Journal of Aerospace Information Systems
Vol. 18, No. 4, April 2021

Flight Software Development, Migration, and Testing in Desktop and Embedded Environments

Mar Cols Margenet,* Hanspeter Schaub,† and Scott Piggott‡
University of Colorado Boulder, Boulder, Colorado 80309

https://doi.org/10.2514/1.I010820

This paper investigates different strategies for end-to-end flight software development that support having both desktop and embedded environments while minimizing the existing gap between them, in order to facilitate reiteration back and forth of the flight application. For desktop prototyping, the use of Python as a user-facing language wrapping C/C++ algorithm source code is considered. The Basilisk software testbed is presented as a specific incarnation of this desktop development proposal. For embedded development and testing, two different approaches are reviewed and demonstrated: the use of NASA's core Flight System, which is a well-known middleware layer, and the use of MicroPython, which is a new, lean, and efficient implementation of the Python 3 programming language optimized to run on constrained environments. The migration flow of flight algorithms from the Basilisk desktop environment into each of the considered embeddable targets is described and numerical results from embedded testing are shown. While the Basilisk–core Flight System strategy is explained through the experience of its use in an actual mission, the Basilisk-MicroPython strategy is proposed as a promising and novel strategy that is still under investigation.

Downloaded by UNIVERSITY OF COLORADO on April 14, 2021 | http://arc.aiaa.org | DOI: 10.2514/1.I010820

I. Introduction

THE complete engineering cycle to develop a flight software system encompasses an involved path of deploying and running the flight algorithms within different testbed environments. In a standard spacecraft mission, there are three distinct computing environments to consider as flight algorithm targets: desktop computer (for algorithm 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). The two latter computing environments (hardware or emulated flight processor) are considered to be embedded. Because a regular desktop environment and an embedded environment are very different (in terms of resources, capabilities, deployability, and end-user programmability among other), 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 and debugging tools and procedures that each testbed currently allows, hence potentially leaving room for a lack of testing continuity and fidelity across environments.

This paper investigates end-to-end FSW development strategies that support having both desktop and embedded environments separately while minimizing the existing gap between them (in terms of programmability, deployability, and testability), with the aim of facilitating migration back and forth of the flight application. Regarding programmability, in order to mitigate changes across environments it seems critical that the flight algorithm source code remains unmodified. The underlying idea being to stay as close as possible to the long-held NASA saying of "test what you fly, flying what you test"—since the first day of desktop development until the last day of embedded testing. Regarding mitigation of portability efforts and changes, the use of middleware is considered. As for high-fidelity and continuous testability across environments, distributed multiplatform simulations are suggested. On these lines, three FSW development proposals are presented throughout the paper: one desktop development proposal and two embedded development ones. The combination of the desktop proposal and each one of the embedded proposals constitutes an end-to-end FSW development approach by itself.

The desktop development proposal suggests the use of Python as a user-facing language for prototyping and testing flight algorithm code that is actually written in either C or C++. The idea behind this strategy is to develop algorithms directly in an embeddable programming language while leveraging Python for setting up simulation scenarios faster and analyzing results more easily. The Basilisk (https://hanspeterschaub.info/bskMain.html) software testbed is presented as a specific incarnation of this desktop development proposal. In Basilisk in particular, the flight algorithm code is written in C as per common requirement of spacecraft missions although C++ is also supported. In the recent years, the combination of Python and C/C++ for aerospace tools/applications has seen increasing interest. Examples of aerospace applications, other than Basilisk, that use the same strategy are MONTE [1] (developed by the Jet Propulsion Laboratory, JPL, for design and analysis of deep-space navigation) and Dshell [2] (a physical simulator for both robotic and spacecraft simulations also created at JPL).

Regarding embedded development, the first strategy considered in this paper suggests the use of the core Flight System (cFS) middleware and the same C flight algorithm source code as in the desktop environment. This strategy is showcased through its application into an ongoing interplanetary spacecraft mission [3]. The second embedded development proposal contemplates the replacement of cFS for the novel MicroPython (https://micropython.org), using a C++ version of the same flight algorithm as in the desktop environment. This strategy is being investigated in the context of a Ph.D. thesis [4]. All the development proposals are demonstrated and tested through distributed simulations; they 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 testing flight algorithms on it is not an easy task. However, many small-satellite missions or startup companies with limited resources and without extensive flight heritage would highly benefit from having available an easily deployable, easily testable embedded flight system. An interesting new trend in some missions is to use commercial processors in redundant configurations instead of a single radiation hardened processor [5,6]. The increasing interest on alternatives to classic radiation-hardened processors reveals the need for improvement in existing embedded flight systems. As mentioned, the use of middleware can aid portability of the flight application across different

Received 16 December 2019; revision received 14 September 2020; accepted for publication 5 January 2021; published online 22 February 2021. Copyright © 2021 by the American Institute of Aeronautics and Astronautics, Inc. All rights reserved. All requests for copying and permission to reprint should be submitted to CCC at www.copyright.com; employ the eISSN 2327-3097 to initiate your request. See also AIAA Rights and Permissions www.aiaa.org/randp.

*Graduate Student, Aerospace Engineering Sciences. Member AIAA.
†Professor, Glenn L. Murphy Chair, Aerospace Engineering Sciences. Associate Fellow AIAA.
‡Attitude Dynamics and Control Software Integrated Simulation Software Lead, Laboratory for Atmospheric and Space Physics. Member AIAA.

targets. In addition to cFS, which is an established and widely used middleware layer [7,8], different and more modern frameworks have been developed in the recent years to support FSW development and testing across environments. For instance, JPL's F Prime (F′) [9] is a software framework for rapid development and deployment of embedded applications, which is specifically designed for small-scale flight systems. In the context of robotic FSW for Mars surface exploration missions, JPL has also developed the Surface Simulation (SSim) [10], which uses actual FSW instead of a simplified model to perform rapid desktop testing. The present paper investigates yet another modern alternative for a middle ground between desktop and embedded environments: MicroPython. Although the promising MicroPython is being investigated by ESA for onboard control procedures in particular [11,12], its application as middleware is different and novel.

The paper is outlined as follows. Section II describes the general features of desktop development environments and discusses two of the most popular approaches for FSW prototyping: model-based development and Python-wrapping of C/C++ flight algorithm code. Section III explains the features of embedded environments, introduces the concept of middleware as a transition point, and describes how this paper aims to mitigate the impact of changing environments. Section IV describes the use of Basilisk for desktop FSW development. Section V describes the migration of flight algorithms from the desktop environment into cFS and showcases numerical results from testing the embedded cFS-FSW application in an emulated flat-sat. Section VI explores, in turn, the migration of flight algorithms from the desktop environment into MicroPython and provides a first proof-of-concept. Section VII concludes the paper and, at the end, there is Appendix with pseudocode for some of the tools presented.

Downloaded by UNIVERSITY OF COLORADO on April 14, 2021 | http://arc.aiaa.org | DOI: 10.2514/1.I010820

II. Desktop Development Environment

Desktop computers are the most flexible of the environments thanks to the use of state-of-the-art processors and operating systems. This flexibility is shown in terms of computing speed, memory availability, and user friendliness among others. 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 and mission-specific requirements are met. For the purposes of prototyping FSW in a desktop environment, the use of high-level scripting languages like Python or Matlab is extremely convenient as it enables rapid development and iteration. However, regular desktop scripting languages are not suitable for embedded flight applications requiring low memory footprint and bounded use of resources like CPU and RAM. For this reason, if flight algorithm source code is firstly prototyped in the desktop environment using desktop scripting languages, it is then usually translated into programming languages like Fortran, C, or C++ for migration into an embedded flight target. Note that the previous statement refers specifically to "regular desktop scripting" languages, in the sense that they are meant for general-purpose programming (like Python, Matlab, Ruby, Perl, etc.). In the field of space engineering, command sequencing languages like VML [13], PLEXIL [14], or Timeliner [15] are often also referred to as scripting languages because they express spacecraft commands using high-level concepts. However, sequencing languages are much simpler and memory lightweight yet less flexible than general-purposes scripting languages, because they have to guarantee spacecraft safety under all possible execution paths. In this paper, the term "scripting" language is used to refer specifically to the former type, i.e. general-purpose scripting languages like Python or Matlab that are used for all sorts of desktop software applications.

In the context of FSW, there are two different approaches commonly adopted for desktop development: model-based development (MBD) and Python-wrapping of underlying C/C++ code. Each of these approaches is discussed in the next subsections. Be aware that this discussion is limited to desktop development, and the migration into actual flight targets is treated in later sections of the paper.

A. Model-Based Development

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 (https://www.mathworks.com/products/simulink.html) and National Instruments LabVIEW (http://www.ni.com/en-us/shop/labview.html). Next, an automated source code generation software tool is used to translate the graphic design into programming source code. This step is often known as autocoding. The MBD process is depicted in Fig. 1. In spite of its convenience, MBD introduces another step in the flow of flight algorithms between environments that adds on into the continuity problem: FSW validity from model-in-the-loop (MIL) simulations to software-in-the-loop (SIL) simulations cannot be readily inferred without further testing [16].

Ultimately, it is, of course, on each mission to decide whether MIL-to-SIL transition requires dedicated validation steps or not. Having said that, autocoding undeniably implies a complete change (or rather a complete generation) of source code and, as pointed in [16], a change in mathematical libraries as well. Additional challenges with automatically generated code are that 1) it can be 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 lack of readability [17]. With this in mind, for all those FSW mission groups who might be concerned about transparency in the generation of the source code that ultimately gets deployed on the target spacecraft environment, an alternative and popular strategy is described next.

B. Python Interface with Underlying C/C++

An alternative to MBD is to use Python for wrapping C/C++ source code. This approach is inspired on the internal workings of the Python language itself: built-in modules that require speed, like Numpy (https://numpy.org), are actually written in C/C++ and then wrapped into Python using Python-language bindings. As a matter of fact, there are several ways to extend the Python language with custom C/C++ modules. Although CPython is the native way of extending Python with custom C/C++ modules, there are also high-level and easy-to-use libraries like Simplified Wrapper and Interface Generator (SWIG; http://swig.org) that handle this extension. Using the same logic, Python could serve as an excellent testbed for FSW development if the flight algorithm code is written exclusively in C/C++ and then wrapped into Python for simulation setup and analysis of results.

Such development proposal is depicted in Fig. 2. The advantage of this approach is that that there is no MIL development and, from a testing perspective, the transition from MIL to SIL is skipped. Although this improvement in continuity comes at the expense of developers writing the algorithm source code directly in C/C++, testing and postprocessing can be done entirely in Python, hence taking advantage of built-in libraries and other optimized mechanisms that scripting languages provide for these very specific purposes. Regarding migration into the embedded target, the C/C++ source code remains unmodified and the Python portion in Fig. 2 is simply removed. The transition from SIL to hardware-in-the-loop

Fig. 1 Model-based development: from model in the loop, through software, to hardware.

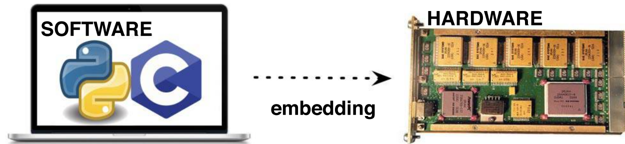


Fig. 2 Python wrapper with underlying C/C++ code: from software in the loop to hardware.

(HWIL), which requires dedicated testing, is discussed in later sections of this paper.

As far as desktop development is concerned, the point of the Python-wrapping C/C++ approach is to take advantage of two extremely powerful and different languages, exploiting each one for what it is best at. Specific advantages of using Python as wrapper and user-facing language are that it presents an especially clean and straightforward syntax (which leads to faster development and less cognitive load), it has a very large standard library, and, because Python is generally not compiled, Python interpreters are great for rapid testing and exploration. In turn, some advantages of having underlying C++ source code are that the runtime performance is better and more predictable and, because it is a low-level language, it can target just about every known platform, including embedded systems.

III. Embedded Development Environment

Time-critical applications like those of FSW usually demand the use of onboard processors with drastically fewer resources than a typical desktop computer. Therefore, FSW systems are said to be constrained or embedded. Embedded environments are, in essence, electronic systems that are managed by a microprocessor (like a hardware flight processor) or a microcontroller that operates the whole system with precise timing. Embedded flight processor environments are defined by the selection of two items: the microprocessor board and the real-time operating system.

When programmed appropriately, a real-time system can guarantee that tasks consistently execute in a specified time constraint. Determinism is, precisely, the characteristic that describes how consistently a system executes tasks within a time constraint. A perfectly deterministic system would experience no variation in timing for tasks. Typical flight systems demand determinism in both operations and CPU cycles. In addition, they present reduced memory availability (RAM/ROM).

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 [18]. Radiation hardening of processors is important in order to ensure their uninterrupted operation over long durations in the harsh space environment. Figure 3 shows several radiation-hardened processors commonly used for space exploration (RAD750, ColdFire, LEON, etc.), all of them being very expensive and presenting similar limitations in performance.

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, tending to be mission specific. An alternative target for flight algorithms is a middleware layer, which is described in the next subsection.

A. Middleware

Middleware can be regarded as an abstraction layer or “glue code” that ensures portability of the flight algorithms among different processors and RTOS. An example of middleware is the cFS, which is an open-source product provided by NASA Goddard Spaceflight Center [7,8]. Although targeting middleware can be worthwhile in the long run to ensure portability of the flight application, small missions do not tend to follow this approach given the complexity and steeper learning curve of the work entailed [17]. However, if a user-friendly, easily deployable middleware layer existed, the number of missions embracing reusability through middleware would most likely increase.

Recently, a lean (i.e., memory lightweight), efficient (i.e., with fast execution), and highly portable implementation of the Python 3 programming language has been developed. This new implementation is named “MicroPython” and it is very compelling for use in embedded FSW systems as it includes a small subset of the Python standard library, and it is optimized to run on microcontrollers and in constrained environments. The difference between MicroPython and conventional programming languages is that it provides many advanced features (characteristic of scripting languages) while having little memory footprint and being extremely compact (characteristic of compiled programming languages). Regarding MicroPython’s portability, it currently supports about 15 different ports available on GitHub (https://github.com/micropython/micropython/tree/master/ports). Some of these ports include *unix*, *windows*, *stm32*, *qemu-arm*, *bare-arm*, and *est32*. Supporting both 32-bit and 64-bit platforms, MicroPython’s potential as a middleware layer is very compelling.

B. Impact of Changing the Target Environment

While FSW migration from desktop environments to embedded flight targets can be facilitated by the use of middleware, a change in

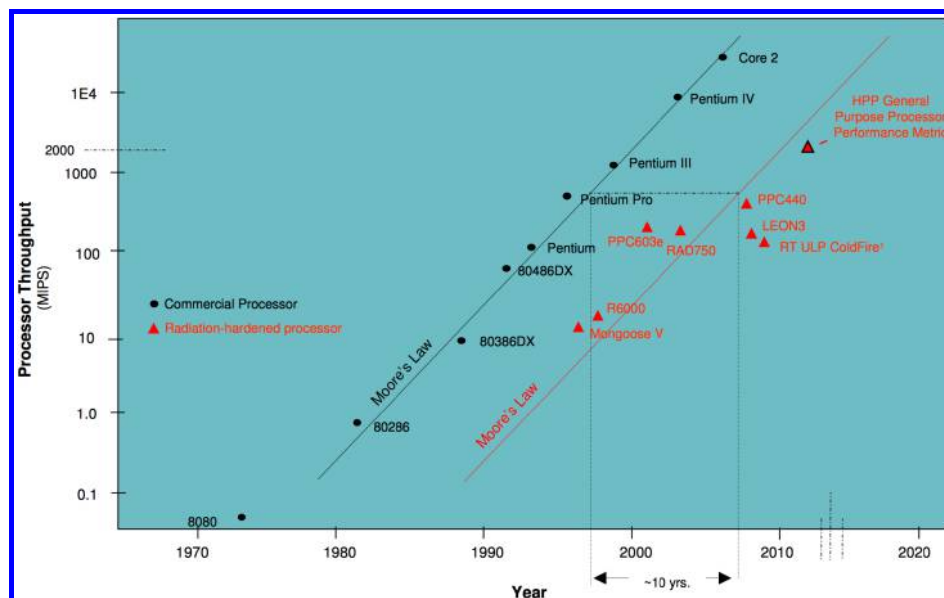


Fig. 3 Radiation-hardened microprocessors. Image extracted from [18].

Downloaded by UNIVERSITY OF COLORADO on April 14, 2021 | http://arc.aiaa.org | DOI: 10.2514/1.I010820

computing environments always requires dedicated testing, regardless of the chosen approach for desktop development. Having said that, the strategies presented in this paper aim to mitigate the impact of changing targets by avoiding introduction of nonessential modifications, which is achieved by following these guidelines:

1) Use the same source code for flight algorithms and mathematical libraries in both environments. Note that this is the rationale behind the desktop development proposal of using Python for wrapping C/C++ instead of using a model-based approach.

2) Use middleware to avoid the need of developing target-specific code and to ensure portability.

3) Use distributed simulations such that external models (like the spacecraft physical simulation against which flight algorithms are tested) always run in the same platform, being then FSW the only component of the simulation that changes environments.

4) Match test scenarios (e.g., launch separation, Mars orbit insertion, and science maneuvers) in both environments.

Without further sources of change and/or uncertainty other than the FSW migration itself, numerical results from runs in the desktop environment can be compared with their counterparts in the target environment. Such comparison of runs can be found, for example, in Ref. [6], which considers a Raspberry Pi as the target hardware.

For the purposes of embedded testing, this paper also suggests the use of flight processor emulations instead of their hardware counterpart. An example of a processor board emulator is the open-source QEMU (http://qemu.org). The advantage of using an emulation is that it provides pure software substitutions for an expensive and limited piece of hardware and, in this way, it allows simultaneous testing among different mission groups [19–21]. Another advantage of an emulation is that it can accommodate for times when FSW asleep in a simulation scenario.

The complete FSW development cycle, according to the described guidelines and using an emulated board for embedded testing, is depicted in Fig. 4. In this paper, the Basilisk software is applied for FSW development in the desktop computer environment, and two different middleware tools, cFS and MicroPython, are analyzed. The focus of the paper is on the flight algorithm migration from the desktop computer into the target middleware. Numerical simulations testing FSW on middleware running on an emulated flight processor are shown but, for full details on embedded FSW testing, the reader is referred to Ref. [4]. With this in mind, the next section in this paper provides an introduction to the Basilisk framework.

IV. Desktop Development Through Baslisk

The desktop FSW development proposal being suggested in this paper encompasses the use of Python as a user-facing language for prototyping and testing flight algorithm code that is actually written in C/C++. The Basilisk software testbed is presented, next, as a specific incarnation of this proposal.

Basilisk is an open-source, cross-platform, desktop testbed for designing flight algorithms and testing them in closed-loop dynamics simulations. The Basilisk testbed is 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 order to support an interplanetary spacecraft mission.

Basilisk is architected in a modular and highly reconfigurable fashion using C++ modules that perform spacecraft physical simulation tasks and C modules that perform mission-specific GN&C tasks. The SWIG library is used to wrap the C/C++ modules and make them available at the Python layer for three purposes:

1) Setup of C/C++ algorithms for specific simulation scenarios

2) Desktop execution of the simulation scenarios (i.e., running the main control loops)

3) Postprocessing of simulation results

Some of the advantages of using Python as the user-facing interface are ease of data analysis (which is comfortably leveraged through built-in libraries like Numpy, Matplotlib, and PANDAS, among others), capability of automated regression tests (via py-test), and rapid Monte Carlo handling.

Figure 5 illustrates the nominal (but not necessarily required) layout of a Basilisk scenario. This layout is composed of two independent processes: an FSW process and a spacecraft physical simulation process. During a simulation run, the C and C++ modules from the different processes communicate with each other through a message passing interface based on a publish–subscribe pattern. The beauty of using a message interface is that it delineates a very clean separation between the different processes. This separation facilitates, later on, the migration of the FSW application into a

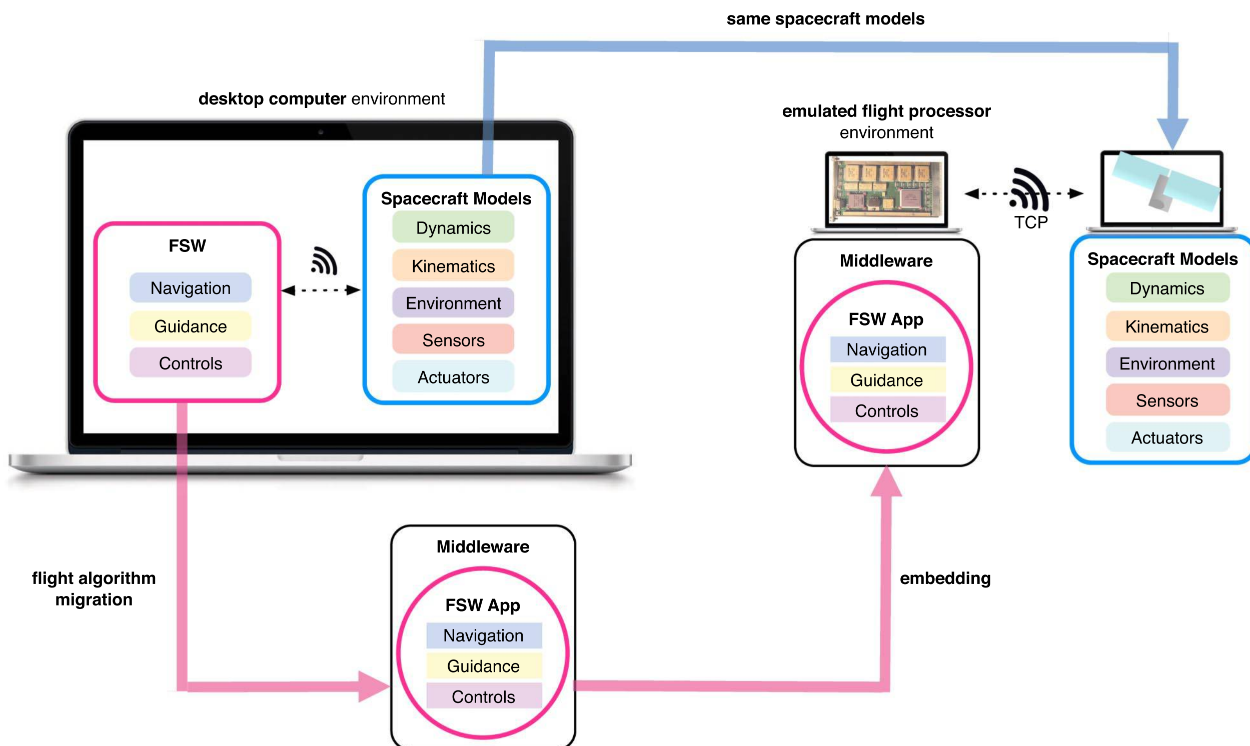


Fig. 4 FSW development cycle.

Downloaded by UNIVERSITY OF COLORADO on April 14, 2021 | http://arc.aiaa.org | DOI: 10.2514/1.I010820

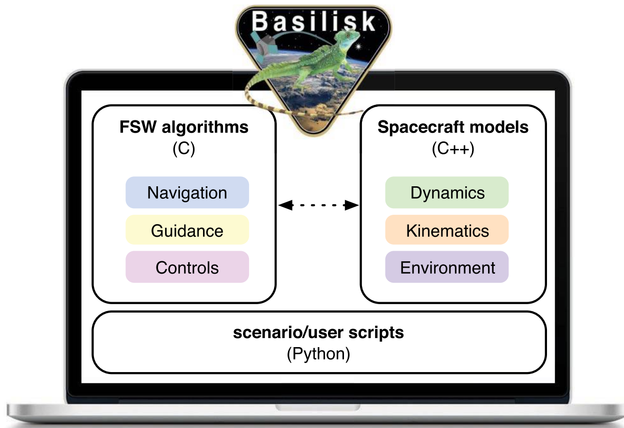
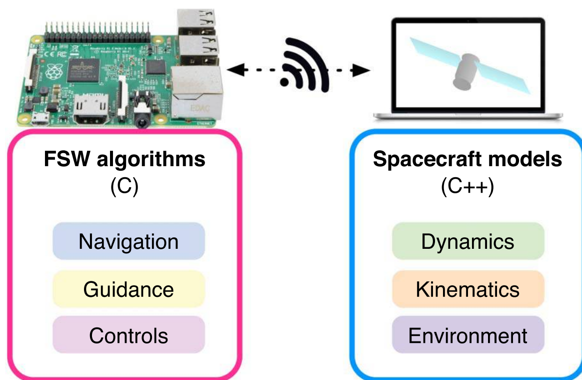
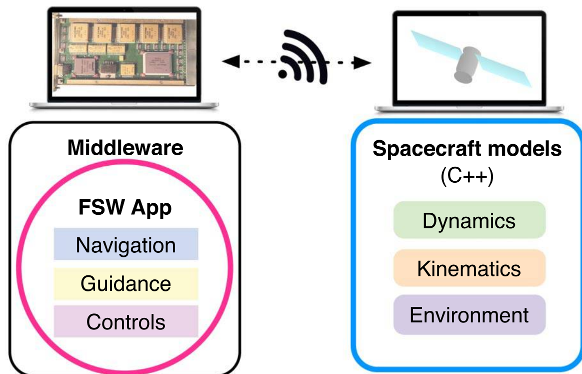


Fig. 5 Basilisk (BSK) desktop environment.



a) FSW on the Raspberry Pi: ARM processor and Linux OS



b) FSW inside cFS on an SBC (single-board computer) emulator

Fig. 6 Migration of the flight application.

different processor. Figure 6 showcases two different processor targets to which Basilisk-developed flight algorithms have been migrated.

The target processor in Fig. 6a is a Raspberry Pi, which has a built-in ARM processor and comes with the Linux OS out of the box. Since Basilisk is cross platform in nature, a regular Basilisk FSW process can seamlessly run on the Pi platform. Reference [6] showcases a numerical simulation with the setup of Fig. 6a running on soft real time. The target processor in Fig. 6b is an emulated radiation-hardened processor. Such target is currently being used for testing in the aforementioned interplanetary mission in which LASP and the AVS laboratory are collaborating. For this mission, the emulated board is a LEON microprocessor with RTEMS running on top. Since the emulated system is embedded, the Basilisk process containing the FSW algorithms cannot natively run on this system; hence, the flight algorithms are firstly integrated into a cFS application that is actually embeddable. The next section in this paper describes the details behind integrating Basilisk-developed flight algorithms into a cFS application that is then tested in an emulation of a LEON board.

V. Embedded Development Through cFS

First and foremost, let us provide some more insight on the cFS itself. The cFS is a middleware layer that ensures portability of a flight application among different RTOS and processor boards. It is an open-source product by NASA Goddard that has inherited software from flight missions for over 20 years, and it is written mostly in C.

The architectural design of cFS is depicted in Fig. 7. Starting from the highest level of the architecture to the lowest, first, there is the application layer, which is where the mission-specific flight algorithms reside; therefore this layer is always customized by the user. Below, there is a library layer, where common components that are typically part of an FSW system are available for sharing and reuse (e.g., file delivery protocol, checksum, and housekeeping). In the middle, there is the core Flight Executive (cFE) layer, which is the central piece of cFS and provides five core services: executive, event, software bus, table, and time services. One level lower, there is the platform and OS abstraction layer, which are the key pieces enabling portability. The very bottom is where the RTOS/processor boot software resides.

Because cFS it is mostly written in C, a mission that decides to use this middleware layer needs to implement its FSW application in C as well. Therefore, if in the desktop environment the FSW algorithms and their setup are written in a combination of C and Python, the Python portion has to be removed (or rather translated) for migration. The next section introduces a novel mechanism to facilitate the required translation. This mechanism is generally applicable to any desktop testbed that, similar to Basilisk, leverages the use of Python as a wrapper for C/C++ flight algorithm code.

A. Flight Algorithm Migration into a cFS Application

To understand the process of migrating Basilisk-developed flight algorithms into a cFS application, it is necessary to look back at the desktop development proposal of using Python for setup, desktop

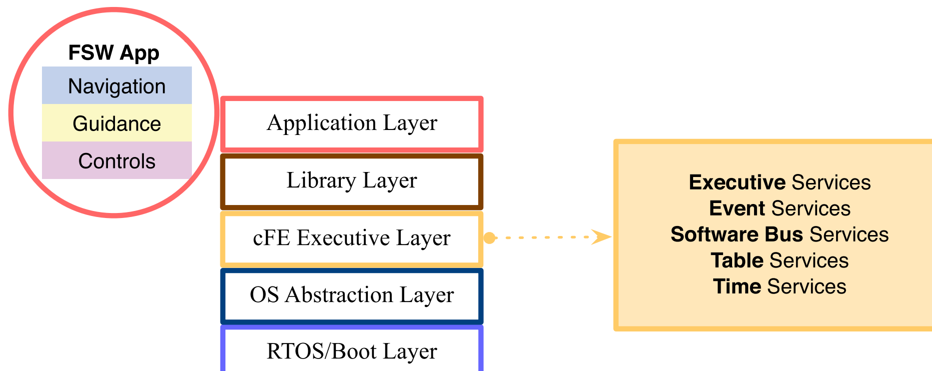


Fig. 7 Architecture of the core flight system.

Downloaded by UNIVERSITY OF COLORADO on April 14, 2021 | http://arc.aiaa.org | DOI: 10.2514/1.I010820

execution, and postprocessing of underlying C/C++ flight algorithms. There is one of these Python functionalities that need to be translated into C for migration: the setup for the C flight algorithms. Once the flight application is all written in C, it can be readily integrated within cFS. The next question to be answered is what "setup" means exactly. 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. These two setup items are further explained next.

1) *C module initialization:* Each Basilisk C-module is a standalone model or self-contained piece of logic. In the context of FSW, a module could be a specific navigation filter, a control law, a torque-to-voltage converter, or, simply, a container for static vehicle configuration data. All Basilisk C modules are characterized for having a C configuration struct and four main methods operating on the defined struct. In functionality, these main methods are common to all modules, and they perform module self-initialization, cross-initialization, update, and reset. These generic functions are externally called from Python during desktop execution. Listing 1 shows a snippet of code from a very simple module, the vehicle configuration one. This module simply contains static data of the spacecraft vehicle, like inertia and center of mass.

Listing 1: C module source code (vehicleConfigSource.h)

```
// Configuration struct
typedef struct{
    double ISCPntB_B[9]; // inertia
    double CoM_B[3]; // center of mass
    char outputMsgName[MAX_LENGHT]  // unique name for the output message
}VehicleConfigStruct;
// Main algorithms
void SelfInit_vehConfig(VehicleConfigStruct *data, ...);
void CrossInit_vehConfig(VehicleConfigStruct *data, ...);
void Update_vehConfig(VehicleConfigStruct *data, ...);
void Reset_vehConfig(VehicleConfigStruct *data, ...);
```

In the desktop environment, SWIG automatically handles the conversion of types from C and C++ into Python. A complete list of the C and C++ features that can be converted is found in the official SWIG webpage (http://swig.org/compare.html). Initializing the C and C++ variables of all the modules in Python is handy because it makes the simulation completely reconfigurable: changing the initialization values from Python does not force recompilation of the C code again. This principle, which consist on decoupling the high-level software functionality from the low-level implementation, is also know as the principle of dependency inversion in object-oriented programming. A snippet of Python code initializing the C vehicle configuration module is shown in Listing 2. While, in the desktop environment, the module variables are initialized through Python, in the cFS environment the variables are initialized with the same values through C.

Listing 2: Python setup code (for vehicle configuration module)

```
# Instantiate C config struct as a Python object
self.VehicleConfigObj = VehicleConfigStruct()
# Initialize variables
def SetVehicleConfig(self):
    # Define a unique model tag for the Python object
    self.ModelTag = "veh"
    # Initialize the C struct variables as if they were Python object variables
    self.VehicleConfigObj.ISCPntB_B = [600.0, 0.0, 0.0,
                                        0.0, 600.0, 0.0,
                                        0.0, 0.0, 600.0]
    self.VehicleConfigObj.CoM_B = [0.0, 0.0, 1.0]
    self.VehicleConfigObj.outputMsgName = "adcs_config_data"
    return
```

Downloaded by UNIVERSITY OF COLORADO on April 14, 2021 | http://arc.aiaa.org | DOI: 10.2514/1.I010820

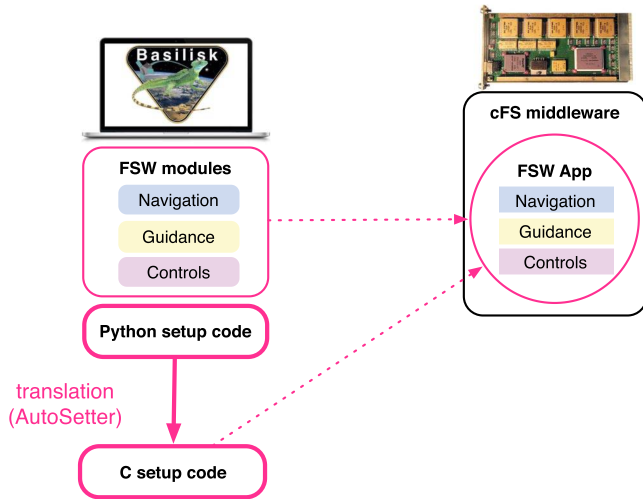


Fig. 8 Translation of setup code from Python to C.

2) *Task groups and rates:* The other setup item leveraged from Python in the desktop environment is the instantiation of C/C++ tasks that run at the defined task rates. Any number of modules can be added to a task, and calling priorities are also established from Python. In the desktop simulation, the Python code itself loops through the tasks cyclically and, for each task, calls the update method of all the modules in that task. In the embedded environment, it is desired to maintain the same task groups. Therefore, equivalent C functions, retaining the same groups and priority of tasks, have to be implemented.

Now that the kind of setup code that needs to be translated from Python to C has been explained, let us describe the interesting part: the translation mechanism. Figure 8 illustrates the conversion of the flight application from a Basilisk desktop simulation into a pure-C application that can be integrated into cFS. A key remark here is that the flight algorithm source code remains unchanged. The pure-C application is conformed by the unmodified algorithm source code plus one additional header and source file containing the setup code written in C.

The translation of the setup code from Python to C is handled automatically via an independent script written in Python: the `AutoSetter`. The beauty of the `AutoSetter`, compared with autocoding in model-based development, is that it is not a convoluted black box; rather, the `AutoSetter` is a simple template mapping Python variable types/values into their C counterparts. The resulting C setup code is minimal and completely human readable.

The workings of the `AutoSetter` essentially rely on Python's introspection capabilities. Looking at oneself is something that neither C nor C++ can accomplish without significant investment in source parsing. In contrast, Python can easily realize that, inside the FSW simulation process (written in C but wrapped in Python), there is 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, these modules now present built-in Python properties like *module*, *name*, *type()*, *dir()*, *getattr()*, and so on, which are the key to introspection.

Listing 3 shows a snippet of the C code automatically generated by the `AutoSetter`. Note that this C setup code (output of the `AutoSetter`) corresponds to the Python code shown previously in Listing 2 (input of the `AutoSetter`). Let us take a closer look, for instance, at the inertia variable (*ISCPntB_T*). In Python, the inertia is initialized as a list of nine floats, with only three of them being actually nonzero values; for the `AutoSetter` this unambiguously translates into a C array of nine doubles, with the same indices filled with nonzero values as in the Python list.

Listing 3: Sample of AutoGenerated C Setup Code

```
typedef struct{ // Struct with all FSW modules
        VehicleConfigStruct veh;
        // [...] More modules below
} AllConfig;
void AllConfig_DataInit(AllConfig *data){ // Modules initialization
        memset(data, 0x0, sizeof(AllConfig));
        // VehicleConfig module init
        data->veh.CoM[1] = 1.0;
        data->veh.ISCPntB_B[0] = 600.0;
        data->veh.ISCPntB_B[4] = 600.0;
        data->veh.ISCPntB_B[8] = 600.0;
        strcpy(data->veh.outputMsgName, "adcs_config_data");
        // [...] More modules below
}
```

Downloaded by UNIVERSITY OF COLORADO on April 14, 2021 | http://arc.aiaa.org | DOI: 10.2514/1.I010820

Downloaded by UNIVERSITY OF COLORADO on April 14, 2021 | http://arc.aiaa.org | DOI: 10.2514/1.I010820

It is worth clarifying that absolutely no naming convention is imposed on the module variables in order for the `AutoSetter` to find them and parse them appropriately. Developing this tool was a matter of investigating which Python built-in properties would provide the information required to create C code out of the SWIGed modules instantiated and initialized in the existing Python scenarios. The last section of this paper is an Appendix containing pseudocode for the `AutoSetter`. It is important to notice that this script is specifically linked to the workings of Basilisk, because this is how it was built. However, it constitutes a proof of Python's effectiveness in introspection and parsing. Any other FSW testbed that uses Python to wrap C/C++ code could use an equivalent translation approach.

A final remark is that, as seen in Listing 3, the automatically generated C code is minimal and completely human readable, allowing for rapid syntactic checking. In addition, because the generated code only involves integration and initialization, it is less likely to cause run-time failures. During the development of the generator, malfunctions and improper handling were caught by the linker and compiler when rebuilding the automatically generated code together with the unmodified flight algorithms. Further verification comes in the form of integrated tests and analysis of the closed-loop performance, which is done in the next section.

B. Embedded cFS-FSW Testing in an Emulated Flat-Sat

As illustrated earlier in Fig. 8, the unmodified FSW algorithms plus the autogenerated C setup code constitute a cFS application that is embeddable. The resulting cFS-FSW application can be embedded, for example, in an emulated flight processor and then tested in an emulated flat-sat. The flat-sat is emulated in the sense that all the different components are actually software models replicating its hardware counterparts. The concept of emulating a flat-sat configuration for the purposes of integrated testing is depicted in Fig. 9. Here, the cFS-FSW application runs within a processor board (or SBC) emulator and interacts with external applications like the spacecraft physical simulation and a ground system model. However, once FSW is integrated within cFS and embedded into the SBC emulator, enabling interaction between FSW and the external world is not simple. To achieve this communication, it has also been necessary to model several FPGA registers within the SBC emulator. These registers have been modeled as a memory map for the input and output of raw binary data. The layout of the combined cFS-FSW and modeled registers, both within the SBC emulator, is illustrated in Fig. 10.

For the aforementioned interplanetary mission (in which the Basilisk-cFS development approach has been put in practice), the general concept of an emulated flat-sat has actually materialized in the configuration depicted in Fig. 11. Figure 11 illustrates the four main components of the emulated flat-sat: flight processor emulator, ground system (GS) model, spacecraft physical models, and visualization. Note that within the flight processor emulator, there is a total of four different registers. Through these registers, FSW interacts with the external world by reading and writing in a hardware-like fashion that also replicates interrupts. For instance, it receives commands and returns telemetry (using CCSDS packets from and to the GS emulator) and, similarly, it commands the actuators in the spacecraft simulation and also receives sensor data back.

With the emulated flat-sat shown in Fig. 11, it is possible to replicate the same scenarios that are executed from Python in the

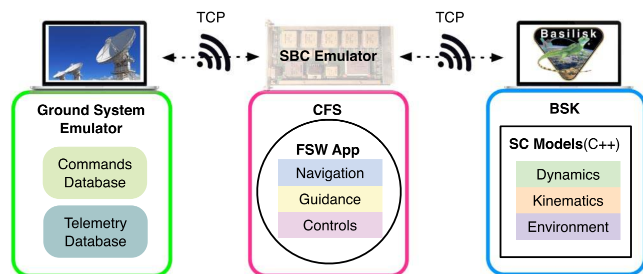


Fig. 9 Concept of emulated flat-sat.

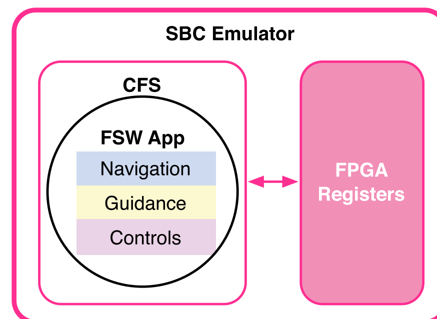


Fig. 10 FPGA register emulation.

Basilsk desktop environment. One of these scenarios is, for instance, a sequence of pointing maneuvers that happens during cruise:

1) Nav-monitoring followed by inertial pointing
2) Ephemeris correlation followed by Mars pointing
3) Switch to sun pointing
4) Back to Mars pointing

Numerical results of executing this scenario in the emulated flat-sat, where all the different components are running in a distributed fashion, are shown in Fig. 12. These plots correspond to the modified Rodrigues parameter (MRP) [22] attitude of the spacecraft main body frame, as simulated in the spacecraft physical simulation. During this run, the user sends several 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. These data are used by the navigation filters of the cFS-FSW application. In this way, FSW estimates the spacecraft's current pointing 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 back to the spacecraft physical simulation, where a set of four reaction wheels is used to apply the commanded control torque. Figure 12 shows the effect of the commands sent and proves the flow of data between the different components in the emulated flat-sat. Additional instrument-pointing plots (not included in this paper but shown in Ref. [4]) demonstrate convergence into the commanded pointing modes.

When the same FSW scenarios are executed in both desktop and embedded environments, using with the same spacecraft physical models for closed-loop testing, the impact of changing the FSW target can be analyzed by comparing the plots of the embedded run with its desktop counterpart. Because Fig. 12 shows the closed-loop behavior of the spacecraft simulation interacting with FSW, changes in the FSW performance will be reflected on the closed-loop results as well. In addition, it is also possible to compare directly FSW data by postprocessing the telemetry received in the GS system (in the embedded run) and analyzing it against the FSW results from the desktop run. An apple-to-apple comparison of a simple pointing scenario run first in the desktop environment (using a single platform for both FSW and the spacecraft physical simulation) and run next in a distributed fashion (with FSW running on the flight target) is shown in Ref. [6].

C. cFS Approach Summary

As a brief summary of this section, the presented approach for embedded FSW testing uses cFS, the FSW application is written purely in C, and, for emulated flat-sat testing, it has been necessary to emulate the FPGA registers within the processor board emulator. This approach is being applied into an actual interplanetary spacecraft mission and, so far, it has enabled efficient development, deployment, and testing across environments. The transition between desktop and embedded environments is enhanced by the use of the `AutoSetter`. In addition, emulated flat-sat testing has proved to be a very cost-effective means of performing system-wide testing early on in the mission's program, alleviating schedule constraints by using software models only. Having said that, the cFS migration approach

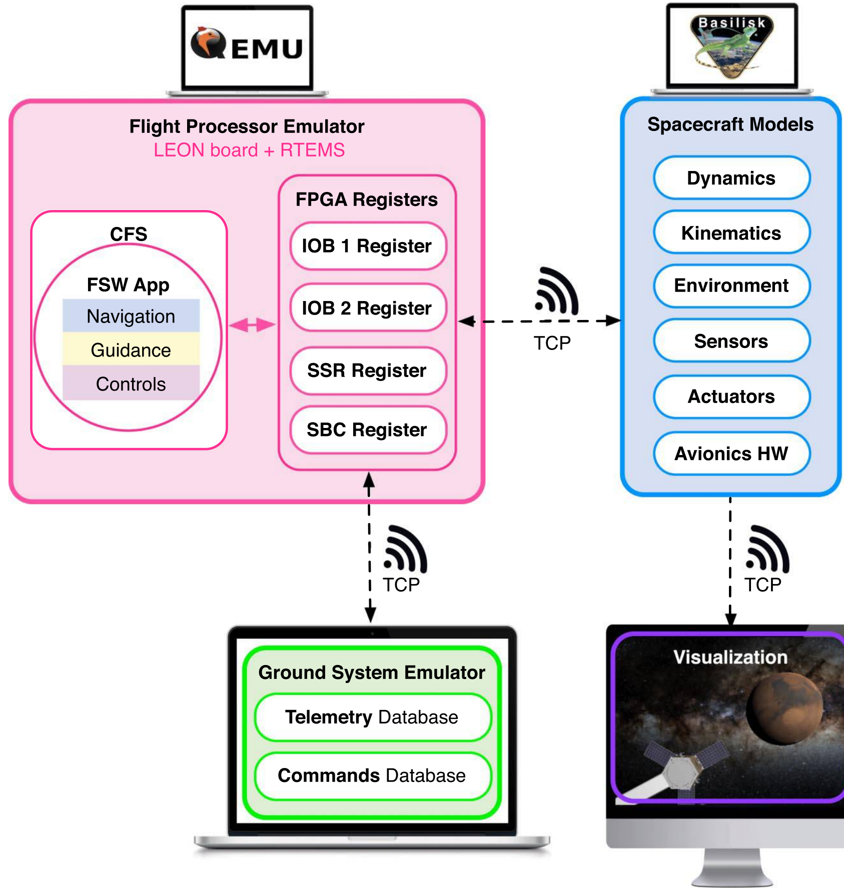


Fig. 11 Mission's emulated flat-sat configuration.

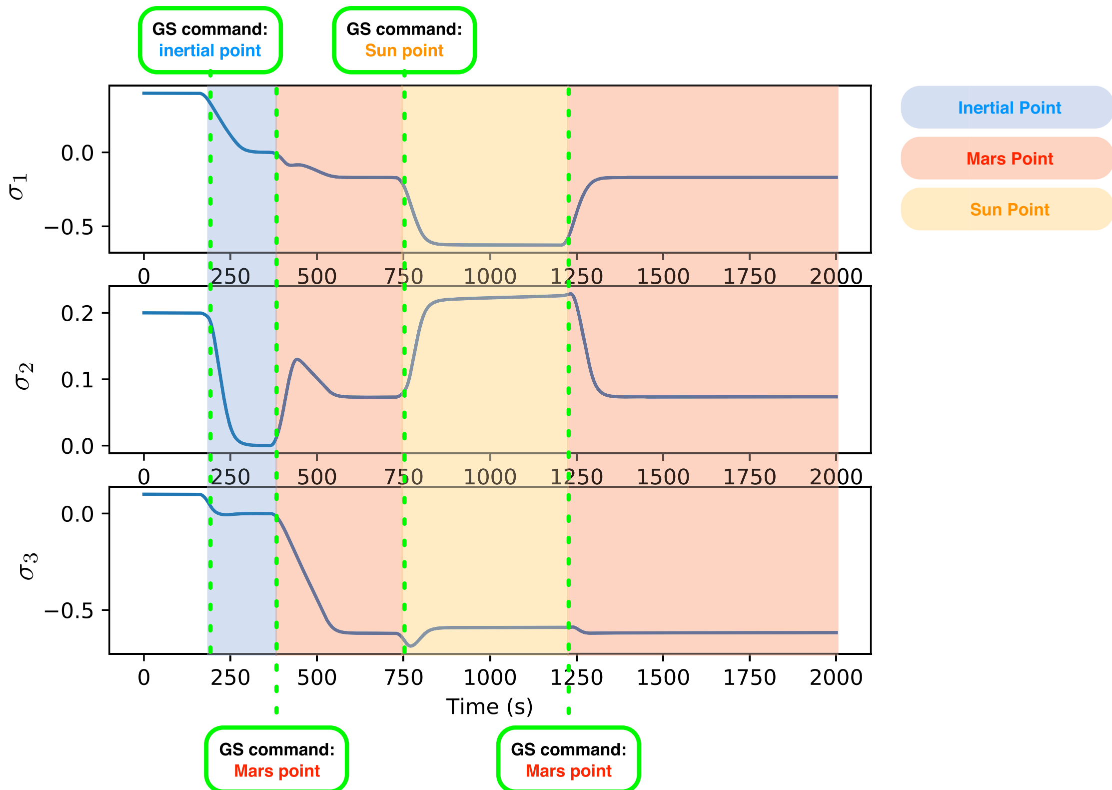


Fig. 12 Closed-loop simulation: evolution of the spacecraft main body attitude.

Downloaded by UNIVERSITY OF COLORADO on April 14, 2021 | http://arc.aiaa.org | DOI: 10.2514/1.I010820

presents three clear cons: migration effort, difficult interaction between FSW and the external world for embedded testing, and replicated cFS functionality within the flight application. These caveats are further detailed next.

Migration effort: The *AutoSetter.py* produces specific code for every single FSW configuration defined in Python. Therefore, the *AutoSetter.py* needs to run for every Python scenario/configuration that is to be tested in the embedded environment.

Difficult interaction with FSW: The modeling of FPGA registers, which is necessary to enable interaction between the embedded cFS-FSW application and the external world, is far from simple. The general idea is that each register has an associated memory buffer, and specific FSW states are mapped to specific addresses within these buffers. The challenges associated with this mapping and with the handling of FSW states are thoroughly described in Ref. [4]. Additionally, in the embedded environment of Fig. 11, it is not possible to fully capture all the cFS-FSW states but only those who are eventually sent as telemetry to the GS model.

Replicated cFS functionalities: Last but not least, cFS has revealed some inflexibilities in its design. Recalling Fig. 7, the cFE executive layer provides five core services, which cannot be removed (even if not used) or customized. For the mission application presented in this paper, examples of unused cFS functionality are services like software bus, time, and events. In addition, the fact that C++ is not natively supported within cFS also seems a limitation toward the future.

After seeing both the feasibility and the drawbacks of the cFS-FSW approach, a different and more modern tool is researched in the next section of this paper: MicroPython.

VI. Embedded Development Through MicroPython

Seeing the generalized interest in Python for desktop FSW development, it makes good sense to consider MicroPython as a middleware for embedded development. As a quick recapitulation, MicroPython is a lean and efficient implementation of the Python 3 programming language that includes a small subset of the Python standard library and that is optimized to run in microcontrollers and in other constrained environments. It presents many advanced scripting features while being compact enough to fit and run within just 256 KB of code space and 16 KB of RAM. Supporting both 32- and 64-bit architectures, the investigation of MicroPython as a middleware layer for FSW applications is very compelling. On these lines, the following section describes the migration of Basilisk-developed flight algorithms into MicroPython. Then, numerical results of a distributed simulation are shown as a first proof-of-concept.

A. Integration of FSW Modules into MicroPython

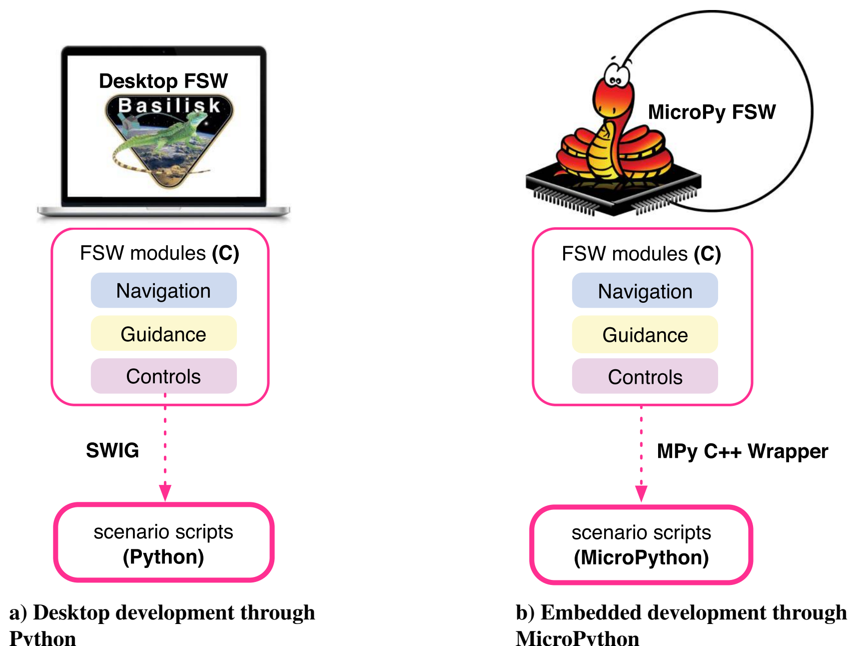
The idea proposed in this paper is to use MicroPython for embedded setup and execution of the same, unmodified C flight algorithm code as in the desktop environment. However, the standard way of extending MicroPython with custom C modules involves a lot of boilerplate code. For this reason, another open-source software tool is introduced: the MicroPython C++ Wrap (https://github.com/stinos/micropython-wrap), which is a header-only C++ library that provides some interoperability between C++ and the MicroPython programming language. The equivalence between the desktop and embedded development strategies is shown in Fig. 13. Note that, in the embedded environment, the MicroPython C++ Wrap has the same functionality as SWIG in its desktop counterpart.

Using the MicroPython C++ Wrap, the process of integrating C++ modules within MicroPython is drastically reduced. However, this comes at the cost of requiring all the modules to be written in C++ rather than in C. Recall that, currently, all the FSW modules within Basilisk are written in C. With this in mind, the technical work required to migrate Basilisk flight algorithms into MicroPython can be broken down into three tasks:

1) *Creating a C++ class for every C FSW module:* Because the MicroPython C++ wrapper is specially designed to wrap C++ code, the suggested approach for wrapping the unmodified C FSW algorithms, as they currently exist in Basilisk, is to a create a C++ 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. To achieve this behavior, it is necessary to recompile MicroPython after having declared and registered the different C++ classes, functions, and types.

3) *Adapting existing desktop Python scenarios into MicroPython:* Because MicroPython is only a light version of the Python 3 programming language, some advanced Python functionalities and large libraries (like those usually employed for postprocessing) are not supported. If this constraint is accounted for, the desktop Python scripts could, in principle, be seamlessly used within MicroPython, provided that they are written in version 3 of the language. With respect to Basilisk, however, MicroPython scripts currently import



a) Desktop development through Python

b) Embedded development through MicroPython

Fig. 13 FSW development: Python (desktop) and MicroPython (embedded).

Downloaded by UNIVERSITY OF COLORADO on April 14, 2021 | http://arc.aiaa.org | DOI: 10.2514/1.I010820

and instantiate the FSW modules differently from the Python desktop scripts: desktop Python scripts instantiate directly the C FSW modules, whereas MicroPython scripts instantiate their corresponding C++ wrapper classes. Therefore, some small adaptations are needed.

Figure 14 illustrates these three items, which are to be done. Interestingly, two of them, which are 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` produces specific C setup code for integration within cFS, an equivalent script has been developed to automate the integration within MicroPython. This new introspective script will be referred to as `AutoWrapper`. The process of migrating Basilisk FSW modules from the desktop environment into MicroPython through the `AutoWrapper` is illustrated in Fig. 15. Note that the input to the AutoWrapper is simply a desktop Python scenario script and the outputs are the corresponding C++ wrapper classes and the MicroPython integration code patch.

The last section of this paper is an Appendix containing pseudocode for the `AutoWrapper`. The AutoWrapper tool uses the same mechanism as the `AutoSetter` to figure out the variable and method names of the underlying C modules. Once introspection is granted, the code for the C++ wrapper classes and the MicroPython integration can be generated through templates. Listing 5 showcases a sample C++ wrapper class that has been automatically generated, whereas Listing 6 provides pseudocode for the Python template defining how the C++ classes are to be written.

With the layout in Fig. 14, Basilisk FSW simulations can be set up and executed from MicroPython in the same way as they are in the Python desktop environment. Recall from earlier that in the desktop prototyping environment Python is used for 1) setup, 2) desktop execution, and 3) postprocessing of the simulation results. However, MicroPython cannot handle postprocessing because it is meant to be embedded and, not surprisingly, large libraries for analysis and plotting are not supported. The question that arises immediately is how to validate the results from a MicroPython simulation run. As a matter of fact, MicroPython is capable of logging all the data from an execution run. Because the problem is about pulling and plotting such data within the constrained environment, an alternative solution 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 into the desktop Python environment for regular postprocessing. Figure 16 illustrates the suggested postprocessing mechanism through an archived binary file. The key aspect of this approach is that the postprocessing simulation (on the right) is agnostic of the archive file being created out of the Python desktop simulation or out of the MicroPython embedded simulation. Such agnosticism contributes toward a more homogenous and smooth process for cross-environment testing.

To elaborate further on how the archived binary file is structured, it is necessary to go back to Basilisk's architecture. All Basilisk simulations are instances of a C++ class that contains a Message Logger variable. In the desktop environment, the user defines from Python which module messages have to be logged and at which rate; recall that Basilisk modules communicate with each other through a publish–subscribe messaging interface. Each message has a unique name and associated ID and, when a message needs to be logged, the Message Logger creates a memory buffer for this message. This memory buffer increases over time as the simulation executes, and once the simulation is over, each message buffer can be retrieved back at the Python layer for postprocessing. In a similar way, the Message Logger can also write the message buffers into a file (i.e., the binary archive) using functions from the C++ standard library.

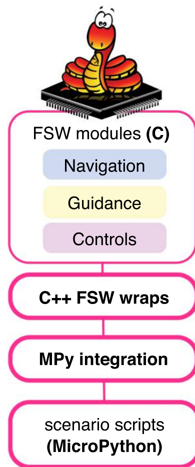


Fig. 14 Complete layout of the MicroPython-FSW application.

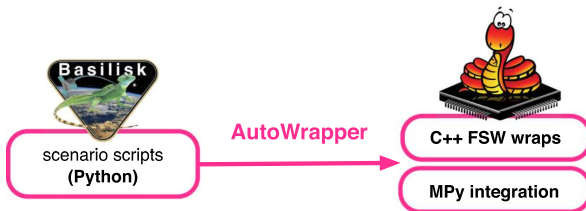


Fig. 15 Input and output of the AutoWrapper.

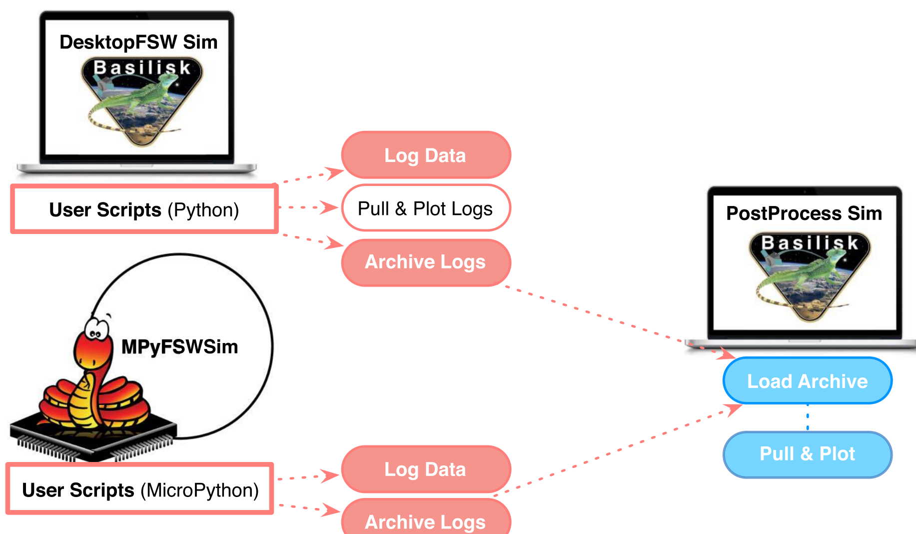


Fig. 16 Postprocessing Python (desktop) and MicroPython (embedded) execution runs.

Downloaded by UNIVERSITY OF COLORADO on April 14, 2021 | http://arc.aiaa.org | DOI: 10.2514/1.I010820

Downloaded by UNIVERSITY OF COLORADO on April 14, 2021 | http://arc.aiaa.org | DOI: 10.2514/1.I010820

B. MicroPython-FSW Testing

To prove the validity of MicroPython as an FSW target, a sample MicroPython-FSW application is tested in a distributed closed-loop with spacecraft physical models. To create this sample application, several Basilisk modules are integrated within MicroPython: a subset of the C FSW modules, the basic C++ architectural modules (e.g., process containers, task containers, messaging system), and a register-like module that enables interaction between the embedded FSW and the external world. This distributed-simulation setup is illustrated in Fig. 17.

The scenario executed corresponds to an inertial-pointing guidance maneuver. The FSW modules required for this sample scenario are the following:

Vehicle configuration module: It contains vehicle static data used by other modules.

Reaction wheel (RW) configuration module: It contains RW static data used by other modules.

Inertial pointing (guidance module): It computes an inertial 3D reference, which is conformed by an MRP attitude set, angular rate, and angular acceleration.

Attitude tracking error (guidance module): It computes the tracking error between the current state and the desired reference.

MRP feedback (controls module): It computes a 3D control torque.

Reaction-wheel motor torque (controls module): It maps the 3D control torque into individual motor torques for the RW pyramid (a set of four RWs is used in the physical simulation).

Figure 18 illustrates the numerical closed-loop behavior of the spacecraft during the inertial-pointing maneuver. The plots illustrate that the simulated spacecraft is initially tumbling and the FSW algorithms take it into an inertial-pointing state. In particular, Fig. 18a displays the attitude tracking error evolution as computed by FSW, whereas Fig. 18b illustrates the control torques commanded to the reaction wheel pyramid. After executing the distributed simulation, the FSW states computed within MicroPython are retrieved and postprocessed in the desktop environment through the archive mechanism shown in Fig. 16.

C. Potential of the MicroPython-FSW Approach

The previous section has presented an initial feasibility analysis for the use of the Basilisk flight architecture together with a MicroPython interpreter in order to yield a user-friendly, lightweight, and flexible flight operating system that can seamlessly run in desktop environments and constrained flight environments. In this first proof-of-concept, however, MicroPython is simply running on Unix. Although the real-world use case of the Basilisk-MicroPython FSW application is to run in a constrained environment on top of an RTOS, this initial technical demonstration serves to show that, by making use of the MicroPython C++ Wrap library, MicroPython can easily interface with Basilisk's C/C++ algorithm source code.

As stated earlier, common requirements of spacecraft FSW involve not only interfacing to native C and C++ code, but also a demand for real-time determinism, concurrency, and low use of resources (CPU, RAM, and ROM). Ongoing research that is not included in this paper involves, precisely, analyzing the suitability of the Basilisk-MicroPython application to run in constrained environments with limited resources and real-time constraints. The work under development

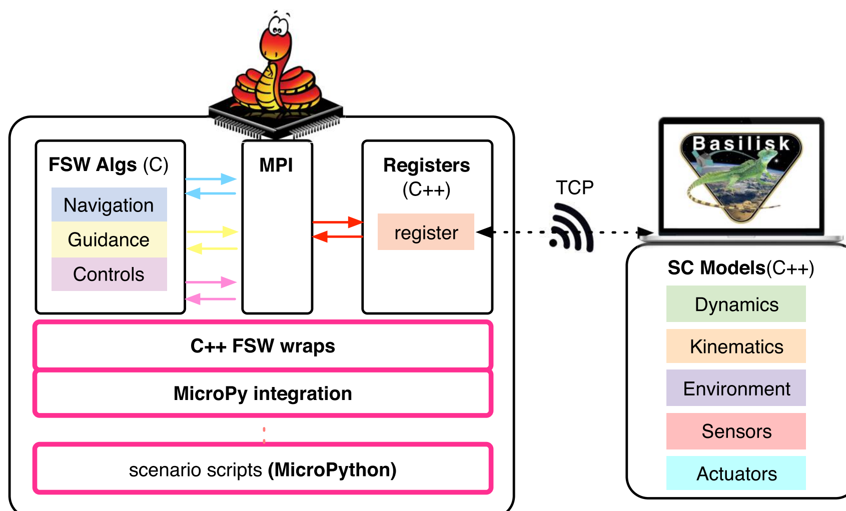
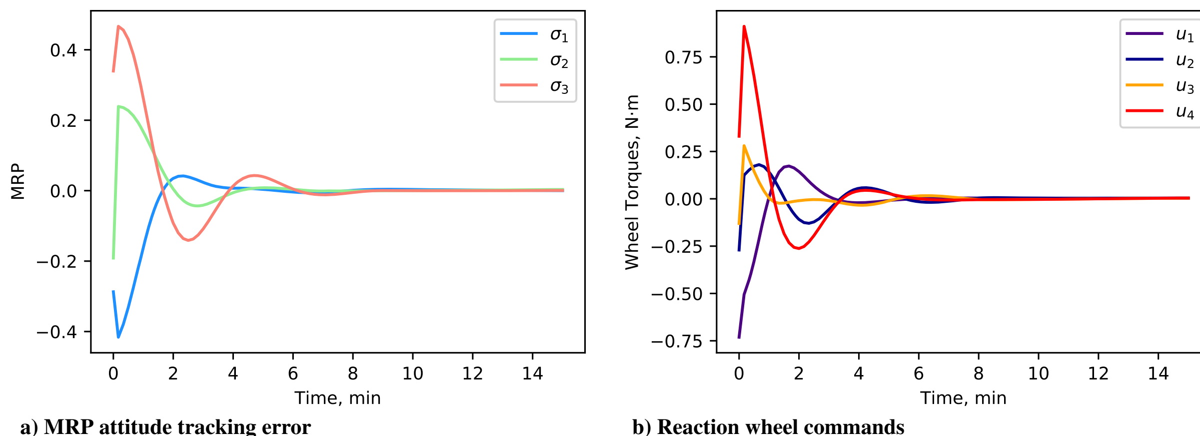


Fig. 17 Closed-loop testing of MicroPython flight algorithms.



a) MRP attitude tracking error

b) Reaction wheel commands

Fig. 18 MicroPython FSW closed-loop testing in an inertial-pointing maneuver.

Downloaded by UNIVERSITY OF COLORADO on April 14, 2021 | http://arc.aiaa.org | DOI: 10.2514/1.I010820

involves profiling the memory and CPU usage of the Basilisk-MicroPython application on Unix as well targeting the application into several 32-bit processors, like the STM32 microcontroller of the PyBoard and the family of LEON boards.

Given that the potential of MicroPython as a middleware for space applications is already showing in this first proof-of-concept, some initial comparisons between MicroPython and cFS can be drawn. Firstly, by means of MicroPython, the migration effort is reduced because the generated integration code is no longer specific but reconfigurable. The setup code generated by the `AutoSetter` for cFS integration is specific to each Python scenario script. In contrast, with the code generated by the `AutoWrapper`, all the FSW states are reconfigurable from the MicroPython layer without need of recompiling the source code again (seeing again the principle of dependency inversion). Secondly, because MicroPython has access to the messaging system of the FSW application, it is possible to fully capture all the FSW states at any point in a simulation run. Further, the modeling of the FPGA registers can be greatly simplified for the purpose of emulated flat-sat testing. The simplification of the registers is possible thanks to the fact that, for an external-world application, it is much easier to interact with MicroPython than with cFS. Last but not least, MicroPython guarantees the portability of a middleware layer without the replicated functionality imposed by cFS.

VII. Conclusions

This paper has presented two different strategies for end-to-end flight software development. Both strategies use the Basilisk testbed as a desktop development environment, but they differ on the targeted middleware: the cFS in one case and the novel MicroPython in the other.

The feasibility of the cFS approach (more conventional than the MicroPython one) is seen through the experience of its application into an interplanetary spacecraft mission. Integrating Basilisk-developed flight algorithms into an embeddable cFS application can be achieved by automatically generating a minimal set of C integration code through Python's introspection capabilities.

In turn, the feasibility of combining a lightweight version of the Basilisk flight architecture with a MicroPython interpreter is also investigated with the objective of yielding a flexible flight operating system that can directly run in constrained environments (hardware flight processor or its virtual counterpart). Although the MicroPython investigation is still ongoing research, the current results are promising. A complete implementation of this strategy would enhance the testing capabilities of flight software within constrained flight processor testbeds and would therefore minimize the gap between desktop and flight environments. Furthermore, such flight architecture would offer the same portability as a middleware layer while minimizing migration and integration costs.

Future work encompasses running the Basilisk-MicroPython flight system in a constrained environment on top of an real-time operating system. The first proof-of-concept for the feasibility of the Basilisk-MicroPython approach, as shown in this paper, has been implemented in Unix. Future work involves (but is not limited to) testing the Basilisk-MicroPython system on conventional 32-bit flight processor.

Appendix: Pseudocode for the AutoSetter and AutoWrapper Tools

A. AutoSetter Tool

Listing 4 provides the pseudocode for the `AutoSetter` tool and shows its working mechanisms: looping through the C modules of each FSW task defined in a given Python scenario and parsing the modules' main algorithms as well as their variables and values. Note that the variables have to be parsed recursively in order to handle nested structures and arrays. Listing 4 also includes comments exemplifying the parsing of the vehicle configuration module (defined earlier in Listing 1 and initialized in Listing 2). The pseudo-code provided in Listing 4 focuses, particularly, on the introspection part of the tool. Once introspection is granted, C output can be generated by defining output templates. The template strategy is shown for the AutoWrapper in Listing 5 and Listing 6.

Listing 4: Pseudocode for the AutoSetter.py

```
def main():
    #  FSW process containing GN&C tasks and modules
    sim = FSW_process()
    # Define the specific tasks to be handled by the autosetter.
    # These correspond to a subset of all the tasks created in the sim
    task_list = ["initialization", "sensor_read", "inertial_point", "feedback_control"]
    # Run the autosetter
    parse_modules(sim, task_list, output_file_name="c_setup_code")

# Look for the tasks in the sim that are also defined in the task_list.
# For each of these tasks, start looping through the modules contained in the task
def parse_modules(sim, task_list, output_file_name):
    source_file = open(output_file_name '.c', w+)  # create a C source file
    for i in range(0, len(sim.tasks)):
        task = sim.tasks[i]
```

```
        if task.name in task_list:
            for j in range(0, len(task.models)):
                model = task.models[j] # refers to the Python object wrapping the C module
                # e.g. model = Basilisk.fswAlgorithms.vehicleConfigSource.VehicleConfigStruct;
                # proxy of <Swig Object of type 'VehicleConfigStruct *' at address 0x...
                model_tag =  task.models[j].ModelTag # e.g. model_tag = "veh"
                autocode_variables(model, model_tag, source_file)
                autocode_methods(model, model_tag, source_file)

# This function looks for the specific names of the main methods in each module
# (e.g. SelfInit_vehConfig, CrossInit_vehConfig, Update_vehConfig and Reset_vehConfig)
def autocode_methods(model, model_tag, source_file):
    # Get the name of the C module underneath the Python/SWIG layers
    c_module_name = model.__module__  # Basilisk.fswAlgorithms.vehicleConfigSource
    # Create the header line to include int the AutoSetter output file

    split_names = module.split('.') # e.g. [Basilisk, fswAlgorithms, vehicleConfigSource]
    create_header(split_names) # e.g. '#include "Basilisk/fswAlgorithms/vehicleConfigSource.h"'
   # Get the name of the C structure
    c_struct_name = str(type(model).__name__) # e.g. VehicleConfigStruct
    # Get the actual C module to perform some more introspection
    system_model = sys.modules[c_module_name] # sys.modules is a Python built-in dictionary
    methods_list = dir(system_model) # contains the Python and C functions of the module
    #e.g. methods_list = ['Update_VehicleConfig, SelfInit_VehicleConfigData, __file__, ...)
    for method_name in methods_list: # parse the methods
        method_object = eval('sys.modules["' + module + '"].' + method_name)
            if type(method_object).__name__ == "SwigPyObject":
                # then you have found the name of the module-specific methods
                # e.g. "SelfInit_VehicleConfig"
   ...
   # Once the main method names in a module are known,
   # they can be used in a C template to create part of the AutoSetter's output
   ...
   return(...)
```

Downloaded by UNIVERSITY OF COLORADO on April 14, 2021 | http://arc.aiaa.org | DOI: 10.2514/1.I010820

```
# This is a recursive method evaluating which (and how) variables are to be translated into C
def autocode_variables(model, model_tag, source_file):
    # e.g. model = Basilisk.fswAlgorithms.vehicleConfigSource.VehicleConfigStruct;
    # e.g. model_tag = "veh"
    field_names = dir(module) # this gives all you the module variables (C and Python)
    # e.g. field_names = ["CoM", "ISCPntB_B", "__class__", "__del__", ..., "outputMsgName"]
    for k in range(0, len(field_names)):
        field = field_names[i] # e.g. field = "CoM"
        field_value = getattr(module, field)  # e.g. field_value = [0.0, 0.0, 1.0]
        field_type = type(field_value).__name__ # e.g. field_type = "list"
        #  Here we parse the current field and decide whether to auto-code it or not
        if <the field is a Python/SWIG built-in variable>:
            # these variables either have names starting with "__" or "this->"
            # or the field_type is "SwigPyObject" or "instancemethod"
            continue #  ignore this variable and move on to the next one
elif field_type=="class":
    # recursion:
    nested_model_name = model_tag + field_name
    autocode_model(model=field_value, model_tag=nested_model_name, ...)
elif field_type=="list" and type(field_value[0])=="class":
    # recursion: it's a list of class/struct objects
    for <each class element in the field_value list>:
        autocode_model(...)
elif field_type=="list": # numeric lists
    # Translate non-zero elements of the list into C
    for m in range(0, len(field_value)):
        if field_value[m] != 0: # e.g. for CoM, field_value[1] = 1.0
            write_val_to_source(...) # Output example: veh.CoM[1] = 1.0;
elif field_type=="str": # character array
    write_str_to_source(...)
    # Output example: strcpy(veh_model_tag.outputMsgName, "adcs_config_data");
else: # non-array type. E.g. X=2.0, letter="a", etc.
    write_val_to_source(...)
```

B. AutoWrapper Tool

As mentioned earlier, the objectives of the AutoWrapper tool are to generate a C++ wrapper class around each C FSW module and to generate the integration (or glue) code between MicroPython and the C++ classes. Listing 5 shows the C++ wrapper class that has been automatically generated for the vehicle configuration C module. The wrapper class is described next. The C++ class contains the original C struct of the vehicle configuration module as a private variable. For reference, recall that this C struct is provided earlier in Listing 1. For each member in the C struct, a setter function and a getter function are created in the C++ wrapper class. The reason for this is that the MicroPython C++ Wrapper library does not support direct interoperability between MicroPython and C++ class variables (only between MicroPython and a C++ class functions). In addition, the C++ class in Listing 5 also contains callbacks to the main four C algorithms of the vehicle configuration module. Recall that main

Downloaded by UNIVERSITY OF COLORADO on April 14, 2021 | http://arc.aiaa.org | DOI: 10.2514/1.I010820

algorithms calls are used for execution in the Python desktop environment, and they will also be used for execution in MicroPython.

The combination of the C++ wrapper class with setters and getters (Listing 5) and the MicroPython integration code (which is not shown in this paper) is equivalent to the functionality that SWIG provides out of the box for Python in a regular desktop environment. Although the functionality achieved is the same, the memory footprint with the MicroPython-wrapping approach is drastically reduced. the AutoWrapper tool uses the same mechanism as the `AutoSetter` to figure out the variable and method names of the underlying C modules. Once introspection is granted, the code for the C++ wrapper class and the MicroPython integration patch can be generated through templates. Pseudocode for the Python template describing how to generate the C++ wrapper class of a C module is provided in Listing 6.

Listing 5: AutoGenerated C++ wrapper class (vehicleConfigSource.hpp)

```
#ifndef WRAP_vehConfigData_HPP
#define WRAP_vehConfigData_HPP
#include <iostream>
#include "utilities/linearAlgebra.h"
#include "_GeneralModuleFiles/sys_model.h"
#include "vehicleConfigData/vehicleConfigData.h"
class vehClass: public SysModel {
public:
   /* Constructor: memset 0 the C struct member variable */
   vehClass(){ memset(&this->config_data, 0x0, sizeof(VehicleConfig));}
   ˜ vehClass(){return;}
   /* Callbacks to the C model's generic algorithms */
   void SelfInit(){ SelfInit_vehicleConfig(&(this->config_data), ...); }
   void CrossInit(){ CrossInit_vehicleConfig(&(this->config_data), ...); }
   void UpdateState(uint64_t callTime){ Update_vehicleConfig(&(this->config_data), ...); }
   void Reset(uint64_t callTime){ Reset_vehicleConfig(&(this->config_data), ...); }
   /* Setter and getter for the "outputPropsName" variable of the C struct */
   void Set_outputPropsName(std::string new_outputPropsName){
      memset(this->config_data.outputPropsName, '\0', sizeof(char) * MAX_STAT_MSG_LENGTH);
      strncpy(this->config_data.outputPropsName, new_outputPropsName.c_str(), ...);
   }
   std::string Get_outputPropsName() const{
      std::string local_outputPropsName(this->config_data.outputPropsName);
      return(local_outputPropsName);
   }
   /* Setter and getter for the inertia "ISCPntB_B" variable of the C struct */
   void Set_ISCPntB_B(std::vector<double>new_ISCPntB_B) {
      m33Copy(RECAST3X3 new_ISCPntB_B.data(),  RECAST3X3 this->config_data.ISCPntB_B);
   }
   std::vector<double> Get_ISCPntB_B() const {
      std::vector<double> local_ISCPntB_B(this->config_data.ISCPntB_B, ...);
      return (local_ISCPntB_B);
   }
```

Downloaded by UNIVERSITY OF COLORADO on April 14, 2021 | http://arc.aiaa.org | DOI: 10.2514/1.I010820

```
    /* Setter and getter for the center of mass "CoM_B" variable of the C struct */
    void Set_CoM_B(std::vector<double>new_CoM_B) {
        v3Copy(new_CoM_B.data(), this->config_data.CoM_B);
    }
    std::vector<double> Get_CoM_B() const {
        std::vector<double> local_CoM_B(this->config_data.CoM_B, ... );
        return (local_CoM_B);
    }
private:
    /* Define the model's C struct  as a private member variable */
    VehicleConfig config_data;
};
#endif
```

Listing 6: Python pseudocode for the C++ templates

```
class CppWrapClassTemplate(object):
    def __init__(self):
        ...
    def create_new_template(self, model_tag, header_line, c_struct_name, algs_dict, hpp_line):
        # e.g. model_tag = "veh"
        # hpp_line = '#include "vehicleConfigSource.h"'
        # c_struct_name = "VehicleConfigStruct"
        # algs_dict = ["CrossInit_vehicleConfig", "SelfInit_vehicleConfig", ...]
        self.current_model = model_tag
        # Create C++ class
        compile_def_name = 'WRAP_%s_HPP' % model_tag
        str_compile_def = '#ifndef ' + compile_def_name + '\n' + \
                          '#define ' + compile_def_name + '\n\n'
        class_name = model_tag + "Class" // e.g. class_name = "vehClass"
        str_class = 'class %s: public SysModel {\n' % class_name + \
            # Constructor
            'public: \n' + \
            '\t%s(){ memset(&this->config_data, 0x0, sizeof(%s));}\n' %
            (class_name, c_struct_name) + \
            # Destructor
            '\t~%s(){return;}\n' % class_name
        ...
        str_c_data = 'private: \n' + \
                     '\t%s config_data;' % c_struct_name
        str_end = '\n}; \n\n#endif'
```

Downloaded by UNIVERSITY OF COLORADO on April 14, 2021 | http://arc.aiaa.org | DOI: 10.2514/1.I010820

```
def add_string_property(self, field_name): #e.g. field_name = "outputPropsName"
    # Getter
    getter_str = "\tstd::string Get_%s() const{\n" % field_name + \
                 "\t\tstd::string local_%s(this->config_data.%s);\n" %
                 (field_name, field_name) + \
                 "\t\treturn(local_%s);\n" % field_name + \
                 "\t}\n"
    # Setter (...)
    return (...)
```

Downloaded by UNIVERSITY OF COLORADO on April 14, 2021 | http://arc.aiaa.org | DOI: 10.2514/1.I010820

References

[1] 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*, edited by S. Benthall, and S. Rostrup, 2016, pp. 62–68. https://doi.org/10.25080/Majora-629e541a-009

[2] 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*, AIAA Paper 2016-5421, Sept. 2016. https://doi.org/10.2514/6.2016-5421

[3] 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, Dec. 2016.

[4] Cols-Margenet, M., "End-to-End Flight Software Development and Testing: Modularity, Transparency and Scalability Across Testbeds," Ph.D. Thesis, Univ. of Colorado, Boulder, CO, Aug. 2020.

[5] 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, Dec. 2015, pp. 73–89. https://doi.org/10.1016/j.actaastro.2015.08.002

[6] 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, 2017.

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

[8] McComas, D., "NASA/GSFC' Flight Software Core Flight System," *Flight Software Workshop*, 2012.

[9] Bocchino, R., Canham, T., Watney, G., Reder, L., and Levison, J., "F Prime: An Open-Source Framework for Small-Scale Flight Software Systems," *AIAA/USU Conference on Small Satellites*, Paper SSC-18-XII-04, 2018.

[10] Vandi Verma, C. L., "SSim: NASA Mars Rover Robotics Flight Software Simulation," *IEEE Aerospace Conference*, IEEE, New York, 2019, pp. 1–11. https://doi.org/10.1109/AERO.2019.8741862

[11] George, D., Sanchez de la Llana, D., and Jorge, T., "Porting of MicroPython to Leon Platforms," *DASIA 2016—Data Systems in Aerospace*, edited by L. Ouwehand, Vol. 736, European Space Agency, Paris, Aug. 2016, p. 3.

[12] 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—Data Systems in Aerospace*, May 2018.

[13] Grasso, C., "The Fully Programmable Spacecraft: Procedural Sequencing for JPL Deep Space Missions Using VML (Virtual Machine Language)," *Proceedings, IEEE Aerospace Conference*, IEEE, New York, 2002. https://doi.org/10.1109/AERO.2002.1036829

[14] Verma, V., Jónsson, A., Pasareanu, C., and Iatauro, M., "Universal Executive and PLEXIL: Engine and Language for Robust Spacecraft Control and Operations," *AIAA Space*, AIAA, Reston, VA, Sept. 2006. https://doi.org/10.2514/6.2006-7449

[15] Busa, J., Braunstein, E., Brunet, R., Grace, R., Vu, T., Brown, R., and Dwyer, W., "Timeliner: Automating Procedures on the ISS," *SpaceOps 2002 Conference*, AIAA, Reston, VA, 2002. https://doi.org/10.2514/6.2002-T3-02

[16] Arregi, A., Schriever, F., Arias, C., and Jung, A., "Ensuring Numerical Reproducibility for Model-Based Software Engineering," *8th European Conference for Aeronautics and Aerospace Sciences (EUCASS)*, Madrid, July 2019. https://doi.org/10.13009/EUCASS2019-790

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

[18] 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*, June 2007.

[19] Cols Margenet, M., Kenneally, P. W., Schaub, H., and Piggott, S., "Simulation Of Heterogeneous Spacecraft and Mission Components Through the Black Lion Framework," *John L. Junkins Dynamical Systems Symposium*, College Station, TX, May 2018, Paper 7.

[20] Lauretta, D., *OSIRIS-REx Asteroid Sample-Return Mission*, Springer, Cham, Switzerland, 2015, pp. 543–567. https://doi.org/10.1007/978-3-319-03952-7_44

[21] Mangieri, M., and Vice, J., "Kedalion: NASA's Adaptable and Agile Hardware/Software Integration and Test Lab," *AIAA SPACE 2011 Conference & Exposition*, AIAA, Reston, VA, Sept. 2011. https://doi.org/10.2514/6.2011-7176

[22] Schaub, H.,, and Junkins, J. L., *Analytical Mechanics of Space Systems*, 4th ed., AIAA Education Series, AIAA, Reston, VA, 2018, Chap. 3.7. https://doi.org/10.2514/4.102400

M. D. Davies
Associate Editor