Simulation of Heterogeneous Spacecraft and Mission Components through the Black Lion Framework

Mar Cols Margenet[1], Patrick Kenneally[2], Hanspeter Schaub[3] and Scott Piggott[4]

Paper No. #7

AAS John L. Junkins Dynamical Systems Symposium
College Station, TX
May 20–21, 2018

[1] Graduate Student, Aerospace Engineering Sciences, University of Colorado Boulder

[2] Graduate Student, Aerospace Engineering Sciences, University of Colorado Boulder

[3] Professor, Glenn L. Murphy Chair, Department of Aerospace Engineering Sciences, University of Colorado, 431 UCB, Colorado Center for Astrodynamics Research, Boulder, CO 80309-0431. AAS Fellow

[4] ADCS Integrated Simulation Software Lead, Laboratory for Atmospheric and Space Physics, University of Colorado Boulder

AAS No. #7

SIMULATION OF HETEROGENEOUS SPACECRAFT AND MISSION COMPONENTS THROUGH THE BLACK LION FRAMEWORK

Mar Cols Margenet[*], Patrick Kenneally[†], Hanspeter Schaub[‡] and Scott Piggott[§]

This paper describes the design and implementation of Black Lion, a purely software-based modern spacecraft simulation and test environment, with the aim to perform dynamic analysis of mission spaceflight software in a modular distributed simulation framework. The simulation framework is architected to be highly configurable and modular, allowing for heterogenous software components across multiple computing platforms to be integrated into a single simulation. Note that this includes integrating legacy software components that were never designed to be integrated with other software components. This functionality is demonstrated by simulating in software the flat-sat testing of a deep space spacecraft. Here Black Lion provides the capability to start up and run the operational command and telemetry databases, the unmodified flight software executable within a processor emulation application, as well as simulate the physical response using the Basilisk astrodynamics simulation framework. The later provides high-fidelity spacecraft dynamic, sensor, actuator and environment models that are integrated in order to test the flight software system in realistic closed-loop simulations. Simulation results demonstrate how a flat-sat scenario is simulated with software-only integration of stand-alone ground software and the flight computers on virtual machines.

INTRODUCTION

Black Lion (BL) is a communication architecture which enables a distributed software-simulation (SW-sim) of a spacecraft system. Spacecraft software undergoes rigorous levels of integration and validation testing of the complex mission behaviors. Besides testing the nominal functionality and expected spacecraft behaviors, simulations are critical to test off-nominal behaviors where components fail, sensor signals are corrupted and how ground contact communication periods are used to analyze and address mission issues. While the BL SW-sim is being developed in order to support flat-sat testing for an ongoing interplanetary mission in which the Laboratory for Atmospheric and Space Physics (LASP) and the Autonomous Vehicle Systems (AVS) laboratory at the University of Colorado Boulder are collaborating, the BL architecture is general enough to create a range of distributed spacecraft simulations. Multiple software processes across heterogenous computing platforms can be integrated into a single BL simulation. This allows, for example, having a

[*]Graduate Student, Aerospace Engineering Sciences, University of Colorado Boulder.

[†]Graduate Student, Aerospace Engineering Sciences, University of Colorado Boulder.

[‡]Professor, Glenn L. Murphy Chair, Department of Aerospace Engineering Sciences, University of Colorado, 431 UCB, Colorado Center for Astrodynamics Research, Boulder, CO 80309-0431. AAS Fellow.

[§]ADCS Integrated Simulation Software Lead, Laboratory for Atmospheric and Space Physics, University of Colorado Boulder.

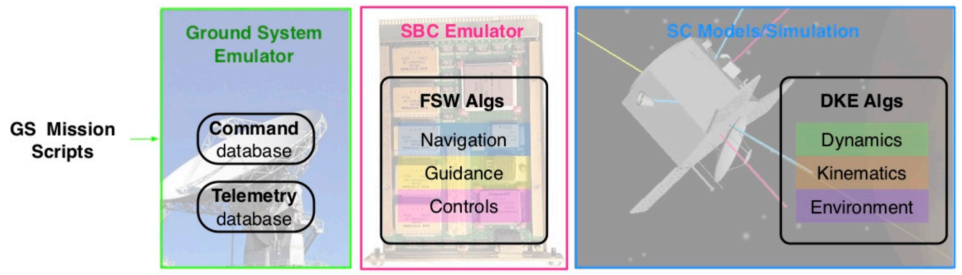

Figure 1. Virtualization of Spaceflight Components.

dedicated computer running a complex space environment model and another computer integrating spacecraft dynamics, both of them exchanging data dynamically through BL. Further, many mission operations rely on software legacy components that were never design to integrated into a larger simulation such as the ground communication system software, or the spacecraft flight software itself.

Considering the flat-sat testing scenario, there are multiple mission components interacting with each other: the ground system (GS), the single board computer (SBC) and its flight software (FSW) algorithms, as well as spacecraft (SC) models to simulate the dynamics, kinematics and environment (DKE). In contrast, SW-sim testing uses virtual models or emulators in place of hardware components such as the GS or SBC. The *raison d' être* of a SW-sim is to provide a comprehensive simulation testbed that is purely software-based. Figure 1 depicts the idea behind the virtualization of the GS, the SBC and the spacecraft's DKE. As illustrated, the GS emulator ingests the same mission scripts as the real ground system and contains also the same command/telemetry databases, while the SBC emulator runs the actual mission FSW.

The virtual models used in a SW-sim are usually pre-existent resources from the particular mission that the SW-sim effort is supporting. These virtual components are stand-alone applications that are generally heterogeneous (written in different programming languages, nominally running at different speeds, etc.). Therefore, a common communication architecture is needed to synchronize exchange of data between the multiple nodes (virtual components).

The Black Lion communication architecture is designed to connect all the nodes of a distributed SW-sim while being as transparent as possible to the internals of these nodes, such that different mission users can plug-and-play virtual models. While the Black Lion effort is currently motivated for a specific spacecraft interplanetary mission, the system is being built under the principles of reusability and scalability, and the BL applications extend beyond the flat-sat example discussed here. Examples of further BL applications include the integration of large clusters of spacecraft, complex simulation components running on super-computers or cloud servers, as well as distributed simulation of both spacecraft sensor and actuation systems.

The main purpose of a SW-sim is to test the onboard FSW executable, a complete binary image that includes the FSW algorithms/application, the real-time operating system (RTOS), the Boot software and the support package for the single board computer as illustrated in Fig. 2. Testing the onboard FSW in response to dynamic stimuli in a spacecraft operational environment is also one of the driving goals for flat-sat testing. While SW-sim testing does not replace flat-sat tests, it can reduce bottlenecks by providing pure software substitutions for hardware components of limited quantity that might be needed simultaneously for testing by different mission groups. This alleviates schedule constraints by using software models only, providing a flexible and cost-effective means of performing mission system testing early on in a mission program's schedule.

Software-sim environments have long been used in the aerospace industry. Notable examples of aerospace missions using a software-only test based approach are James Webb Space Telescope, Space Launch System, Juno and OSIRIS-Rex. Different flavors exist of SW-sim architectures where some groups are developing their own in-house solutions like NASA's Operational Simulator engine[*] while other groups may have acquired and maintain versions of Lockheed Martin's "SoftSim" to support verification and validation (V&V) activities. In general, simulators tend to be sophisticated software products that are developed in parallel with the systems they are intended to test. However, all SW-sims are meant to provide a common functionality: the ability to run the unmodified FSW executable in a software-only simulation environment.

Simulation software architectures take many forms and the characteristics of the architecture strongly influence the functionality and therefore applications of the simulation tool. With some generalization, simulation architectures can being characterized by the degree to which system components are coupled. The coupling between simulation components is manifested by the simulation structure, where the system may be integrated as a single system of required components, integrated as a modular system with optional components or developed as a group of cooperative, yet stand alone components.

An example of a system which has increased coupling between components is Advanced Solutions Inc's (ASI) Spacecraft Object Library in STK (SOLIS). SOLIS is a commercial plug-in to the Analytical Graphics, Inc (AGI) Systems ToolKit (STK™) mission analysis software. The plugin extends STK's orbit and space environment dynamics with the On-board Dynamic Simulation System (ODySSy™). ODySSy is an on-board spacecraft simulator providing additional models for rotational dynamics, sensors, actuators, power and thermal dynamics, and basic spacecraft control and guidance algorithms.[1] Using both SOLIS and ODySSy from ASI provides end-to-end spacecraft simulation functionality. However, it does so by requiring those tools specifically. There are minimal options to substitute one component with another which was not intended to operate with the SOLIS system.

A software suite which demonstrates increasing modularity in its architecture is the NASA Jet

[*]`https://www.nasa.gov/centers/ivv/jstar/jstar_simulation.html`

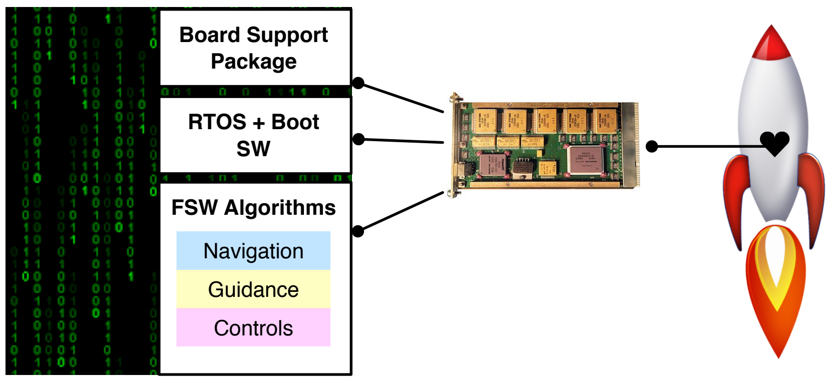

Figure 2. Binary Image Loaded to the Spacecraft (Onboard Flight Software).

Propulsion Laboratory's Dshell system.[2] The Dshell system avoids tightly coupled components by establishing interconnections and communicating data between components via "connector signals". Connector signals allow each component to provide data to other components without requiring knowledge of the other components internals or availability.[3] The Dshell suite of components has grown since it's initial development. Now, using the wide range of components developed for the Dshell system, enginners are able to support a wide variety of simulation configurations including both robotic and spacecraft simulation, software and hardware-in-the-loop testing and mission telemetry visualization.[4]

The NASA Operational Simulator (NOS)* is a simulation system which exemplifies the characteristics of a loosely coupled system architecture. The NOS system is a generic software-only simulation architecture and was developed by NASA's Independent Verification and Validation (IV&V) Independent Test Capability (ITC). The NOS system achieves its flexible architecture by employing a message passing middleware application to connect various simulation components by a virtualized MIL-STD-1553 or SpaceWire messaging bus.[5] This middleware approach allows users to add or remove heterogeneous simulation components, unique to a particular spacecraft mission, without needing to rewrite or recompile model or application code.[6]

The BL SW-sim in particular provides the capability to start up and run the operational command and telemetry databases, as well as the unmodified flight software executable. Virtual models are used for the ground system, the single board computer and for other required hardware components like sensors, actuators and avionics. High-fidelity dynamic, kinematic and environment models are integrated in order to test the flight software system in realistic closed-loop simulations. The scope and coverage of the BL simulator involves assuring that the source code components can reliably perform required capabilities under both nominal and off-nominal (i.e. faulted) conditions and that the system's responses are deterministic.

Whereas the functionality and coverage of the BL SW-sim overlap in parts with those pursued by other groups, the novelty of the BL design is the system architecture. From the outset the architecture was conceived as a distributed system. This distributed nature allows the BL system to incorporate any node on any machine provided that a BL interface is written for the particular node. The distributed architecture allows one to employ BL across a wide spectrum of simulation configurations. The spectrum's extents can be defined as software only simulations to mixed software and hardware node simulation configurations. Black Lion's flexible architecture allows the system to operate beyond the specific focus of a purely software flat-sat replacement and perform simulations of various phases of a spacecraft's mission. Two cursory examples of such novel simulation configurations are a configuration which utilizes models executing on a high performance computing facility and a configuration which incorporates hardware such as a LIDAR module being artificially stimulated in a laboratory.

A guiding principle of BL development is to move away from highly expensive, specialized products and focus on delivering scalable functionality and increased operational efficiency at a faster pace and lower cost. In order to achieve this modern shift, the following directives are driving the system design and implementation: 1) Use of open-source, cross-platform products, 2) Modular architecture by initial design and 3) Agile software development.

The paper is outlined as follows: first, there is a preamble introducing the FSW application that is the central target of the BL SW-sim testing. Next, the different components to be included in the SW-sim are defined. The following section addresses communication aspects between the different

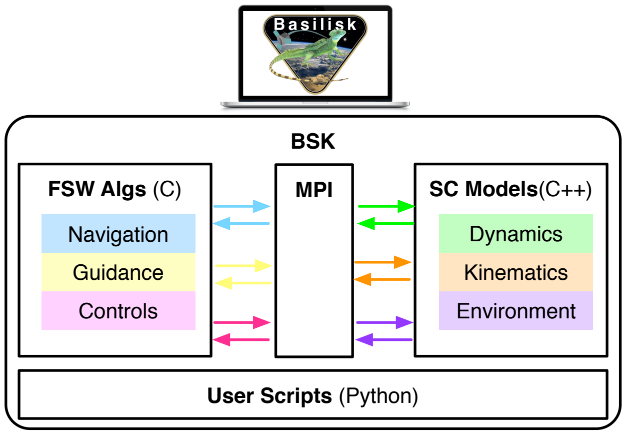

Figure 3. Spacecraft Models and FSW Application Developed within Basilisk (BSK).

nodes, covering: node heterogeneity, communication goals and introduction of the BL architecture. Then, there is a special section focusing on the adoption of modern software technology and its role in the BL development effort. The next section explains the strategies for both the data transport (socket patterns, connections, etc.) and node synchronization. A brief section then highlights the capability of BL to run as a distributed machine system. Finally, some concluding remarks are provided and a road-map for future work is included.

PREAMBLE: THE FSW ALGORITHMS AND APPLICATION DEVELOPMENT

The collaboration between LASP and the AVS laboratory encompasses developing GN&C algorithms for different mission profiles. In order to develop these algorithms a parallel development effort has produced a general software astrodynamics framework aimed at serving as a testbed for prototyping and testing. This astrodynamics framework is named Basilisk (BSK)[*] and is currently available as an open-source product.[7] BSK leverages Python's ease-of-use as a testbed for FSW development provided that the spacecraft models and the flight algorithm code are written exclusively in C/C++, and then automatically wrapped into Python for simulation setup, analysis, and testing. The architecture of the Basilisk software framework is depicted in Fig. 3. As illustrated, the BSK simulation system is decomposed into two main blocks: a high-fidelity simulation of the physical spacecraft written in C++ ("SC Models" in Fig. 3) and a GN&C algorithms suite written in ANSI-C ("FSW Algs." in Fig. 3). All BSK modules are developed in a modular architecture using C, C++ and Python coding languages that communicate with each other through a custom message passing interface ("MPI" in Fig. 3). In the general case, the FSW algorithms are written exclusively in C as per requirements of spacecraft missions.

The BSK desktop environment is used to construct and test different modes of the FSW application until the required functionality is achieved. At this point, the task, parameter, and state configurations are migrated out of the Python environment and embedded onto an actual flight target. The flight target could be either a specific processor and RTOS or a middleware layer like NASA's core Flight System (cFS). Targeting middleware is advantageous in that ensures portability among different processors and RTOS. Reference 8 discusses the ease of transition from the Basilisk desktop environment into a cFS application, ready to be embedded onto an SBC running a standard RTOS.

[*]`http://hanspeterschaub.info/bskMain.html`

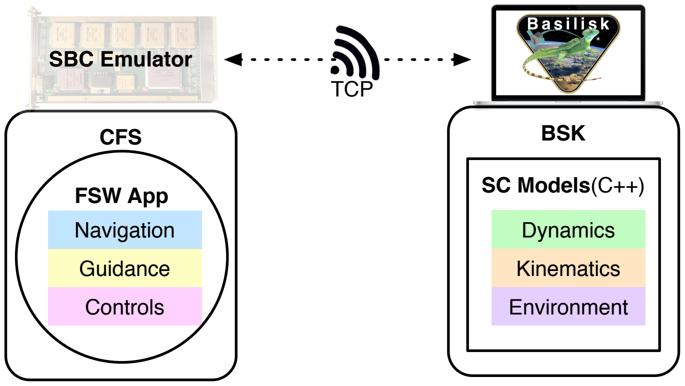

Figure 4. Spacecraft Models in BSK and FSW Application Migrated to a Flight Target.

Figure 4 showcases a system where the FSW algorithms are integrated within cFS, which in turn run on an RTOS inside an SBC emulator. A virtual SBC would be used in a SW-sim, while a physical SBC would be used in a flat-sat system. Figure 4 illustrates that the communication between the two separate systems happens through a TCP connection, allowing the different components to run on different computers. While References 8 and 9 showcase specific strategies to enable peer-to-peer communication between BSK SC models and a given FSW application, the BL SW-sim can now be used to provide greater flexibility by facilitating and routing communication between an indefinite number of peers.

EXPANSION OF SIMULATION MODULES TO MULTIPLE COMPUTER NODES

Once the GN&C flight software application is determined, the next step is to expand the software simulation coverage by simulating multiple spaceflight components in order to allow simultaneous and rapid iterative testing from multiple engineering groups. The right hand side of Fig. 5 illustrates the different heterogeneous legacy and new mission specific components that are to be included in the distributed BL SW-sim. As depicted in Fig. 5, the initial system with only the FSW application and the SC models is expanded to include: a GS virtual model, an SBC virtual model and a visualization tool.

Ground System Emulator: Includes the command and telemetry databases, and runs the same mission scripts/sequences as the physical GS. It sends commands out and receives telemetry back, all in the form of Consultative Committee for Space Data System (CCSDS) packets. An example of GS emulator is the open-source COSMOS software application.[*]

Virtualized Single Board Computer: It contains the unmodified FSW executable for the mission, which runs on a standard Real-Time Operating System or RTOS. The Single Board Computer or SBC emulation includes also Field Programmable Gate Array (FPGA)-like registers. In this case, the actual FPGA registers are emulated/replaced with a memory map for input/output of raw binary data. An example of a processor architecture and system emulator, which is used to emulate the SBC is the open-source QEMU.[†] The BL SW-sim currently makes use of a slightly modified version of QEMU.

[*]`http://cosmos-project.org`
[†]`http://qemu.org`

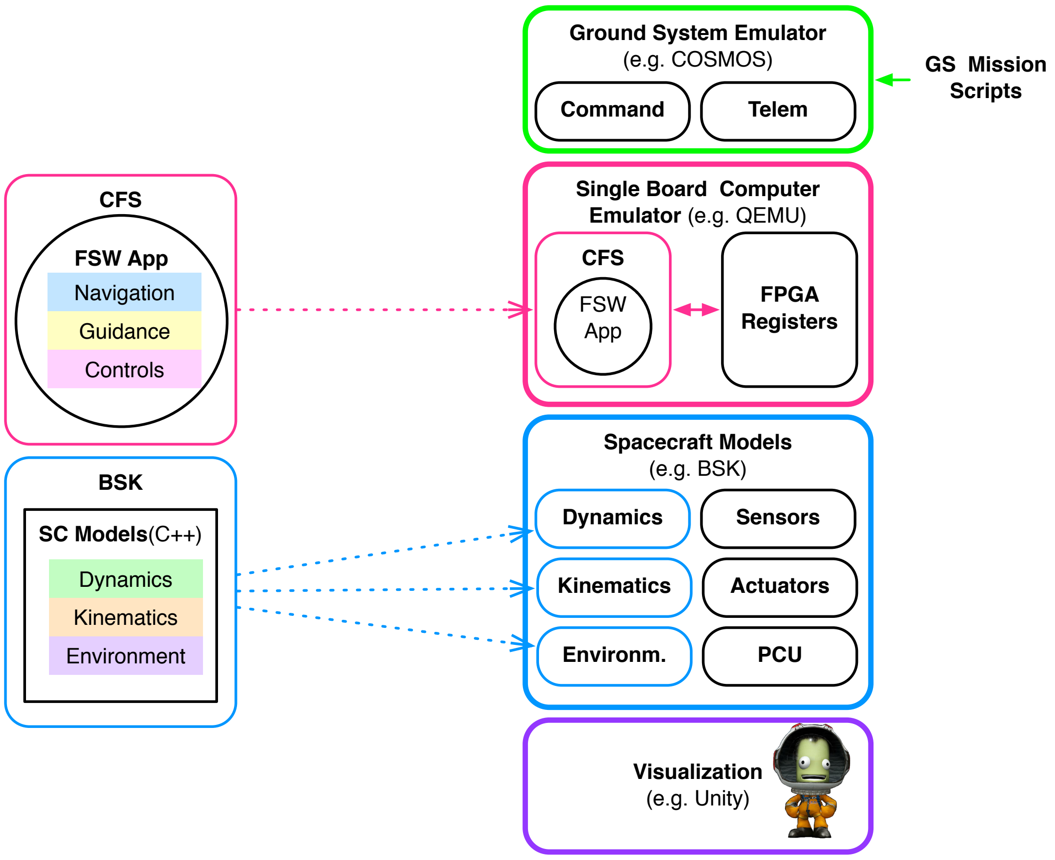

Figure 5. Expansion from GN&C Testing to an Integral SW-Sim Testing Environment.

Spacecraft Models: The BSK simulation framework is used for both high-fidelity DKE models and hardware component models. The hardware component models include sensors (gyro, star tracker, coarse sun sensors, etc.), actuators (reaction wheels, attitude control thrusters, orbit control thrusters) and the power control unit (PCU). Jointly, the DKE and hardware models allow testing of other nodes in closed-loop dynamics simulation.

Visualization: a graphical user interface (GUI) is being developed with the Unity[*] game engine. The GUI shall visually reproduce the spacecraft physical behaviors, which in this case are determined by the BSK simulation. Reference 10 describes the GUI under development that is going to communicate with BSK using the BL message transport protocol.

COMMUNICATION BETWEEN THE COMPONENTS

Recall that the nodes in the SW-sim are stand-alone applications that are initially unaware of any other nodes. They are written in different programming languages, wrap their internal data using different structures or packet types, and run at different speeds.

The differences between the particular nodes currently used in BL are highlighted in Fig. 6. As a quick recapitulation, the GS modeled is written in C++ that uses CCSDS packets with a particular data format. The SBC emulator is based on the open-source product named QEMU. It is written in

[*]`http://unity3d.com`

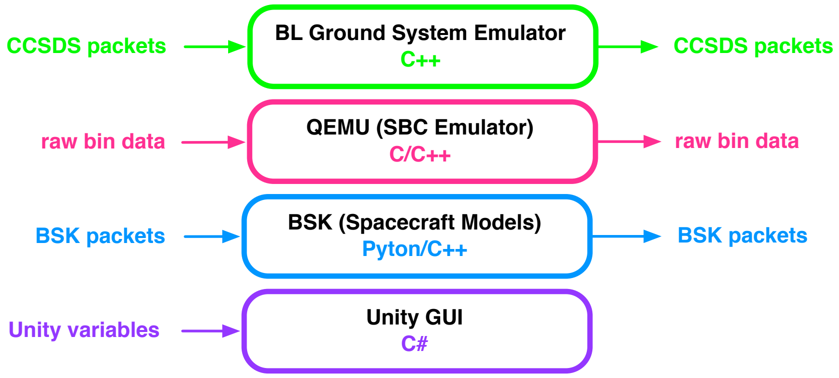

Figure 6. Heterogeneity of Programming Languages and Internal Data Packets in the BL SW-Sim.

a combination of C and C++, and deals directly with raw binary data. The SC models are simulated within BSK. While the BSK source code is written in C++, the application's interface with the external world is Python. The BSK packets are C++ defined structures that come along with a message header. Finally, the Unity-based GUI is written in C#. The challenge for BL is to integrate these heterogenous components while maintaining synchronous operation of the modules.

The heterogeneity between the multiple components drives the need for a dedicated communication architecture. The term communication, as understood here, involves the multiple goals:

1. **Transport** of binary data between nodes
2. **Serialization** of binary data because each node must know how to convert the received bytes into structures that can then manage internally
3. **Synchronization** of nodes to keep all the nodes in lock-step during the simulation run
4. **Dynamicity** in the connections map to allow a more flexible simulation environment that is minimally dependent on static components (the less static, or strictly required, pieces there are in the communication network, the better).

The goal of BL is to achieve the described communication targets while being completely transparent to the internals of each node. This enables such that users to plug and play their models of choice, while having the flexibility to add/remove other nodes because they are not static pieces in the communication map. To this end, the BL architecture depicted in Fig. 7 is comprised of a single central controller and two APIs that are attached to each node.

BL Central Controller: The one and only static piece in the network (i.e. it has a static IP address). The central controller acts as a master in the synchronization of nodes and as a broker in the transactions (exchange of data) between nodes.

Delegate API: This interface component manages sockets and direct connections with the central controller. It is the same script attached to all the nodes. The `Delegate` class is currently implemented for Python nodes and for C++ nodes.

Router API: This is a generic class with node-specific callbacks to create a custom interface to a legacy or newly developed mission simulation component. Its purpose is to route data in and

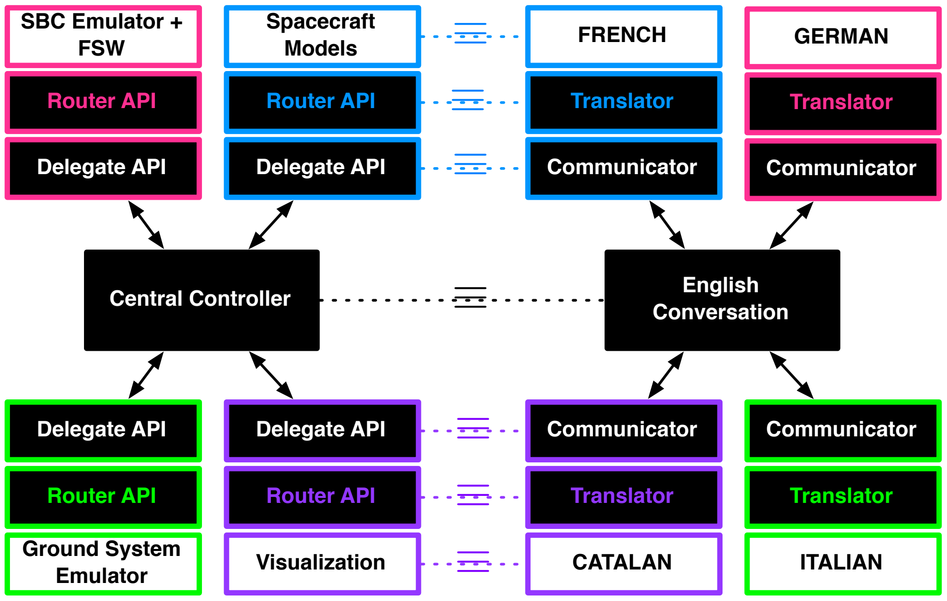

Figure 7. Communication Architecture: Central Controller, Delegate APIs and Router APIs.

out of the internals of the node. For instance, when routing out, the Router API gathers the node internal data, translates the data into a standardized BL system format, and then passes the data to the node's Delegate API, who is ready to ship it across the BL communication network.

In order to explain better the idea behind the given architecture, one can use a human language analogy: each node is an individual that speaks a different language, as illustrated in the right hand side of Fig. 7. The router acts as a translator from the individual's language to a common standardized language, like English in the case of Fig. 7. Once the Router has translated the data, the Delegate communicates over the sockets. The final result is an English conversation in which each individual does not have to learn the particular languages of every other participant in the conversation.

ADOPTION OF MODERN SOFTWARE TECHNOLOGY AND TECHNIQUES

Before moving into the details of the communication hub, it is interesting to step back and emphasize the modern technology and techniques that are critical to the BL effort. BL takes advantage of the following open-source, cutting-edge software technologies:

ZeroMQ Message Library: high-performance asynchronous messaging library, aimed at use in distributed or concurrent applications.[*] It allows the transport of data to be fast, reliable and protocol independent. The ZMQ interfaces are available in a wide range of programming languages, which can perfectly interact with each other.

Google Protobuffers: Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data (like XML, but smaller, faster, and simpler) [†]. The user defines the

[*]http://zeromq.org
[†]http://developers.google.com/protocol-buffers

structure of the data once and then it is possible to use special generated source code to easily write and read the structured data to and from a variety of data streams and using a variety of languages.

In terms of techniques, the BL software is being developed in a **component-based** (modular) approach where nodes are gradually integrated one by one. Further, it is very easy to swap pieces, add new ones, or remove existing ones to perform different kinds of testing. This gradual escalation is granted by having a **dynamic architecture**, where the central controller is the one and only static piece (i.e. server) and the nodes are dynamic clients that can come and go on the fly. Moreover, the software is built through what is known as **agile development**, which implies continuous delivery to mission users and immediate integration of the feedback received - resulting in very fast build, testing and deployment cycles.

DATA TRANSFER AND SYNCHRONIZATION

Socket and Connection Definitions

In order to understand how the communications hub works, it is critical to explain upfront the socket types and connection types used in the system. Two types of ZMQ-socket patterns are used to transport data: the request-reply pattern and the publish-subscribe pattern. The publish-subscribe pattern is applied in two different flavors, as described next:

Request (REQ) - Reply (REP): the central controller has a REQ socket for each node instantiated in the simulation that is used to make requests. In turn, each node has a REP socket that receives and parses the request, performs the commanded task, and replies back indicating accomplishment.

Publish (PUB) - Subscribe (FRONTEND SUB): Every node has a PUB socket to share its own internal data by publication. In turn, the central controller has a frontend with a SUB socket that subscribes to the publications from all nodes.

Publish (BACKEND PUB) - Subscribe (SUB): Additionally the central controller has a SUB-frontend and a PUB-backend. The messages received at the frontend are internally routed to the backend, which then re-publishes the data. In turn, each node has a SUB socket that subscribes to the messages of interest coming from the controller's backend.

The relationship between sockets just described is exemplified in Fig.8. The figure depicts the central controller in the middle and two sample nodes highlighted in magenta and blue. As shown in Fig. 8, the sockets are encapsulated by the Delegate API.

Now that the socket types are defined, the connections of these sockets to a given IP address and port is discussed. All the socket connections in the system fall into either one of these categories: static connection (i.e. **binding** type in ZMQ terms) or dynamic type (i.e. **connecting** type in ZMQ terms). The static connections are all associated to sockets in the central controller, while the dynamic connections are associated to the sockets in each of the nodes' Delegate API.

Central Controller: it is the only static piece in the network thanks to the frontend-backend (broker) approach. The controller acts as a server in the sense that it **binds** to a static IP address. Within the same address, it uses a total of $(2 + N)$ ports, where N is the number of nodes instantiated: One port for the frontend, one port for the backend, and a command port for each of the node-request sockets.

Nodes' Delegate API: through the delegate API attached to each one of the nodes, the nodes become dynamic clients that can come and leave without bringing down the rest of the system. This dynamicity is reflected in the fact that the nodes only **connect** to an address and port, rather than bind.

Through the described strategy, the server is always required and the clients are independent entities that do not intrinsically rely on each other. The use of ZMQ also allows all the connections to be protocol independent (TCP, IPC, etc.). The idea of socket binding (static nature) versus socket connecting (dynamic nature) is illustrated in the topology showcased in Fig. 9. The figure also reflects the fact that there is only one IP address in the entire BL system and within this address multiple ports are used. As before, the figure displays the central controller (server) in the middle and two sample nodes (clients) highlighted in magenta and blue colors.

Request-Reply Communication between the Controller and the Nodes' Delegate

The requests from the Controller to the Delegate on each node are not spacecraft commands, but rather they are communication and synchronization commands exclusive to the SW-sim. In the current BL implementation, the controller can make 5 type of requests, some of which come in the form of multi-part messages. Firstly, there is the **"Initialize" request**, which is a multi-part message containing the "Initialize" signal, the controller's frontend address and port and the controller's backend address and port. The actions taken by the node when this request is parsed are: self initialization, connect its pub-socket to the controller's frontend and connect its sub-socket to the controller's backend. Secondly, there is the **"Provide Desired Message Names" request**, which instructs each node to report all the message names to which the node wishes to subscribe. Thirdly, there is the **"Match Message Names" request**, which is multi-part message with the "Match" signal and a set of all the message names that the other nodes have asked for. The node returns back a reduced list with the message names for which it has found an internal match. Then, there is a **"Tick" request**, which is used at every time-step of the SW-sim run for synchronization purposes and it contains the time duration of the next time-step (i.e. Δt). Once the node has accomplished all the tasks, it sends back a "Tock" reply. Eventually, there is the **"Finish" request**, which is a signal for the node to close the sockets, clean up and shut down.

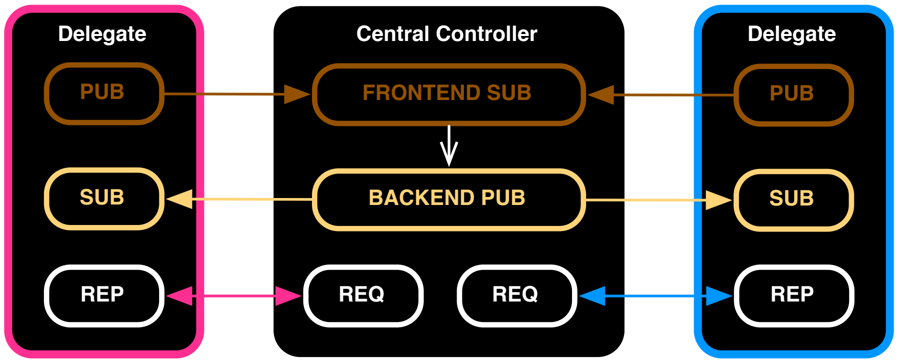

Figure 8. Socket Patterns between the Central Controller and Sample Nodes.

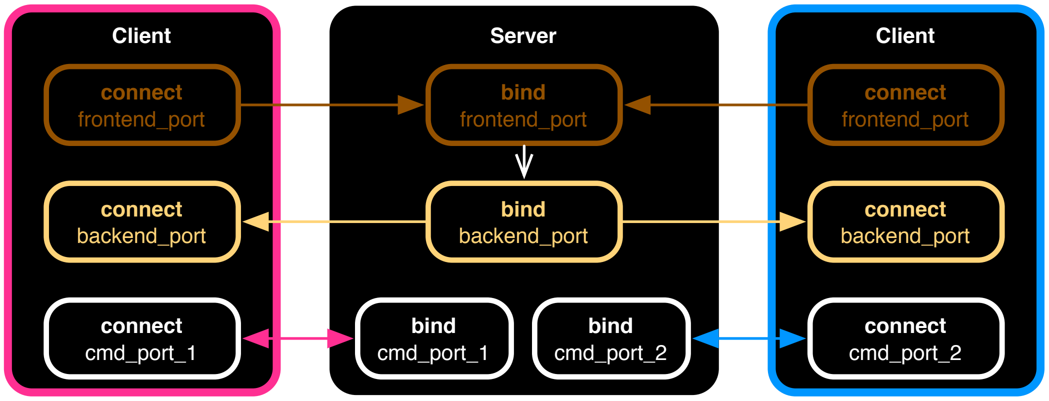

Figure 9. Socket Connections Types (Binding vs. Connecting) and Ports.

Tick-Tock Synchronization

Three actions or tasks happen in sequence inside each node between the parsing of a "Tick"-request and the sending of a "Tock"-reply: **publish**, **subscribe** and **step simulation** forward. Note that these three actions are node internal calls triggered by the "Tick" request. Figure 10 depicts the sequence of interactions and actions happening between the central controller and a sample node highlighted in magenta. In Fig. 10, the words written in white imply interactions between the controller and the node whereas the words written in magenta indicate node internal calls.

Publish: In the publish internal call, the node's Router collects the application internal data and makes it available to the node's Delegate to publish to the controller's frontend.

Subscribe: In the subscribe internal call, the node's Delegate receives external data coming from the controller's backend and hands the data to the node's Router, who is responsible for writing these messages down into the internals of the node application.

Stem Simulation: In general terms, the step simulation internal call implies executing the node's application during Δt in order to generate new data. Recall that Δt is a message part of the "Tick" request sent by the controller. Having said that, there are nuances in the precise meaning of "step simulation" for nodes that are synchronous (i.e. run in cycles, like FSW or the spacecraft simulation) and for nodes that are asynchronous (i.e. are event-based, like the ground system)

Because each node is an independent process that runs at a different speed, the "Tick-Tock" signal ensures that all of them are in lock-step. Let us recall the particular four nodes that are being integrated in the BL system: the SBC emulator, the SC simulation, the GS emulator and the visualization GUI. Figure 11 depicts the synchronous nature of the SBC and SC simulation nodes, the asynchronous nature of the GS emulator, and the listener nature of the visualization node.

Both the SBC node and the SC simulation node are synchronous in nature in the sense that they run in cycles or at predefined rates. Because the FSW executable is running inside the SBC emulator using a RTOS, the speed of the FSW execution is real time. In contrast, the SC simulation runs natively faster than real time. For the synchronous nodes, the "step simulation" call implies running as many cycles as there are within Δt before exchanging data again with the rest of the system. If one node finishes simulating Δt earlier, it sends the "Tock" reply and waits for a new

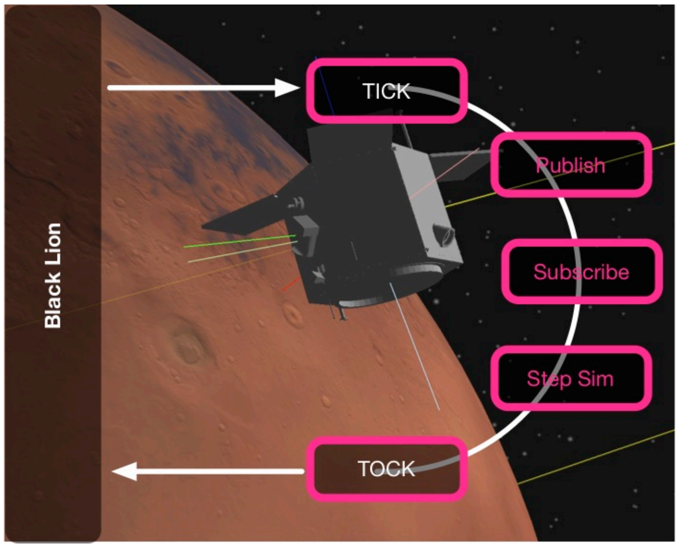

Figure 10. Node Actions between a "Tick-Tock": Publish, Subscribe, Step Simulation.

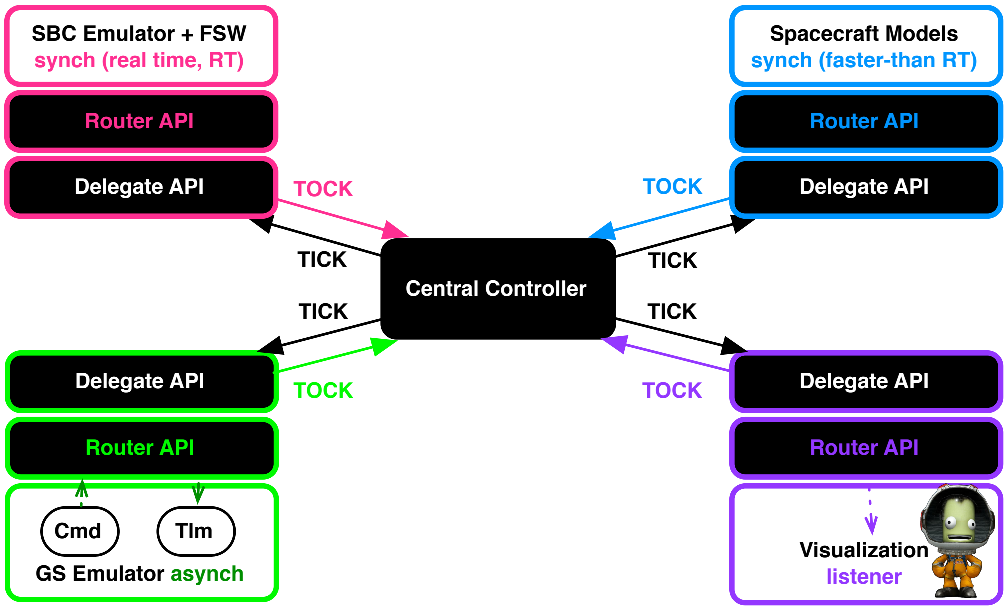

Figure 11. Nodes' Timely Nature: Synchronous, Asynchronous and Listener Behaviors.

request from the controller. Because the controller will not proceed until it has received all the "Tock" signals from all the nodes, the SW-sim speed is naturally driven by the slowest component. In contrast, the GS node is asynchronous: the sending of spacecraft commands and the receiving of telemetry are discrete-time events. The GS also receives a "Tick" command because the exchange of data still happens simultaneously between all the nodes. The asynchronous nature of a node, like the GS, demands a special treatment of the Delegate and Router APIs: the communication through the APIs must happen at a different thread from which the main application is running. The Visualization node is another case on its own: it can be simply regarded as a "listener" governed by the SC simulation. Therefore, it only subscribes to the SC simulation messages and, within the "step simulation" call, its job is to show the spacecraft timely evolution according to the received set of messages.

MULTI-MACHINE FUNCTIONALITY

On a final note, BL is capable of running as a distributed system architecture with multiple machines, different operating systems, talking to each other. Interestingly, this multi-machine functionality has come out-of-the-box thanks to using modern SW technology such as ZMQ. It is *da facto* that the BL SW-sim can perfectly run all the nodes within the same computer, provided that the computer has enough capacity to handle all the concurrent applications.

NUMERICAL SIMULATION

The following BL numerical simulation showcases a distributed-system run with three different nodes: the Basilisk spacecraft physical simulation, the SBC emulator running the mission FSW executable and the Ground System virtual model. The Black Lion Central Controller is also present to synchronize the nodes and act as a message broker between them. The distributed simulation setup operating across three computers running different operating systems is depicted in Fig. 12.

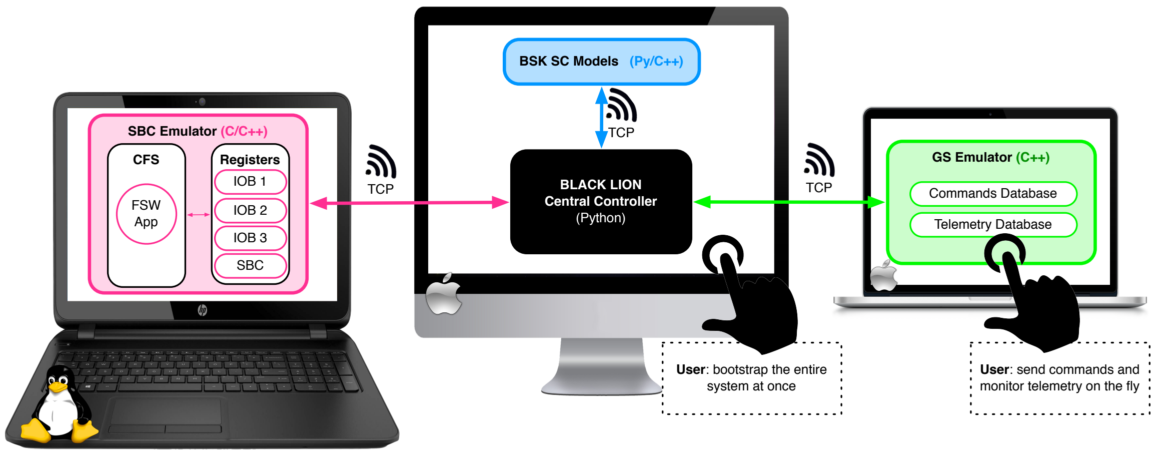

Figure 12. Nodes in the Integral SW-Sim Run.

System Initialization

The computer that runs the Central Controller is the host (unique element in the system with static IP address). Black Lion currently has the capability to simultaneously initialize and configure the Central Controller and all the nodes that are running on the same machine (i.e. nodes that don't require to SSH elsewhere). In the setup depicted in Fig. 12, the Central Controller is initialized and configured together with the BSK node. The FSW-SBC emulator and the GS are initialized independently when they run on different computing platforms, and the only argument they each need is the TCP address of the Central Controller and the specific port on which to bind. From this point onwards, all the setting, configuring and handshaking of nodes is accomplished automatically through the Central Controller's logic without further user interaction.

Node Configurations

Next the particular configuration and application of each node in the showcased simulation is explained.

Basilisk Node Configuration. The Basilisk simulation is configured in an scenario where the spacecraft probe has just separated from the launcher after leaving Earth. The simulated true attitude and rate of the spacecraft are set to the following values:

$$\boldsymbol{\sigma}_{B/N}(t=0) = [0.4, 0.2, 0.1] \tag{1a}$$

$$\boldsymbol{\omega}_{B/N}(t=0) = [0.0, 0.0, 0.0] \tag{1b}$$

where $\boldsymbol{\sigma}_{B/N}$ is a Modified Rodrigues Parameter or MRP based attitude description,[11–13] and $\boldsymbol{\omega}_{B/N}$ is the inertial angular velocity. Hardware components modeled in Basilisk that are relevant in the present simulation include a set of four reaction wheels (which are used to control the spacecraft) and a dual-headed star-tracker (which provide attitude and angular rate measurements).

Ground System Configuration. The virtual GS contains a complete mission suite of command and telemetry databases. The model is used dynamically by the GS user during the Black Lion run to send commands (manually or scripted) on the fly and monitor telemetry. In the specific scenario showcased here, the GS user places two commands. In the presented simulation the first command issued to the FSW places FSW into a monitoring mode where the spacecraft monitors states and reports back the requested states telemetry. The second command encompasses the performance of a maneuver: driving the spacecraft to coarse sun pointing.

FSW Configuration. The FSW executable runs as a CFS application on an RTOS in the Single Board Computer emulator. Four registers are emulated from/to which FSW reads/writes data. These register devices are the SBC register and three different Input/Output Breakout device registers (IOB1, IOB2 and IOB3). Each register stores in memory different data packets. For instance, when commanded to, FSW writes reaction wheel voltage commands for wheels 1 and 3 to the IOB1 and voltage commands for wheels 2 and 4 to the IOB2. The IOB3 register is where star tracker sensor data is stored when it becomes available for FSW to read. Finally, the SBC register stores the uplink commands for FSW to pick as well as the downlink telemetry for ground to process.

Simulation Run

The simulation starts and all the nodes begin to step forward in lock-step. The communication exchange happens once every node has executed for $\Delta t = 0.1$ seconds of virtual time. The total simulation time is 10 virtual minutes.

Throughout the simulation, BSK produces true data of the spacecraft states and, based on that, the modelled hardware sensors create corrupted sensor data. At the start of the simulation, FSW does not know about the spacecraft states and runs like this for several seconds until the GS user sends, through the GUI of the virtual GS, the command to start monitoring the spacecraft ("monitoring only" command). This command comes across as a CCSDS packet that FSW parses. Once the command is parsed, FSW starts ingesting star tracker data produced by the BSK simulation: fused attitude measurements and rate measurements from the Magnetohydrodynamic Inertial Reference Unit (MIRU). The GS user is able to monitor through the GUI the telemetry reported by FSW, which is in turn filtering the sensor data in order to acquire locked attitude knowledge. Once attitude lock is achieved, the GS user sends a new command to the spacecraft to maneuver to sun pointing ("sun

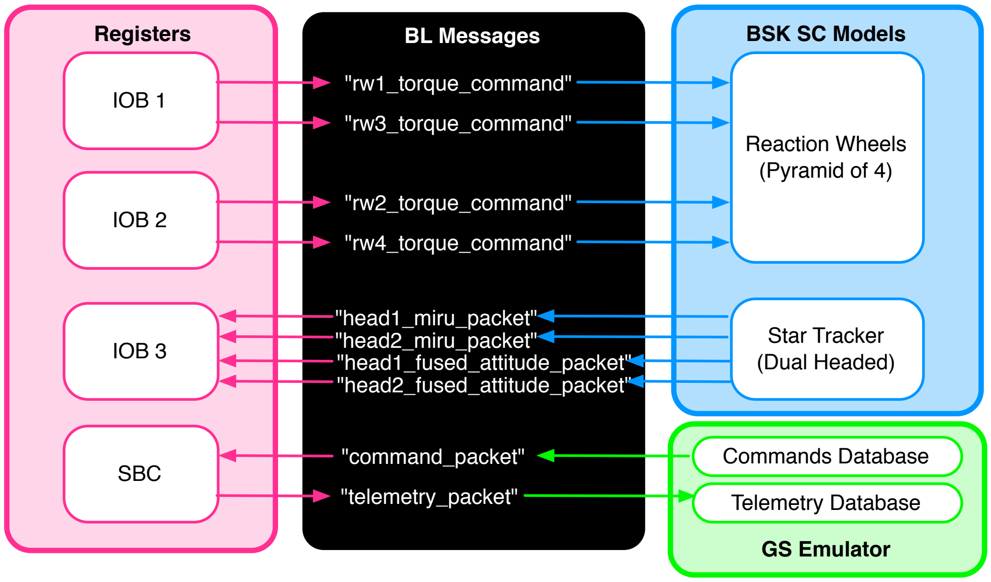

Figure 13. Messages Shipped through the Black Lion Central Controller.

pointing maneuver" command). Once FSW receives the command, the GN&C algorithm tasks start computing the control torque required to drive the spacecraft from its current estimated state to the desired one. These torques come across the BL system as voltage commands for each of the spacecraft's wheels that the BSK simulation ingests and integrates in its dynamics simulation. The GS user does not send any other command, and FSW keeps closing the loop with the spacecraft's dynamic simulation as well as reporting back telemetry to the GS. Figure 13 illustrates the start-point, end-point and direction of all the message connections that happen in the described closed-loop scenario.

Simulation Results

The plots in Fig. 14 show the true MRP set and angular rates as logged from the Basilisk numerical simulation. On the bottom of the plot, the GS commands are marked in green boxes at the time they are sent by the GS user. The first GS command ("monitoring only") is sent after the BL simulation has run for approximately 20 virtual seconds. During the time elapsed between the first and second GS commands, FSW starts filtering the sensor data until convergence. This FSW monitoring period is marked at the top of the plot within a pink box. During the monitoring period, the GS user can observe the attitude locking process (convergence) through the reported telemetry. Of course, the FSW filtering process is not reflected in the MRP and rates evolution of Fig. 14, which only reflect the true simulation data. The second GS command is sent when about 1.5 virtual minutes have elapsed since the beginning of the simulation. It is observed in the plots that the reaction wheel torques commanded by FSW cause the spacecraft dynamics to evolve: through a steering control law, the spacecraft is driven into the new attitude state of sun pointing.

FUTURE WORK AND DEVELOPMENTS

Black Lion is currently operational and functional for SW-sim testing. Features and functionality are continuously being enhanced, with on-going efforts including:

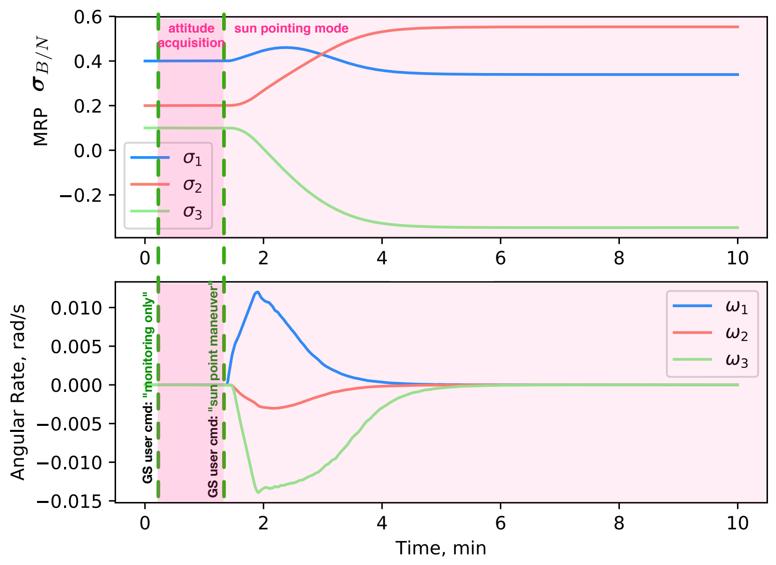

Figure 14. True Data Plots from the Basilisk Spacecraft Simulation.

Dynamic discovery of nodes: This implies 1) Configuring nodes automatically to enter into a particular mode or scenario and 2) Launch/start-up nodes remotely from a single common call within the Black Lion central controller.

Graceful handling of node failure: If one node fails, the user can decide whether to continue running the SW-sim test with the remaining nodes or to restart the simulation run. Further, description of failure modes and recovery logs are *de facto*.

Direct pipe for fault injection: Corruption of spacecraft models (sensors and actuators) and corruption of CCSDS packets can be triggered dynamically from the central controller.

Command and telemetry checking routines: This implies 1) Integration of telemetry post-processing to view flow of commands and telemetry during run time and 2) Addition of simple callbacks to provide telemetry checks (reporting passed/failed states) for runs in which post-processing of telemetry is not desired.

Single-step execution and halt command: The user can place halt commands within the central configuration script in order to run the SW-sim to a specific point and then perform peek and poke, for example: reading out memory locations, inserting new values into memory locations, examining stacks, attaching debuggers to various processes, and so on.

Unification/definition of data interfaces between applications: Electronic Datasheets (EDS). The use of protobuffers enables backwards compatibility regarding changes on the overlay data definition. Without backwards compatibility, changes on data structures (i.e. change of versions) may result in insidious errors, because the most efficient way to pass data is binary format and mismatches in format/versions can go undetected until much later and so can be extremely hard to find.

Handling different SW-sim build configurations: This implies handling of 1) Nodes built in 32 bits and others in 64 bits. Note that some data sizes change when these different options are used, which can disrupt the access of the defined data structures), 2) Teams using different compilers (there can be very subtle differences in data representations) and 3) Nodes with different endianness.

CONCLUSIONS AND FUTURE WORK

The present work has covered the basic aspects of Black Lion, a communication architecture that can be configured to provide an integral SW-sim functionality. Black Lion is currently supporting validation and verification activities for an ongoing interplanetary mission. Yet, what makes the architecture interesting is its flexibility and its scalability. These features are in turn granted by the adoption of modern software tools and techniques. An abstracted communication layer across a diversity of nodes is achieved by means of a unique central controller and two generic APIs attached to each of the nodes. Currently, these APIs have been implemented for nodes whose heterogeneity spans from: multithreaded vs. single-threaded nodes, asynchronous vs. synchronous nodes, little-endian vs. big endian nodes, as well as a variety of programming languages: Python, C and C++. The C# counterpart of the APIs is currently under development.

REFERENCES

[1] J. Cuseo, "STK/SOLIS and STK/ODySSy Flight Software: Supporting the Entire Spacecraft Lifecycle," *2011 Workshops on Spacecraft Flight Software*, Johns Hopkins University Applied Physics Laboratory, Laurel, MD, October 19-21 2011.

[2] J. J. Biesiadecki, D. A. Henriques, and A. Jain, "A reusable, real-time spacecraft dynamics simulator," *Digital Avionics Systems Conference, 1997. 16th DASC., AIAA/IEEE*, Vol. 2, 1997, pp. 8.2–8–8.2–14 vol.2, 10.1109/DASC.1997.637259.

[3] C. S. Lim and A. Jain, "Dshell++: A component based, reusable space system simulation framework," *Proceedings - 2009 3rd IEEE International Conference on Space Mission Challenges for Information Technology, SMC-IT 2009*, 2009, pp. 229–236, 10.1109/SMC-IT.2009.35.

[4] J. Cameron, A. Jain, B. Dan, E. Bailey, J. Balaram, E. Bonfiglio, H. Grip, M. Ivanov, and E. Sklyanskiy, "DSENDS: Multi-mission Flight Dynamics Simulator for NASA Missions," *Aiaa Space 2016*, No. September, 2016, pp. 1–18, 10.2514/6.2016-5421.

[5] S. A. Zemerick, J. R. Morris, and B. T. Bailey, "NASA Operational Simulator (NOS) for V and V of complex systems," Vol. 875205, No. May 2013, 2013, p. 875205, 10.1117/12.2015246.

[6] M. Grubb, J. Morris, S. Zemerick, and J. Lucas, "NASA Operational Simulator for Small Satellites (NOS3): Tools for Software-based Validation and Verification of Small Satellites," *Proceedings of the AIAA/USU Conference on Small Satellites*, Logan, Utah, August 2016.

[7] J. Alcorn, H. Schaub, S. Piggott, and D. Kubitschek, "Simulating Attitude Actuation Options Using the Basilisk Astrodynamics Software Architecture," *67th International Astronautical Congress*, Guadalajara, Mexico, Sept. 26–30 2016.

[8] S. Piggott, J. Alcorn, M. C. Margenet, P. W. Kenneally, and H. Schaub, "Flight Software Development Through Python," *2016 Workshop on Spacecraft Flight Software*, JPL, California, Dec. 13–15 2016.

[9] M. Cols Margenet, H. Schaub, and S. Piggott, "Modular Platform for Hardware-in-the-Loop Testing of Autonomous Flight Algorithms," *International Symposium on Space Flight Dynamics*, Matsuyama-Ehime, Japan, June 3–9 2017.

[10] J. Wood, M. Cols-Margenet, P. Kenneally, H. Schaub, and S. Piggott, "Flexible Basilisk Astrodynamics Visualization Software Using the Unity Rendering Engine," *AAS Guidance and Control Conference*, Breckenridge, CO, February 2–7 2018.

[11] H. Schaub and J. L. Junkins, "Stereographic Orientation Parameters for Attitude Dynamics: A Generalization of the Rodrigues Parameters," *Journal of the Astronautical Sciences*, Vol. 44, No. 1, 1996, pp. 1–19.

[12] S. R. Marandi and V. J. Modi, "A Preferred Coordinate System and the Associated Orientation Representation in Attitude Dynamics," *Acta Astronautica*, Vol. 15, No. 11, 1987, pp. 833–843.

[13] T. F. Wiener, *Theoretical Analysis of Gimballess Inertial Reference Equipment Using Delta-Modulated Instruments*. Ph.D. dissertation, Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, Cambridge, MA, March 1962.