(Preprint) AAS 26-078

SIMULATION OF GENERALLY ARTICULATED SPACECRAFT FOR GN&C ALGORITHM DESIGN AND VALIDATION

Juan Garcia-Bonilla*, Will Schwend†, and Hanspeter Schaub‡

High-fidelity guidance, navigation, and control (GN&C) simulation increasingly requires representing spacecraft as generally articulated multi-body systems rather than as single rigid bodies. Deployables, reaction wheels, manipulators, and other internal mechanisms introduce coupled dynamics, joint constraints, and configuration-dependent inertia that are difficult to capture in rigid-body astrodynamics simulators. Robotics-focused multi-body engines, on the other hand, often lack orbital propagation and space-environment modeling needed for GN&C analysis. This paper introduces a model-based, message-passing simulation paradigm for articulated spacecraft that treats forward kinematics and forward dynamics as solver-provided services and represents environmental effects, actuation, and constraints as modular models that exchange information exclusively through typed messages. This separation allows GN&C developers to assemble high-fidelity simulations by composing reusable force/torque and constraint models, without deriving system-specific equations of motion, while preserving the common workflow of evaluating environment forces at integrator substeps and running flight software at a fixed rate. The paradigm is implemented in the open-source Basilisk framework by integrating the MuJoCo dynamics engine as an alternate multi-body backend. Three demonstrations validate the approach: staged deployment of a branching solar-array mechanism with message-based joint locking, closed-loop attitude control using reaction wheels modeled as explicit articulated bodies to capture coupled hub-wheel dynamics, and six-degree-of-freedom control using thrusters mounted on actuated robotic arms.

INTRODUCTION

A well designed numerical simulation is a foundational capability for spacecraft guidance, navigation, and control (GN&C) development, verification, and operations. Such a simulation enables rapid iteration on control laws, assessment of robustness to disturbances and modeling uncertainty, and risk reduction without reliance on expensive hardware testing.[1–3] During operations, simulation supports mission rehearsals and operator training, helping teams explore contingencies and make informed decisions.[4] Post-launch, simulation also supports telemetry reconstruction and anomaly investigation, and it increasingly supplies training data for learning-based spaceflight algorithms when flight data are sparse.[5]

A persistent limitation of many astrodynamics simulators is the assumption that a spacecraft behaves as a single rigid body. This abstraction is adequate for numerous analyses, but it obscures

*Ph.D. Student, Department of Aerospace Engineering Sciences, University of Colorado Boulder, 431 UCB, Boulder, CO 80309, USA

†Ph.D. Student, Department of Aerospace Engineering Sciences, University of Colorado Boulder, 431 UCB, Boulder, CO 80309, USA

‡Ann and H. J. Smead Department of Aerospace Engineering Sciences, University of Colorado Boulder, 431 UCB, Boulder, CO 80309, USA

(a) "Haven-2" space station proposal*.

(b) Dextre two-armed robot attached to the ISS†.

(c) Starliner docking with the ISS rendering‡.

Figure 1: Examples of spacecraft configurations that benefit from multi-body dynamics modeling.

internal momentum exchange and configuration-dependent inertia that arise in modern spacecraft equipped with reaction wheels, deployable arrays, gimbaled instruments, articulated manipulators, or docking interfaces.[6] Emerging mission concepts, including on-orbit servicing, assembly, modular stations, and autonomous robotic operations, further motivate simulation environments that represent spacecraft as generally articulated multi-body systems as visualized in Fig. 1.[7–9]

Robotics simulators (e.g., Gazebo§, MuJoCo¶, Simscape Multibody‖) provide efficient and robust algorithms for articulated dynamics, joint constraints, and contact modeling,[10–12] but they typically do not provide orbital propagation, ephemerides, and space-environment models needed for GN&C analysis. Conversely, mission analysis tools (e.g., GMAT, STK, FreeFlyer) offer mature astrodynamics capabilities,[13–15] yet commonly treat spacecraft as rigid bodies and provide limited support for general articulated structures. Existing spacecraft-focused multi-body tools such as JPL's Dshell-DARTS** demonstrate strong capability but are not broadly accessible due to closed-source distribution.[1]

Basilisk†† is a modular, open-source simulation toolkit for astrodynamics and GN&C.[2] It supports accurate orbital and attitude propagation with real-time performance. Basilisk includes efficient multi-body capabilities based on the back-substitution method (BSM), but these capabilities are constrained in the topology and joint combinations they support, and become difficult to extend without deriving and implementing complicated equations of motion.[16–18] This limits rapid prototyping for branching mechanisms found with complex spacecraft robotic arms and solar arrays, and other generally articulated configurations of increasing interest to the GN&C community.

This paper introduces a general multi-body simulation paradigm that applies Basilisk's message-passing design principles directly to spacecraft dynamics, extending earlier conference work that demonstrated an initial message-driven multi-body architecture.[19] The key idea is to treat kinematics and forward dynamics as solver-provided services, and to represent environmental effects, actuation, and constraints as composable message-driven models that publish generalized forces and time-varying inertial properties.

The primary contributions are:

§ https://classic.gazebosim.org
¶ https://mujoco.org
‖ https://www.mathworks.com/products/simscape-multibody.html
** https://dartslab.jpl.nasa.gov/DSHELL/index.php
†† https://avslab.github.io/basilisk

1. A message-passing architecture that decomposes multi-body simulation into forward kinematics, modular force/torque models, and forward dynamics.

2. An implementation of this architecture in Basilisk that integrates MuJoCo as an alternate dynamics engine while preserving Basilisk tasking and messaging semantics.

3. Demonstrations showing that conventional GN&C loops operate directly on articulated dynamics, including staged solar-array deployment with joint constraints, reaction-wheel attitude control with wheels modeled as explicit bodies, and control using thrusters mounted on articulated robotic arms.

The remainder of this paper is organized as follows. The next section states the simulation problem and GN&C-oriented requirements. The following section presents the message-passing dynamics paradigm. The Basilisk+MuJoCo implementation is then described, followed by an extension that supports auxiliary continuous-time states and time-varying inertial properties. The paper then demonstrates the approach on three GN&C-relevant scenarios. Finally, the concluding section summarizes key takeaways and outlines directions for future work.

PROBLEM STATEMENT AND GN&C-ORIENTED REQUIREMENTS

A generally articulated spacecraft can be represented as a set of rigid bodies connected by joints forming a kinematic tree, optionally augmented by constraints (e.g., joint locks or closed-chain conditions). Let $\boldsymbol{q} \in \mathbb{R}^n$ denote the generalized coordinates, consisting of the free-flying hub pose and twist together with all internal joint coordinates. The system dynamics are expressed as

$$\frac{\mathrm{d}\boldsymbol{q}}{\mathrm{d}t} = f(\boldsymbol{q},\, \boldsymbol{\tau}(t, \boldsymbol{q}))\,, \tag{1}$$

where $\boldsymbol{\tau}(t, \boldsymbol{q})$ denotes generalized forces. These include externally applied forces and torques (e.g., thrusters), internal joint torques (e.g., motors and reaction wheel drives), and environment-induced effects such as gravity, drag, contact, and constraint reactions.

From a GN&C software perspective, a simulation framework intended for *efficient*, *general* articulated-spacecraft studies should satisfy three practical requirements:

1. **Topology generality:** support branching articulated structures and mixed joint types without requiring problem-specific equation derivations.

2. **Modular physical effects:** represent environment and actuation effects as reusable components that can be composed across vehicles and scenarios.

3. **GN&C task compatibility:** integrate cleanly with fixed-rate navigation, guidance, and control loops while allowing the dynamics to be integrated at smaller internal timesteps.

[¶] "Starliner makes first docking with ISS on OFT-2 mission", `https://www.nasaspaceflight.com/2022/05/oft-2-docking/`, Accessed: 2026-01-18.

[‖] "NASA Image and Video Library: View of Dextre and CTC2", `https://images.nasa.gov/details/iss027e016182`, Accessed: 2026-01-18.

[**] "Vast Announces Haven-2, Its Proposed Space Station Designed To Succeed The International Space Station (ISS)", `https://www.vastspace.com/updates/vast-announces-haven-2-its-proposed-space-station-designed-to-succeed-the-international-space-station-iss`, Accessed: 2026-01-18.

These requirements motivate decomposing Eq. (1) into three coupled computations:

1. **Forward kinematics:** map $(\boldsymbol{q}, \dot{\boldsymbol{q}})$ to frame-level pose and twist for bodies and joints of interest.
2. **Force/torque evaluation:** compute contributions to $\boldsymbol{\tau}$ from environment models, actuators, and control laws.
3. **Forward dynamics:** combine $\boldsymbol{\tau}$ with inertial properties to evaluate $\dot{\boldsymbol{q}}$.

Forward kinematics and forward dynamics admit mature, efficient, and system-agnostic implementations in open-source multi-body engines developed by the robotics community.[10,11] Force and torque evaluation, by contrast, is inherently application-specific: it encodes the disturbances, actuation, and GN&C logic a designer wishes to validate. The framework proposed in this paper therefore treats kinematics and dynamics evaluation as solver-provided services and focuses user effort on composing modular, physically meaningful effect models that publish generalized force and torque contributions.

Beyond efficiency, modern robotics multi-body engines also provide capabilities that are frequently desirable in spacecraft studies, including joint limits, prescribed motion, and constraint and contact handling. While not strictly required for all GN&C analyses, these features expand the class of scenarios that can be simulated without changing the surrounding GN&C software workflow.

MESSAGE-PASSING ARCHITECTURE FOR ARTICULATED SPACECRAFT DYNAMICS

Basilisk is an open-source astrodynamics simulation framework built on a modular, message-passing architecture.[20] In this architecture, discrete models represent components such as sensors, actuators, and flight software, and they communicate by publishing and subscribing to structured messages. This design promotes code reuse, encapsulation, and testability via strict interfaces.

To extend these benefits to generally articulated spacecraft dynamics, we propose a message-passing paradigm that decomposes Eq. (1) into three model classes that communicate exclusively through messages:

1. A **forward kinematics model** computes the pose and twist of selected frames on the spacecraft given the generalized coordinates $\boldsymbol{q}$ as well as joint states.
2. A set of **dynamic-effect models** compute generalized forces $\boldsymbol{\tau}$ acting on the system's bodies and joints using kinematic and environmental information.
3. A **forward dynamics model** aggregates generalized forces and computes the generalized coordinate derivative $\frac{\mathrm{d}\boldsymbol{q}}{\mathrm{d}t}$.

In this architecture, solver complexity is isolated within the forward-kinematics and forward-dynamics services. Simulation engineers implement and compose dynamic-effect models for gravity, thrusters, aerodynamic drag, solar radiation pressure, joint actuation, constraints, control laws, etc. Each model reads standardized input messages (e.g., frame pose and twist) and publishes standardized outputs (e.g., body forces, body torques, or joint torques).

This design yields three practical advantages for GN&C simulation:

- **Reusability:** Models are written against generic input and output messages, making them portable across vehicles and scenarios.
- **Clarity:** Each model implements a physically meaningful effect with a strict interface, which improves debugging and verification.
- **Composability:** Complex environments are constructed by combining well-tested modules without modifying the dynamics solver.

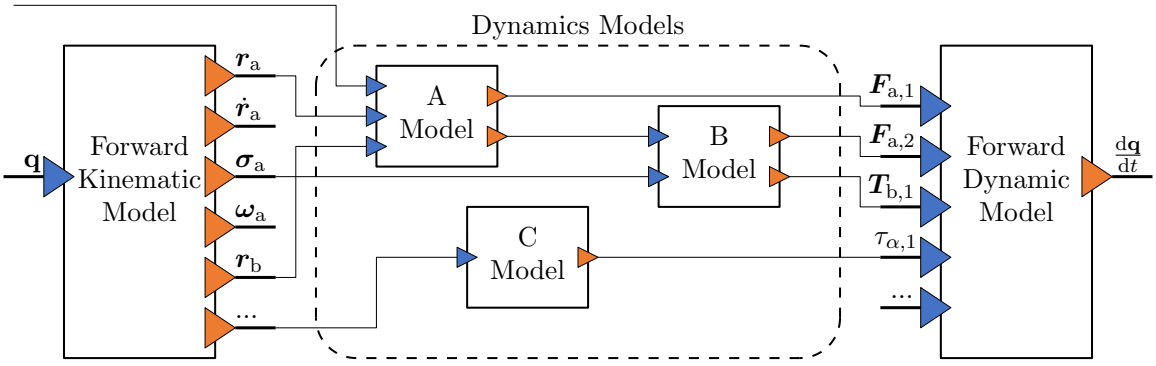

Figure 2: Schematic of the message-passing, model-based simulation architecture. Each model (black square) communicates via messages: inputs (blue triangles) and outputs (orange triangles). The forward kinematics model provides frame-level pose and twist; dynamics models compute forces and torques; the forward dynamics model aggregates these to compute the generalized coordinates' time derivative.

Figure 2 illustrates this paradigm. The forward kinematics model reads the system state and publishes frame poses and twists (e.g., position $\boldsymbol{r}$, velocity $\dot{\boldsymbol{r}}$, attitude $\boldsymbol{\sigma}$, angular velocity $\boldsymbol{\omega}$). Multiple dynamic-effect models subscribe to this information to compute spatial forces and torques ($\boldsymbol{F}$, $\boldsymbol{T}$) or joint torques (τ). The forward dynamics model collects these generalized forces and produces $\dot{\boldsymbol{q}}$, which is integrated forward in time.

Solver backend and solver-agnostic interface

MuJoCo is used as the multi-body backend in the implementation presented here because it provides efficient articulated rigid-body dynamics, joint constraints, collision detection, and contact handling.[11] However, the proposed paradigm is solver-agnostic: any engine that can evaluate forward kinematics for selected frames and evaluate forward dynamics given generalized forces can serve as the backend. In this work, MuJoCo's capabilities are exposed through message interfaces that preserve Basilisk's modular workflow.

IMPLEMENTATION IN BASILISK WITH MUJOCO

This section describes how the message-passing dynamics paradigm is realized within the Basilisk simulation framework by integrating MuJoCo as an alternate multi-body dynamics backend. Note that MuJoCo is integrated as a software package as is and no modifications to MoJoCo were made. The integration process described in this section is purely on the Basilisk side. Thus, the integration is designed to preserve the standard Basilisk workflow for GN&C analysis: dynamics are propagated through numerical integration, while flight software executes in fixed-rate tasks. In particular,

models whose outputs must be available at integrator substeps (e.g., gravity or other state-dependent forces) are scheduled in a dedicated *dynamics task*, whereas GN&C models execute in a separate task at a user-defined rate (e.g., 10 Hz). This separation ensures physically consistent force evaluation during integration without forcing GN&C code to run at the integrator's internal step size.

At the implementation level, the Basilisk-MuJoCo bridge is organized around two ideas. First, the MuJoCo model file is treated as a *vehicle description* that is loaded once and then accessed through Basilisk-native objects. Second, all interactions between user models (controllers, environment effects, schedulers) and the multi-body plant occur through Basilisk messages, so that actuation and constraints follow the same publish-subscribe workflow as the rest of the Basilisk ecosystem.

MuJoCo scene representation and Basilisk wrappers

MuJoCo represents articulated systems using an XML description of bodies, joints, geometries, and (optionally) inertial properties*. In our implementation, a Basilisk `MJScene` object loads this XML and encapsulates the underlying MuJoCo data structures. The `MJScene` wrapper exposes a Basilisk-friendly interface for retrieving model components by name and for configuring simulation execution, including selection of a Basilisk numerical integrator and access to message endpoints used for actuation and constraints.

Listing 1 illustrates a typical setup sequence. An `MJScene` is constructed from an XML file, a Basilisk integrator is instantiated and assigned to the scene via `scene.setIntegrator(...)`, and user scripts then query the scene to retrieve handles to XML-defined bodies and joints. In the branching solar-array deployment demonstration presented later in this paper, this pattern is used to iterate over the six panel bodies and collect their hinge joints into a dictionary that is later used to attach controllers and staged constraints.

```
from Basilisk.simulation import mujoco
from Basilisk.simulation import svIntegrators

scene = mujoco.MJScene.fromFile(XML_PATH)
integ = svIntegrators.svIntegratorRKF45(scene)
scene.setIntegrator(integ)

joints: dict[str, mujoco.MJScalarJoint] = {}
for panelID in ["10", "1p", "1n", "20", "2p", "2n"]:
    bodyName = f"panel_{panelID}"
    body: mujoco.MJBody = scene.getBody(bodyName)

    jointName = f"panel_{panelID}_deploy"
    joints[panelID] = body.getScalarJoint(jointName)
```

Listing 1: Loading a MuJoCo XML file in Basilisk, setting an integrator, and querying joints and bodies.

Two implementation details are worth emphasizing. First, users do not interact with MuJoCo directly. After the XML is loaded, scenario scripts remain entirely in Basilisk: components are retrieved by name (see Listing 1) and commanded through message connections, so the multi-body plant behaves like a standard Basilisk module from the perspective of GN&C models. Second, the

*MuJoCo XML Reference, `https://mujoco.readthedocs.io/en/stable/XMLreference.html`, Accessed: 2026-01-18.

multi-body propagation is integrated through Basilisk's standard integrator interface, which preserves the usual step control, logging, and task scheduling mechanisms used in existing Basilisk simulations.

Message interfaces for actuation and constraints

The previous subsection described how a multi-body model is instantiated and accessed through Basilisk wrappers. This subsection describes how user-defined models *influence* that multi-body plant while preserving the message-passing philosophy end-to-end: all solver inputs that modify dynamics are exposed as Basilisk message endpoints.

In the simplest case, joint actuation is achieved by subscribing a MuJoCo actuator's input message to the output of a controller model. This allows conventional Basilisk GN&C code (or custom controllers) to command joint torques without invoking solver calls: controllers publish torque commands, and the MuJoCo-backed forward dynamics model consumes them during force aggregation at each evaluation.

Constraints are handled in the same way. MuJoCo provides a rich set of constraint mechanisms, and the integration exposes commonly used constraints as message-controlled inputs so they can be scheduled and composed like any other Basilisk behavior. The solar-array deployment demonstration, for example, uses temporary joint "locks" to enforce staged motion. In our interface, each scalar joint exposes a `constrainedStateInMsg`; connecting this input to a message specifying a desired joint value requests that the dynamics backend enforce a constraint at that value, while disconnecting the message restores the joint's freedom.

Listing 2 shows how joint locking is performed in a scenario script. The `lockJoint` function constructs a `ScalarJointStateMsg` containing the desired fixed angle and subscribes the joint's constraint input to that message. The `unlockJoint` function simply unsubscribes, removing the constraint command. In the branching solar-array deployment demonstration, these operations are used both to immobilize non-active panels during each deployment phase thus emulating mechanical latching at stowed or fully deployed angles.

```
def lockJoint(panelID: str, angle: float):
    jointConstraintMsg = messaging.ScalarJointStateMsg()
    jointConstraintMsgPayload = messaging.ScalarJointStateMsgPayload()
    jointConstraintMsgPayload.state = angle
    jointConstraintMsg.write(jointConstraintMsgPayload, 0, -1)

    joints[panelID].constrainedStateInMsg.subscribeTo(jointConstraintMsg)

def unlockJoint(panelID: str):
    joints[panelID].constrainedStateInMsg.unsubscribe()
```

Listing 2: Locking and unlocking specific joints on the multi-body using message connections.

Exposing both actuation and constraints through messages reflects the broader design goal of the framework: solver capability should appear to the simulation engineer as typed message inputs and outputs on modular components, rather than as bespoke solver calls embedded in scripts. This preserves composability (e.g., constraints can be commanded by a scheduler or state machine model), supports consistent unit testing of control and environment models, and keeps GN&C integration identical across rigid-body and articulated dynamics plants.

EXTENSION TO AUXILIARY CONTINUOUS-TIME STATES

The multi-body generalized coordinates $\boldsymbol{q}$ capture the articulated configuration (hub pose/twist and joint coordinates), but many GN&C simulations require additional continuous-time states to represent vehicle subsystems and estimators. Examples include propellant mass and tank states, battery charge, actuator internal states, thermal states, and navigation filter states. Let $\boldsymbol{x} \in \mathbb{R}^m$ denote a vector of such auxiliary states evolving according to

$$\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t} = g(t, \boldsymbol{x}, \boldsymbol{q}), \tag{2}$$

where the right-hand side may depend on time, the auxiliary states themselves, and the current multi-body configuration.

Model-owned states in a message-passing simulation

A key design goal of the proposed paradigm is to allow auxiliary states to be implemented *within* the same modular model structure used for forces, sensors, and controllers. Concretely, a model may declare (register) internal continuous states that become part of the overall integrated state vector. At each dynamics evaluation, the integrator provides the model with its current internal state value $\boldsymbol{x}$ and the model returns the corresponding derivative $\dot{\boldsymbol{x}}$ through the standard model evaluation interface. This preserves encapsulation: the state definition and derivative computation remain localized within a single model.

Although the state propagation interface itself is not message-based, these stateful models still participate in Basilisk's message-passing ecosystem for coupling. They typically *consume* message inputs (e.g., commanded thrust, measured joint rate, or power demand) to compute $\dot{\boldsymbol{x}}$, and *publish* message outputs (e.g., actuator commands, consumption rates, or diagnostics) derived from their internal states. For example, a PID controller maintains an integral error state, an actuator model may maintain internal states representing rate limits or motor dynamics, and a propellant subsystem model may integrate remaining mass based on commanded thrust.

Coupling auxiliary states into articulated dynamics

Auxiliary states can influence the articulated dynamics through two pathways. First, they can modify applied generalized forces, such as thruster force magnitude depending on propellant feed conditions, or motor torque limits depending on battery voltage. This is captured by allowing dynamic-effect models to compute generalized forces as functions of $(t, \boldsymbol{q}, \boldsymbol{x})$:

$$\boldsymbol{\tau} = \boldsymbol{\tau}(t, \boldsymbol{q}, \boldsymbol{x}).$$

Second, auxiliary states can modify the inertial properties of the multi-body system, such as total mass, center of mass, and inertia tensor as propellant depletes or as deployables reconfigure mass distribution.

To account for both effects, we write the generalized coordinate propagation as

$$\frac{\mathrm{d}\boldsymbol{q}}{\mathrm{d}t} = f(\boldsymbol{q},\, \boldsymbol{\tau}(t, \boldsymbol{q}, \boldsymbol{x}),\, \mathcal{M}(t, \boldsymbol{q}, \boldsymbol{x}))\,, \tag{3}$$

where $\mathcal{M}$ denotes the set of inertial properties required by the dynamics backend (e.g., body masses, centers of mass, and inertia tensors).

TIME-VARYING MASS PROPERTIES AS MESSAGE INPUTS

The mapping from auxiliary states $\boldsymbol{x}$ to inertial properties $\mathcal{M}$ is highly application-specific (e.g., slosh models, tank geometry, moving payloads). Rather than prescribing a fixed formulation, the proposed architecture exposes inertial properties as *message-controlled inputs* to the forward dynamics service. In practice, an "inertia model" publishes updated mass properties for one or more bodies, and the forward dynamics service consumes these messages when assembling the multi-body system used for kinematics and dynamics evaluation.

This message-level interface has two advantages. First, it decouples subsystem modeling from the dynamics backend: the same fuel-tank or payload model can be reused regardless of whether the backend is MuJoCo or another solver, as long as the inertial-property message schema is supported. Second, it allows inertia updates to be scheduled and tested like any other model: an inertia model can be run in the dynamics task (for substep-consistent updates) or at a lower rate if appropriate, depending on the application.

Figure 3 illustrates this extension. Models D and E each maintain internal continuous states and publish their derivatives for integration. In addition, these models publish time-varying inertial properties (e.g., updated center of mass $\boldsymbol{c}_A$) to the forward dynamics model, enabling coupled evolution of subsystem states and articulated spacecraft dynamics within the same message-passing simulation workflow.

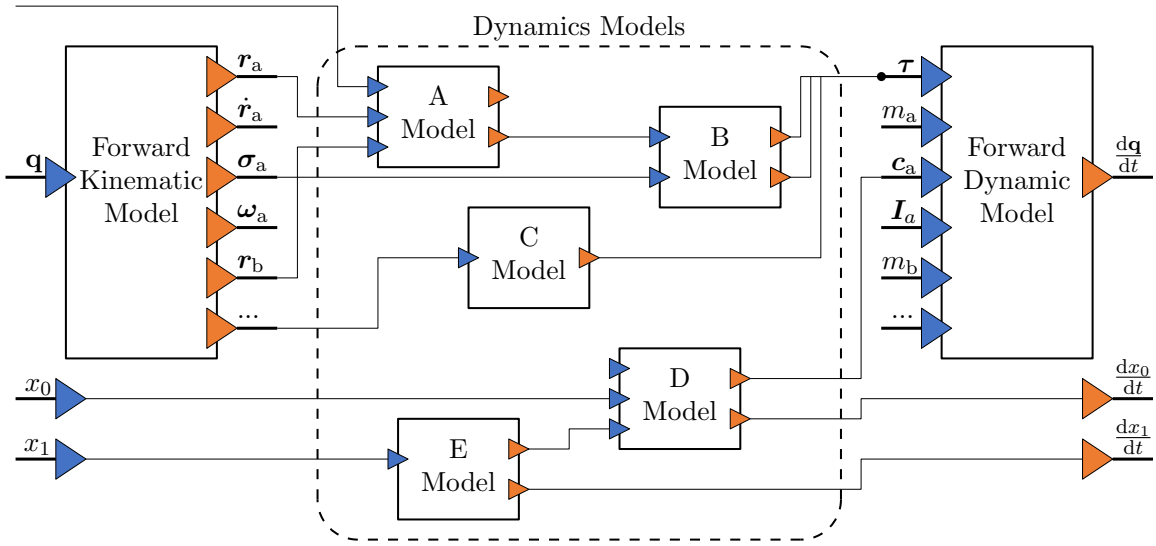

Figure 3: Extension of the model-based simulation paradigm to include auxiliary continuous-time states and time-varying inertial properties. Models D and E manage continuous states (x_0, x_1), publish their derivatives for integration, and publish updated inertial properties to the forward dynamics model.

DEMONSTRATIONS FOR GN&C DEVELOPMENT AND VALIDATION

This section demonstrates end-to-end workflows enabled by the proposed message-driven dynamics architecture, implemented in Basilisk with MuJoCo as the underlying multi-body dynamics engine. Each demonstration highlights a GN&C-relevant capability:

- Branching articulation with staged actuation and constraint locking.*
- Classical reaction-wheel attitude control on an articulated model.†
- Tightly coupled dynamics under non-traditional actuation using arm-mounted thrusters.

Branching solar-array deployment with staged actuation and joint locking

Objective This scenario demonstrates that a generally articulated, branching kinematic tree can be simulated and controlled without deriving problem-specific equations of motion, while still supporting GN&C-relevant features such as staged deployment logic, joint limits, and joint locking through message interfaces.

Setup The spacecraft consists of a rigid central hub with six solar panels arranged in a 3×2 branching configuration. Each panel is connected through a single-degree-of-freedom hinge joint and is deployed by applying an actuator torque about that hinge. To emulate a realistic deployment sequence and reduce dynamic coupling during actuation, deployment is performed in three phases: the center panels deploy first, followed by the right-side panels, and finally the left-side panels (Fig. 4). Joint limits and temporary joint "locks" are enforced using MuJoCo's constraint capabilities, enabling both hard stops and latch-like behavior at specified angles.

The complete multi-body definition is specified in MuJoCo XML and provided in Appendix A. The XML defines the kinematic tree by nesting `<body>` elements (with child bodies representing downstream links) and associating joints and geometries with each body‡. Degrees of freedom are introduced with `<joint>` elements (or `<freejoint>` for free-flying bodies); in this scenario, each deployable panel uses a `type="hinge"` joint to represent a single rotational DOF about a specified axis. Geometries specify both visualization and physical properties, including contact surfaces and mass properties. MuJoCo can either accept inertial parameters explicitly or infer them from geometry. Here, panels are represented as low-density thin rectangular prisms with denser cylindrical support beams, and MuJoCo is used to infer consistent inertial properties from this structural description, avoiding manual inertia calculations.

Message connections and control logic Each hinge is driven by a PID controller that tracks a smooth reference profile. The desired joint position and velocity are produced by two interpolator models that depend only on simulation time. The measured joint position and velocity are provided by the multi-body forward kinematics outputs associated with each joint. The controller computes a hinge torque command that is routed to the MuJoCo joint actuator via a message connection.

Figure 5 summarizes the message-level structure for a single joint. The interpolators output the desired joint position $\theta^*_{1\mathrm{n}}$ and velocity $\dot{\theta}^*_{1\mathrm{n}}$. The controller subscribes to these desired values as well as to the measured joint state $(\theta_{1\mathrm{n}}, \dot{\theta}_{1\mathrm{n}})$. Internally, the controller maintains an integral error state $e_{1\mathrm{n}}$, whose derivative $\frac{\mathrm{d}e_{1\mathrm{n}}}{\mathrm{d}t}$ is returned through the model state interface for numerical integration. The controller outputs a torque command $\tau_{1\mathrm{n}}$, which is applied at the joint by subscribing the MuJoCo actuator input to the controller output message.

*Full scenario available at: `https://github.com/AVSLab/basilisk/blob/develop/examples/mujoco/scenarioBranchingPanels.py`

†Full scenario available at: `https://github.com/AVSLab/basilisk/blob/develop/examples/mujoco/scenarioAttitudeFeedbackRWMuJoCo.py`

‡MuJoCo XML Reference, `https://mujoco.readthedocs.io/en/stable/XMLreference.html`, Accessed: 2026-01-19.

(a) Solar array stowed.

(b) Center solar panels deployed.

(c) Right solar panels deployed.

(d) Complete solar array deployed.

Figure 4: Rendering of a simulated spacecraft multi-body system during solar array deployment. The six panels are deployed in a staged sequence, illustrating the ability to simulate complex, branching articulation.

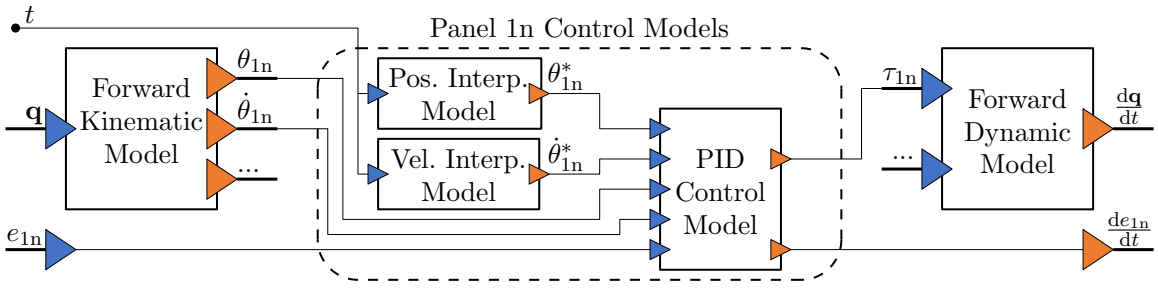

Figure 5: Message-based control architecture for a solar panel joint using PID control. Interpolator models provide desired position and velocity; the PID controller reads measured and desired values, computes control torque, and publishes it to the forward dynamics model. The integral error is tracked as a continuous-time state.

Listing 3 shows the corresponding setup in code for one hinge. A MuJoCo single-input joint actuator is instantiated for the hinge DOF, and message connections are created from the interpolators to the controller, from the joint kinematics messages to the controller, and from the controller to the actuator. The controller and interpolators are then scheduled into the dynamics task, ensuring that reference generation and torque computation occur consistently with dynamics propagation.

```
def addJointController(panelID: str, initialAngle: float, timeOffset: int):
    joint = joints[panelID]
    act = scene.addJointSingleActuator(f"panel_{panelID}_deploy", joint)

    pidController = PIDController(K_p=0.1, K_d=0.002, K_i=0.0001)

    positionInterpolator, velocityInterpolator = generateProfiles(
        initialPoint=initialAngle, finalPoint=0,
        vMax=np.deg2rad(0.05), aMax=np.deg2rad(0.0001),
        timeOffset=timeOffset
    )

    pidController.desiredInMsg.subscribeTo(positionInterpolator.
    interpolatedOutMsg)
    pidController.desiredDotInMsg.subscribeTo(velocityInterpolator.
    interpolatedOutMsg)
    pidController.measuredInMsg.subscribeTo(joint.stateOutMsg)
    pidController.measuredDotInMsg.subscribeTo(joint.stateDotOutMsg)
    act.actuatorInMsg.subscribeTo(pidController.outputOutMsg)

    scene.AddModelToDynamicsTask(positionInterpolator, priority=50)
    scene.AddModelToDynamicsTask(velocityInterpolator, priority=49)
    scene.AddModelToDynamicsTask(pidController, priority=25)

addJointController("10", initialAngle=np.pi/2, timeOffset=0)
addJointController("20", initialAngle=np.pi,   timeOffset=0)
addJointController("1p", initialAngle=np.pi,   timeOffset=operationTime)
addJointController("2p", initialAngle=np.pi,   timeOffset=operationTime)
addJointController("1n", initialAngle=np.pi,   timeOffset=2*operationTime)
addJointController("2n", initialAngle=np.pi,   timeOffset=2*operationTime)
```

Listing 3: Adding a PID controller for a panel hinge to follow a smooth deploy trajectory.

Staging is achieved by offsetting the start-time of the interpolators (Listing 2). Non-active joints are temporarily constrained to fixed angles by connecting their `constrainedStateInMsg` inputs to a constant joint-state message, effectively "locking" the corresponding hinge. When a deployment phase completes, the newly deployed joints are latched at their final angles and the next set of joints is unlocked and commanded. This staging logic exercises both message-routed actuation and message-controlled constraints within a single articulated simulation.

Results Figure 6 shows the commanded and actual hinge angles for all six panels throughout the three deployment phases. The commanded trajectories (solid blue) are closely tracked by the joint motion (dashed red), and the phase transitions occur at the intended times, confirming that the controller/actuator message wiring, time-offset reference scheduling, and joint locking/unlocking via constraints operate correctly together within the MuJoCo-backed dynamics task.

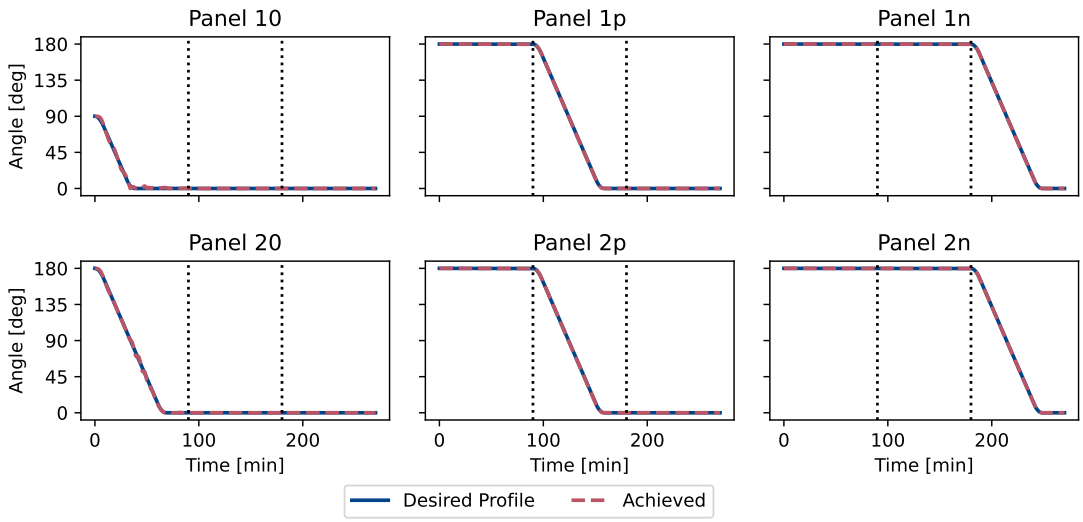

Figure 6: Time evolution of joint angles during staged solar array deployment. Each subplot shows the commanded (solid blue) and actual (dashed red) joint angle for one panel. Vertical dashed lines indicate deployment phases.

Reaction-wheel attitude control on an articulated model

Objective This scenario demonstrates that a conventional Basilisk GN&C control loop can operate directly on a general multi-body model, while the coupled hub-reaction-wheel dynamics are handled transparently by the dynamics engine. The example also highlights the benefits of representing reaction wheels as explicit articulated bodies: the same model naturally captures momentum exchange and back-torques on the hub, and it is straightforward to extend to non-ideal configurations (e.g., wheel misalignment or imbalances) by modifying joint axes or inertial properties in the XML.

Multi-body configuration The spacecraft is modeled in MuJoCo as a free-flying rigid hub with three reaction wheel bodies attached through single-degree-of-freedom hinge joints. The hub body is assigned a `<freejoint>`, enabling full translational and rotational motion, while each reaction wheel introduces a rotational degree of freedom aligned (nominally) with a principal body axis. Control authority is generated by applying motor torques at these hinge joints, which accelerate or decelerate the wheels. The resulting back-torques acting on the hub arise naturally from the multi-body dynamics solution, avoiding the need to explicitly derive and validate challenging coupled hub-reaction-wheel equations of motion.[18]

The complete MuJoCo XML description defining the hub, reaction wheel bodies, joints, and inertial properties is provided in Appendix B. Listing 4 illustrates how the hub and wheel bodies are retrieved from the `MJScene` and how the wheel spin joints are accessed. These joint handles are later used to instantiate actuators and connect GN&C torque commands (Listing 7).

```
busBody: mujoco.MJBody = scene.getBody("bus")
rw1Body: mujoco.MJBody = scene.getBody("rw1")
rw2Body: mujoco.MJBody = scene.getBody("rw2")
rw3Body: mujoco.MJBody = scene.getBody("rw3")

rw1Joint = rw1Body.getScalarJoint("rw1Spin")
rw2Joint = rw2Body.getScalarJoint("rw2Spin")
rw3Joint = rw3Body.getScalarJoint("rw3Spin")
```

Listing 4: Retrieving hub and wheel bodies from the MuJoCo scene and accessing wheel spin hinge joints.

This formulation also makes it easy to model common non-idealities. For example, wheel misalignment can be represented by perturbing the hinge joint axis in the XML, and wheel imbalance effects can be approximated by modifying the wheel inertial properties. Because the multi-body coupling is handled by the solver, these variations do not require re-deriving equations of motion or changing the GN&C interface.

Environmental forces and task structure To place the spacecraft in a representative orbital environment, a point-mass gravity model is added to the dynamics task. Gravity is configured with a central Earth source and applied to the hub and all reaction wheel bodies, as shown in Listing 5. Placing gravity at the level of individual bodies allows first-order gravity-gradient effects to arise naturally when bodies are spatially separated, although such effects are not dominant in this scenario.

```
gravity = NBodyGravity.NBodyGravity()
gravity.ModelTag = "gravity"
scene.AddModelToDynamicsTask(gravity)

muEarth = 0.3986004415e15  # [m^3/s^2]
earthPm = pointMassGravityModel.PointMassGravityModel()
earthPm.muBody = muEarth
gravity.addGravitySource("earth", earthPm, isCentralBody=True)

gravity.addGravityTarget("bus", busBody)
gravity.addGravityTarget("rw1", rw1Body)
gravity.addGravityTarget("rw2", rw2Body)
gravity.addGravityTarget("rw3", rw3Body)
```

Listing 5: Configuring a point-mass gravity model and applying gravitational forces to all bodies.

Figure 7 summarizes the overall message-based architecture and task separation. The gravity model is placed in the special *dynamics* task and is evaluated at every integrator substep, ensuring that gravitational forces are available at any intermediate integration point. The GN&C models are placed in a separate task and executed at a fixed rate of 10 Hz. The listings in this subsection illustrate these two scheduling patterns: dynamic-effect models are added via `scene.AddModelToDynamicsTask(...)` (Listing 5), while GN&C models are added via `sim.AddModelToTask(gncTaskName,...)` (Listings 6 and 7).

GN&C integration Navigation is performed using a standard Basilisk navigation model that consumes the hub center-of-mass state produced by the MuJoCo scene, as shown in Listing 6. The resulting attitude and angular velocity estimates are passed through a conventional guidance,

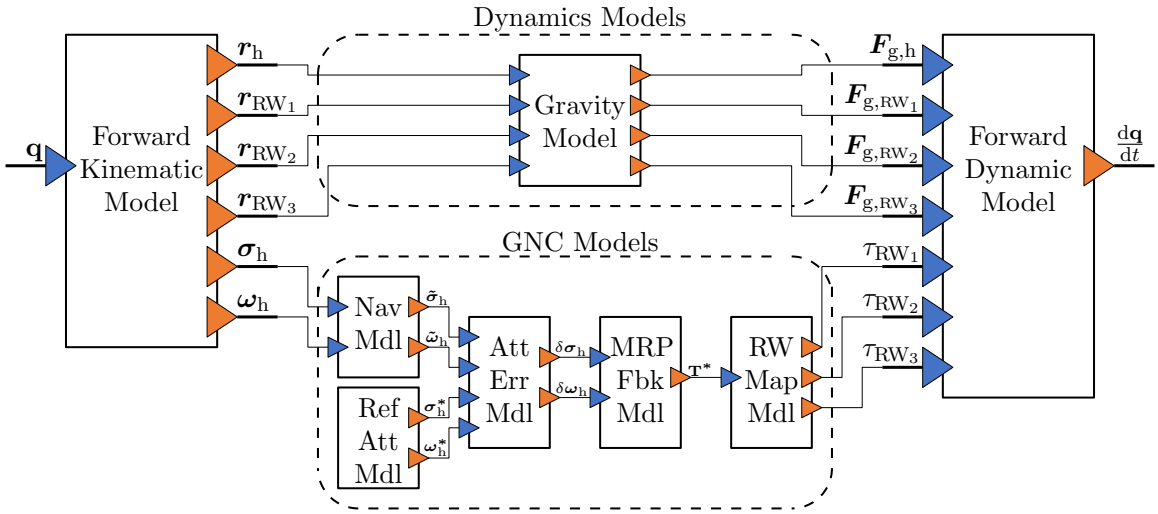

Figure 7: Message-based architecture for reaction wheel attitude control. The gravity model operates in the dynamics task, taking as input the positions of all bodies in the scene ($\boldsymbol{r}_{\text{h}}$, $\boldsymbol{r}_{\text{RW}_i}$) and outputting the corresponding gravitational forces ($\boldsymbol{F}_{g,hub}$, $\boldsymbol{F}_{g,RW_i}$). The GN&C models execute in a separate task at a fixed rate, producing reaction wheel torques (τ_{RW_i}) based on the hub attitude ($\boldsymbol{\sigma}_{\text{h}}$) and angular velocity ($\boldsymbol{\omega}_{\text{h}}$).

tracking error, and feedback control chain (Fig. 7).

```
simpleNavObj = simpleNav.SimpleNav()
simpleNavObj.ModelTag = "simpleNav"
simpleNavObj.scStateInMsg.subscribeTo(busBody.getCenterOfMass().stateOutMsg)
sim.AddModelToTask(gncTaskName, simpleNavObj)
```

Listing 6: Connecting a navigation model to the hub center-of-mass state from the multi-body engine.

The commanded control torques are mapped to individual reaction wheel motor torques and applied at the wheel joints through MuJoCo single-input actuators. Listing 7 illustrates creating one actuator per wheel spin DOF using the joint handles obtained in Listing 4 and then connecting the per-wheel torque command messages from the GN&C stack to the actuator input messages.

```
# Create one MuJoCo actuator per wheel spin DOF
rw1Act = scene.addJointSingleActuator("rw1Act", rw1Joint)
rw2Act = scene.addJointSingleActuator("rw2Act", rw2Joint)
rw3Act = scene.addJointSingleActuator("rw3Act", rw3Joint)

# Connect per-wheel motor torque commands to MuJoCo actuators
rw1Act.actuatorInMsg.subscribeTo(rwTorqueMap.actuatorOutMsgs[0])
rw2Act.actuatorInMsg.subscribeTo(rwTorqueMap.actuatorOutMsgs[1])
rw3Act.actuatorInMsg.subscribeTo(rwTorqueMap.actuatorOutMsgs[2])
```

Listing 7: Creating MuJoCo wheel actuators and connecting GN&C torque commands.

Results The simulation is initialized in Earth orbit with a perturbed initial attitude and angular rate, as well as nonzero initial reaction wheel spin rates. The control objective is to regulate the spacecraft to a fixed inertial attitude with zero angular rate. Figure 8 shows the time evolution of the attitude tracking error (Modified Rodrigues Parameters) together with the angular rate tracking error. The controller successfully drives both errors to zero, demonstrating that classical GN&C loops integrate seamlessly with the articulated dynamics backend while capturing the coupled hub-wheel dynamics through the multi-body solver.

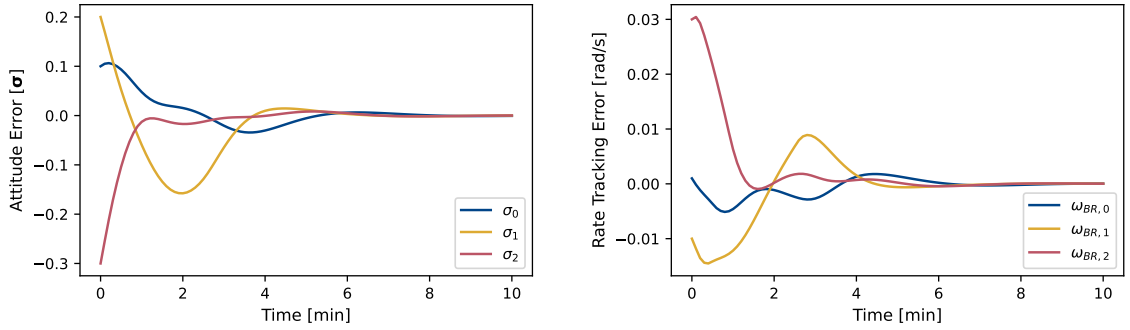

Figure 8: Attitude tracking error (Modified Rodrigues Parameters) and angular rate tracking error for the reaction wheel attitude control scenario.

Six-DOF control using thrusters mounted on robotic arms

Objective This scenario demonstrates a six-degree-of-freedom (DOF) spacecraft controlled exclusively using thrusters mounted on articulated robotic arms. The example highlights how the proposed message-passing simulation paradigm, implemented using Basilisk with MuJoCo as the underlying multi-body dynamics engine, enables rapid prototyping of non-traditional spacecraft control architectures.

Multi-body configuration The simulated spacecraft, shown in Figure 9, consists of a rigid central hub (red) connected to two identical robotic arms. Each arm comprises a primary (green) and secondary (orange) segment, connected by revolute joints, resulting in four rotational degrees of freedom per arm. A thruster is mounted at the end of each secondary segment, yielding two off-body thrusters capable of producing both translational forces and control torques on the spacecraft. Each of the eight joints is actuated by a motor, and an additional hub torque actuator is included to compensate for reaction torques generated during joint motion. The complete multi-body configuration, including body geometry, joint definitions, and inertial properties, is defined using a MuJoCo XML file provided in Appendix C.

Control architecture and message flow Control of the spacecraft is achieved using a cascaded architecture. An outer-loop controller computes the desired net force and torque required to regulate the spacecraft's position, velocity, attitude, and angular velocity $(\boldsymbol{r}_\mathrm{h}, \dot{\boldsymbol{r}}_\mathrm{h}, \boldsymbol{\sigma}_\mathrm{h}, \boldsymbol{\omega}_\mathrm{h})$. These desired quantities are mapped into thruster forces (F_1, F_2), a compensating hub torque ($\boldsymbol{T}_\mathrm{h}$), and target joint angle and velocity commands $(\boldsymbol{\theta}^*, \dot{\boldsymbol{\theta}}^*)$. An inner-loop controller then drives the joint angles to their commanded values using PD control torques $\boldsymbol{\tau}$, while the resulting thruster forces generate the commanded spacecraft motion. The full details of the cascaded control architecture will be presented in a forthcoming paper.

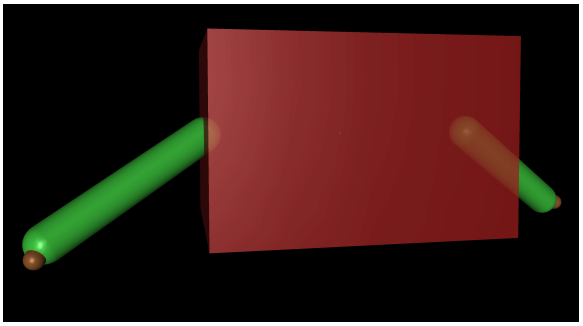

Figure 9: Rendering of a simulated spacecraft multi-body system utilizing robotic-arm-mounted thrusters. The spacecraft motion is controlled using these thrusters, demonstrating the ability to simulate highly coupled spacecraft–manipulator dynamics.

Figure 10 illustrates the message-based control architecture used in this scenario. Forward kinematics models provide spacecraft and joint states to the flight software models, while controller outputs are routed to joint actuators, the hub torque actuator, and thruster force actuators. As in the previous examples, the `MJScene` interface provides direct access to kinematic states and actuator interfaces through standardized message connections.

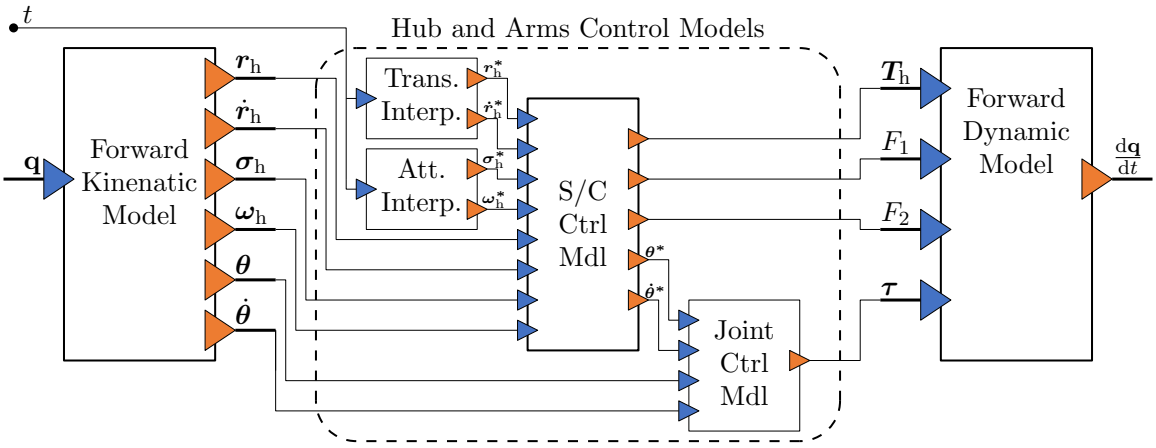

Figure 10: Message-based control architecture for spacecraft control using thrusters mounted on robotic arms. Forward kinematics models provide spacecraft and joint states to the control stack. The outer-loop controller computes desired thruster forces and joint angle commands, while the joint controller computes the corresponding joint and hub torques, which are applied through the multi-body dynamics engine.

Listing 8 shows how the hub and joint states are retrieved from the MuJoCo scene and connected to flight software input messages. The hub state is obtained from the hub origin frame, while each arm joint publishes scalar joint position and rate messages that feed the flight software model.

```python
hubOriginFrame: mujoco.MJSite = scene.getBody("hub").getOrigin()

joints: list[mujoco.MJScalarJoint] = []
for armNum in range(1,3):
    armBaseBody: mujoco.MJBody = scene.getBody(f"arm_{armNum}_base")
    for axis in ("x", "y"):
        jointName = f"arm_{armNum}_base_joint_{axis}"
        joints.append(armBaseBody.getScalarJoint(jointName))

    armTipBody: mujoco.MJBody = scene.getBody(f"arm_{armNum}_tip")
    for axis in ("z", "y"):
        jointName = f"arm_{armNum}_tip_joint_{axis}"
        joints.append(armTipBody.getScalarJoint(jointName))

fsw.scStatesInMsg.subscribeTo(hubOriginFrame.stateOutMsg)

for msgIdx, joint in enumerate(joints):
    fsw.jointInMsgs[msgIdx].subscribeTo(joint.stateOutMsg)
    fsw.jointDotInMsgs[msgIdx].subscribeTo(joint.stateDotOutMsg)
```

Listing 8: Retrieving hub and joint states from the MuJoCo scene and connecting them to flight software input messages.

Listing 9 shows the corresponding actuation wiring. Joint torque commands are routed to MuJoCo single-input joint actuators, commanded thruster forces are routed to thruster actuators, and the hub torque command is routed to a dedicated hub torque actuator. Together, Listings 8 and 9 demonstrate that the full cascaded controller can be connected to a highly coupled articulated plant using only message subscriptions, with no solver-specific calls embedded in the flight software logic.

```python
for msgIdx, joint in enumerate(joints):
    actName = f"{joint.getName()}_act"
    act: mujoco.MJSingleActuator = scene.addJointSingleActuator(actName, joint)
    act.actuatorInMsg.subscribeTo(fsw.jointTorqueOutMsgs[msgIdx])

for thrusterNum in range(1,3):
    actName = f"F_thruster_{thrusterNum}"
    act: mujoco.MJSingleActuator = scene.getSingleActuator()
    act.actuatorInMsg.subscribeTo(fsw.thrForceOutMsgs[thrusterNum-1])

hubTorqueAct: mujoco.MJTorqueActuator = scene.addTorqueActuator(
    "hub_torque_act", hubOriginFrame
)
hubTorqueAct.torqueInMsg.subscribeTo(fsw.hubTorqueOutMsg)
```

Listing 9: Connecting flight software torque and force outputs to joint, hub, and thruster actuators in the MuJoCo scene.

Results Figure 11 shows the time evolution of the spacecraft's translational and attitude errors during the simulation. The results demonstrate stable closed-loop control of both position and attitude using arm-mounted thrusters, validating the proposed message-driven multi-body dynamics architecture under a highly non-traditional control configuration. This example illustrates how the

framework supports tightly coupled spacecraft and manipulator dynamics with minimal additional simulation infrastructure.

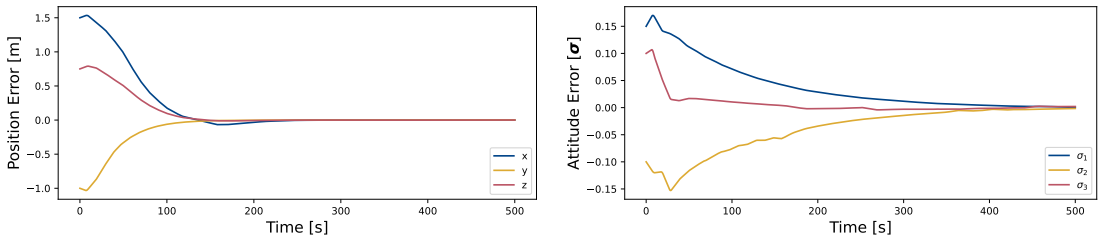

Figure 11: Time evolution of spacecraft translational and attitude regulation errors for the robotic-arm-mounted thruster scenario. The results demonstrate stable closed-loop performance using only off-body thrusters.

CONCLUSION

This paper presents a message-passing paradigm for simulating generally articulated spacecraft that is explicitly aligned with GN&C development and validation workflows. The central idea is to separate *how* multi-body motion is computed from *what* physical effects and control actions are applied: forward kinematics and forward dynamics are treated as solver-provided services, while gravity, actuation, constraint logic, and other mission-specific effects are implemented as modular models that communicate through typed messages. This division isolates solver-specific complexity, improves transparency of modeling assumptions, and encourages reuse of validated dynamic-effect models across vehicles and missions.

A key practical outcome is that the paradigm preserves Basilisk's standard simulation semantics. State-dependent environment forces are evaluated at integrator substeps within a dedicated dynamics task, while navigation, guidance, and control models execute in a separate fixed-rate task. The resulting workflow supports physically consistent integration without forcing the GN&C stack to operate at the dynamics integrator's internal step size.

The paradigm is implemented in Basilisk by integrating MuJoCo as an alternate dynamics backend while retaining Basilisk messaging and tasking. Three demonstrations exercise the framework on GN&C-relevant problems: staged deployment of a branching solar-array mechanism using PID actuation and joint locking; reaction-wheel attitude control with wheels modeled as articulated bodies to capture coupled hub-wheel dynamics without hand-derived equations; and six-degree-of-freedom control using thrusters mounted on actuated robotic arms, illustrating highly coupled and non-traditional actuation concepts. Collectively, these examples show that message-driven multi-body simulation bridges robotics-grade articulated dynamics and spaceflight-focused astrodynamics environments while remaining accessible to GN&C engineers.

FUTURE WORK

Future work will benchmark and validate the proposed approach against Basilisk's existing back-substitution method in regimes where the two methods overlap.[21] Additional demonstrations will target GN&C-relevant scenarios that benefit from general articulation and constraints, including multi-vehicle simulations for proximity operations and formation flying, contact-rich interactions such as docking, and expanded environmental and actuator model libraries.

ACKNOWLEDGMENT

The research was supported by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration (80NM0018D0004).

Artificial intelligence (AI) tools were used in the preparation of this manuscript in accordance with AIAA guidelines. Specifically, AI was employed as a software development assistant to support code workflows relevant to the work. Additionally, AI was used to review and improve the clarity, grammar, and readability of the manuscript text.

REFERENCES

[1] J. Garcia-Bonilla, C. Leake, A. Elmquist, T. D. Hasseler, V. Steyert, A. Gaut, and A. Jain, "Dshell-DARTS: A Reusability-Focused Multi-Mission Aerospace and Robotics Simulation Toolkit," *2025 IEEE Aerospace Conference*, 2025, pp. 1–13, 10.1109/AERO63441.2025.11068690.

[2] P. W. Kenneally, S. Piggott, and H. Schaub, "Basilisk: A Flexible, Scalable and Modular Astrodynamics Simulation Framework," *Journal of Aerospace Information Systems*, Vol. 17, No. 9, 2020, pp. 496–507, 10.2514/1.I010762.

[3] M. Cols-Margenet, H. Schaub, and S. Piggott, *Modular Attitude Guidance Development using the Basilisk Software Framework*, 10.2514/6.2016-5538.

[4] D. M. Geletko, M. D. Grubb, J. P. Lucas, J. R. Morris, M. Spolaor, M. D. Suder, S. C. Yokum, and S. A. Zemerick, "NASA Operational Simulator for Small Satellites (NOS3): the STF-1 CubeSat case study," 2019.

[5] M. Tipaldi, R. Iervolino, and P. R. Massenio, "Reinforcement learning in spacecraft control applications: Advances, prospects, and challenges," *Annual Reviews in Control*, Vol. 54, 2022, pp. 1–23, https://doi.org/10.1016/j.arcontrol.2022.07.004.

[6] A. K. Banerjee, "Contributions of Multibody Dynamics to Space Flight: A Brief Review," *Journal of Guidance, Control, and Dynamics*, Vol. 26, No. 3, 2003, pp. 385–394, 10.2514/2.5069.

[7] A. Seddaoui, C. M. Saaj, and M. H. Nair, "Modeling a Controlled-Floating Space Robot for In-Space Services: A Beginner's Tutorial," *Frontiers in Robotics and AI*, Vol. 8, 12 2021, p. 725333, 10.3389/FROBT.2021.725333/FULL.

[8] Y. Qi and M. Shan, "Simulation on Flexible Multibody System with Topology Changes for In-space Assembly," *Journal of the Astronautical Sciences*, Vol. 71, Apr. 2024, p. 11, 10.1007/s40295-024-00431-0.

[9] J. Vaz Carneiro and H. Schaub, "Scalable architecture for rapid setup and execution of multi-satellite simulations," *Advances in Space Research*, Vol. 73, No. 11, 2024, pp. 5416–5425. Recent Advances in Satellite Constellations and Formation Flying, https://doi.org/10.1016/j.asr.2023.11.026.

[10] W. Qian, Z. Xia, J. Xiong, Y. Gan, Y. Guo, S. Weng, H. Deng, Y. Hu, and J. Zhang, "Manipulation task simulation using ROS and Gazebo," *2014 IEEE International Conference on Robotics and Biomimetics (ROBIO 2014)*, 2014, pp. 2594–2598, 10.1109/ROBIO.2014.7090732.

[11] E. Todorov, T. Erez, and Y. Tassa, "MuJoCo: A physics engine for model-based control," *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012, pp. 5026–5033, 10.1109/IROS.2012.6386109.

[12] G. Boschetti and T. Sinico, "Designing Digital Twins of Robots Using Simscape Multibody," *Robotics*, Vol. 13, No. 4, 2024, 10.3390/robotics13040062.

[13] S. P. Hughes, R. H. Qureshi, S. D. Cooley, and J. J. Parker, *Verification and Validation of the General Mission Analysis Tool (GMAT)*. 2014, 10.2514/6.2014-4151.

[14] A. Wall, *Systems Tool Kit (STK)*, pp. 11–32. CRC Press, 1 2024, 10.1201/9781003321811-4/SYSTEMS-TOOL-KIT-STK-ALEXIS-WALL.

[15] G. Dell, M. Hametz, P. Noonan, L. Newman, D. Folta, and J. Bristow, *EOS AM-1 and EO-1 support using FreeFlyer and AutoCon*. 1998, 10.2514/6.1998-4196.

[16] J. a. Vaz Carneiro, C. Allard, and H. Schaub, "General Dynamics for Single- and Dual-Axis Rotating Rigid Spacecraft Components," *Journal of Spacecraft and Rockets*, Vol. 61, No. 4, 2024, pp. 1099–1113, 10.2514/1.A35865.

[17] A. M. Morell, J. V. Carneiro, L. Kiner, and H. Schaub, *Multi-Arm Post-Docking Spacecraft Dynamics Using Penalty Methods*, 10.2514/6.2024-1870.

[18] J. Alcorn, C. Allard, and H. Schaub, "Fully Coupled Reaction Wheel Static and Dynamic Imbalance for Spacecraft Jitter Modeling," *Journal of Guidance, Control, and Dynamics*, Vol. 41, No. 6, 2018, pp. 1380–1388, 10.2514/1.G003277.

[19] J. Garcia-Bonilla and H. Schaub, "A Message-Passing Simulation Framework for Generally Articulated Spacecraft Dynamics," *Proceedings of the AAS/AIAA Astrodynamics Specialist Conference*, Boston, Massachusetts, Aug. 2025.

[20] S. Carnahan, S. Piggott, and H. Schaub, "A New Messaging System For Basilisk," *AAS Guidance and Control Conference*, Breckenridge, CO, Jan. 30 – Feb. 5 2020. AAS 20-134.

[21] C. Allard, M. Diaz Ramos, and H. Schaub, "Computational Performance of Complex Spacecraft Simulations Using Back-Substitution," *Journal of Aerospace Information Systems*, Vol. 16, No. 10, 2019, pp. 427–436, 10.2514/1.I010713.

APPENDIX A: SOLAR ARRAY SCENARIO MULTI-BODY DEFINITION

The following is the content of the XML file used to define the multi-body system discussed in Section “Multi-body Configuration”.

```
<mujoco>
  <option gravity="0 0 0">
    <flag contact="disable"/>
  </option>

  <worldbody>
    <body name="hub">
      <freejoint/>
      <geom name="hub_box" type="box" size="1 1 1" density="2700" rgba="1 0 0 0.5"/>

      <body name="panel_10" pos="1 0 -1" xyaxes="0 -1 0 0 0 -1">
        <joint name="panel_10_deploy" type="hinge" axis="1 0 0" range="0 90"/>
        <geom name="panel_10_box" type="box"  pos="0 0 1" size="1 0.05 0.8" density="200" rgba="0 1 0 1"/>
        <geom name="panel_10_bar" type="cylinder" fromto="0 0 0 0 0 2" size=".075" density="1000" rgba="0 1 1 1"/>
        <geom name="panel_10_cbar" type="cylinder" fromto="1 0 1 -1 0 1" size=".075" density="1000" rgba="0 1 1 1" />

        <body name="panel_1p" pos="1 0 1" axisangle="0 1 0 90">
                  <joint name="panel_1p_deploy" type="hinge" axis="1 0 0" range="0 180"/>
                  <geom name="panel_1p_box" type="box"  pos="0 0 1" size="0.8 0.05 0.8" density="200" rgba="0 1 0 1"/>
          <geom name="panel_1p_bar" type="cylinder" fromto="0 0 0 0 0 1.8" size=".075" density="1000" rgba="0 1 1 1" />
                </body>

        <body name="panel_1n" pos="-1 0 1" axisangle="0 -1 0 90">
                  <joint name="panel_1n_deploy" type="hinge" axis="1 0 0" range="0 180"/>
                  <geom name="panel_1n_box" type="box"  pos="0 0 1" size="0.8 0.05 0.8" density="200" rgba="0 1 0 1"/>
          <geom name="panel_1n_bar" type="cylinder"  fromto="0 0 0 0 0 1.8" size=".075" density="1000" rgba="0 1 1 1" />
                </body>

                <body name="panel_20" pos="0 0 2">
                  <joint name="panel_20_deploy" axis="-1 0 0" range="0 180"/>
                  <geom name="panel_20_box" type="box"  pos="0 0 1" size="1 0.05 0.8" density="200" rgba="0 1 0 1"/>
          <geom name="panel_20_bar" type="cylinder"  fromto="0 0 0 0 0 1.8" size=".075" density="1000" rgba="0 1 1 1" />
          <geom name="panel_20_cbar" type="cylinder" fromto="1 0 1 -1 0 1" size=".075" density="1000" rgba="0 1 1 1" />

          <body name="panel_2p" pos="1 0 1" axisangle="0 1 0 90">
            <joint name="panel_2p_deploy" type="hinge" axis="1 0 0" range="0 180"/>
            <geom name="panel_2p_box" type="box"  pos="0 0 1" size="0.8 0.05 0.8
```

```
    " density="200" rgba="0 1 0 1"/>
38            <geom name="panel_2p_bar" type="cylinder"  fromto="0 0 0 0 0 1.8"
    size=".075" density="1000" rgba="0 1 1 1" />
39          </body>
40
41          <body name="panel_2n" pos="-1 0 1" axisangle="0 -1 0 90">
42            <joint name="panel_2n_deploy" type="hinge" axis="1 0 0" range="0 180
    "/>
43            <geom name="panel_2n_box" type="box"  pos="0 0 1" size="0.8 0.05 0.8
    " density="200" rgba="0 1 0 1"/>
44            <geom name="panel_2n_bar" type="cylinder"  fromto="0 0 0 0 0 1.8"
    size=".075" density="1000" rgba="0 1 1 1" />
45          </body>
46              </body>
47      </body>
48
49    </body>
50
51  </worldbody>
52
53 </mujoco>
```

APPENDIX B: REACTION WHEEL ATTITUDE CONTROL MULTI-BODY DEFINITION

The following is the content of the XML file used to define the multi-body system discussed in Section "Reaction Wheel Attitude Control Example".

```
<mujoco>
  <option>
    <flag contact="disable"/>
  </option>

  <worldbody>
    <body name="bus" pos="0 0 0">
      <!-- Free-flying 6-DOF base -->
      <freejoint name="busFree"/>

      <!-- Bus rigid-body inertial properties -->
      <inertial pos="0 0 0" mass="750.0"
                diaginertia="900.0 800.0 600.0"/>

      <!-- Simple visual geometry -->
      <geom type="box" size="0.6 0.5 0.4" rgba="0.6 0.6 0.6 1"/>

      <!-- RW1: spin about +X, colocated at bus origin -->
      <body name="rw1" pos="0 0 0">
        <joint name="rw1Spin" type="hinge" axis="1 0 0" limited="false"/>
        <inertial pos="0 0 0" mass="9.0"
                  diaginertia="0.07957747154594767 0.039788735772973836
    0.039788735772973836"/>
        <geom type="cylinder" size="0.08 0.03" rgba="0.2 0.2 0.8 1" euler="0 90
    0"/>
      </body>

      <!-- RW2: spin about +Y, colocated at bus origin -->
      <body name="rw2" pos="0 0 0">
        <joint name="rw2Spin" type="hinge" axis="0 1 0" limited="false"/>
        <inertial pos="0 0 0" mass="9.0"
                  diaginertia="0.039788735772973836 0.07957747154594767
    0.039788735772973836"/>
        <geom type="cylinder" size="0.08 0.03" rgba="0.2 0.8 0.2 1" euler="90 0
    0"/>
      </body>

      <!-- RW3: spin about +Z, offset at (0.5, 0.5, 0.5) m -->
      <body name="rw3" pos="0.5 0.5 0.5">
        <joint name="rw3Spin" type="hinge" axis="0 0 1" limited="false"/>
        <inertial pos="0 0 0" mass="9.0"
                  diaginertia="0.039788735772973836 0.039788735772973836
    0.07957747154594767"/>
        <geom type="cylinder" size="0.08 0.03" rgba="0.8 0.2 0.2 1"/>
      </body>

    </body>
  </worldbody>
</mujoco>
```

APPENDIX C: ROBOTIC-ARM-MOUNTED THRUSTER MULTI-BODY DEFINITION

The following is the content of the XML file used to define the multi-body system discussed in Section “Robotic-arm-mounted Thruster Example”.

```
<mujoco>
  <option>
    <flag contact="disable"/>
  </option>

  <worldbody>
    <body name="hub">
      <freejoint />
      <geom name="hub_box" type="box" size="0.79 0.69 0.52" rgba="1 0 0 0.5"
    mass="330" />

      <body name="arm_1_base" pos="0.79 0 0">
        <joint name="arm_1_base_joint_x" axis="1 0 0" />
        <joint name="arm_1_base_joint_y" axis="0 1 0" />
        <geom name="boom_1" type="capsule" size="0.1" fromto="0 0 0 1 0 0" rgba=
    "0 1 0 1" mass="8" />

        <body name="arm_1_tip" pos="1 0 0">
          <joint name="arm_1_tip_joint_z" axis="0 0 1" />
          <joint name="arm_1_tip_joint_y" axis="0 1 0" />
          <geom name="tip_1" type="capsule" size="0.05" fromto="0 0 0 0.1 0 0"
    rgba="0.733 0.32 0 1" mass="2" />
          <site name="thruster_1" pos="0.1 0 0" />
        </body>
      </body>

      <body name="arm_2_base" pos="-0.79 0 0" xyaxes="-1 0 0 0 -1 0">
        <joint name="arm_2_base_joint_x" axis="1 0 0" />
        <joint name="arm_2_base_joint_y" axis="0 1 0" />
        <geom name="boom_2" type="capsule" size="0.1" fromto="0 0 0 1 0 0" rgba=
    "0 1 0 1" mass="8" />

        <body name="arm_2_tip" pos="1 0 0">
          <joint name="arm_2_tip_joint_z" axis="0 0 1" />
          <joint name="arm_2_tip_joint_y" axis="0 1 0" />
          <geom name="tip_2" type="capsule" size="0.05" fromto="0 0 0 0.1 0 0"
    rgba="0.733 0.32 0 1" mass="2" />
          <site name="thruster_2" pos="0.1 0 0" />
        </body>
      </body>
    </body>
  </worldbody>

  <actuator>
    <motor name="F_thruster_1" site="thruster_1" gear="-1 0 0 0 0 0" />
    <motor name="F_thruster_2" site="thruster_2" gear="-1 0 0 0 0 0" />
  </actuator>
</mujoco>
```