Demonstrating proper state initialization is the primary purpose of this test.

Table 5.2: 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 5.3: 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 5.4: Attitude and Attitude Rate Test Cases

Run Directory	**Class Name** [a]	**Source Frame**	**Subj. Frame**	**Subj. Vehicle**	**Att. Err.** (d)	**Rate Err.** (d/s)	**Status**
RUN_1230	LvlhRot	Tgt. LVLH	Body	ISS	0.0	7.8×10^{-19}	Passed
RUN_2100	Rot	Inertial	Body	ISS	3.0×10^{-14}	0.0	Passed
RUN_3771	Lvlh	Tgt. LVLH	Body	STS 114	2.1×10^{-14}	1.2×10^{-17}	Passed
RUN_3822	Ned	Ref. NED	Struct	Pad 39A	1.8×10^{-14}	9.8×10^{-19}	Passed
RUN_4451	Rot	Tgt. Struct	Struct	STS 114	2.7×10^{-14}	2.2×10^{-17}	Passed
RUN_4681	NedRot	Tgt. NED	Struct	STS 114	1.4×10^{-14}	4.5×10^{-19}	Passed
RUN_5461	Rot	Tgt. Point	Point	STS 114	1.3×10^{-14}		Passed
	LvlhRot	Tgt. LVLH	Body			0.0	Passed

[a]See the footnote to table 5.3 regarding the abbreviated class naming convention.

Test DynBody_2: Attach/Detach

Background The primary purpose of this test is to test that the Dynamic Body Model attach and detach mechanisms operate as required. This includes satisfying the basic attach and detach requirements from the Mass Body Model and satisfying the new DynBody requirement to obey the conservation laws while attaching and detaching objects.

Test description This test places dynamic bodies in rather contrived circumstances to ensure the bodies come into alignment. The bodies attach at just the point in time when they do come into alignment. Bodies also detach in this test. The expectation is that the attachments should result in properly calculated mass properties and that attachments and detachments should conserve linear and angular momenta.

Test directory `models/dynamics/dyn_body/verif/SIM_verif_attach_detach`
This simulation defines three dynamic bodies and various Body Action Model attach/detach objects that make the simulation's dynamic bodies attach to and detach from one another.

The simulation contains two run directories. The `RUN_simple_attach_detach` directory uses only two of the dynamic bodies. The two bodies are configured to come into alignment ten seconds into the simulation, at which time they attach to one another. After another ten seconds the bodies detach. Then, a rotational spin is applied to body1 and the body2 is kinematically attached to a mass point on the body1. After some time, the spinning body is attached to the central point making all bodies attached to the integration frame with no motion at an offset. Finally, body1 is detached and reattached to the integration frame with no offset.

The `RUN_complex_attach_detach` directory uses all three vehicles. The simulation bodies combine in two steps to form a three-body composite body. The latter attachment specifies an attachment that, if performed as requested, would create a non-tree structure. One body detaches from this composite and reattaches later. Finally, the specific detachment capability is tested.

Success criteria Table 5.5 lists the items to be tested at specific events. Two types of criteria exist: Logical tests of the connectivity status, and numerical tests of mass properties and states. The connectivity must represent the attach/detach event that just occurred. The numerical results from the simulation should match analytic results to within numerical precision.

Test results All tests pass.

Applicable requirements This test demonstrates the partial or complete satisfaction of the following requirements:

- DynBody_2. Mass properties are properly initialized and are properly updated due to attachments and detachments.
- DynBody_6. Angular acceleration, including that due to inertial torque, is properly calculated.
- DynBody_7. The equations of motion are properly integrated.
- DynBody_8. Vehicle points are properly constructed and used during attachments.
- DynBody_9. Demonstrating this capability is the primary purpose of this test.

Table 5.5: Attach/Detach Test Criteria

Simple Attach/Detach

Time	Event	Test Items
0	Mass properties initialized States initialized	Body mass properties Body states
10	Bodies 1 and 2 come into alignment Body 1 attaches to body 2	Relative state Connectivity (Body 1 attached to 2) Composite mass properties Composite state
20	Body 1 detaches	Connectivity (Body 1 detached from 2) Body states
30	Body 1 begins rotational spin	Body states
35	Body 2 kinematic attach to Body 1	Connectivity (Body 2 attached to 1) No change to composite mass props for either body Body 2 relative to Body 1 is static Body 1 state continues undisturbed
40	Body 1 kinematic attach to Central Point	Connectivity (Body 1 attached to "Central Point") Body states are static
50	Body 1 detaches from the Central Point Body 1 kinematic attaches to the Central Point with no offset	Connectivity (Body 1 attached to "Central Point") Body 1 has a zero state Body states are static

Complex Attach/Detach

Time	Event	Test Items
0	Mass properties initialized States initialized	Body mass properties Body states
10	Bodies 1 and 2 come into alignment Body 1 attaches to body 2	Relative state Connectivity (Body 1 attached to 2) Composite mass properties Composite state
32.78	Bodies 1 and 3 come into alignment Body 1 attaches to body 3	Relative state Connectivity (Body 1 attached to 2 to 3) Composite mass properties Composite state
50	Body 1 detaches	Connectivity (Body 1 detached from 2) Body states
55	Bodies 1 and 2 come into alignment Body 1 attaches to body 2	Relative state Connectivity (Body 1 attached to 2 to 3) Composite mass properties Composite state

Test DynBody_3: Force/Torque

Background The primary purpose of this test is to test that external forces and torques are properly accumulated and propagated to parent bodies.

Test description This test applies forces and torques to a pair of joined dynamic bodies flying through otherwise empty space. The situation is contrived to test that forces and torques, including the non-transmitted forces and torques, are properly accumulated and propagated. The bodies take a jaunt along the inertial z axis, return to the origin, then another jaunt along the inertial x axis and return once again to the origin. The bodies perform a 90 degree yaw and then a 90 degree roll during the latter excursion.

Test directory `models/dynamics/dyn_body/verif/SIM_force_torque`
The simulation S_define file defines two dynamic bodies, body actions that operate on these bodies, and forces and torques that can be applied to these bodies. The forces and torques are collected via the *v*collect mechanism. There is one run for this test. Table 5.6 summarizes the actions, forces and torques that are applied during the course of the run and identifies the expected results from those operations.

Success criteria The analytic and simulated results for the accumulated/propagated forces and torques, and for the resulting accelerations, must agree to within numerical precision. Deviations between the analytic and simulated integrated states must be small and attributable solely to the small numeric errors in the accelerations.

Test results The final position of the composite center of mass is within 1.7×10^{-12} meters of the origin. This small error is fully attributable to the integration of small errors. Along the way, small errors are made in the accumulation and propagation of forces and torques and calculations of accelerations. These small errors are purely numerical.

The test passes.

Applicable requirements This test demonstrates the partial or complete satisfaction of the following requirements:

- DynBody_2. Mass properties are properly initialized and are properly updated due to attachments and detachments.
- DynBody_6. Forces and torques are properly accumulated and propagated. Accelerations are properly calculated from forces and torques.
- DynBody_7. The equations of motion are properly integrated.
- DynBody_8. Vehicle points are properly constructed and used during attachments.
- DynBody_9. JEOD's magical teleportation capability works as advertised.

Table 5.6: Force/Torque Test Summary

Time	**Action**	**Expected Result**
0	Bodies initialized and attached	Composite body has spherical mass distribution and is at rest at the inertial origin.
0-3	Effector & environmental z-axis forces increasingly applied to body2	Composite acceleration builds from 0 to 3 m/s^2 $\hat{z}$ in steps of 1 m/s^2.
4-6	Non-xmitted forces on body2	Composite acceleration remains at 3 m/s$^2\hat{z}$.
7	Body2 detaches from body1	Bodies are detached.
7-9	Non-xmitted z-axis force on body2	Body2 zooms back toward the origin, coming to rest with respect to body1 at t=9.
9	Body2 reattaches to body1	Attachment does not change body1 core state.
9-19	Non-transmitted z-axis forces on body1	Composite acceleration is -3 m/s^2 $\hat{z}$.
21-27	Non-transmitted z-axis forces on body1	Composite body moves along z toward origin, coming to rest at the origin at t=27.
30-32	Environmental x-axis force on body2, non-xmitted y-axis torque on body1	Composite acceleration is 1 m/s^2 $\hat{x}$.
32-52	Environmental z-axis torque on body2, reversing sign at t=42	Composite body rotates about its z-axis, completing a 90 degree yaw at t=52.
52-56	Environmental y-axis force on body2, non-transmitted x-axis torque on body1	Composite acceleration is -1 m/s^2 $\hat{x}$, making composite move back toward origin.
56-76	Environmental x-axis torque on body2, reversing sign at t=66	Composite body rotates about its x-axis, completing a 90 degree roll at t=76.
76-78	Non-transmitted z-axis force on body1	Composite acceleration is 1 m/s^2 $\hat{x}$.
78		Composite body frame is at rest at the origin.

Test DynBody_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.

Test directory `models/dynamics/body_action/verif/SIM_frame_switch`
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 partial or complete satisfaction of the following requirements:

- DynBody_3. Demonstrating this capability is the primary purpose of this test.
- DynBody_4. States are properly converted to the new integration frame.
- DynBody_6. Accelerations are properly calculated in the new integration frame.
- DynBody_7. The equations of motion are properly integrated.

Test DynBody_5: Structure Integration

Background The primary purpose of this test is to test that integrating the structural origin yields very similar results compared to integrating the composite body center of mass.

Test description This test uses a near-carbon copy of `$JEOD_HOME/verif/SIM_dyncomp`, with a StructureIntegratedDynBody replacing the DynBody used in that simulation. `SIM_dyncomp` has been cross-validated with a number of external simulations. Cross-checking the near-carbon copy demonstrates that the structure-integrated dynamic body code provides valid results.

Test directory `models/dynamics/dyn_body/verif/SIM_dyncomp_structure`
The simulation S_define file replaces the DynBody used in `SIM_dyncomp` with a StructureIntegratedDynBody. The `S_overrides.mk` file generates the `SET_test` run directories by copying the corresponding python input files from the corresponding `SIM_dyncomp` run directories. The The `S_overrides.mk` file also makes the `Modified_data` and `Log_data` directories be symbolic links to the corresponding directories in `SIM_dyncomp`. These acts ensure that `SIM_dyncomp_structure` simulation runs start under the same initial conditions as do the corresponding `SIM_dyncomp` runs.

Success criteria Corresponding runs of `SIM_dyncomp` and `SIM_dyncomp_structure` should agree to high degree of precision throughout the eight hour span of the `SIM_dyncomp` runs. To test this, position discrepencies must be small, on the order of millimeters at most.

Test results The magnitudes of the position discrepancies between `SIM_dyncomp` and the structure-integrated replacement for the the four most sensitive `IM_dyncomp` runs are depicted in figure 5.1. The largest discrepancies are less than 100 micrometers, even on the one test case that shows the greatest deviation. These small errors are purely numerical.

The test passes.

Applicable requirements This test demonstrates the partial or complete satisfaction of the following requirements:

- DynBody_6. Forces and torques are properly accumulated and propagated. Accelerations are properly calculated from forces and torques.
- DynBody_7. The equations of motion are properly integrated.

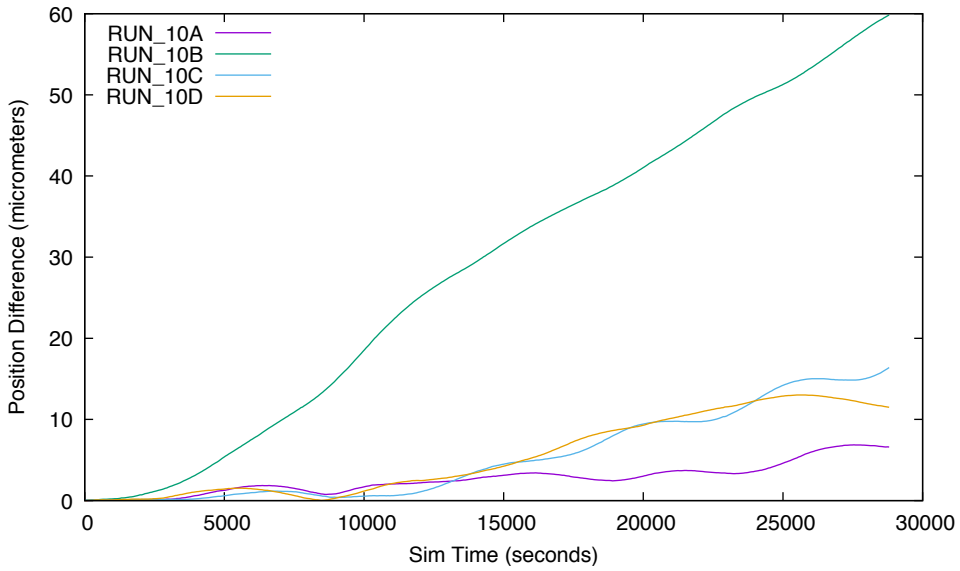

Figure 5.1: Structural origin vs center of mass integration position discrepancies

5.3 Metrics

Table 5.7 presents coarse metrics on the source files that comprise the model.

Table 5.7: Coarse Metrics

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

Table 5.8 presents the extended cyclomatic complexity (ECC) of the methods defined in the model.

Table 5.8: Cyclomatic Complexity

Method	File	Line	ECC
jeod::release_vector (Collect Type & vec)	include/body_force_collect.hh	78	3
jeod::CollectType::collect_ insert (CollectType & collect_in, value_type & elem)	include/body_force_collect.hh	98	5
jeod::CollectType::collect_ push_back (CollectType & collect_in, value_type & elem)	include/body_force_collect.hh	130	5
jeod::JPVCollectForce::std:: perform_insert_action (const std::string & value)	include/body_force_collect.hh	178	1
jeod::JPVCollectForce::push_ back (CollectForce* const & elem)	include/body_force_collect.hh	193	1

Table 5.8: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::BodyWrenchCollect:: accumulate (Wrench& sum)	include/body_wrench_ collect.hh	125	1
jeod::BodyWrenchCollect:: accumulate (const double point[3], Wrench& sum)	include/body_wrench_ collect.hh	137	1
jeod::DynBody::activate ()	include/dyn_body.hh	146	1
jeod::DynBody::deactivate ()	include/dyn_body.hh	154	1
jeod::DynBody::JeodPointer Vector¡er7_utils::get_ integrable_objects ()	include/dyn_body.hh	302	1
jeod::DynBody::get_ initialized_states ()	include/dyn_body.hh	524	1
jeod::DynBody::(RefFrame Items::initialized_states_ contains (RefFrameItems:: Items test_items)	include/dyn_body.hh	533	1
jeod::DynBodyGenericFrame Attachment::Vector3:: initialize_attachment (Ref Frame &parent_frame, const RefFrameState &attach_state)	include/dyn_body_generic_ rigid_attach.hh	95	1
jeod::DynBodyGenericFrame Attachment::clear_ attachment ()	include/dyn_body_generic_ rigid_attach.hh	107	1
jeod::DynBodyGenericFrame Attachment::isAttached ()	include/dyn_body_generic_ rigid_attach.hh	112	1
jeod::DynBodyGenericFrame Attachment::get_parent_ frame ()	include/dyn_body_generic_ rigid_attach.hh	117	1
jeod::DynBodyGenericFrame Attachment::get_attach_ offset ()	include/dyn_body_generic_ rigid_attach.hh	122	1
jeod::CollectForce:: operator== (const Collect Force &other)	include/force.hh	185	1
jeod::Force::operator[] (const unsigned int index)	include/force_inline.hh	70	1

Continued on next page

Table 5.8: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::Force::operator[] (const unsigned int index)	include/force_inline.hh	83	1
jeod::CollectForce::is_active (void)	include/force_inline.hh	98	3
jeod::CollectForce::operator[] (const unsigned int index)	include/force_inline.hh	112	1
jeod::CollectForce::operator[] (const unsigned int index)	include/force_inline.hh	125	1
jeod::StructureIntegratedDyn Body::get_vehicle_properties ()	include/structure_integrated_ dyn_body.hh	266	1
jeod::CollectTorque:: operator== (const Collect Torque &other)	include/torque.hh	180	1
jeod::Torque::operator[] (const unsigned int index)	include/torque_inline.hh	70	1
jeod::Torque::operator[] (const unsigned int index)	include/torque_inline.hh	83	1
jeod::CollectTorque::is_active (void)	include/torque_inline.hh	98	3
jeod::CollectTorque:: operator[] (const unsigned int index)	include/torque_inline.hh	112	1
jeod::CollectTorque:: operator[] (const unsigned int index)	include/torque_inline.hh	125	1

Continued on next page

Table 5.8: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::VehicleProperties::SolverTypes::VehicleProperties (SolverTypes::Vector3RefT parent_to_structure_offset_in, SolverTypes::Matrix3x3RefT parent_to_structure_transform_in, double& mass_in, SolverTypes::Vector3RefT structure_to_body_offset_in, SolverTypes::Matrix3x3RefT inertia_in, SolverTypes::Matrix3x3RefT structure_to_body_transform_in, double& inverse_mass_in, SolverTypes::Matrix3x3RefT inverse_inertia_in)	include/vehicle_properties.hh	102	1
jeod::VehicleProperties::SolverTypes::get_parent_to_structure_offset ()	include/vehicle_properties.hh	149	1
jeod::VehicleProperties::SolverTypes::get_parent_to_structure_transform ()	include/vehicle_properties.hh	159	1
jeod::VehicleProperties::get_mass ()	include/vehicle_properties.hh	168	1
jeod::VehicleProperties::SolverTypes::get_structure_to_body_offset ()	include/vehicle_properties.hh	176	1
jeod::VehicleProperties::SolverTypes::get_inertia ()	include/vehicle_properties.hh	186	1
jeod::VehicleProperties::SolverTypes::get_structure_to_body_transform ()	include/vehicle_properties.hh	195	1
jeod::VehicleProperties::get_inverse_mass ()	include/vehicle_properties.hh	204	1
jeod::VehicleProperties::SolverTypes::get_inverse_inertia ()	include/vehicle_properties.hh	212	1

Continued on next page

Table 5.8: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::Wrench::Vector3:: Wrench (const double torque_in[3], const double force_in[3], const double point_in[3], bool active_in = true)	include/wrench.hh	107	1
jeod::Wrench::Vector3:: Wrench (const double point_in[3], bool active_in = true)	include/wrench.hh	127	1
jeod::Wrench::Vector3:: operator+= (const Wrench& other)	include/wrench.hh	173	3
jeod::Wrench::activate ()	include/wrench.hh	195	1
jeod::Wrench::deactivate ()	include/wrench.hh	204	1
jeod::Wrench::is_active ()	include/wrench.hh	213	1
jeod::Wrench::Vector3::reset_ force_and_torque ()	include/wrench.hh	222	1
jeod::Wrench::Vector3::reset_ torque ()	include/wrench.hh	232	1
jeod::Wrench::Vector3::reset_ force ()	include/wrench.hh	241	1
jeod::Wrench::Vector3::reset_ point ()	include/wrench.hh	250	1
jeod::Wrench::Vector3::set (const double torque_in[3], const double force_in[3], const double point_in[3])	include/wrench.hh	259	1
jeod::Wrench::Vector3::set_ torque (const double torque_in[3])	include/wrench.hh	276	1
jeod::Wrench::Vector3::set_ force (const double force_ in[3])	include/wrench.hh	286	1
jeod::Wrench::Vector3::set_ force (const double force_ in[3], const double point_ in[3])	include/wrench.hh	296	1

Continued on next page

Table 5.8: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::Wrench::Vector3::set_ point (const double point_ in[3])	include/wrench.hh	309	1
jeod::Wrench::Vector3::scale_ torque (double scale)	include/wrench.hh	320	1
jeod::Wrench::Vector3::scale_ force (double scale)	include/wrench.hh	330	1
jeod::Wrench::get_torque ()	include/wrench.hh	340	1
jeod::Wrench::get_force ()	include/wrench.hh	349	1
jeod::Wrench::get_point ()	include/wrench.hh	358	1
jeod::Wrench::std::accumulate (const std::vector¡ Wrench*¿& collection)	include/wrench.hh	367	2
jeod::Wrench::std::accumulate (const std::vector¡ Wrench*¿& collection, const double new_point[3])	include/wrench.hh	384	1
jeod::Wrench::Vector3:: transform_to_point (const double new_point[3])	include/wrench.hh	399	1
jeod::Wrench::Vector3:: transform_to_parent (const MassPointState& point_ state)	include/wrench.hh	414	1
jeod::BodyForceCollect::Body ForceCollect (void)	src/aux_classes.cc	40	1
jeod::BodyForceCollect::~ BodyForceCollect (void)	src/aux_classes.cc	80	1
jeod::FrameDerivs::Frame Derivs (void)	src/aux_classes.cc	104	1
jeod::BodyWrenchCollect:: BodyWrenchCollect ()	src/body_wrench_collect.cc	26	1
jeod::BodyWrenchCollect::~ BodyWrenchCollect ()	src/body_wrench_collect.cc	35	1
jeod::DynBody::DynBody ()	src/dyn_body.cc	62	1
jeod::DynBody::~DynBody ()	src/dyn_body.cc	112	8

Continued on next page

Table 5.8: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::DynBody::set_name (const std::string & name_in)	src/dyn_body.cc	160	1
jeod::DynBody::is_root_body ()	src/dyn_body.cc	167	1
jeod::DynBody::get_parent_body ()	src/dyn_body.cc	175	1
jeod::DynBody::get_parent_body_internal ()	src/dyn_body.cc	184	1
jeod::DynBody::get_root_body ()	src/dyn_body.cc	192	1
jeod::DynBody::get_root_body_internal ()	src/dyn_body.cc	203	2
jeod::DynBody::add_control (GravityControls * control)	src/dyn_body.cc	220	1
jeod::DynBody::initialize_controls (GravityManager & grav_manager)	src/dyn_body.cc	230	1
jeod::DynBody::reset_controls ()	src/dyn_body.cc	240	1
jeod::DynBody::sort_controls ()	src/dyn_body.cc	250	1
jeod::DynBody::get_dynamics_integration_group ()	src/dyn_body.cc	259	3
jeod::DynBody::add_integrable_object (er7_utils::IntegrableObject & new_object)	src/dyn_body.cc	288	2
jeod::DynBody::remove_integrable_object (er7_utils::IntegrableObject & del_object)	src/dyn_body.cc	307	2
jeod::DynBody::clear_integrable_objects ()	src/dyn_body.cc	328	1
jeod::DynBody::migrate_integrable_objects (void)	src/dyn_body.cc	335	3

Continued on next page

Table 5.8: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::DynBody::attach_validate_parent (const DynBody & parent, bool generate_message)	src/dyn_body_attach.cc	55	7
jeod::DynBody::attach_validate_child (const DynBody & child, bool generate_message)	src/dyn_body_attach.cc	107	11
jeod::DynBody::add_mass_body_validate (const MassBody & child, bool generate_message)	src/dyn_body_attach.cc	174	8
jeod::DynBody::attach_to (const char * this_point_name, const char * parent_point_name, DynBody & parent)	src/dyn_body_attach.cc	228	1
jeod::DynBody::attach_to (const double offset_pstr_cstr_pstr[3], const double T_pstr_cstr[3][3], DynBody & parent)	src/dyn_body_attach.cc	246	1
jeod::DynBody::attach_to_frame (const char * parent_ref_frame_name)	src/dyn_body_attach.cc	267	2
jeod::DynBody::attach_to_frame (RefFrame &parent)	src/dyn_body_attach.cc	287	1
jeod::DynBody::attach_to_frame (const char * this_point_name, const char * parent_ref_frame_name, const double offset_pframe_cpt_pframe[3], const double T_pframe_cpt[3][3])	src/dyn_body_attach.cc	296	4
jeod::DynBody::attach_to_frame (const double offset_pframe_cstr_pframe[3], const double T_pframe_cstr[3][3], RefFrame & parent)	src/dyn_body_attach.cc	355	1

Continued on next page

Table 5.8: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::DynBody::attach_child (const char * this_point_ name, const char * child_ point_name, DynBody & child)	src/dyn_body_attach.cc	369	7
jeod::DynBody::attach_child (const double xyz_cstr_wrt_ pstr[3], const double T_ pstr_to_cstr[3][3], DynBody & child)	src/dyn_body_attach.cc	505	4
jeod::DynBody::add_mass_ body (const char * this_ point_name, const char * child_point_name, Mass Body & child)	src/dyn_body_attach.cc	591	6
jeod::DynBody::add_mass_ body (const double xyz_ cstr_wrt_pstr[3], const double T_pstr_to_cstr[3][3], MassBody & child)	src/dyn_body_attach.cc	715	3
jeod::DynBody::add_mass_ body_frames (MassBody &subbody)	src/dyn_body_attach.cc	802	6
jeod::DynBody::attach_ establish_links (DynBody & parent)	src/dyn_body_attach.cc	843	2
jeod::DynBody::attach_ update_properties (const double offset_pstr_cstr_ pstr[3], const double T_ pstr_cstr[3][3], DynBody & child)	src/dyn_body_attach.cc	868	7
jeod::DynBody::process_ dynamic_attachment (const double offset_pstr_cstr_ pstr[3], const double T_ pstr_cstr[3][3], DynBody & root_body, DynBody & child_body)	src/dyn_body_attach.cc	951	1

Continued on next page

Table 5.8: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodPointerVector¡ CollectForce¿::accumulate_ forces (const JeodPointer Vector¡CollectForce¿::type & vec, double * cumulation)	src/dyn_body_collect.cc	52	3
jeod::JeodPointerVector¡ CollectTorque¿:: accumulate_torques (const JeodPointerVector¡Collect Torque¿::type & vec, double * cumulation)	src/dyn_body_collect.cc	74	3
jeod::DynBody::collect_forces_ and_torques ()	src/dyn_body_collect.cc	96	10
jeod::DynBody::detach (Dyn Body & other_body)	src/dyn_body_detach.cc	47	10
jeod::DynBody::detach (void)	src/dyn_body_detach.cc	136	4
jeod::DynBody::remove_mass_ body (MassBody & child)	src/dyn_body_detach.cc	164	5
jeod::DynBody::detach_mass_ body_frames (MassBody &subbody)	src/dyn_body_detach.cc	238	6
jeod::DynBody::detach_mass_ internal (MassBody & child)	src/dyn_body_detach.cc	283	1
jeod::DynBody::find_body_ frame (const char * frame_ id)	src/dyn_body_find_body_ frame.cc	48	5
jeod::DynBody::initialize_ model (BaseDynManager & dyn_manager_in)	src/dyn_body_initialize_ model.cc	43	5
jeod::DynBody::set_integ_ frame (EphemerisRefFrame & new_integ_frame)	src/dyn_body_integration.cc	58	5
jeod::DynBody::set_integ_ frame (const char * new_ integ_frame_name)	src/dyn_body_integration.cc	125	2

Continued on next page

Table 5.8: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::DynBody::switch_integration_frames (EphemerisRefFrame & new_integ_frame)	src/dyn_body_integration.cc	146	3
jeod::DynBody::switch_integration_frames (const char * new_integ_frame_name)	src/dyn_body_integration.cc	188	2
jeod::DynBody::create_body_integrators (const er7_utils:: IntegratorConstructor & generator, er7_utils:: IntegrationControls & controls, const Jeod IntegrationTime & time_mngr)	src/dyn_body_integration.cc	208	2
jeod::DynBody::create_integrators (const er7_utils:: IntegratorConstructor & generator, er7_utils:: IntegrationControls & controls, const er7_utils:: TimeInterface & time_if)	src/dyn_body_integration.cc	251	2
jeod::DynBody::destroy_integrators (void)	src/dyn_body_integration.cc	281	1
jeod::DynBody::reset_integrators (void)	src/dyn_body_integration.cc	291	3
jeod::er7_utils::integrate (double dyn_dt, unsigned int target_stage)	src/dyn_body_integration.cc	307	4
jeod::er7_utils::trans_integ (double dyn_dt, unsigned int target_stage)	src/dyn_body_integration.cc	360	1
jeod::er7_utils::rot_integ (double dyn_dt, unsigned int target_stage)	src/dyn_body_integration.cc	380	2

Continued on next page

Table 5.8: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::DynBody::compute_ref_point_transform (const BodyRefFrame & source_frame, const MassPoint ** const ref_point, MassPoint State & rel_state)	src/dyn_body_propagate_state.cc	48	8
jeod::DynBody::compute_derived_state_forward (const BodyRefFrame & source_frame, const Mass Point & rel_state, BodyRef Frame & derived_frame)	src/dyn_body_propagate_state.cc	157	1
jeod::DynBody::compute_state_elements_forward (const BodyRefFrame & source_frame, const Mass Point & rel_state, const Ref FrameItems & state_items, BodyRefFrame & derived_frame)	src/dyn_body_propagate_state.cc	213	5
jeod::DynBody::compute_derived_state_reverse (const BodyRefFrame & source_frame, const MassPoint & rel_state, BodyRefFrame & derived_frame)	src/dyn_body_propagate_state.cc	276	1
jeod::DynBody::compute_state_elements_reverse (const BodyRefFrame & source_frame, const Mass Point & rel_state, const Ref FrameItems & state_items, BodyRefFrame & derived_frame)	src/dyn_body_propagate_state.cc	331	5
jeod::DynBody::update_integrated_state ()	src/dyn_body_propagate_state.cc	394	15
jeod::DynBody::propagate_state ()	src/dyn_body_propagate_state.cc	572	5
jeod::DynBody::propagate_state_from_structure ()	src/dyn_body_propagate_state.cc	606	9

Continued on next page

Table 5.8: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::DynBody::propagate_state_from_composite ()	src/dyn_body_propagate_state.cc	698	9
jeod::DynBody::compute_vehicle_point_states (Ref FrameItems::Items set_items)	src/dyn_body_propagate_state.cc	787	4
jeod::MessageHandler::check_frame_ownership (const BodyRefFrame & frame, const DynBody * dyn_body, const char * file, unsigned int line)	src/dyn_body_set_state.cc	53	2
jeod::DynBody::set_state (Ref FrameItems::Items set_items, const RefFrameState & state, BodyRefFrame & subject_frame)	src/dyn_body_set_state.cc	77	5
jeod::DynBody::set_state_source (RefFrameItems:: Items items, BodyRefFrame & frame)	src/dyn_body_set_state.cc	130	5
jeod::DynBody::set_position (const double position[3], BodyRefFrame & subject_frame)	src/dyn_body_set_state.cc	182	1
jeod::DynBody::set_velocity (const double velocity[3], BodyRefFrame & subject_frame)	src/dyn_body_set_state.cc	199	1
jeod::DynBody::set_attitude_left_quaternion (const Quaternion & left_quat, BodyRefFrame & subject_frame)	src/dyn_body_set_state.cc	216	1
jeod::DynBody::set_attitude_right_quaternion (const Quaternion & right_quat, BodyRefFrame & subject_frame)	src/dyn_body_set_state.cc	235	1

Continued on next page

Table 5.8: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::DynBody::set_attitude_ matrix (const double matrix[3][3], BodyRefFrame & subject_frame)	src/dyn_body_set_state.cc	254	1
jeod::DynBody::set_attitude_ rate (const double attitude_ rate[3], BodyRefFrame & subject_frame)	src/dyn_body_set_state.cc	273	1
jeod::DynBody::set_state_ source_internal (RefFrame Items::Items items, Body RefFrame & frame)	src/dyn_body_set_state.cc	291	5
jeod::DynBody::add_mass_ point (const MassPointInit & mass_point_init)	src/dyn_body_vehicle_point.cc	50	4
jeod::DynBody::find_vehicle_ point (const char * pt_ name)	src/dyn_body_vehicle_point.cc	99	4
jeod::DynBody::compute_ vehicle_point_derivatives (const BodyRefFrame & vehicle_pt, FrameDerivs & pt_derivs)	src/dyn_body_vehicle_point.cc	122	3
jeod::Force::Force (void)	src/force.cc	41	1
jeod::Force::~Force (void)	src/force.cc	53	1
jeod::CollectForce::Collect Force (void)	src/force.cc	64	1
jeod::CollectForce::Collect Force (Force & source_ force)	src/force.cc	77	1
jeod::CollectForce::Collect Force (double force_3vec[3])	src/force.cc	92	1
jeod::CollectForce::Collect Force (CollectForce & source_force)	src/force.cc	107	1
jeod::CollectForce::~Collect Force (void)	src/force.cc	122	1
jeod::CInterfaceForce::C InterfaceForce (void)	src/force.cc	134	1

Continued on next page

Table 5.8: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::CInterfaceForce::C InterfaceForce (double * force_3vec)	src/force.cc	149	1
jeod::CInterfaceForce::~C InterfaceForce (void)	src/force.cc	164	3
jeod::CollectForce::create (Force & source_force)	src/force.cc	177	1
jeod::CollectForce::create (Force * source_force)	src/force.cc	192	1
jeod::CollectForce::create (double * force_3vec)	src/force.cc	207	1
jeod::CollectForce::create (CollectForce & source_ force)	src/force.cc	221	1
jeod::CollectForce::create (CollectForce * source_force)	src/force.cc	236	1
jeod::StructureIntegratedDyn Body::StructureIntegrated DynBody ()	src/structure_integrated_dyn_ body.cc	36	1
jeod::StructureIntegratedDyn Body::~StructureIntegrated DynBody ()	src/structure_integrated_dyn_ body.cc	59	1
jeod::JeodPointerVector¡ CollectForce¿::accumulate_ forces (const JeodPointer Vector¡CollectForce¿::type & vec, double * cumulation)	src/structure_integrated_dyn_ body_collect.cc	33	3
jeod::JeodPointerVector¡ CollectTorque¿:: accumulate_torques (const JeodPointerVector¡Collect Torque¿::type & vec, double * cumulation)	src/structure_integrated_dyn_ body_collect.cc	54	3
jeod::StructureIntegratedDyn Body::collect_forces_and_ torques ()	src/structure_integrated_dyn_ body_collect.cc	75	7
jeod::StructureIntegratedDyn Body::collect_local_forces_ and_torques ()	src/structure_integrated_dyn_ body_collect.cc	178	5

Continued on next page

Table 5.8: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::StructureIntegratedDynBody::PropagateForcesAndTorques ()	src/structure_integrated_dyn_body_collect.cc	234	5
jeod::StructureIntegratedDynBody::compute_inertial_torque ()	src/structure_integrated_dyn_body_collect.cc	328	1
jeod::StructureIntegratedDynBody::compute_rotational_acceleration ()	src/structure_integrated_dyn_body_collect.cc	353	1
jeod::StructureIntegratedDynBody::compute_translational_acceleration ()	src/structure_integrated_dyn_body_collect.cc	385	1
jeod::StructureIntegratedDynBody::complete_translational_acceleration ()	src/structure_integrated_dyn_body_collect.cc	415	1
jeod::er7_utils::trans_integ (double dyn_dt, unsigned int target_stage)	src/structure_integrated_dyn_body_integration.cc	36	1
jeod::er7_utils::rot_integ (double dyn_dt, unsigned int target_stage)	src/structure_integrated_dyn_body_integration.cc	50	2
jeod::StructureIntegratedDynBody::compute_vehicle_point_derivatives (const BodyRefFrame & vehicle_pt, FrameDerivs & pt_derivs)	src/structure_integrated_dyn_body_pt_accel.cc	31	3
jeod::StructureIntegratedDynBody::attach_update_properties (const double offset_pstr_cstr_pstr[3], const double T_pstr_cstr[3][3], DynBody & child)	src/structure_integrated_dyn_body_solve.cc	36	4
jeod::StructureIntegratedDynBody::detach (DynBody & other_body)	src/structure_integrated_dyn_body_solve.cc	69	4

Continued on next page

Table 5.8: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::StructureIntegratedDynBody::set_solver (DynBodyConstraintsSolver& solver_in)	src/structure_integrated_dyn_body_solve.cc	106	2
jeod::StructureIntegratedDynBody::add_constraint (DynBodyConstraint* constraint)	src/structure_integrated_dyn_body_solve.cc	123	2
jeod::StructureIntegratedDynBody::solve_constraints ()	src/structure_integrated_dyn_body_solve.cc	140	5
jeod::Torque::Torque (void)	src/torque.cc	41	1
jeod::Torque::~Torque (void)	src/torque.cc	53	1
jeod::CollectTorque::CollectTorque (void)	src/torque.cc	64	1
jeod::CollectTorque::CollectTorque (Torque & source_torque)	src/torque.cc	77	1
jeod::CollectTorque::CollectTorque (double torque_3vec[3])	src/torque.cc	92	1
jeod::CollectTorque::CollectTorque (CollectTorque & source_torque)	src/torque.cc	107	1
jeod::CollectTorque::~CollectTorque (void)	src/torque.cc	122	1
jeod::CInterfaceTorque::CInterfaceTorque (void)	src/torque.cc	134	1
jeod::CInterfaceTorque::CInterfaceTorque (double * torque_3vec)	src/torque.cc	149	1
jeod::CInterfaceTorque::~CInterfaceTorque (void)	src/torque.cc	164	3
jeod::CollectTorque::create (Torque & source_torque)	src/torque.cc	177	1
jeod::CollectTorque::create (Torque * source_torque)	src/torque.cc	192	1
jeod::CollectTorque::create (double * torque_3vec)	src/torque.cc	207	1

Continued on next page

Table 5.8: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::CollectTorque::create (CollectTorque & source_ torque)	src/torque.cc	221	1
jeod::CollectTorque::create (CollectTorque * source_ torque)	src/torque.cc	236	1

5.4 Requirements Traceability

Table 5.9 summarizes the inspections and tests that demonstrate the satisfaction of the requirements levied on the model.

Table 5.9: Requirements Traceability

Requirement	Inspection or test
DynBody_1 Project Requirements	Insp. DynBody_1 Top-level Inspection
DynBody_2 Mass	Insp. DynBody_2 Design Inspection Test DynBody_1 State Initialization Test DynBody_2 Attach/Detach Test DynBody_3 Force/Torque
DynBody_3 Integration Frame	Insp. DynBody_2 Design Inspection Test DynBody_1 State Initialization Test DynBody_4 Frame Switch
DynBody_4 State Representation	Insp. DynBody_2 Design Inspection Test DynBody_1 State Initialization Test DynBody_4 Frame Switch
DynBody_5 Staged Initialization	Insp. DynBody_2 Design Inspection Test DynBody_1 State Initialization
DynBody_6 Equations of Motion	Insp. DynBody_2 Design Inspection Insp. DynBody_3 Mathematical Formulation Test DynBody_2 Attach/Detach Test DynBody_3 Force/Torque Test DynBody_4 Frame Switch Test DynBody_5 Structure Integration
DynBody_7 State Integration and Propagation	Insp. DynBody_2 Design Inspection Insp. DynBody_3 Mathematical Formulation Test DynBody_2 Attach/Detach Test DynBody_3 Force/Torque Test DynBody_4 Frame Switch Test DynBody_5 Structure Integration
DynBody_8 Vehicle Points	Insp. DynBody_2 Design Inspection Test DynBody_2 Attach/Detach Test DynBody_3 Force/Torque Test DynBody_4 Frame Switch
DynBody_9 Attach/Detach	Insp. DynBody_2 Design Inspection Insp. DynBody_3 Mathematical Formulation Test DynBody_2 Attach/Detach Test DynBody_3 Force/Torque

Bibliography

[1] Generated by doxygen. *JEOD Dynamic Body Model 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.

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

[3] Hammen, D. Integration Model. Technical Report JSC-61777-utils/integration, NASA, Johnson Space Center, Houston, Texas, July 2023.

[4] Hammen, D. Memory Allocation Routines. Technical Report JSC-61777-utils/memory, NASA, Johnson Space Center, Houston, Texas, July 2023.

[5] Hammen, D. Quaternion. Technical Report JSC-61777-utils/quaternion, NASA, Johnson Space Center, Houston, Texas, July 2023.

[6] Hammen, D. Body Action Model. Technical Report JSC-61777-dynamics/body_action, 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] NASA. NASA Software Engineering Requirements. Technical Report NPR-7150.2, NASA, NASA Headquarters, Washington, D.C., September 2004.

[9] Shelton, R. Message Handling Class. Technical Report JSC-61777-utils/message, NASA, Johnson Space Center, Houston, Texas, July 2023.

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

[11] Shelton, R. Simulation Engine Interface Model. Technical Report JSC-61777-utils/sim_interface, NASA, Johnson Space Center, Houston, Texas, July 2023.

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

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

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

[15] Thompson, B. Gravity Gradient Torque Model. Technical Report JSC-61777-interactions/gravity_torque, NASA, Johnson Space Center, Houston, Texas, July 2023.

[16] Thompson, B. and Morris., J. Gravity Model. Technical Report JSC-61777-environment/gravity, NASA, Johnson Space Center, Houston, Texas, July 2023.

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