Available online at www.sciencedirect.com

ScienceDirect

Advances in Space Research 73 (2024) 5416–5425

ADVANCES IN SPACE RESEARCH (a COSPAR publication)

www.elsevier.com/locate/asr

Scalable architecture for rapid setup and execution of multi-satellite simulations

João Vaz Carneiro *, Hanspeter Schaub

Ann and H.J. Smead Department of Aerospace Engineering Sciences, University of Colorado Boulder, 431 UCB, Colorado Center for Astrodynamics Research, Boulder, CO 80309, United States

Received 9 November 2022; received in revised form 10 November 2023; accepted 17 November 2023
Available online 20 November 2023

Abstract

This work introduces a novel architecture that allows for easy integration of multiple satellites in a single simulation. The new design focuses on modularity, expandability, and easy scriptability while maintaining Basilisk's high-fidelity and speed features. Modularity is important to make highly specialized simulations, which include parameters related to the environment but also specific to each spacecraft. Expandability and scriptability are also key, as one of the goals is to facilitate the creation of simulations with a large number of satellites. The architecture is implemented in Basilisk, an open-source, flight-proven physics and flight software engine, although the fundamental principles can be applied to any software application. Through an overhauled messaging system, the architecture also allows for easy addition of homogeneous or heterogeneous satellites with reduced overhead for the user. This paper shows code snippets and multithreading simulation performance to discuss how the architecture achieves its underlying objectives of simplicity, accessibility, and performance.

Keywords: Spacecraft simulations; Software architecture; Spacecraft formation flying

1. Introduction

The interest in spacecraft constellations has rapidly developed over the years, attracting government agencies to sponsor both science and defense-oriented projects and private commercial companies in the broadband communication field. Spacecraft constellations have several when compared to single-spacecraft missions. They allow for better, more continuous Earth coverage, which is helpful for communications and science missions alike. Moreover, by spreading the effort among multiple spacecraft, the mission risk decreases, as a critical failure on one of the satellites does not necessarily mean the end of the mission.

The Global Positioning System (GPS), designed initially by the U.S. Department of Defense in the 1970s, was the first global navigation satellite system to be operational and is arguably one the most important constellation of satellites orbiting Earth (Enge, 1994). The National Aeronautics and Space Administration (NASA) has also invested in projects that take advantage of multiple spacecraft in orbit like CYGNSS (Carreno-Luengo et al., 2021; Ruf et al., 2017). This project uses eight micro-satellites to collect the first frequent space-based measurements of surface wind speeds in the inner core of tropical cyclones. The European Space Agency (ESA) launched the Swarm mission in 2013 to study Earth's magnetic field and electric currents (Macmillan and Olsen, 2013; Friis-Christensen et al., 2008; Friis-Christensen et al., 2006). All these projects are examples of missions that take advantage of the flexibility of having multiple satellites orbiting Earth,

* Corresponding author.
E-mail addresses: joao.carneiro@colorado.edu (J. Vaz Carneiro), hanspeter.schaub@colorado.edu (H. Schaub).

https://doi.org/10.1016/j.asr.2023.11.026
0273-1177/

whether to take measurements of Earth or to allow for accurate position determination.

As for private endeavors, the most notable is the Starlink project (SpaceX, 2022), which focuses on delivering internet services to remote and under-served locations. SpaceX finances it and has already launched over two thousand satellites, with tens of thousands in development. Amazon's Project Kuiper (Amazon, 2022) and OneWeb (OneWeb, 2022), while in an earlier development stage, aim to tackle a similar problem. The greater coverage and decreased cost of multiple satellites in lower Earth orbits (LEO and MEO) have allowed these communication companies to populate large areas of Earth's orbit with smaller satellites, providing cost-effective internet services to remote populations.

Undoubtedly, large spacecraft constellations will continue to become a core part of the spacecraft population in Earth's orbit, especially with the recent commercialization of space (Valinia et al., 2019). Their successful implementation requires sophisticated software that accurately simulates each spacecraft in orbit. Extensive simulations are a vital part of any mission, from dynamical and hardware-in-the-loop validation and verification of mission requirements to post-launch telemetry analysis. Many software tools can simulate a single spacecraft with high fidelity and speed. MATLAB/Simulink (MathWorks, 2022) is widely used in industry for its ease of use and the ability to auto-generate C code. NASA's General Mission Analysis Tool (GMAT) (NASA, 2022b) and Analytic Graphic, Inc.'s Systems Tool Kit (STK) (AGI, 2022) are powerful software packages that focus on simulating high-fidelity orbital dynamics, including orbit propagation, common orbital perturbations, as well as trajectory design and optimization. NASA's '42' (NASA, 2022a) and Jet Propulsion Laboratory's (JPL) Dynamics Algorithms for Real-Time Simulation (DARTS) are both capable of simulating complex spacecraft behavior, including attitude dynamics and closed-loop control. While very powerful, all these software packages have drawbacks. Matlab/Simulink runs slower than other software packages based on C/C++. At the same time, there is a C auto-generation feature, but the resulting code is not human-readable, making it very difficult to optimize and debug. GMAT and STK are developed with orbit simulations in mind and currently cannot simulate complex spacecraft attitude dynamics. NASA's '42' does not mention being able to run on different threads, and while DARTS does have that capability, it is not readily available to the public.

A simulation with many satellites can become cumbersome if the software architecture is not built for multi-spacecraft prototyping. Creating a simulation with multiple satellites brings new challenges that many software packages are unprepared to deal with. Usually, the user must manually include and specify every spacecraft, which for large constellations gets increasingly time-consuming and yields cluttered and hard-to-follow scripts. Moreover, if the architecture is not built with multi-spacecraft simulations in mind, it likely simulates every spacecraft in series. This means that the additional time it takes to run the simulation will roughly increase linearly with the number of spacecraft, which becomes a problem if the goal is to simulate tens or hundreds of satellites at a time. Recently, the industry has taken note of these issues, and there has been a push to make the simulations of a large number of satellites viable. For example, STK version 12.4 allows users to easily create a constellation of hundreds or thousands of satellites by specifying a multi-shelled Walker constellation, a subset of the public space catalog, or by using their custom orbit elements or ephemerides. The architecture proposed in this paper aims to tackle the challenges of multi-satellite simulations to make them simple to set up and maintain while using an engineering-friendly Python scripting interface.

While the focus of this paper is to propose a general architecture framework that supports simulations of multiple spacecraft, the specific software implementation is also addressed. The Basilisk (https://hanspeterschaub.info/basilisk) astrodynamics software tool implements the architecture and creates the example scenarios for this work. Basilisk is a flight-proven modular mission simulation framework used to set up high-fidelity simulations (Kenneally et al., 2020). Its modular nature (Allard et al., 2018a) allows for the simple integration of complex simulation tasks, such as power generation and consumption, fully-coupled attitude control devices (Alcorn et al., 2018; Sasaki et al., 2018), complex multi-body dynamics (Allard et al., 2018b; Panicucci et al., 2018), and orbital perturbations. Modules are created using C/C++ for rapid execution, while the user interacts and connects modules using Python for easy scriptability and rapid prototyping. While multi-satellite simulations have been developed using version 1 of Basilisk, the associated messaging system made creating multiple satellites very cumbersome as each message had to have a unique message name. (Kenneally et al., 2020).

This paper explores a new architecture that is able to create modular and extendable multi-satellite simulations. Implementing this redefined architecture is made possible by deploying the new messaging system in Basilisk version 2 (Carnahan et al., 2020). While multi-satellite simulations had been created using the old messaging system with Basilisk 1, the overhauled messaging system makes the simulation design substantially easier to code and scale to a large number of satellites. With its peer-to-peer message connections, Basilisk 2 allows the modules to be easily connected upon initialization without having the user figure out how to connect and name modules from a single message pool.

This paper is organized as follows. Section 2 investigates the underlying principles used to build the proposed architecture. Section 3 goes into detail about how the architecture is built and what its components are. Section 4 explains the capabilities of the new messaging system and does a comparison with the previous implementation. Section 5 dives into how a simulation runs under the new

architecture. Section 6 gives some qualitative examples regarding the implementation of the new architecture. Finally, Section 7 summarizes the points discussed in this paper.

2. Architecture design

The proposed redesign aims to create an architecture that effectively simulates multiple satellites while preserving computational speed and modular model fidelity. This novel framework is based on four principles: modularity, scalability, parallelization, and scriptability.

2.1. Modularity

In this context, modularity is the ability of a software framework to be divided into smaller pieces that can be linked together to create the simulation. It means that each simulation feature or module is detached from another, and multiple modules must be added and linked to run the simulation as intended. While this takes a toll on simulation time, as the modules are run separately, it saves development time.

This is particularly important in the context of complex spacecraft simulation. For example, different missions may require different attitude control systems. Some may use reaction wheels for their precise pointing characteristics, while others may use control moment gyroscopes (CMGs) for their larger torque needs. While physically different in generating requested torques, both systems have the same objective of generating a requested torque from an attitude control law. By modularizing the software framework, the developer can quickly switch between each attitude control device, implemented as distinct modules, without overhauling the entire attitude control system. Basilisk has been built from the ground up as a modular system, and this work takes advantage of this structure.

However, for the simulation to run, the individual modules must communicate and share information. The requested torque from the attitude control module must be passed onto the attitude control device module (reaction wheels or CMGs) for the attitude control system. In Basilisk, the information is shared through a messaging system, which is discussed in-depth in Section 4.

2.2. Scalability/expandability

Scalability, or expandability, represents the ability to increase the number of satellites in a single simulation. This is critical for scenarios with a large number of satellites, which sets this architecture apart from usual software designs.

For this work, scalability is achieved by standardizing the class creation that sets up the simulation environment and the classes responsible for simulating each spacecraft's dynamics and flight software (FSW) routines. Adding a new spacecraft is implemented through a loop that creates and connects every module necessary for every spacecraft. This way, the user can add and customize as many spacecraft as needed with no major changes to the framework.

2.3. Parallelization

Parallelization allows the software to exploit the architecture of modern CPUs. Nowadays, most processing units contain multiple cores, with some cores consisting of multiple threads. Different processes can be run simultaneously using different threads.

The importance of parallelization for multi-spacecraft simulation is clear. Running each spacecraft's process in parallel lessens the otherwise steep linear increase in simulation time for an increasing number of satellites. Simulating each spacecraft in parallel decreases the simulation time associated with more satellites, although the number of available threads limits this improvement .

This parallelization allows for multithreading when multiple threads are used simultaneously for different processes. Multithreading has implementation challenges, such as when two threads read and modify the same data simultaneously or when two threads are not properly coordinated in time. Making a program multithread-safe is non-trivial. For this work, the proposed architecture allows for multithreading of spacecraft constellations, not spacecraft formations. The difference relies on the fact that spacecraft in a constellation do not depend on each other. In contrast, their trajectory or flight modes in a formation can depend on other spacecraft.

It should be noted that it is impossible to parallelize a single spacecraft's dynamics, as it contains strongly coupled nonlinear differential equations. However, it is possible to run the dynamics of each spacecraft in separate threads. Thus, with the presented multi-threaded approach, the simulation speed of individual spacecraft is not increased. Rather, the numerical speed-up is achieved by simulating many spacecraft simultaneously.

2.4. Scriptability

Scriptability relates to the ease of simulation setup and development. Performing a simulation with multiple spacecraft usually implies code repetition, inefficient routines, and hard-to-follow scripts. Since this architecture is focused on allowing the implementation of multi-satellite simulations in a user-friendly way, the effort of creating multiple satellites (whether homogeneous or heterogeneous) is greatly simplified. This allows the user to focus on adding the necessary modules to create the intended simulation. Another advantage is that a user can create different classes that, if implemented correctly, can be changed between each other in a plug-and-play fashion. For example, two different environment classes, one around Earth and another around Mars, can be created and swapped as easily as changing a single line of code in the main script. It also means that debugging is vastly simpli-

fied, as the scripts are organized per spacecraft, and bugs are easier to track.

In addition, no recompiling is necessary to create and modify the simulation scripts. The user can add, connect, and change all simulation modules in Python without having to go through the lengthy process of recompiling the C and C++ Basilisk files.

3. Architecture framework

Guided by the goals and constraints expressed in the previous section, the multi-satellite architecture is now presented. The architecture diagram is shown in Fig. 1.

There are four classes: Master, Environment (Env), Dynamics (Dyn), and Flight Software (FSW). Only one Master and Environment classes exist, while there exist as many instantiations of Dynamics and Flight Software classes as the number of satellites in the simulation. The connections between class instances are done through modules taking advantage of the messaging system, which is used to share information between them.

Each spacecraft's Dynamics and Flight Software class instances are independent, so they can be run simultaneously, saving computation time in multithreaded scenarios. This architecture is also easily expandable by attaching more Dyn and FSW class instantiations according to the total number of satellites.

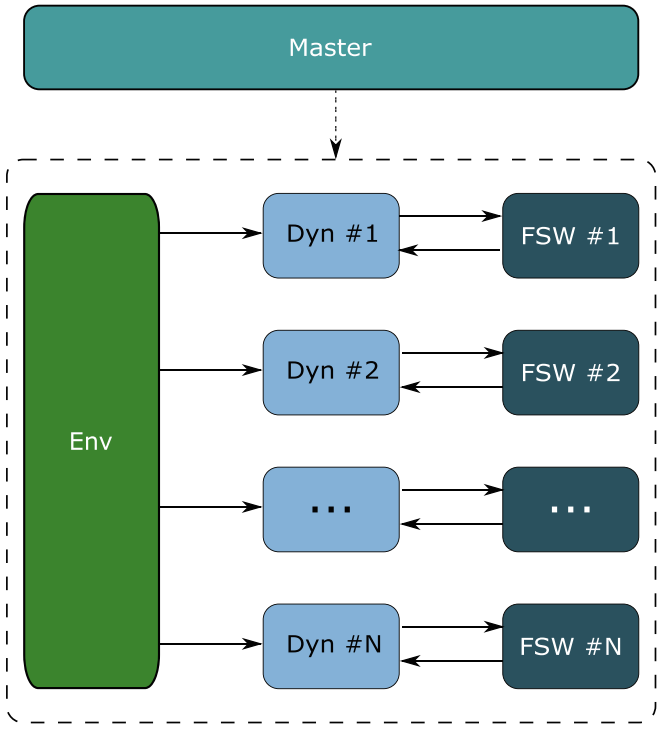

Fig. 1. Multi-satellite architecture diagram. Solid arrows represent information sharing through messages. The dashed arrow below the Master class corresponds to the functions that initialize and access all classes. The dashed box encompasses all the simulation classes, which contain the modules run during the simulation.

3.1. Master class

The Master class contains the methods that create and manage all other classes. This class is created in the scenario script, where these methods can be called for initialization and retrieval. To attach the Environment class to the simulation, see the code snippet in Listing 1. Here, `BSK_EnvEarth` is a Python script that contains the Environment class, with its modules inside.

Similarly, the Dynamics and Flight Software classes are attached through the function call shown in Listing 2. Again, `BSK_MultiSatDyn` and `BSK_MultiSatFsw` are Python scripts that contain the Dynamics and Flight Software classes, respectively. The argument of the methods shown is a list of Dyn or FSW classes. In this case, all spacecraft are the same, so a list of identical classes is added as an input. If a heterogeneous set of satellites is to be implemented, the list would contain different classes in the proper order, as shown in Listing 3. The user would have to create each class to suit the simulation requirements.

The methods that access the Env, Dyn, and FSW classes are also found in the Master class. Within the scenario script, the functions in Listing 4 are called to retrieve and access all classes and their modules.

3.2. Environment class

The environment class contains modules that are not spacecraft-specific but that instead describe the simulation environment. The gravity field is modeled within this class, along with atmospheric perturbations such as the effect of drag through an atmospheric density model. Ground stations for communications or imaging are also added to this class.

```
self.set_EnvModel(BSK_EnvEarth)
```

Listing 1. Attach an environment model to the simulation.

```
self.set_DynModel([BSK_MultiSatDyn]*numberSpacecraft)
self.set_FswModel([BSK_MultiSatFsw]*numberSpacecraft)
```

Listing 2. Attach dynamic and flight software models to the simulation.

```
self.set_DynModel([BSK_MultiSatDyn1, BSK_MultiSatDyn2,
↪  BSK_MultiSatDyn3])
self.set_FswModel([BSK_MultiSatFsw1, BSK_MultiSatFsw2,
↪  BSK_MultiSatFsw3])
```

Listing 3. Attach heterogeneous Dynamic and Flight Software models to the simulation.

```
EnvModel = self.get_EnvModel()
DynModels = self.get_DynModel()
FswModels = self.get_FswModel()
```

Listing 4. Retrieve simulation classes.

Since this class is not spacecraft-specific, it is shared among all spacecraft classes. This also means the user can readily change between different environments, such as the Martian or Cislunar environments, without making changes to each spacecraft. However, one must be careful about setting each spacecraft's initial conditions: a reasonable orbit around the Moon might not work around Earth. To solve this, the spacecraft's initial conditions are set using orbital elements with a canonical semi-major axis value, i.e., the semi-major axis depends on the main body's radius, with the constraint that the orbit's periapsis must be larger than the main body's radius.

3.3. Dynamics class

The dynamics class contains the modules that recreate the spacecraft and its components, which means one must exist for each spacecraft. While in most cases, all spacecraft are identical, there may be situations where the simulated spacecraft may have to be different. This might be the case for a mission where a single mother ship centralizes information from several smaller spacecraft. The proposed architecture allows for both scenarios to be simulated, and it is up to the user to configure different dynamics classes if a heterogeneous constellation is required.

Critical subsystems are implemented within the dynamics class. This includes the power system, which contains solar panels for charging energy and battery storage. Components that require energy, such as attitude control devices, transmitters, or cameras, are also properly integrated into the power system to account for energy consumption and generation. The attitude control system is also implemented in the dynamics class, and it includes reaction wheels, control moment gyroscopes, and thrusters. It should be noted that only the dynamics of these attitude control devices and the corresponding effect on the spacecraft are simulated within this class. All control law algorithms are part of the flight software class. Finally, the modules related to the instrument system, which includes cameras, transmitters, and data buffers, are also included in the dynamics class.

3.4. Flight software class

The FSW class contains the logic to go into the spacecraft's onboard computer. While the environment and dynamics classes simulate physical phenomena, the flight software class includes the modules that make decisions based on the spacecraft's position, velocity, attitude, etc. It contains the instructions and logic to make the spacecraft meet its mission objectives.

The attitude modes are set within the FSW class, which dictates where the spacecraft should point. These may include pointing the solar panels at the Sun for battery charging (Sun pointing), pointing the antenna at the Earth for downlinking data (nadir pointing), or pointing a sensor at a target on the planet's surface for imaging (target pointing). This class also contains the logic for the attitude control system. Given a reference attitude and attitude rate, a required torque is computed and mapped onto the attitude control devices (reaction wheels, control moment gyroscopes, thrusters), which drives the spacecraft's attitude to the reference attitude.

Beyond attitude control, relative orbit control is also implemented, calculating the necessary burns to enable specific formation flying maneuvers. The attitude and relative orbit control laws are derived in Schaub and Junkins (2018).

4. Messaging system

Due to its modular nature, a messaging system is necessary for this architecture. Its purpose is to transfer information from one module to the other so that all modules have the most up-to-date data at run time. For the simulation to work correctly, the messaging system must be fast while still retaining accuracy in the information it delivers between modules. Another important aspect is its user-friendliness: the more intuitive the system, the quicker the user can connect modules without making mistakes.

Basilisk's messaging system uses messages, which, at their core, consist of C/C++ structures. This architecture would not have been possible without substantial modifications to the messaging system in the release of Basilisk 2.0. The new messaging system is explained in Carnahan et al. (2020).

The prior version 1 Basilisk messaging system relied on a message pool. All messages were stored in a container and were available to all modules. A particular module would grab the correct message by searching the container for the message by its name. The name was auto-generated, which meant that the message connections were implicit: the user did not have to set a name for every message, and the modules were developed in such a way that they would search for the message with an expected predefined name. For example, the attitude control device module would expect an input message with the same name as the output message of the attitude control law module. A diagram showing the structure of the old messaging is shown in Fig. 2. The advantages of the old messaging system included the speed of the connections, simplicity, and readability of message identifiers to users, and implicit message connections, where the user did not need to worry about connecting the correct messages between modules (Carnahan et al., 2020). Another feature was the ability to recreate the module connections from the single messag-

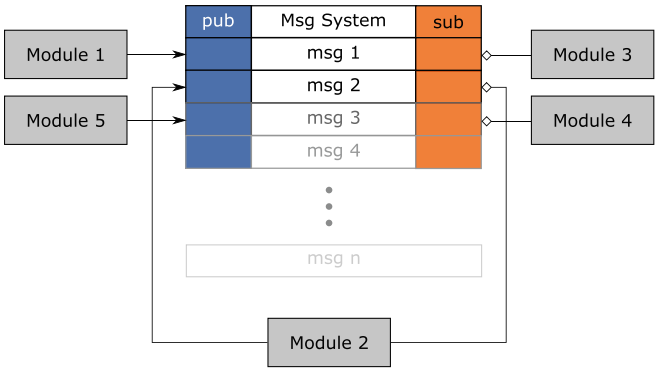

Fig. 2. Old messaging system diagram. Modules read and write messages in a universal message pool, which is categorized by message name. Carnahan et al. (2020).

ing pool, which allowed the user to reconstruct the simulation structure from the message graph.

However, the system had fundamental challenges, particularly evident in multi-spacecraft simulations. First, because multiple instances of the same module were created, the user had to manually change the name of each affected message for the connections to be set appropriately. Take a simulation of 3 spacecraft, each with an attitude feedback controller and a set of three reaction wheels. The user would have to manually assign a name for each output message of the three attitude feedback controller modules so that the required torque is passed onto the correct set of reaction wheels. It is easy to imagine how cumbersome this would become to simulate tens of satellites. Moreover, typos in message names made modules unable to access the proper data, requiring the user to go through a complicated troubleshooting process. The system also had no way to verify that the appropriate message type was being connected, leading to the simulation running with incorrect information if it was not correctly configured. The false configuration issues become stronger as the number of satellites increases. Therefore, an overhaul of this system was implemented.

Instead of a message storage container, the new system uses message classes. Messages are now explicitly connected between modules by the user, which tackles most of the drawbacks of the old system. There is no need to name the messages anymore, which takes care of the old system's naming problem. Moreover, connecting messages between modules allows for strong type checking, as the message identifier is now a class instance instead of a string. When the simulation is initialized, each module verifies that the input messages correspond to the expected type; if not, an error flag is thrown, and the user can quickly troubleshoot the problem.

A diagram for the new messaging system structure is shown in Fig. 3. Here, modules 1 and 2 each have an output message, which connects to two input messages to module 3. Module 3 has two output messages, which are both connected to module 4. Finally, module 5 also takes

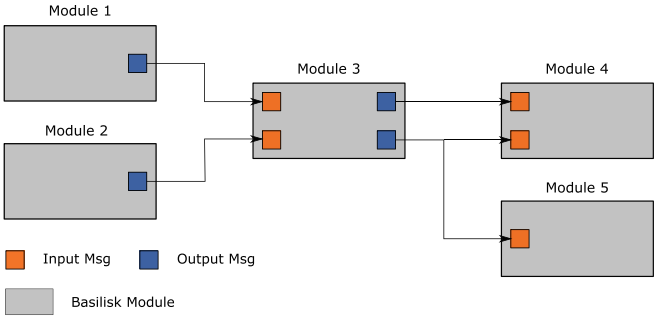

Fig. 3. New messaging system diagram. Messages are subscribed to a specific module in a peer-to-peer system.

in the second output message of module 3 as an input message. To set up all module connections, each input message needs to be subscribed to the incoming output message of the other module. A code snippet is provided as an example in Listing 5. Further information can be found in https://hanspeterschaub.info/basilisk/Learn/bskPrinciples/bskPrinciples-3.html.

Most importantly, the new explicit connections allow for much easier expandability for this architecture. There is no need to name messages for instances of the same module anymore. Moreover, it is possible to automatically connect messages between modules in a loop for every spacecraft instance, which makes creating simulations of hundreds of satellites as easy as simulations of a single one.

An important consideration is that while the same message can be used to input multiple modules, multiple messages cannot be funneled into the same module input. This problem mainly arises when multiple modules output the same message type, which is processed by a single module. For example, it is common to have multiple modules that output reference attitude messages but only a single module that processes the information and computes the attitude error. The solution is to create a stand-alone message, connect the multiple output messages to it, and subscribe the input message to the stand-alone message. See more in https://hanspeterschaub.info/basilisk/Learn/bskPrinciples/bskPrinciples-7.html.

5. Process and module design

Knowing the process and module design is essential to understanding the simulation flow. The process architecture for Basilisk is explained in depth in Kenneally et al.

```
1 module3.inputMsg1.subscribeTo(module1.outputMsg)
2 module3.inputMsg2.subscribeTo(module2.outputMsg)
3 module4.inputMsg1.subscribeTo(module3.outputMsg1)
4 module4.inputMsg2.subscribeTo(module3.outputMsg2)
5 module5.inputMsg.subscribeTo(module3.outputMsg2)
```

Listing 5. Message subscriptions.

(2020). A simplified diagram of this process architecture is shown in Fig. 4.

Basilisk processes (called 'processes' moving forward) correspond to the top-level structures in Basilisk. They can contain one or more tasks, which have individual modules. Modules within a task run at the same integration rate. This allows the user to group modules that require similar timestep fidelity with each other. For example, attitude control devices should be updated more frequently due to their dynamics, while orbit propagation usually needs to be updated less regularly to capture the dynamics accurately. To stop running the modules within a task, a task can be disabled. Further information on process and module implementation can be found in https://hanspeterschaub.info/basilisk/Learn/bskPrinciples/bskPrinciples-1.html and https://hanspeterschaub.info/basilisk/Learn/bskPrinciples/bskPrinciples-2b.html.

For this work, each class contains its process. The environment and dynamics processes only contain one task, as all modules can be updated at the same rate and will never be disabled. However, each FSW process has multiple tasks associated with each flight mode. This happens because when a flight mode is active, all others should be disabled. Each process is assigned to a single thread when running the scenario using multithreading. It is impossible to separate and assign different tasks within a process to other threads. Nonetheless, more than one process can run within each available thread.

For the simulation to run as intended, the order of initialization and execution is significant. Wrong initialization of the simulation class instances can lead to modules trying to connect messages to other modules that do not exist, as they have not been created yet. Poor execution order leads to mismatched modules and potentially outdated information, impacting the guidance and control algorithms.

For this architecture, the initialization and execution orders are identical. This is because the flow of information dictates how the modules are created and how they are updated at each time step. The modules that do not depend on others to run are created/updated first, and the ones with the most dependencies are last in line.

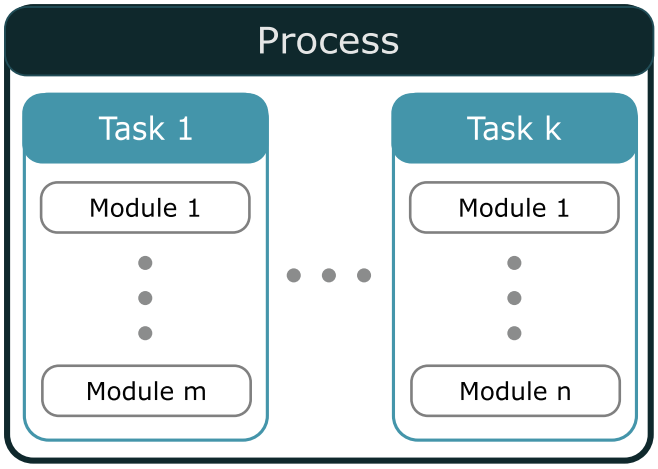

Fig. 4. Basilisk process architecture. Kenneally et al. (2020).

Following this hierarchy, the environment class is the first to be initialized and updated. It contains modules that compute the position and velocity of the gravitational bodies, the density of the atmosphere, or even the position of ground stations. All these modules do not depend on the spacecraft and, therefore, have no external dependencies. The gravitational bodies' information is drawn from a catalog, the atmosphere's density follows a statistical model, and the position of ground locations can be calculated from initial conditions and the planet's rotation.

The dynamics class is updated next. Some modules are self-contained and do not need information from the environment class. For example, the attitude is propagated using the spacecraft's previous attitude, its inertia, and the dynamics from attitude control devices such as reaction wheels. However, the environment modules do influence some of the dynamics modules. Orbit propagation is done in the dynamics class, and it depends on where the gravitational bodies are located and their properties. Therefore, this class directly depends on the environment class modules.

The final class to be updated is the Flight Software class. FSW contains the modules most dependent on other classes: ground locations for tracking and spacecraft attitude for the control law are examples of this. However, the FSW class can have some effect on dynamics modules. For example, given the current and reference attitudes, one FSW module uses a control law to request a desired torque. This torque is mapped onto the attitude control devices, which impacts their dynamics. However, these devices are simulated within the dynamics class. This can create problems, as the dynamics have already been updated once the FSW modules are run. Ultimately, this is not a problem because the information passed from FSW to the dynamics class only needs to affect the simulation at the next time step, when the Environment-Dynamics-FSW loop is rerun.

With multiple spacecraft, this execution order becomes even more critical. All dynamics class instantiations are initialized and run before any FSW class instances are updated. This ensures all FSW class instances have the most up-to-date information about the spacecraft's properties. While most FSW modules concern their spacecraft, there are times when information regarding other spacecraft is used. Suppose a formation of satellites is set to do science on Earth. Depending on the current scientific objective, the satellite formation might take different shapes, such as a string of pearls or a double echelon. Assuming that there is a chief satellite, the other spacecraft must receive information about its position and velocity to maintain or change from one formation shape to the other. Therefore, it is critical that the FSW class instantiations of each deputy satellite have updated information regarding the dynamics of the chief and potentially others in the constellation. Unfortunately, while this scenario can be simu-

lated with the current implementation, it is incompatible with the current multithreading implementation.

6. Numerical simulations

Some qualitative and quantitative examples of the software package in practice are shown in this section. The intention is to illustrate how the topics explained in previous chapters can be used in simulations with any number of satellites. In particular, for this paper, the ease of use of this architecture is demonstrated through an example scenario where the environment and the number of spacecraft are changed by modifying only one or two lines of code. The goal is to show how, by generalizing the architecture, it is possible to readily change the number of spacecraft or the simulation environment without significantly changing the code. The example scenario consists of three satellites orbiting Earth. Then, the environment is changed to Mercury, showcasing the plug-and-play attributes of the multi-satellite framework. Then, the environment is changed back to Earth, while the number of satellites increases to twenty. All these changes are done through minimal code change, which underlines the benefits of the proposed architecture.

Many examples that use the proposed architecture can be found online. The *Basic Orbit Scenario* (https://hanspeterschaub.info/basilisk/examples/MultiSatBskSim/scenariosMultiSat/scenario_BasicOrbitMultiSat.html) creates a constellation of satellites around Earth or Mercury. The *Attitude Guidance Scenario* (https://hanspeterschaub.info/basilisk/examples/MultiSatBskSim/scenariosMultiSat/scenario_AttGuidMultiSat.html) expands on the previous scenario and adds attitude control devices and their corresponding algorithms. Finally, the *Station Keeping Scenario* (https://hanspeterschaub.info/basilisk/examples/MultiSatBskSim/scenariosMultiSat/scenario_StationKeepingMultiSat.html) also adds algorithms that can change the orbit of each spacecraft through DV-burn scheduling based on orbital element errors. The underlying files can be found in the `MultiSatBskSim` directory (https://hanspeterschaub.info/basilisk/examples/MultiSatBskSim/index.html).

Moreover, a qualitative analysis of the multithreading performance is also shown to underline the importance of parallelization in a simulation with multiple satellites. The objective is to convey that parallelizing Basilisk's processes into different threads decreases the cost of simulating additional satellites, particularly for more satellites. An example of a multithreading implementation can be found in *Multithreaded Basic Orbit Scenario* (https://hanspeterschaub.info/basilisk/examples/MultiSatBskSim/scenariosMultiSat/scenario_BasicOrbitMultiSat_MT.html).

6.1. Example scenario

As an illustrative example of the proposed architecture's capabilities, a three-spacecraft simulation around low-Earth orbit is created in Basilisk. Since the focus is on the simulation setup, no flight software modules are included. Each spacecraft is treated as a singular rigid body in orbit. The inertial orbits of all spacecraft are shown in Fig. 5.

Here, an environment class with Earth as the main gravity body is added to the simulation. All spacecraft are homogeneous, which means they have the same dynamics and FSW classes. The code for the class setup is shown in Listing 6, where the number of spacecraft is set to `numberSpacecraft` = 3.

The main gravity body is now changed to Mercury. Assuming an environment class with Mercury as the main body exists and is appropriately set up, changing the environment requires little effort. The setup in the scenario script is almost identical to the one with the Earth environment, as shown in Listing 7. The inertial orbits for this scenario are shown in Fig. 6. The scale of this plot is purposely the same as the one Fig. 5 to show the different sizes of Mercury and the satellite's orbits. This happens because the spacecraft's initial conditions are set through orbital elements, with the major axis being proportional to the main body's equatorial radius. This allows the user to freely change the gravity body without worrying about the orbits intersecting the planet, keeping them in the low orbit regime (see third line of Listing 8).

For the final example, the environment class is set back to Earth as the main body. The number of spacecraft is increased to twenty, which is done through setting `numberSpacecraft` = 20. With no other change, a twenty-satellite simulation is created, together with dynamics and FSW modules for each spacecraft instance. This includes an attitude control system, power system, etc. The orbits for this scenario are shown in Figs. 7. Since

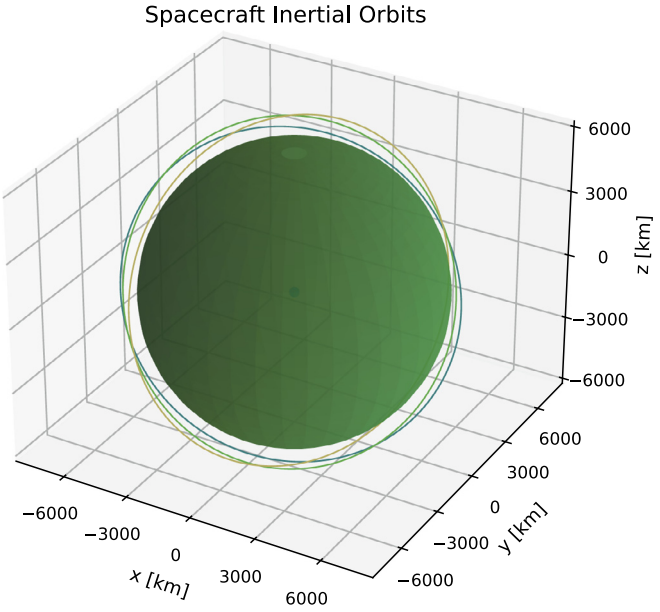

Fig. 5. Orbits for a three-spacecraft simulation around Earth.

```
self.set_EnvModel(BSK_EnvironmentEarth)
self.set_DynModel([BSK_MultiSatDynamics]*numberSpacecra
self.set_FswModel([BSK_MultiSatFsw]*numberSpacecraft)
```

Listing 6. Setup for a simulation around Earth.

```
self.set_EnvModel(BSK_EnvironmentMercury)
self.set_DynModel([BSK_MultiSatDynamics]*numberSpacecra
self.set_FswModel([BSK_MultiSatFsw]*numberSpacecraft)
```

Listing 7. Setup for a simulation around Mercury.

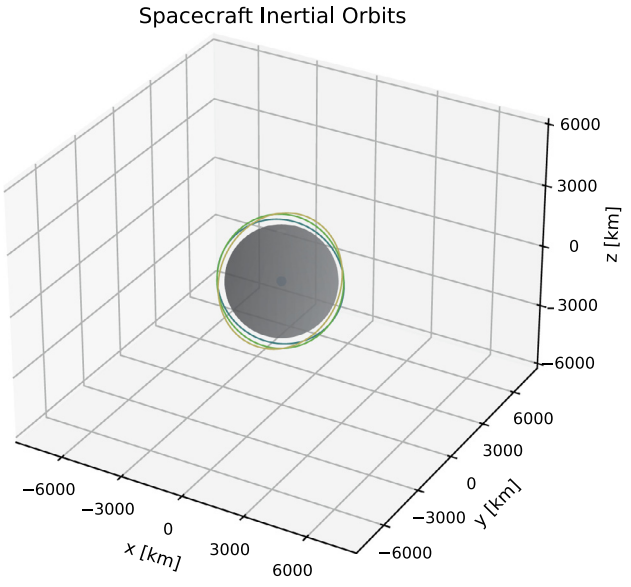

Fig. 6. Orbits for a three-spacecraft simulation around Mercury.

```
for i in range(self.numberSpacecraft):
    self.oe.append(orbitalMotion.ClassicElements())
    self.oe[i].a = 1.1 * EnvModel.planetRadius + 1E5*(i
    self.oe[i].e = 0.01 + 0.001*i
    self.oe[i].i = (45.0 + 15.0 * i) * macros.D2R
    self.oe[i].Omega = (48.2 + 5.0*i) * macros.D2R
    self.oe[i].omega = 347.8 * macros.D2R
    self.oe[i].f = 85.3 * macros.D2R
```

Listing 8. Initial conditions loop.

the initial conditions are set in a loop for all spacecraft, the software can adapt to any number of spacecraft that the user intends to simulate. This loop is shown in Listing 8.

6.2. Multithreading performance

To show how parallelization impacts performance, simulations using single and multiple threads are run with an

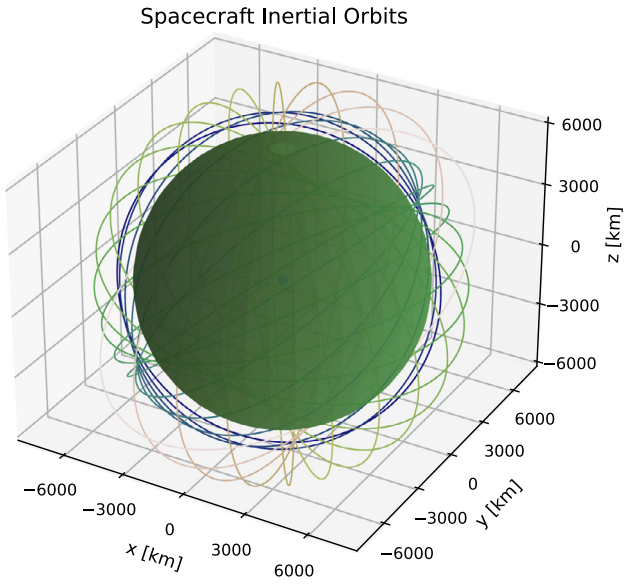

Fig. 7. Orbits for a twenty-spacecraft simulation around Earth.

increasing number of spacecraft. Performance is measured through simulation time, including all initialization routines and the simulation itself. A decrease in simulation time implies better performance. The results are shown in Fig. 8.

As expected, the multithreading application consistently delivers faster simulation times when compared to the single-threaded scenario. More importantly, the simulation time slope with the number of satellites for the multithreading curve is lower than for the single thread. This means the additional simulation time incurred from adding more satellites is lower, which is critical for running simulations of tens or hundreds of satellites in a reasonable time frame.

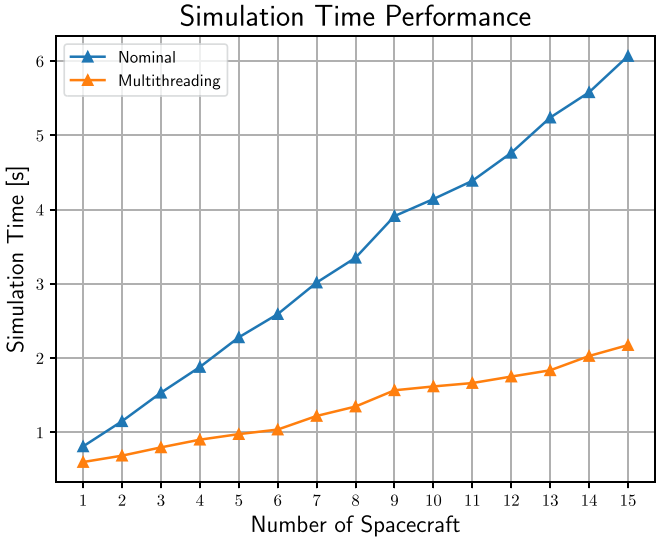

Fig. 8. Multithreading performance in terms of simulation time. $N+1$ threads are used for each simulation, where N is the total number of spacecraft. The time is averaged over ten runs on a CPU with 16 threads.

It should be noted that for this scenario, a constellation of satellites is used instead of a formation of satellites. The difference between these two options is that there is no communication between spacecraft and no active relative formation control in a constellation. This means that all spacecraft dynamics are independent from each other. There is one environment class shared between all spacecraft and one dynamics class instance for each satellite, with no FSW modules present. Therefore, a total of $N + 1$ processes are running concurrently: 1 for each N spacecraft and the environment class shared with all satellites.

While data-sharing is possible using multithreading, enabling a formation of spacecraft, it is harder to implement because all processes must be up to date across all threads before updating the modules that need that information. Failure to do so means that modules (mainly within the respective FSW classes) might run with incorrect or out-of-date information. Therefore, the current implementation is not multithread safe, although there are plans to implement that feature.

7. Conclusion

In this work, the underlying principles of the architecture design (modularity, scalability, parallelization, and scriptability) are presented to justify the design choices made. The architecture framework is presented, aiming to solve the drawbacks of a multi-satellite simulation. While the architecture is implemented using the Basilisk software tool, it is framed in a general enough way to be applied to any other software application. The challenge of sharing information between different modules is tackled using the new messaging system, which is based on peer-to-peer message connections. Combined with the proposed process and module design, this system guarantees that each module has the most up-to-date data at each iteration. The example scenarios show the ease of changing the simulation parameters after the architecture is implemented. This ease of scriptability is important when different simulation parameters are to be evaluated, as it speeds up the process of changing the environment or the spacecraft itself. Although still in development, the multithreading capabilities show promising speed increases, with the most notable changes when the number of simulated satellites is greatest.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors thank Scott Piggott for the multithreading implementation in Basilisk. Part of this research was supported under the NASA STTR Phase 1 Grant No. 80NSSC21C0117.

References

AGI, 2022. Systems Tool Kit, Software Package Version 12.4. http://www.agi.com/products/stk.

Alcorn, J., Allard, C., Schaub, H., 2018. Fully coupled reaction wheel static and dynamic imbalance for spacecraft jitter modeling. AIAA J. Guidance, Control, Dynam. 41 (6), 1380–1388. https://doi.org/10.2514/1.G003277.

Allard, C., Diaz-Ramos, M., Kenneally, P.W., et al., 2018a. Modular software architecture for fully-coupled spacecraft simulations. J. Aerospace Inform. Syst. 15 (12), 670–683. https://doi.org/10.2514/1.I010653.

Allard, C., Schaub, H., Piggott, S., 2018b. General hinged solar panel dynamics approximating first-order spacecraft flexing. AIAA J. Spacecraft Rock. 55 (5), 1290–1298. https://doi.org/10.2514/1.A34125.

Amazon, 2022. Project Kuiper. https://www.aboutamazon.com/news/tag/project-kuiper.

Carnahan, S., Piggott, S., Schaub, H., 2020. A new messaging system for basilisk. In: AAS Guidance and Control Conference. Breckenridge, CO. AAS 20-134.

Carreno-Luengo, H., Crespo, J.A., Akbar, R., et al., 2021. The CYGNSS mission: On-going science team investigations. Remote Sens. 13 (9), 1814.

Enge, P.K., 1994. The global positioning system: signals, measurements, and performance. Int. J. Wireless Inf. Networks 1 (2), 83–105.

Friis-Christensen, E., Lühr, H., Hulot, G., 2006. Swarm: A constellation to study the Earth's magnetic field. Earth, Planets and Space 58 (4), 351–358.

Friis-Christensen, E., Lühr, H., Knudsen, D., et al., 2008. Swarm–an Earth observation mission investigating geospace. Adv. Space Res. 41 (1), 210–216.

Kenneally, P.W., Piggott, S., Schaub, H., 2020. Basilisk: a flexible, scalable and modular astrodynamics simulation framework. J. Aerospace Inform. Syst. 17 (9), 496–507.

Macmillan, S., Olsen, N., 2013. Observatory data and the Swarm mission. Earth, Planets and Space 65 (11), 1355–1362.

MathWorks, 2022. MATLAB/Simulink, Software Package Version R2022b. https://www.mathworks.com/products/matlab.html.

NASA, 2022a. 42: A Comprehensive General-Purpose Simulation of Attitude and Trajectory Dynamics and Control of Multiple Spacecraft Composed of Multiple Rigid or Flexible Bodies. https://software.nasa.gov/software/GSC-16720-1.

NASA, 2022b. General Mission Analysis Tool, Software Package Version R2020a.

OneWeb (2022). OneWeb. https://oneweb.net.

Panicucci, P., Allard, C., Schaub, H., 2018. Volume multi-sphere-model development using electric field matching. J. Astronaut. Sci. 65 (4), 423–447. https://doi.org/10.1007/s40295-018-0133-0.

Ruf, C., Gleason, S., Ridley, A., et al., 2017. The nasa cygnss mission: Overview and status update. In: 2017 IEEE International Geoscience and Remote Sensing Symposium (IGARSS). IEEE, pp. 2641–2643.

Sasaki, T., Schaub, H., Shimomura, T., 2018. Convex optimization for power tracking of double-gimbal variable-speed control moment gyroscopes. J. Spacecraft Rock. 55 (3), 541–551. https://doi.org/10.2514/1.A33944.

Schaub, H., Junkins, J.L., 2018. Analytical Mechanics of Space Systems, 4th ed. AIAA Education Series, Reston, VA. https://doi.org/10.2514/4.105210.

SpaceX, 2022. Starlink. https://www.starlink.com.

Valinia, A., Burt, J., Pham, T. et al., 2019. The role of smallsats in scientific exploration and commercialization of space. In: Micro-and Nanotechnology Sensors, Systems, and Applications XI. SPIE volume 10982, pp. 278–283.