End-to-End Flight Software Development and Testing: Modularity, Transparency and Scalability across Testbeds

Mar Cols Margenet*

MOTIVATION

Space missions rely highly on the efficiency and reliability of the on-board flight algorithms in order to perform autonomous attitude control or orbit corrections. These critical software functions undergo a stringent review and validation process prior to flight which can be both costly and time consuming. The complete engineering cycle to develop a FSW system encompasses an involved path of deploying and running the flight algorithms within different testbed environments. In a standard spacecraft mission there are mainly three testbed environments to consider: desktop computer (for algorithm design, prototyping and rapid iteration), hardware flight processor (for flat-sat testing and eventually flying) and emulated flight processor in a virtual machine (for emulated flat-sat testing). These environments are illustrated in Fig. 1, where the term single board computer (SBC) is used to refer to the flight processor. The two latter environments –hardware or emulated flight processor– are considered to be embedded. Since a regular desktop computer environment and an embedded flight processor environment are very different in terms of resources, capabilities and end-user programmability, migrating the flight algorithms from one environment to the other generally demands a significant engineering effort. Further, there is also a disparity in the testing tools and procedures that each testbed currently allows. This thesis investigates **end-to-end FSW development strategies** and working implementations that support having both desktop and embedded environments separately while minimizing the existing gap between them in order to ensure: firstly, **transparent migration** of the flight application and, secondly, **consistent testing** throughout the different testbeds.

The term *end-to-end* used in this thesis implies that the entire FSW development cycle is covered: starting from a preliminary desktop design and analysis all the way to testing on the flight hardware. Regarding desktop FSW development, this thesis focuses particularly on **architecting flight algorithms through modular designs and shared coding standards**. At its aim, FSW is intended to support the rest of the system for which it has been

Figure 1. FSW testbeds: desktop and embedded

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

designed; yet in practice it often ends up slaying the other system components due to lacking architecture and proper implementation[1]. The development of inflexible, mission-specific flight algorithms is, indeed, a recurrent problematic pattern in the aerospace industry that needs to be addressed[2]. Architectural design of flight algorithms takes place in the desktop prototyping phase and the design decisions made here impact portability and testability across all testbeds. Regarding flight hardware, this thesis puts special emphasis on **emulated flat-sat testing of the embedded FSW.** The emulation of embedded systems is particularly interesting because it provides pure software substitutions for expensive hardware components of limited quantity that might be needed simultaneously for testing by different mission groups[3, 4, 5].

As for transition between desktop and embedded environments, the flight application shall be able to migrate in a way that is transparent. Otherwise, inconsistency is introduced and the validity of the flight application from one environment to the other cannot be readily inferred[6]. In order for the flight algorithm migration to be transparent, it is critical that the source-code itself remains unchanged during migration – the underlying idea being, as the long-held NASA saying goes, "test what you fly, fly what you test", since the first day of development until the last one, from the desktop environment all the way into the embedded one. Regarding consistent and high-fidelity testing throughout environments, such endeavour can be effectively achieved my making use of a distributed communication architecture. Alongside the prototyping of flight algorithms for a given mission, spacecraft physical models are also built with the purpose of testing the FSW algorithm set in a simulated closed-loop. In a flat-sat configuration, there are additional external models interacting with FSW like the ground system. On these lines, flexible and scalable communication architectures enable integration of heterogeneous and independent mission components into a single simulation run, which works seamlessly whether FSW executes from desktop or embedded environment. The concept of an emulated flat-sat for embedded FSW testing is illustrated in Fig. 2.

The end-to-end FSW development strategies pursued in this thesis consider exclusively open-source products and strive for the embedded system to be as close as possible to the desktop testbed in terms of user-friendliness and interaction functionalities, while still adhering to the needs of space: determinism, concurrency and low use of resources. Currently, deploying an embedded flight system and migrating flight algorithms on it is not an easy task. However, many small-satellite missions or start-up companies without extensive FSW legacy would highly benefit from having available an end-to-end FSW development tool-suite.

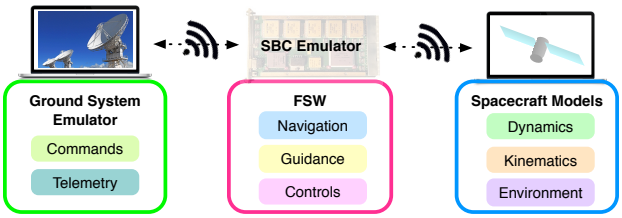


Figure 2. Concept of emulated flat-sat

BACKGROUND

This section aims to provide the reader with further context on desktop and embedded systems, by describing their features as well as the development approaches that each environment supports. In addition, the concept of middleware as a transition point from software to hardware is introduced.

Desktop Systems

Thanks to the use of state-of-the-art processors and operating systems, desktop systems are the most flexible environments in terms of computing speed, memory available and user friendliness. Because of its flexibility, the desktop environment is used in the preliminary step of prototyping mission-specific flight algorithms. These FSW algorithms are usually tested in closed-loop dynamics simulations with spacecraft physical models until the desired algorithm performance is achieved. For the purposes of desktop FSW development, the use of high-level scripting languages like Python or Matlab is extremely convenient as it enables rapid prototyping and iteration. However, scripting languages are not suitable for embedded flight applications requiring absolute control of timing and deterministic behavior. Hence, the source-code of embedded flight targets is usually required to be written in programming languages like Fortran, C or C++, in order to ensure execution speed and determinism. Two different desktop development approaches that are commonly adopted in the aerospace community are discussed next: model-based development (MBD) and Python interface with underlying C/C++ code.

MBD consists on performing architecture design and modeling of both software functions and hardware subsystems using block-diagram programming software tools like, for example, Mathworks's Simulink* and National Instruments LabVIEW†. Next, an automated source-code generation software tool is used to translate the graphical design into programming source-code. This step is also known as auto-coding. The MBD process is depicted in Fig. 3. In spite of its convenience, MBD does not respect the principle of migration transparency: FSW validity from model-in-the loop (MIL) simulations to software-in-the-loop (SIL) simulations cannot be readily inferred without extensive additional testing[6]. Additional challenges with automatically generated code are that: 1) it is usually less efficient in either size or execution than optimized hand-written code and 2) it can be very challenging to edit and debug due to the lack of readability[9].

Figure 3. Model-based development

An alternative to MBD is the use of Python wrapping C/C++ source code. The Python language is recognized as an excellent scripting environment and code development testbed that would lend itself very well to the FSW development process if the code could run as FSW. Although the Python runtime is generally recognized as insufficiently well-

*https://www.mathworks.com/products/simulink.html
†http://www.ni.com/en-us/shop/labview.html

controlled for time-critical applications like those required for aerospace FSW, looking at the internals of the language reveals that most built-in modules requiring speed (e.g. numpy*) are actually written in C/C++ and then wrapped into Python using Python language bindings. Using this logic, it makes good sense that Python could serve as an excellent testbed for FSW development if the FSW code is written exclusively in C/C++ and then wrapped into Python for user-interaction. This development proposal is depicted in Fig. 4. The advantage of this approach is that the C/C++ source-code does not change for migration into embedded systems; the Python wrapper is simply removed.

The Basilisk astrodynamics framework is a desktop FSW testbed that seeks to capitalize on the potential of using Python as a wrapper testbed for C/C++ algorithm code[10, 11, 12]. Other aerospace software tools using Python as a wrapper are MONTE, a deep-space navigation tool developed by the Jet Propulsion Laboratory[13], or Dshell, a physical simulator also developed by JPL that supports both robotic and spacecraft simulations[14].

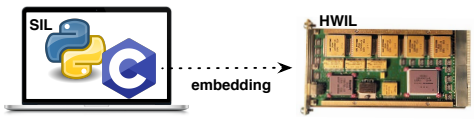


Figure 4. Model-based development

Embedded Systems

Time-critical applications like those of FSW usually demand the use of on-board processors with drastically fewer resources available than the typical desktop environment. Therefore, FSW systems are said to be constrained or embedded. Embedded environments are, in essence, electronic systems which are managed by a microprocessor (like a hardware flight processor) or micro-controller that operates the whole system with precise timing, hence ensuring deterministic behaviors. A significant problem of using scripting languages or big libraries in a flight application is their memory footprint, since there is reduced memory (RAM/ROM) available on a typical flight system. Embedded flight processor environments are defined by the selection of two items: the microprocessor board and the Real-Time Operating System (RTOS). An alternative to using a hardware processor is to emulate it on avirtual machine[3, 4, 5]. Regardless of the flight processor board being physical or emulated, it still represents an embedded environment. Embedded flight processors lag state-of-the-art processors (like those in a desktop computer) by about 10 years due to flight heritage and radiation-hardening requirements[16]. Radiation hardening of processors is important in order to ensure un-interrupted operation over long durations. Some of the most common radiation-hardened processors used in space are RAD750, Coldfire or LEON, all of them being very expensive and presenting similar limited performance. Examples of processor board emulators for any of the aforementioned hardware boards are QEMU†, which is an open-source product, or EMU, created by the ESOC center of the European Space Agency[17].

*https://numpy.org
†http://qemu.org

Middleware Systems

Because a regular desktop computer environment and a flight processor environment operate differently, migrating the flight application from one to another demands a significant migration effort. Furthermore, this effort is intrinsically linked to the specific processor board and RTOS chosen, becoming then mission-specific. An alternative target for mission flight algorithms is a middleware layer. Middleware can be regarded as an abstraction layer or "glue-code" that ensures portability of the flight application among different processors and RTOS. An example of middleware is the Core Flight System (CFS), which is an open-source product provided by NASA Goddard Spaceflight Center[18, 19]. While targeting middleware can be worthwhile in the long run, small missions do not tend to follow this approach given the complexity and steeper learning curve of the work entailed[9]. However, if a user-friendly, easily-deployable middleware layer existed, the number of missions embracing reusability through middleware would most likely increase. Recently, a new, lean and efficient implementation of the Python 3 programming language has appeared that is named MicroPython and that is very compelling for use in embedded FSW systems. MicroPython includes a small subset of the Python standard library and is optimized to run on micro-controllers and in constrained environments. This promising language is already being investigated by ESA for onboard-control procedures (OBCPs) in spacecraft payloads [15, 20]. OBCPs are flight procedures that provide a flexible way to operate the spacecraft either by extending the onboard software functionality or by modifying the behaviour of onboard applications. The application of MicroPython for OBCPs is then different that its application as a middleware layer.

STATE-OF-THE-ART FSW DEVELOPMENT TOOLS

Currently, several software tools –or rather combinations of them– exist that support the entire FSW development cycle, from the desktop prototyping environment up to the embedded one. With some generalization, FSW development architectures are characterized by the degree to which system components are coupled. The coupling between simulation components is manifested by the simulation structure, where the overall 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. Some of the most comprehensive tool-suites include: MAX Flight Software* together with the On-Board Dynamic Simulation System (ODySSy)[21], both provided by Advanced Solutions Inc (ASI); Matlab's Simulink† and Matlab's Embedded Coder‡ together with NASA's Trick engine[22]; JPL's Dshell system[14] together with the NASA Operational Simulator (NOS)§; or the Basilisk¶ desktop testbed together with the Black Lion[3] communication architecture.

These state-of-the-art options differ on the specific tools they use and on the degree of coupling between them.

*http://www.go-asi.com/solutions/max-flight-software
†https://www.mathworks.com/products/simulink.html
‡https://www.mathworks.com/products/embedded-coder.html
§https://www.nasa.gov/centers/ivv/jstar/jstar_simulation.html
¶https://hanspeterschaub.info/bskMain.html

ASI's proposal is an example of a tightly coupled tool-suite, where MAX Flight Software is used for FSW design and ODySSy is used for closed-loop testing. While this option provides testability in both desktop and embedded environments, it does so by requiring these tools specifically and there are minimal options to substitute one component with another which was not intended to operate with ASI's system. A more flexible combination is the one encompassing the use of Mathwork's Simulink for FSW design, Mathwork's Embedded Coder for FSW migration and NASA's Trick engine for closed-loop testing. This specific tool-suite is currently being applied to the Orion mission[23]. The main problem with both of these proposals is that they rely on an MBD approach for desktop development, which does not respect the principle of migration transparency. Within ASI's proposal, auto-coding is performed by MAX Flight Software; within the Orion mission proposal, auto-coding is performed by Mathwork's Embedded Coder. In addition, none of these FSW development proposals addresses the integration of code into flight targets beyond auto-coding. Note that auto-coding simply produces source-code in the target programming language, but this code still needs to be integrated either into a middleware layer or a specific board and RTOS. Further these proposals do not natively offer a distributed testing environment: both ODySSy and Trick (i.e. the spacecraft physical simulations) are meant to migrate with FSW, which is a fact that also raises questions about the level of fidelity reachable.

A software suite which demonstrates increasing modularity in its architecture is JPL's proposal through Dshell and NOS. The Dshell system is a physical simulator while NOS constitutes the communication framework to run distributed software simulations of independent mission components[24, 25], with Dshell being one of its potential components. Although the combination of Dshell and NOS conform a very compelling option for distributed closed-loop testing, which would work seamlessly whether FSW resides in the desktop or in the embedded environment, these software tools are not open-source. Further, this tool-suite does not provide a FSW development environment on itself, which is let to be chosen by the specific mission.

In overall, the state-of-the-art tool-suites that have been reviewed fall short in one or more of the following categories: completeness of the tool-suite as a FSW testbed, transparency of the fight algorithm flow between testbeds, architectural flexibility to include external models for testing, support of distributed simulations and open-sourcing of the tools to the community.

OUTLINE

The niche identified in existing end-to-end FSW development tool-suites has motivated the development of the Basilisk (BSK) software testbed and the Black Lion (BL) communication architecture. This thesis has contributed to both. Basilisk is an open-source, cross-platform, desktop testbed for designing flight algorithms and testing them in closed-loop dynamics simulations. Basilisk 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++. In turn, BL is a purely software-based communication architecture aimed at integrated testing of independent spacecraft mission

models –with Basilisk being one of its potential components. Both Basilisk and Black Lion are currently being implemented by the Autonomous Vehicle Systems (AVS) laboratory at the University of Colorado Boulder and the Laboratory for Atmospheric and Space Physics (LASP) in support of an ongoing interplanetary spacecraft mission. Yet both tools are being built under the principles of flexibility and reusability and, as demonstrated in this thesis, their application extends far beyond this particular mission. The different phases of the FSW development process that are covered in this thesis, as well as their scope, are outlined next:

1. **Desktop FSW developing and testing: modular attitude guidance reference generation for distinct mission profiles.** The relevance of architecting flight algorithms through modular designs and shared coding standards is showcased through a guidance application: onboard attitude reference generation in a modular fashion. The proposed work involves: 1) Breaking up the mathematical transformations required to build up complex guidance motions into atomic functions; 2) developing a suite of atomic guidance software modules; and 3) integrating these modules in Basilisk for simulated closed-loop testing.

2. **Migration of the flight application into portable flight targets.** This thesis describes the migration process of flight algorithms from a desktop prototyping environment, Basilisk, to several flight targets: commercial processors (the Raspberry Pi) as well as embeddable middleware systems (the Core Flight System and MicroPython). The migration strategies to embeddable systems are novel implementations that ensure complete transparency throughout the process.

3. **Embedded FSW development and testing in an emulated flight processor.** This thesis uses an emulation of a LEON board to execute and test FSW applications on top of the RTEMS real-time operation system. Two different embedded FSW applications are considered: a Core Flight System application and a MicroPython application. The Core Flight System application corresponds to the FSW executable of the spacecraft mission in which the AVS laboratory and LASP are collaborating. The work presented here includes the modelling of FPGA registers and complex avionics hardware in order to interact with the embedded FSW application. Numerical simulation results from emulated flat-sat testing runs are shown.

4. **Distributed closed-loop testing (Black Lion communication architecture).** The design and implementation of the Black Lion communication architecture are presented as part of the core work in this thesis. BL is shown to be suitable for both distributed desktop testing as well as embedded testing. Some of its current applications are: 1) Embedded FSW testing in an emulated flat-sat configuration and 2) Hardware-in-the-loop simulation of a constellation of satellites performing distributed attitude guidance.

DESKTOP FSW PROTOTYPING AND TESTING: MODULAR ATTITUDE GUIDANCE

Architectural design of flight algorithms takes place in the desktop prototyping phase and it is enabled by flexible development environments that allow complete user customization of both the algorithm structure and contents, while leveraging the recurrent yet time-consuming tasks of deploying, building and compiling the executable to test. To this end, the Basilisk environment is presented as a flexible, desktop FSW testbed. Upon Basilisk, a modular scheme for generating attitude reference motions is architected and implemented. The layered approach of building an attitude reference is interesting because it promotes code reusability in a topic area that tends to be highly mission specific. Once proved, the validity of the modular guidance approach still holds out of the Basilisk environment.

The Basilisk Testbed

Basilisk is architected in a modular, highly reconfigurable fashion using C++ modules that perform spacecraft physical simulation tasks and C modules that perform mission-specific GN&C tasks. Currently, the Simplified Wrapper and Interface Generator (SWIG) is used within Basilisk to wrap the C/C++ modules and make them available at the Python layer for setup, desktop execution and post-processing.

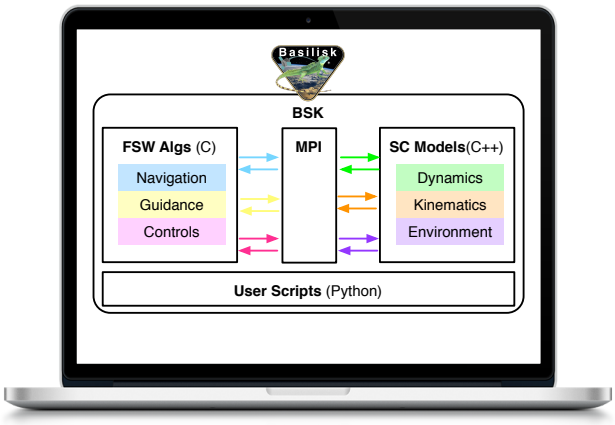


Figure 5. Basilisk (BSK) desktop environment

Figure 5 illustrates the nominal setup –but not necessary required– of a Basilisk desktop simulation. This setup is decomposed into two main processes: a high-fidelity simulation of the physical spacecraft and a suite of GN&C flight algorithms. Each of the processes has multiple tasks and, to each task, any number of C/C++ modules can be added. During a simulation run, the different C and C++ modules communicate with each other through a custom message passing interface (*MPI* in Fig. 5). The modularity of the Basilisk system implies that each process is decomposed into a series of simpler steps and exchangeable components. The cascading of modules is set at the Python level, allowing different levels of simulation fidelity and flight software sophistication. Such data flow allows, for example, the modular guidance application that is presented next.

Modular Attitude Guidance: Generating Rotational Reference Motions for Distinct Mission Profiles

In software development, the idea of modularity consists on breaking up the overall functionality into a series of simpler steps or modules, instead of having a single, large piece of software performing a complex function. Generally, talking about modularity of GN&C would mean that there are separate modules used for sensory input, parameter identification, reference trajectory selection/generation, position error determination, control, and output. This kind

of split is seen, for example, in the GN&C sequences presented in [26, 27]. The guidance application shown in this thesis focuses on the attitude reference trajectory generation in particular. The novelty lays on bringing modularity one step further by dividing the reference generation itself into multiple, exchangeable subcomponents. The advantage of fractionating the reference generation is that, for a given set of core modules, a wide variety of guidance behaviors can be achieved through combination and distinct initialization.

Without loss in generality, this work investigates a methodology to modularize any attitude reference motion into three atomic parts: a base pointing reference, a dynamic reference that is relative to the base, and an attitude offset. The final, desired reference motion is a result of cascading these three attitude reference parts, as depicted in Fig. 6. The proposed modular strategy exploits the fact that different spacecraft attitude guidance profiles often share some core functionalities, as illustrated in Fig. 7.

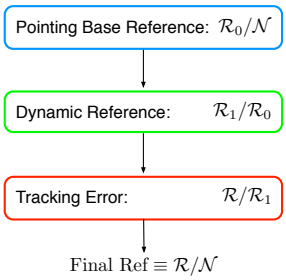


Figure 6. Attitude Guidance Generation

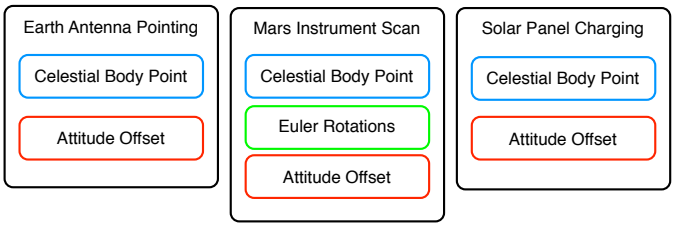


Figure 7. Break-down of sample mission profiles

Problem Statement. For a given spacecraft, the goal of the onboard GN&C flight software is to estimate the current state of the spacecraft body $\mathcal{B}$ (navigation task), generate a reference state $\mathcal{R}$ that can be time-varying or not (guidance task), derive the attitude tracking error between the current state $\mathcal{B}$ and the desired one $\mathcal{R}$ (also guidance task), and apply the required control torque to align $\mathcal{B}$ with $\mathcal{R}$ (control task). Both the spacecraft-body state $\mathcal{B}$ and the reference state $\mathcal{R}$ computed onboard are expressed with respect to an inertial frame $\mathcal{N}$. Note that $\mathcal{N}$ is kept as a generic inertial frame to be chosen (it could be J2000, Earth fixed-frame, Mars centered frame, etc.) since all that matters is consistency on the picked inertial frame. The computed reference state $\mathcal{R}$ is, in this work, composed of three parameters:

$$\mathcal{R} = [\boldsymbol{\sigma}_{R/N}, {}^{\mathcal{N}}\boldsymbol{\omega}_{R/N}, {}^{\mathcal{N}}\dot{\boldsymbol{\omega}}_{R/N}] \tag{1}$$

These parameters are: an inertial attitude measure, denoted through the Modified Rodrigues Parameters (MRP) set $\boldsymbol{\sigma}_{R/N}$ [28], an inertial angular rate vector ${}^{\mathcal{N}}\boldsymbol{\omega}_{R/N}$ expressed in inertial frame $\mathcal{N}$ components, and an inertial angular acceleration vector ${}^{\mathcal{N}}\dot{\boldsymbol{\omega}}_{R/N}$ also in $\mathcal{N}$-frame components. The left-superscript denotes the frame with respect to which the vector components are taken.

Figure 8 illustrates the generation of a compounded compounded final reference $\mathcal{R}$. The final reference is computed through addition of a base pointing reference, two dynamic reference motions (each one relative to the former one) and an attitude offset. The tracking error module is always the last component in the guidance chain. This module reads the output of the latest reference module and adds an attitude offset if applicable. An attitude offset is applied when the generated attitude reference is meant for a spacecraft fixed frame that is not the main one. The output of the tracking error module is the output of the entire attitude guidance block and it is used to feed the control block next. Note that any number of dynamic references modules could be sequentially chained to create increasingly complex rotational patterns. The key to this scalability is that, as shown in Fig. 8, all the reference generation modules (both pointing base and dynamic) output the same message structure defined in Eq. (1).

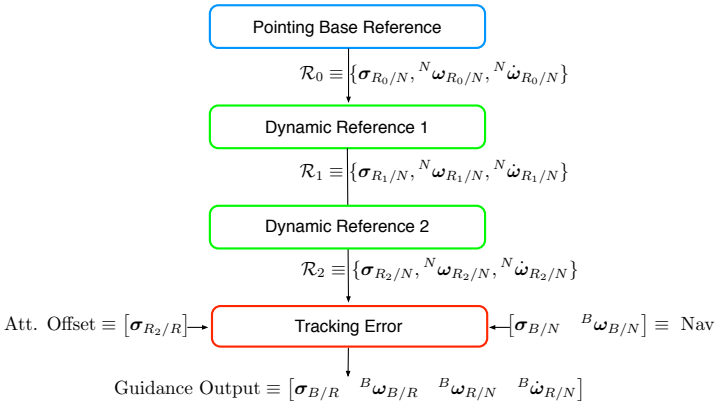
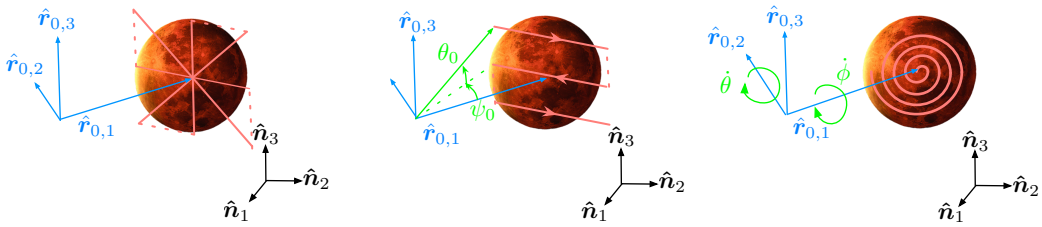


Figure 8. Inputs and outputs of a compounded attitude reference chain

Software Modules. For each of the aforementioned core functionalities (base pointing reference, dynamic reference, and attitude offset) several software modules are implemented, which can be plugged-and-played as lego-like pieces in order to achieve very different rotational patterns.

The first guidance stage consists of modules that generate a base pointing reference $\mathcal{R}_0$ that can be either inertial or non-inertial. The common feature of the base modules is that the generated reference does not depend on any prior reference frame. The base reference modules developed and later integrated to the Basilisk software framework are: inertial pointing, Hill orbit pointing, velocity orbit pointing, and constrained two-body pointing[29] . Both the inertial and orbit frame references are widely used and well documented. The novelty lies on the scheme upon which they are architected: through the modular stack and interface definition, base modules can be used in stand-alone mode or as the base of complex dynamic behaviors. The constrained two-body pointing module is a novel kinematic solution to a constrained attitude pointing requirement.

The second stage consists of modules that define a dynamic reference motion relative to the former one. The dynamic reference modules developed and integrated into Basilisk are: inertial 3D spinning and 3-2-1 Euler angle rotation. The 3-2-1 Euler angle rotation module is particularly interesting because complex dynamic motions can be achieved through elegantly simple constant Euler rates. While Euler angles are often avoided in guidance algorithms because of their mathematical singularities, here the advantages of the Euler sets are exploited in a robust and safe manner that is free of numerical issues[12]. Multiple Euler modules can be cascaded with one another and, through different initialization/setup of the same module, a wide variety of rotational patterns can be achieved. Figure 9



(a) Rectangular Raster Maneuver. (b) Asterisk Raster Maneuver. (c) Spiral Maneuver.

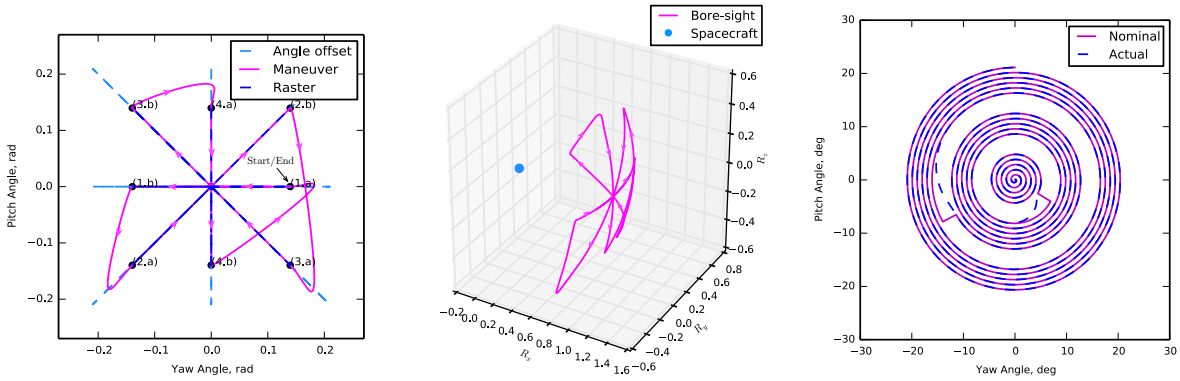
Figure 9. Sample Scanning Patterns.

illustrates three sample scanning patterns that can be performed by consecutive requests of Euler angle offsets and rates. In Fig. 9, the scanning is performed relative to the time-varying base reference frame $\mathcal{R}_0 : \{\hat{\boldsymbol{r}}_{0,1}, \hat{\boldsymbol{r}}_{0,2}, \hat{\boldsymbol{r}}_{0,3}\}$, which keeps track of the orbited celestial body.

The last component in the guidance chain is always the tracking error module, responsible of ensembling the final reference $\mathcal{R}$ and computing the tracking errors with respect to the principal body frame $\mathcal{B}$. The novelty of this module is found in the strategy used for controlling a spacecraft-fixed frame $\mathcal{B}_c$ that is not the principal one. Instead of expressing the guidance output in terms of the specific control frame $\mathcal{B}_c$ (which would then have to be accounted for in the control law), the offset between the main body frame $\mathcal{B}$ and the control frame $\mathcal{B}_c$ is added to the generated final reference $\mathcal{R}$. The advantage of this trick is that it reduces the number of transformations required to guide and control a generic spacecraft fixed-frame that is not the principal one.

Integrated Numerical Simulations. All the presented guidance modules have been integrated within Basilisk for dynamic closed-loop testing[12]. In this thesis, the spacecraft behaviour over two different scanning maneuvers is shown. These scans are the asterisk pattern and and the spiral pattern shown in Fig. 10.

An asterisk scanning pattern is conformed by a total of four raster lines. Using the Euler rotation module, the



(a) 2D view of asterisk scanning (b) 3D view asterisk scanning (c) 2D view of spiral scanning

Figure 10. Euler module scanning patterns

complete pattern can be achieved with only four command steps. Figure 10(a) compares the desired nominal raster lines (dark blue lines) with the actual yaw-pitch angles scanned in the maneuver (magenta lines). Figure 10(b) shows a 3D-view of the performed bore-sight pointing.

An spiral scanning pattern can be easily performed through a 1-2 sequence of constant Euler rates. In order to achieve such sequence, multiple dynamic references are concatenated on top of a base frame (i.e. the same 3-2-1 Euler rotation module is instantiated twice). Figure 10(c) shows a triple-spiral maneuver, where each spiral is determined by a different 1-2 Euler sequence. In this case, the nominal raster lines commanded from guidance are depicted in magenta while the actual angles scanned by the spacecraft bore-sight are painted in blue.

Conclusions

A modular approach for building guidance attitude references has been implemented and demonstrated in the desktop development enviroment. This layered strategy promotes reusability of code throughout distinct mission profiles and, as a matter of fact, it is currently being applied in an interplanetary spacecraft mission to perform both science and nominal maneuvers. Interestingly, Ref. [32] extends the modular guidance strategy into relative ADCS guidance within a formation of satellites, while preserving the same full kinematic properties.

MIGRATION OF THE FLIGHT APPLICATION

Once flight algorithms have been proved to satisfy mission requirements in the desktop development environment, the next step is to migrate them into the mission flight target of choice. This section describes the migration process of flight algorithms from the Basilisk desktop environment to several flight targets: commercial processors (the Raspberry Pi) as well as embeddable middleware systems (the Core Flight System and MicroPython). The transition to commercial flight targets is relevant given the raising interest on using multiple modern processors in redundant configuration for space exploration, instead of a single radiation hardened-processor[7, 30, 31, 8]. In turn, the transition into embeddable systems presented in this thesis is a novel strategy that moves away from auto-coding by ensuring complete transparency throughout the process.

Commercial Processors: The Raspberry Pi

The concept of migrating Basilisk-developed flight algorithms into the Raspberry Pi and testing them in closed-loop simulations is illustrated in Fig. 11. The Raspberry Pi in particular has a built-in ARM processor and comes out-of-the-box with the Debian OS, which a flavour of Linux. While Basilisk is cross-platform in nature and supports Linux, MacOS and Windows, compiling and building the Basilisk software on Debian required additional technical effort,

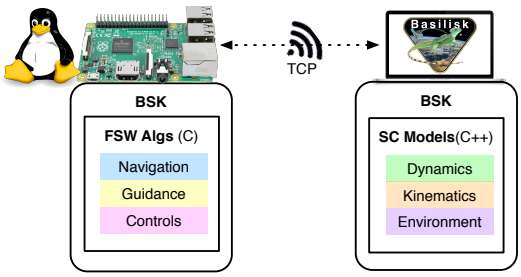


Figure 11. FSW on the Raspberry Pi: ARM processor and Linux OS

mainly due to the SPICE* library included within Basilisk. It was necessary to build from source the Cspice Linux 32bit version but manually modifying the compilation flags in order to target the 64-bit CPU architecture of the latest Raspberry Pi hardware. All the other Basilisk libraries were linked without problems.

While a Basilisk closed-loop simulation could run entirely on the Pi platform (i.e. single platform simulation), the distributed configuration in Fig. 11 replicates reality better. Reference [8] showcases a numerical simulation with the setup of Fig. 11, running in soft real-time. There, an inertial pointing guidance maneuver is performed on a spacecraft that is initially tumbling. The highlight of the results is that the FSW behavior in a distributed hardware-in-the-loop simulation and in a single-platform desktop simulation are identical.

Embeddable Target I: The Core Flight System

The concept of migrating flight targets into a middleware layer is depicted in Fig. 12. Middleware layers, which are characterized by its portability, can be readily embedded into a radiation-hardened flight processor or its emulated counterpart, as in Fig. 12.

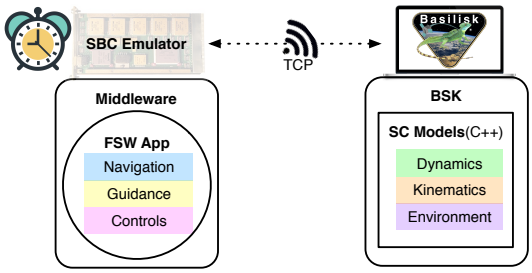


Figure 12. FSW on middleware, embedded in an SBC emulator

This section describes the technical steps required to migrate Basilisk-developed flight algorithms into a Core Flight System application that is actually embeddable. Recall that Basilisk leverages the us of Python for FSW(C/C++) **setup, desktop execution** and **post-processing**. There is one of these Python functionalities that needs to be translated into C for migration: the **setup** for the C flight algorithms. Then the flight application would be all in C, ready to be integrated into CFS. The next question is, of course, what is the concise meaning of "setup" code. In the Basilisk framework, the Python setup encompasses: **1)** variable initialization of each individual C module and **2)** grouping of modules in tasks that run at certain task rates.

The interesting part of the story is that the translation of the setup code from Python to C can be handled automatically via an independent script written in Python: the *AutoSetter.py*. The beauty of the *AutoSetter.py* is that is not a black box but rather an open and transparent python template that maps Python variable types/values into their C counterparts. The resulting C setup code is minimal and completely human readable. Figure 13 illustrates the mechanism of translating the setup code from the desktop Python scripts into C. A key remark here is that the flight algorithm source code remains unchanged: the pure-C application is conformed by the unchanged algorithm source code plus one additional header (*setup.h*) and source file (*setup.c*) containing the setup code written in C.

*https://naif.jpl.nasa.gov/naif/toolkit.html

The workings of the *AutoSetter.py* essentially rely on Python's introspection capabilities. Looking at oneself is something that neither C or C++ can accomplish without significant investment in source parsing. In contrast, Python can readily realize that, within a given Basilisk simulation, there is a FSW process with a list of tasks. And inside each task, there is a list of modules that, despite being written in C, now appear as Python objects; therefore the modules now have all these Python built-in properties like *__module__*, *type( )*, *__name__*, *dir( )* and so on, which are the key to introspection. Figure 14(a) illustrates that the unmodified FSW algorithms plus the auto-generated C setup code constitute a CFS application that is embeddable.

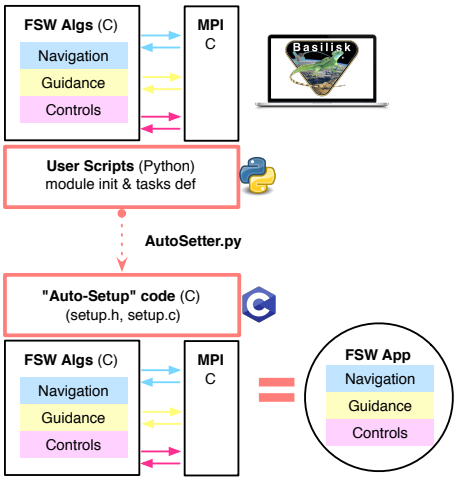
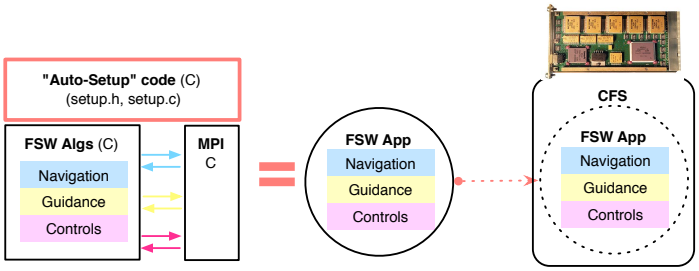
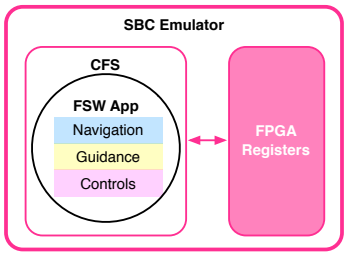


Figure 13. Flight algorithm migration into CFS through the "AutoSetter.py"

The embedded CFS-FSW application can be tested in an emulated flat-sat configuration. In such case, the CFS-FSW application runs within an SBC emulator, which is responsible for enabling interaction between FSW and the external world. Once FSW is embedded, however, it is no longer simple to access the FSW states for reading and writing. For this reason, it has been necessary to emulate the FPGA registers of the SBC. These registers have been modelled as a memory map for input and output of raw binary data. The layout of the combined CFS-FSW and modelled registers within the SBC emulator is represented in Fig. 14(b).



(a) Embedded CFS-FSW Application



(b) Emulated FPGA register space

Figure 14. Embedded testing of CFS-FSW

Due to its complexity, the relationship between the CFS-FSW states and the register space is treated in later sections of this manuscript. Numerical simulations testing the CFS-FSW application in an emulated flat-sat are also shown later on.

Embeddable Target II: MicroPython

Viewing the generalized interest in Python for desktop FSW development, it makes good sense to consider MicroPython as a middleware for embedded development. MicroPython is a lean (i.e. memory lightweight) and efficient (i.e. with fast execution) implementation of the Python 3 programming language that includes a small subset of the Python standard library and that is optimized to run in micro-controllers and in other constrained environments. It is packed full of advanced features (more alike desktop) while being compact enough to fit and run within just 256k of code space and 16k of RAM.

The idea proposed in this thesis is to use MicroPython for **embedded setup** and **embedded execution** of the same, unmodified C/C++ flight algorithm code as in the desktop. However, the standard way of extending MicroPython with custom C or C++ modules involves a lot of boilerplate code. For this reason, another open-source software tool is introduced: the MicroPython C++ Wrap* is a header-only C++ library that provides some interoperability between C/C++ and the MicroPython programming language. Using the MicroPython C++ Wrap, the process of integration with MicroPython is drastically reduced. However, this comes at the cost of requiring all the modules to be written in C++ rather than in C. Currently, all the Basilisk FSW modules are written in C. With this in mind, the technical work required to migrate Basilisk flight algorithms into MicroPython consists of three tasks:

1. **Creating a C++ class for every C FSW module**: the suggested approach for wrapping the unmodified C FSW algorithms as they currently exist in Basilisk is to a create a C++ container/wrapper class (new .hpp file) for every module (.h and .c file) there is.

2. **Generating integration code for every C++ class that needs to be available at the MicroPython layer**. MicroPython is meant to interact directly with the recently created C++ wrapper classes, treating them as if they were native Python modules. In order to achieve this behaviour, it is necessary to recompile MicroPython after having declared and registered the different C++ classes, functions and types.

3. **Adapting existing desktop Python scenario scripts into MicroPython**. Since MicroPython is only a light version of the Python 3 programming language, some advanced Python functionalities or large libraries, like those required for post-processing, are not supported. If this constraint is accounted for, the desktop Python scripts could be used seamlessly within MicroPython. The caveat is that, currently, Basilisk desktop scripts import and instantiate C FSW modules, while MicroPython imports their corresponding C++ wrapper classes.

The interesting part of the described migration tasks is that the creation of the C++ wrapper classes and the generation of the MicroPython integration code can be handled automatically. Let us recall the introspection capabilities that are inherent to the Python language. Similarly to how the *AutoSetter.py* produced specific C setup code for integration

*`https://github.com/stinos/micropython-wrap`

within CFS, an equivalent script has been developed to authomatize the integration within MicroPython. This new introspective script will be referred to as the *AutoWrapper.py*. The process of migrating Basilisk FSW modules from the desktop environment into MicroPython through the *AutoWrapper.py* is illustrated in Fig. 15. Note that the input to the *AutoWrapper.py* is simply a desktop Python scenario script and the output are the corresponding C++ wrapper classes and the MicroPython integration code patch.

While Basilisk uses Python for setup, desktop execution and post-processing, MicroPython can only handle setup and embedded execution but not post-processing. Having said that, MicroPython is capable of logging all the data from an execution. Since the problem is about pulling and plotting such data within the constrained environment, an alternative is to archive the data in a binary file. Thanks to the interoperability between MicroPython and regular Python, the archived results can be loaded without modification back to the desktop Python environment for regular post-processing. Figure 16 illustrates the proposed post-processing mechanism through an archived binary file.

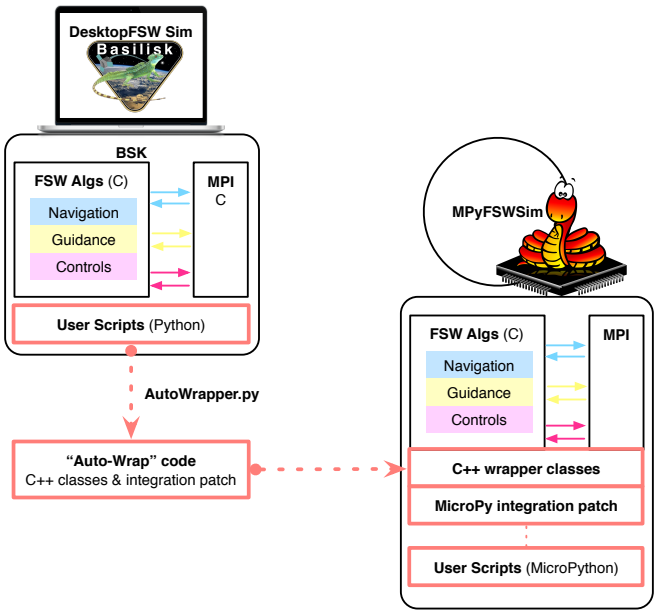


Figure 15. Flight algorithm migration into MicroPython through the "AutoWrapper.py"

The key aspect of this approach is that the post-processing simulation works seamlessly whether the archive file has been created from a Python desktop simulation or from a MicroPython run.

In order to fully prove the validity of MicroPython as middleware for FSW, it is necessary to test the performance of the flight algorithms in simulated closed-loop. With this purpose in mind, a simplified register space has been developed as a Basilisk module that enables interaction between MicroPython and the external world. Figure 17 illustrates the distributed simulation setup used for testing MicroPython FSW. In turn, Fig. 18 shows the numerical simula-

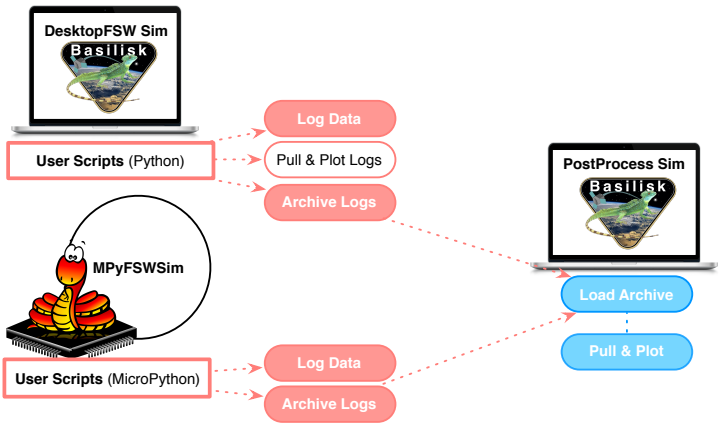


Figure 16. Post-processing mechanism

tion results of a closed-loop guidance maneuver. Here, the FSW algorithms running within MicroPython bring the spacecraft to an inertial 3D pointing state. The dynamics simulation, which runs on a separate desktop platform, starts with the spacecraft initially tumbling while orbiting Mars. Figure 18(a) displays the attitude tracking error evolution as computed by FSW, while Fig. 18(b) displayes the control torques commanded to the reaction wheel pyramid. The FSW states computed within MicroPython have been post-processed and plotted *a posteriori* on a desktop post-processing simulation as described above.

Conclusions

Two novel strategies for migrating flight algorithm code from desktop to embeddable middleware targets have been presented. The novelty of these strategies is found in two different levels: in the transition mechanism itself and in the use-case of MicroPython. The transition out of the Basilisk desktop environment is achieved by automatically generating the integration code required to integrate the unmodified C/C++ flight algorithm code into the corresponding embedded environment –either CFS or MicroPython. The integration code, which is minimal and completely human-readable, is generated through Python's introspection capabilities while the actual source code remains unchanged. Such transition mechanism is generally applicable to any desktop testbed that leverages the use of Python

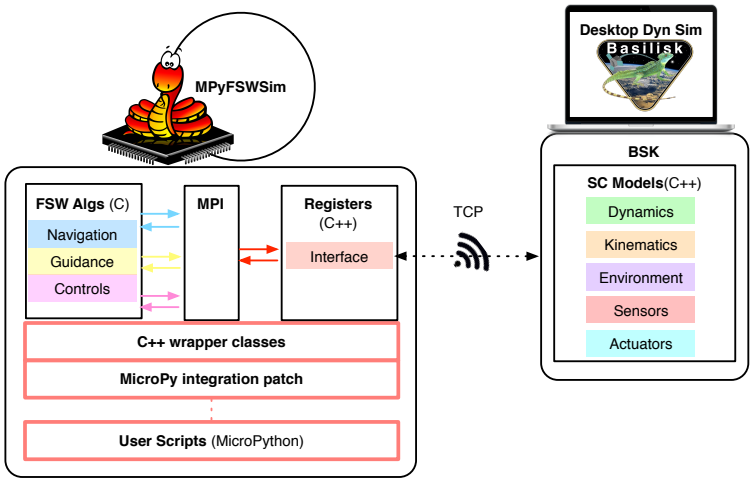
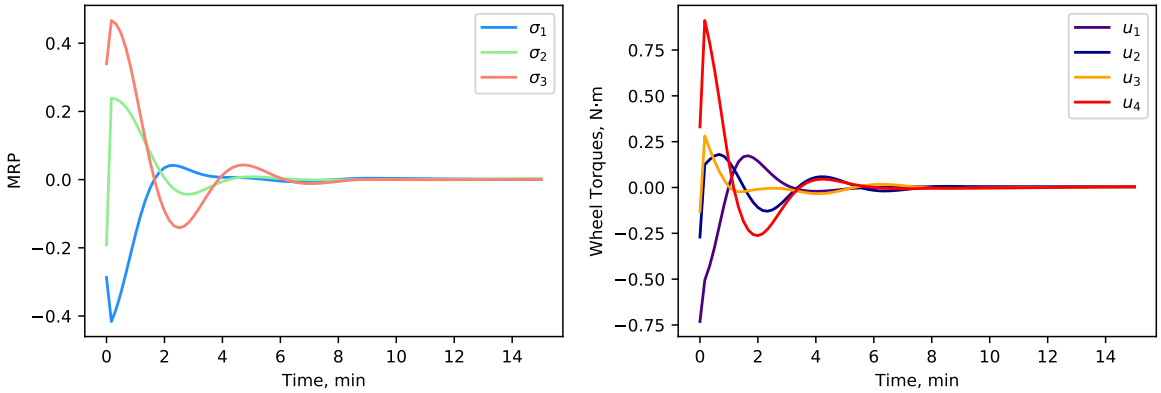


Figure 17. Closed-loop testing of MicroPython flight algorithms



(a) MRP attitude tracking error

(b) Reaction wheel commands

Figure 18. MicroPython FSW closed-loop testing in an inertial 3D pointing maneuver

as a wrapper for C/C++ flight algorithm code. The use-case of MicroPython is novel in the sense that, to the author's knowledge, it has not yet been considered as a potential middleware layer to ensure portability of the onboard FSW among different RTOS and flight processor boards.

EMBEDDED DEVELOPMENT AND TESTING IN AN EMULATED FLIGHT PROCESSOR

Flight processor emulations can be effectively used to test the on-board FSW executable, which is a complete binary image that includes: the FSW algorithms/application, the real-time operating system (RTOS), the board boot-software and the board support package. The present work uses an emulation of a LEON board to run and test FSW applications on top of the RTEMS real-time operation system. Two different embedded FSW applications are considered: a Core Flight System application and a MicroPython application.

CFS-FSW: Avionics Modeling in an Emulated Flight Processor

In an emulated flat-sat configuration, the embedded CFS-FSW application runs within a processor board emulator and interacts with external applications like the spacecraft physical simulation or a ground system model. For the interplanetary spacecraft mission in which LASP and the AVS laboratory are collaborating, the general concept of an emulated flat-sat has materialized in the setup depicted in Fig. 19. In order to access the CFS-FSW states, the FPGA registers have been modelled within the processor board emulator. Four different registers have been implemented: two input/output board registers (IOB), the solid state recorder register (SSR) and the single board computer register (SBC). Through the FPGA registers, FSW reads and writes in a hardware-like fashion that also replicates interrupts. Further, the modeling of registers has been accompanied with an expansion of spacecraft physical models, which now include complex avionics hardware.

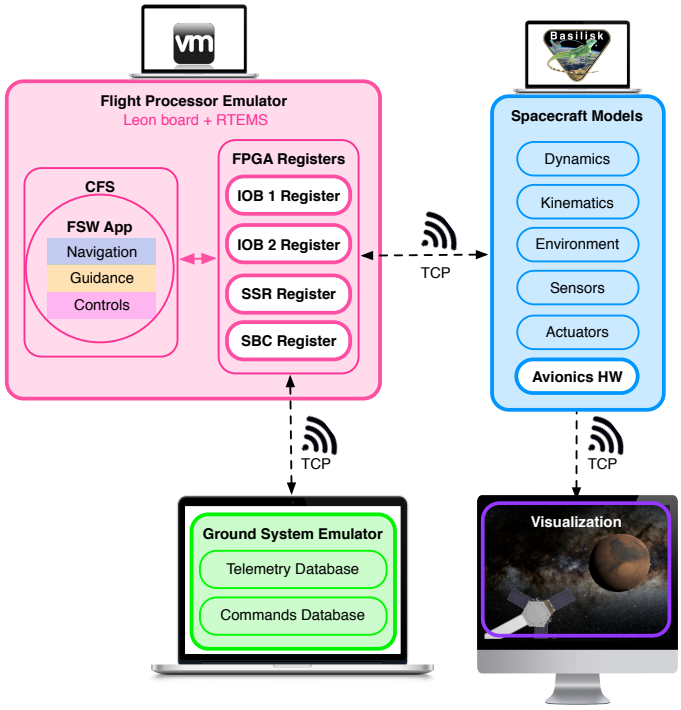


Figure 19. CFS-FSW interaction: emulated FPGA registers and avionics hardware

Regarding registers, the idea is that each of them has an associated memory buffer, and specific FSW states are mapped to specific addresses within these buffers. Not all the internal FSW states are mapped to the register's buffers,

but only those that require interaction with the external world. These shared states will be referred to as snorkels, in the sense that they are direct connection pipes to the internals of the CFS-FSW application. The different snorkels that have been implemented within the register space, as well as the specific connection of these snorkels to the external world, are illustrated in Fig. 20.

In terms of implementation, the challenge is that all these snorkels are intrinsically different from each other: some are unidirectional (they could be either reader or writers with respect to FSW) while other are bidirectional (they have both a reader and a writer associated); some operate by a packet address while others require both packet and descriptor addresses (the descriptor being a separate word that describes something important about the packet); some snorkels store single-word packets while others require queueing packets; some store fixed number of bytes while others handle variable-sized packets; a few snorkels need to remove header bytes before providing this data to FSW; and most of the snorkels are required to handle endianness. From a technical point of view, handling endianness is specially technically.

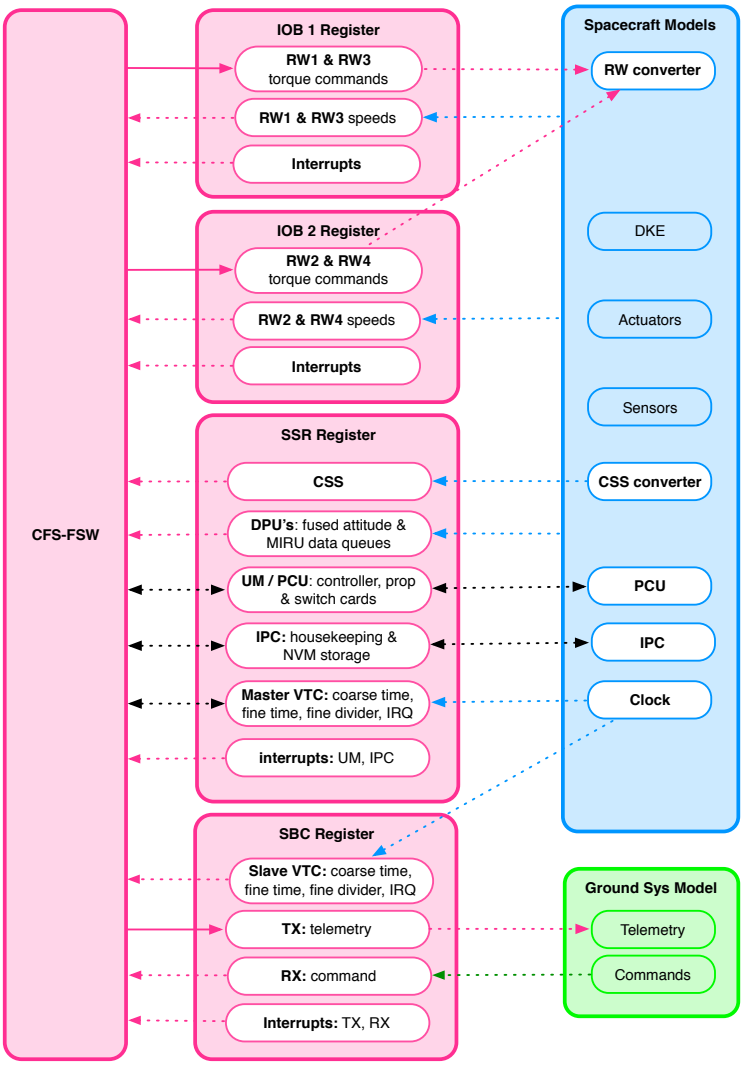


Figure 20. CFS-FSW interaction: emulated FPGA registers and avionics hardware

The avionics hardware models that have been integrated in the spacecraft physical simulation leverage complex functionality that would otherwise have to be implemented within the registers. These avionics models are: a reaction wheel (RW) converter, a coarse sun sensor (CSS) converter, a clock model, the power computing unit (PCU) and the peripheral component interconnect (PCI). The PCU and PCI are particularly complex models responsible of managing avionics cards (controller, propulsion, switch...), storing and retrieving non-volatile-memory commands, and producing house-keeping packets. While the specific snorkels and avionic hardware models described here are mission-specific, the registers space with its readers and

writers constitutes a generic framework that can be applied to any FSW application. As a matter of fact, the registers introduced previously for interaction with the MicroPython-FSW application are a simplified version of the CFS-FSW registers; the underlying register framework is actually the same.

Early numerical simulation results in Fig. 21 showcase the use of the modelled registers during embedded testing of the CFS-FSW application. The plot corresponds to the MRP attitude of the spacecraft main body frame, as simulated in the spacecraft physical simulation. During this run, the user sends four commands from the GS model. These commands are stored in the FPGA registers and picked up by the CFS-FSW application in order to reconfigure the onboard pointing mode. Sensor data from the spacecraft physical simulation is also being continuously updated within the registers. With this data, FSW estimates the spacecraft attitude, derives the associated tracking errors and computes the control torques required to drive the spacecraft into the desired attitude. The control torques are stored in the registers and sent to the spacecraft physical simulation, where a set of four reaction wheels applies the commanded control torque.

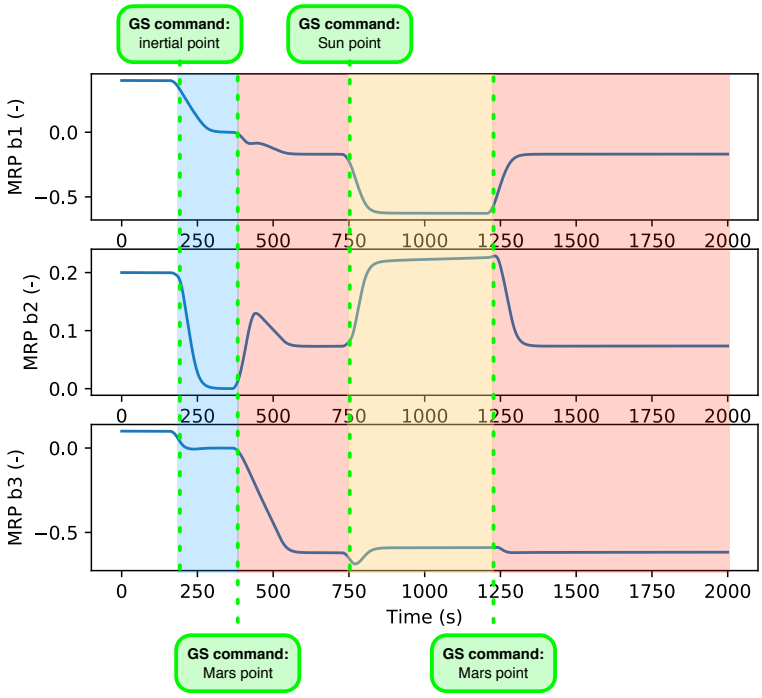


Figure 21. Closed-loop simulation: evolution of the spacecraft main body attitude

MicroPython on RTEMS

Regarding the feasibility of Basilisk-MicroPython as a light-weight and portable flight system, the first proof of concept has been implemented in Unix. Currently, MicroPython supports about 15 different ports*, available on GitHub. Some of these ports include: unix, windows, stm32, qemu-arm, bare-arm or est32. The real-world use case for the Basilisk-MicroPython combination would be to run the system in a constrained environment, on top of a real-time operation system.

Since the CFS-FSW application presented earlier is tested on an emulated LEON 3 board on top of RTEMS, using the same environment to test MicroPython would allow a direct comparison between these two middleware layers. The technical tasks required to pursue such endeavour are the following: **1)** Compile and build an RTEMS toolchain

*https://github.com/micropython/micropython/tree/master/ports

from scratch: the toolchain is conformed by the RTEMS kernel, tools-suite and source-builder and, once created, it can be used to build the board support package (BSP) for the LEON 3 target. **2)** Build an image of the Basilisk-MicroPython application: once the RTOS image, the BSP containing it and the emulated board are bundled together, the FSW executable needs to be compiled and created within this framework.

The processor board emulator considered is the open-source QEMU. Currently, QEMU-LEON is not one of the ports natively supported in the MicroPython repository. Having said that, MicroPython does support and does run on QEMU-ARM, which has a similar instruction set. As a matter of fact, MicroPython has already been ported to LEON platforms as part of an ESA project [33]. However, this extension has not been merged back into the open-source MicroPython main branch. Currently, the LEON port for MicroPython is only available under an ESA license that restricts the export to any country that is not part of the European Union.*

Conclusions

Embedded testing of a mission CFS-FSW application in an emulated flat-sat configuration has been shown. A generic framework for implementing FPGA registers and avionics hardware models has also been presented, as well as its particular application into the aforementioned mission. Such framework yields an unprecedented level of fidelity on the emulation of flat-sats. In addition, a roadmap for testing a MicroPython-FSW application within an equivalent embedded environment has been described. A successful implementation of the described roadmap would move forward the state-of-the-art of middleware layers by proving the viability of the modern MicroPython.

DISTRIBUTED CLOSE-LOOP TESTING THROUGH BLACK LION

In previous sections, the migration of flight algorithms into different types of flight targets (commercial processors and embeddable middleware systems) has been described. So far, the interaction between FSW and the external world has been described simply as a TCP connection. Nonetheless, distributed closed-loop testing of the FSW application is far more complex; integration of independent mission models into a single, distributed simulation run demands for a solid communication architecture underneath. This section describes the design and implementation of Black Lion, a purely software-based communication framework for distributed testing of heterogeneous, stand-alone mission software components.

Architecture Overview

The communication goals handled by BL encompass: **1) Transport** of binary data between nodes/components; **2) Serialization** of binary data (each node must know how to convert the received bytes into structures that can then manage internally); **3) Synchronization** (all nodes run in lock-step during a simulation run); **4) Dynamicity** in the connections map (nodes are not static, required components in the network). The goal of BL is to achieve the described communication goals while being completely abstracted from the internals of each node. In order to achieve the

*`https://essr.esa.int/project/search/microPython`

desired level of abstraction, the BL architecture has been designed as a single central controller and two APIs that are attached to each node. Such architecture is depicted in Fig. 22. The functionality of each BL interface is defined next:

BL Central Controller: it is the only static piece in the network. It acts as a broker between nodes during exchange of data and also governs synchronization through a "tick-tock" mechanism.

Delegate API: it is a generic interface that manages sockets and network connections with the central controller. The same script is attached to all the nodes.

Router API: it is a generic interface class with node-specific callbacks. Its purpose is to route data in and out of the internals of the node. Hence, it is an intermediary between the generic `Delegate` class and the specific node application.

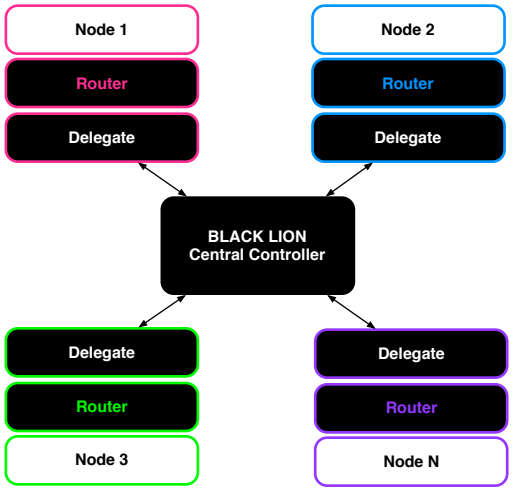


Figure 22. Communication Architecture: Central Controller, Delegate APIs and Router APIs.

Let us use a human language analogy to exemplify the functionality of the BL interfaces: Each node is an individual that speaks a different language (i.e. Spanish / French / German or, in BL, Python / C++ / C#). The `Router` acts as a translator from the individual's language to a common standardized language (i.e. metaphorically English, and hexlified byte-string in BL). If the `Router` is the translator, the `Delegate` can be seen as the communicator (i.e. the person who reads the translation out loud or, in BL, the interface who sends out the standardized data through 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. This property of the communication architecture is key to its scalability.

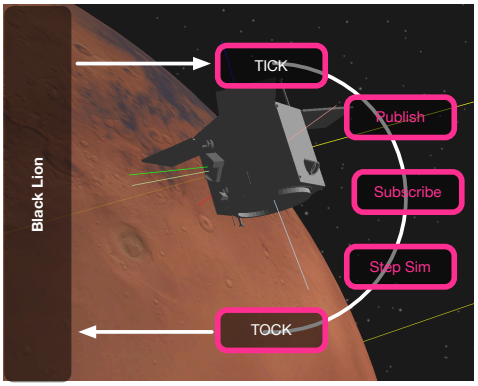


Figure 23. "Tick-tock" synchronization

Black Lion uses a "tick-tock" mechanism to maintain all the applications, which intrinsically have different execution speeds, in lock-step. The synchronization is based on a request-reply pattern where the `Central Controller` requests a "tick" and the `Delegate` of each node replies a "tock" after the node has performed its duty. Figure 23 illustrates the actions that each node performs at each time-step: **publish**, **subscribe** and **step simulation** forward.

Publishing and subscribing implies communicating with the external world in order to refresh the data before executing forward for another time-step. Once a node has finished the commanded time-step execution, it sends a "tock" reply and waits for the next request from the `Central Controller`.

Through the described architecture, BL has been able to enable communication between nodes whose heterogeneity spans from: multithreaded vs. single-threaded nodes, asynchronous vs. synchronous nodes, little-endian vs. big endian nodes, as well as for a variety of programming languages: Python, C, C++ and C#.

Conclusions

Black Lion is suitable for both distributed desktop testing as well as embedded testing. In this thesis, BL has been used for: **1)** Proving the concept of the novel Basilisk-MicroPython flight system in Unix, and **2)** Emulating an entire flat-sat in support of a spacecraft mission. Reference [32] showcases yet another application of BL: hardware-in-the-loop simulation of a constellation of satellites performing distributed, relative guidance maneuvers.

CONCLUSIONS AND FUTURE WORK TIMELINE

Figure 24 shows the Gantt chart of the work accomplished and what is left to do until the proposed graduation date, by the end of summer 2020. On a technical level, the most challenging task yet to be done is the port of MicroPython to LEON. In addition, the migration mechanisms from Basilisk to embeddable systems (recall the "AutoSetter.py" and "AutoWrapper.py") shall become an open-source resource that is available to the community. Similarly, the Black Lion communication architecture also needs to be made open-source. Going back to the motivation of this thesis, completing the described tasks would yield an end-to-end FSW development tool-suite that fills the niche identified, as shown in Fig. 25. Such tool-suite would constitute a complete framework for end-to-end FSW development that guarantees migration transparency across testbeds. It would offer the architectural flexibility to include external models for distributed, integrated closed-loop testing and, last but not least, it would be completely open-source.

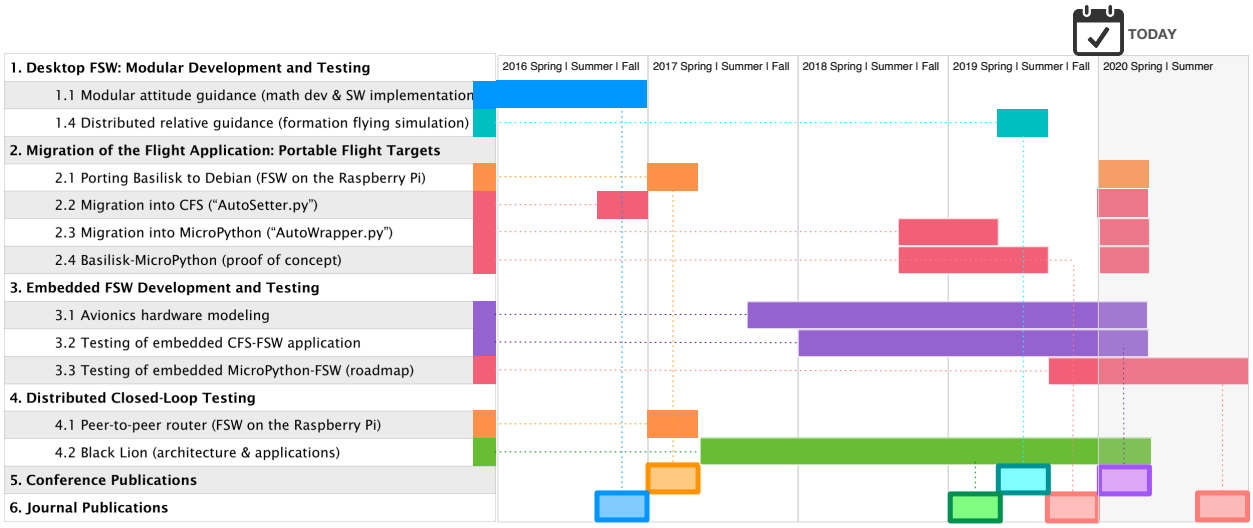


Figure 24. Gantt chart

	Completeness & transparency	External models integration	Open-source
MAX FSW & ODySSy	not transparent		
Matlab & Trick	not transparent		
Dshell & NOS	not complete		
Basilisk & Black Lion			WORK IN PROGRESS

Figure 25. Niche

REFERENCES

[1] Blanchette, S., "Giant Slayer: Will You Let Software be David to Your Goliath System?" *Journal of Aerospace Information Systems*, Vol. 13, No. 10, 2016, pp. 407–417.

[2] Rarick, H. L., Godfrey, S. H., amd R. T. Crumbley, J. C. K., and Wilf, J. M., "NASA Software Engineering Benchmarking Study," SP 2013-604, NASA, May 2013.

[3] Cols Margenet, M., Kenneally, P., Schaub, H., and Piggott, S., "Distributed Simulation of Heterogeneous Mission Subsystems through the Black Lion Framework," *AAS Journal of Astronautical Sciences*, 2019.

[4] Lauretta, D., *OSIRIS-REx Asteroid Sample-Return Mission*, Vol. Handbook of Cosmic Hazards and Planetary Defense, Springer, 2015.

[5] Mangieri, M. and Vice, J., "Kedalion: NASA's Adaptable and Agile Hardware/Software Integration and Test Lab," *AIAA SPACE*, 2011.

[6] Arregi, A. and Schriever, F., "Numerical Reproducibility for Model-Based Software-Engineering," *DASIA*, 2019.

[7] Busch, S., Bangert, P., Dombrovski, S., and Schilling, K., "UWE-3, in-orbit performance and lessons learned of a modular and flexible satellite bus for future pico-satellite formations," *Acta Astronautica*, Vol. 117, December 2015, pp. 73–89.

[8] Cols Margenet, M., Schaub, H., and Piggott, S., "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.

[9] Briggs, M., Benz, N., and Forman, D., "Simulation-Centric Model-Based Development for Spacecraft and Small Launch Vehicles," *32nd Space Symposium*, Colorado Springs, Colorado, April 11–12 2016.

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

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

[12] Cols-Margenet, M., Schaub, H., and Piggott, S., "Modular Attitude Guidance: Generating Rotational Reference Motions for Distinct Mission Profiles," *AIAA Journal of Aerospace Information Systems*, Vol. 15, No. 6, June 2018, pp. 335–352.

[13] Smith, J., Taber, W., Drain, T., Evans, S., Evans, J., Guevara, M., Schulze, W., Sunseri, R., and Wu, H.-C., "MONTE Python for Deep Space Navigation," *Proceedings of the 15th Python in Science Conference (SCIPY)*, 2016.

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

[15] George, D., Sanchez de la Llana, D., and Jorge, T., "Porting of MicroPython to Leon Platforms," Tech. rep., George Robotics Ltd. and ESA ESTEC, 2016.

[16] Keys, A., Watson, M., Frazier, D., Adams, J., Johnson, M., and Kolawa, E., "High Performance, Radiation-Hardened Electronics for Space Environments," *5th International Planetary Probes Workshop*, Bordeaux, France, June 28 2007.

[17] García-Blanco, J., Lacruz-Alcaraz, B., Santos, N., and Silveira, D., "Increasing Representativeness of SIL VV Simulators," *Dasia*, 2019.

[18] McComas, D., "NASA/GSFC' Flight Software Core Flight System," *Flight Software Workshop*, San Antonio, TX, Nov. 7–9 2012.

[19] Cudmore, A., "NASA/GSFC's Flight Software Architecture: Core Flight Executive and Core Flight System," *Flight Software Workshop*, 2011.

[20] Laroche, T., Denis, P., Parisis, P., George, D., Sanchez de la Llana, D., and Tsiodras, T., "MicroPython Virtual Machine for On Board Control Procedures," *Dasia*, 2018.

[21] Cuseo, J., "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.

[22] Penn, J. and Lin, A., "The Trick Simulation Toolkit: A NASA/Opensource Framework for Running Time Based Physics Models," *AIAA Modeling and Simulation Technologies Conference (Sci-Tech)*, San Diego, CA, 2016.

[23] Odegard, R., Henry, J., Milenkovic, Z., and Buttacoli, M., "Model-Based GNC Simulation and Flight Software Development for Orion Missions beyond LEO," *IEEE Aerospace Conference*, Big Sky, Montana, 2014.

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

[25] Grubb, M., Morris, J., Zemerick, S., and Lucas, J., "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.

[26] de Lafontaine, J., Buijs, J., Vuilleumier, P., den Braembussche, P. V., and Mellab, K., "Development of the PROBA Attitude Control and Navigation Software," 2000.

[27] Schulte, P. Z. and Spencer, D. A., "Development of an Integrated Spacecraft Guidance, Navigation, And Control Subsystem for Automated Proximity Operations," October 2014.

[28] Schaub, H. and Junkins, J. L., *Analytical Mechanics of Space Systems*, AIAA Education Series, Reston, VA, 3rd ed., 2014.

[29] Cols Margenet, M., Schaub, H., and Piggott, S., "Modular Attitude Guidance Development using the Basilisk Software Framework," *AIAA/AAS Astrodynamics Specialist Conference*, Sept. 12–15 2016.

[30] Violette, D., "Arduino/Raspberry Pi: Hobbyist Hardware and Radiation Total Dose Degradation," *EEE Parts for Small Missions*, Greenbelt, MD, September 10-11 2014.

[31] Cudmore, A., "Pi-Sat: A Low Cost Small Satellite and Distributed Spacecraft Mission System Test Platform," Tech. rep., NASA Goddard Space Flight Center, Greenbelt, MD, September 9 2015.

[32] Cols-Margenet, M., Schaub, H., and Piggott, S., "Sequentially Distributed Attitude Guidance Across A Spacecraft Formation," *International Workshop on Satellite Constellations and Formation Flying*, University of Strathclyde, Glasgow, Scotland, July 2019.

[33] Limited, G. R., "Porting of MicroPython to LEON platforms," Tech. rep., ESA contract, June 2017.