An End-To-End Flight Software Development Approach Using Micropython and The Basilisk Software Testbed

By Mar COLS-MARGENET,[1)] Hanspeter SCHAUB,[1)] and Scott PIGGOTT[2)]

[1)]*Autonomous Vehicle Systems Laboratory, University of Colorado Boulder, Boulder, Colorado, United States*
[2)]*Laboratory for Atmospheric and Space Physics, Boulder, Colorado, United States*

(Received July 1st, 2019)

This paper investigates novel strategies for end-to-end flight software development that support having both desktop and embedded environments while minimizing the gap between them, in order to allow reiteration back and forth of the flight application. For desktop prototyping, the use of Python as user-interface language wrapping C/C++ algorithm 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 middleware layer, and the use of the novel 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 the flight algorithms from the Basilisk desktop environment into each of the considered embeddable targets is described. The feasibility, migration effort entailed and flexibility of the Core Flight System approach and the MicroPython approach are compared and contrasted.

Key Words: Basilisk, Core Flight System, MicroPython

1. Introduction

Space missions rely highly on the efficiency and reliability of the on-board flight software (FSW) 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 may very well be three testbed environments to consider: desktop computer (for 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 environments –hardware or emulated flight processor– are considered to be embedded. Because 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 paper investigates end-to-end FSW development strategies that support having both desktop and embedded environments separately while minimizing the gap between them in order to allow migration, back and forth, of the flight application. In order for the algorithm migration to be transparent, it is critical that the source-code itself remains unchanged. – 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 all the way into the embedded environment. 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-interface language for prototyping and testing flight algorithm code that is actually written in C/C++. The Basilisk software testbed* is presented as a specific incarnation of this desktop development proposal. The first one of the embedded development proposals suggests the use of the Core Flight System† (CFS) middleware and the same C flight algorithm source code as in the desktop. The second embedded development proposal contemplates the replacement of the CFS for the novel MicroPython‡, using C++ algorithm source code.

It is important to remark that all the development proposals made in this paper 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 start-up companies with limited resources and without extensive flight software legacy, 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.[1,2)] The increasing interest on alternatives to classic radiation-hardened processors reveals the need for improvement in existing embedded flight systems.

The novelty of the work presented throughout this paper is

* `https://hanspeterschaub.info/bskMain.html`
† `https://cfs.gsfc.nasa.gov`
‡ `https://micropython.org`

found in two different levels: in the migration strategy of the flight application and in the use-case of MicroPython. The migration of the flight application refers to the transition from the Basilisk desktop environment into any of the considered embedded environments. This transition 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. The use-case of MicroPython is novel in the sense that, to the authors knowledge, has not yet been considered as a potential middleware layer to ensure portability of the onboard FSW among different RTOS and flight processor boards.

This paper is outlined as follows. First the features of a desktop development environment are described and two different methodologies for desktop FSW prototyping are discussed: model-based development and Python wrapping of C/C++ flight algorithm code. Next, the features of an embedded environment are outlined. From this point onwards, the three FSW development proposals that constitute the core of this paper are presented, and their feasibility is demonstrated. These are: **1)** desktop FSW development in the Basilisk testbed, **2)** integration of the developed flight application into CFS and **3)** integration of the developed flight application into MicroPython.

2. Desktop Development Environment

The desktop is the most flexible of the environments, thanks to the use of state-of-the-art processors and operating systems, in terms of computing speed, memory available and user friendliness, among other. 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, scripting languages are not suitable for embedded flight applications requiring absolute control of timing and deterministic behavior. For this reason, if flight algorithm source code is first prototyped in the desktop environment using scripting languages, it is then usually translated into programming languages like Fortran, C or C++ for migration into an embedded flight target. Two different desktop development approaches that are commonly adopted in the aerospace community are discussed next: *model-based development* and *Python interface with underlying C/C++ code*. The main difference between them is that, with a *model-based* approach, the source code changes for migration into an embedded system.

2.1. Model-based development

This approach focuses 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[¶]. After designing the flight algorithms, the next step is typically to select an automated source-code generation software tool that is compatible with the block-diagram modeling tools selected above and that auto-generates source code in the required programming language (aka auto-coding). Both Simulink and LabVIEW software can produce C code directly from their drag and drop environment with the use of add-on packages. Nevertheless, a problem with automatically generated code is that, usually, it is less efficient in either size or execution than optimized hand-written code. Further, it can be very challenging to edit and debug due to the lack of readability. Although some code generators incorporate their own optimization features, the challenges remain.[3)]

2.2. Python interface with underlying C/C++ 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 be run as FSW. However, the Python runtime is generally recognized as insufficiently well-controlled for time-critical applications like those required for aerospace FSW. Having said that, looking at the internals of the Python language itself reveals that most built-in modules requiring speed are actually written in C/C++ and then wrapped into Python with 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 simulation, analysis, and testing. There are several ways to extend the Python language with custom C/C++ modules. CPython is the native way of creating these bindings but there are also higher level libraries like SWIG (Simplified Wrapper and Interface Generator) that handle this extension.

The Basilisk astrodynamics framework is a desktop FSW testbed that seeks to capitalize on the potential of wrapping C/C++ dynamics simulation code and flight algorithm code through SWIG and make it available at the Python level for setup, desktop execution and post-processing. Apart from Basilisk, there are other well-known FSW tools that also use Python as the user-interface language while maintaining the source code in C/C++. For instance, in the context of spacecraft navigation, there is the Mission Analysis, Operations, and Navigation Toolkit Environment (MONTE) developed by the Jet Propulsion Laboratory to support their deep space exploration program.[4)]

The generalized interest in Python for FSW development is indeed not surprizing; this high-level language has many powerful features (like classes, list comprehension, exceptions handling...), it is open-source and has a large existing community, it is very easy to learn and still extremely powerful for advanced users and, last but not least, it presents lots of opportunities for optimization (since Python is actually compiled despite being a scripting language).[5)]

[§] `https://www.mathworks.com/products/simulink.html`
[¶] `http://www.ni.com/en-us/shop/labview.html`

3. Embedded environment

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 ensuring deterministic behaviors. A significant problem of using scripting languages or big libraries in a flight application is their memory footprint as 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 flight processor is to emulate it on a virtual machine. The advantage of using an emulation is that it provides a pure software substitution for an expensive and limited piece of hardware, therefore allowing simultaneous testing among different mission groups.[6–8)]

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.[9)] Radiation hardening of processors is important in order to ensure their 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. An example of a processor board emulator for any of the aforementioned hardware boards is the open-source QEMU[‖].

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 the flight algorithms is a middleware layer. 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 Core Flight System (CFS),[10,11)] which is an open-source product provided by NASA Goddard Spaceflight Center. While 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. 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. MicroPython is packed full of advanced features while still being compact and having little memory footprint. Currently, MicroPython supports about 15 different ports available on GitHub, among which there are: unix, windows, stm32, qemu-arm, bare-arm or est32[**]. This promising language has already captured the attention of the European Space Agency, where the use of MicroPython is being considered for onboard-control procedures in spacecraft payloads.[5,12)]

4. Desktop FSW Development: The Basilisk Testbed

The desktop FSW development proposal suggested in this paper encompasses the use of Python as a user-interface language for prototyping and testing flight algorithm code that is actually written in C/C++. The Basilisk software testbed is presented next as an incarnation of such 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, highly reconfigurable fashion using C++ modules that perform spacecraft physical simulation tasks and C modules that perform mission-specific GN&C tasks. Currently, 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**. Some of the advantages of using Python as user-interface are: ease of data analysis (which is comfortably leveraged through the use of built-in libraries like Numpy, Matplotlib and PANDAS among other), capability of automated regression tests (via py-test) and rapid Monte-Carlo handling.

Figure 1 illustrates the nominal setup – but not necessary required – of a Basilisk desktop simulation. During a simulation run, the different C and C++ modules communicate with each other through a custom message passing interface (*MPI* in Fig. 1). This MPI which is written in C/C++ and it is based on a publish-subscribe pattern. The beauty of using an MPI

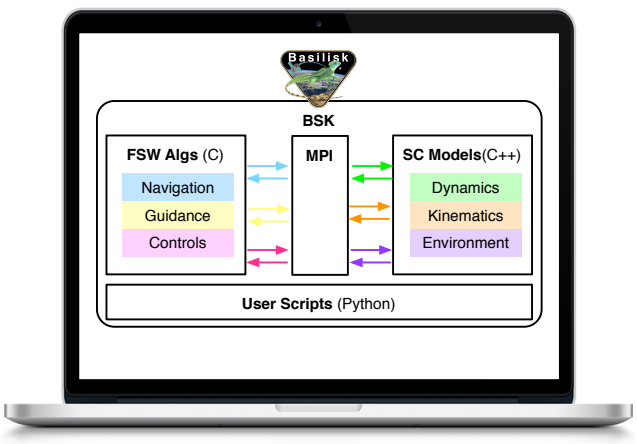


Fig. 1.: Basilisk (BSK) Desktop Environment

[‖] http://qemu.org

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

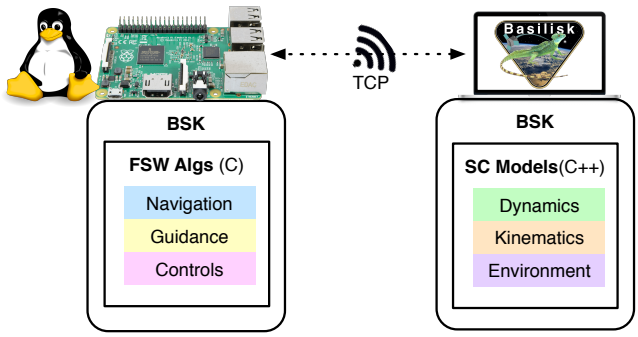
is that it allows a clear separation between the different processes – dynamics simulation process and FSW process for the setup in Fig. 1. This separation facilitates, later on, the migration of the FSW application into a different processor. Figure 2 showcases two different processor targets to which Basilisk-developed flight algorithms have been migrated.

The target processor in Fig. 2(a) 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 runs readily on the Pi platform. Reference 2 showcases a numerical simulation with the setup of Fig. 2(a) running on soft real-time. Reference 13 proposes a formation flying scenario with multiple Raspberry Pi's involved, communicating flight data to each other.

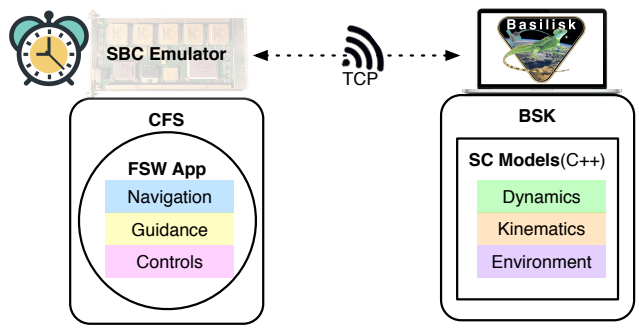
The target processor in Fig. 2(b) is an emulated radiation-hardened processor. Such target is currently being used for testing in the aforementioned interplanetary mission in which LASP and the AVS laboratory are collaborating. The emulated board is a Leon, with RTEMS running on top. Since the emulated system is embedded, the Basilisk process containing the FSW algorithms cannot run natively on this system; for this mission, the flight algorithms are first integrated into a CFS application that is actually embeddable. The next section describes the details of this integration.

5. Embedded FSW Development: The CFS

First and foremost, let us provide some more insight on the Core Flight System (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 missions over



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



(b) FSW inside CFS on an SBC emulator: emulated Leon board and RTEMS

Fig. 2.: Migration of the Flight Application

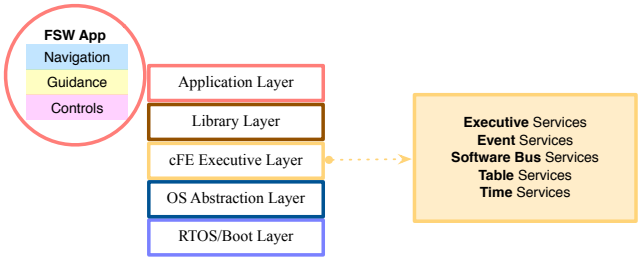


Fig. 3.: Architecture of the Core Flight System

the previous 20 years or more. The architectural design of CFS is depicted in Fig. 3.

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 –therefore this layer is customized by the user. Below there is a library layer, where common components that are typically part of a FSW system are available for sharing and reuse (e.g. file delivery protocol, checksum, housekeeping, etc.). In the middle, there is the core Flight Executive layer (cFE), 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 enables portability. At the very bottom, the mission-specific RTOS and processor boot software reside.

5.1. Flight Algorithm Migration into a CFS Application

This section explains what it takes to migrate the Basilisk-developed flight algorithms into a CFS application that is actually embeddable.

5.1.1. What needs to be translated: Python setup code

Recall the desktop FSW development proposal of using 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 is all in C and can readily be integrated within CFS. The next question is, of course, what does "setup" mean exactly? In the Basilisk framework, the Python setup encompasses: variable initialization of each individual C module and grouping of modules in tasks that run at certain task rates. These two setup items are further explained next.

C Module Initialization: Each Basilisk C-module is a stand-alone 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 "generic" method calls operating on the defined config struct –they are generic in the sense that they perform the same type of operation: module self-initialization, cross-initialization, update and reset. These "generic" functions are externally called from Python in the desktop execution. Listing 1 shows a snippet of code from a very simple module, the vehicle configuration one, which contains vehicle static data needed by other FSW modules.

Listing 1: C Module Code

```c
// Configuration struct
typedef struct{
    double ISCPntB_B[9]; // Inertia
    double CoM_B[3]; // Center of mass
    char outputMsgName[MAX_LENGHT]
}VehicleConfig;
// Generic algorithms
void SelfInit_vehConfigData(...);
void CrossInit_vehConfigData(...);
void Update_vehConfigData(...);
void Reset_vehConfigData(...);
```

Through SWIG, the C config struct of each module can be instantiated in Python as if it was a Python object with the same variables as in the struct. SWIG automatically handles the conversion of types from Python to C, including nested C structures. As a matter of fact, the authors have not yet found a C or C++ variable type that cannot be SWIG-ed. Initializing the 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 feature is specially useful to handle Monte-Carlo testing. A snippet of Python code initializing the C vehicle configuration module is shown in Listing 2.

Listing 2: Python Setup Code

```python
# Instantiate C config struct as a Python object
self.VehicleData = VehicleConfig()
# Initialize variables
def SetVehicleConfigData(self):
    self.VehicleData.ISCPntB_B = [600.0, 0.0, 0.0,
                                  0.0, 600.0, 0.0,
                                  0.0, 0.0, 600.0]
    self.VehicleData.CoM_B = [0.0, 0.0, 1.0]
    self.VehicleDatas.outputMsgName = "adcs_config_data"
    return
```

All the module variables that in the desktop environment are initialized from Python, in the CFS embedded environment must be initialized directly in C. Further, in order to keep consistency in the testing throughout both environments, the values of these variables need to match exactly.

Task Groups and Rates: The other setup-item leveraged from Python in the desktop environment is the instantiation of C++ task containers that run at the defined task rates. Any number of modules can be added to each task created, and priorities between the modules within the same task are also set by the user from Python.

Figure. 4 illustrates a simple Python setup of tasks within a FSW process (*Config Init Task* running at 0 Hz because it is a one-time initialization task, and *Sensor Read Task* running at 1Hz). In the desktop simulation, Python itself loops through the C++ tasks cyclically and, for each task, calls the update method of all the modules in that task. Relating back to Fig. 4, all the modules that belong to *Sensor Read Task* will be updated every 1 second in the simulation.

In the embedded environment, it is desired to maintain the same task groups. Therefore, C calls will need to be implemented that, for each task, contain callbacks to all the methods of the modules belonging to the task. These C callbacks, of course, shall respect the module priority established previously in the Python setup. The reason why the C++ tasks created from Python in the desktop environment cannot be preserved is because CFS does not natively support C++, which is a limitation.

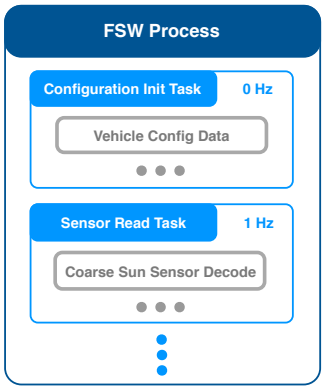


Fig. 4.: Python setup: task groups and rates

5.1.2. How the setup code is translated: the *AutoSetter.py*

As shown in Fig. 5, the setup code of the desktop Python scripts needs to be translated into C. A key remark here is that the flight algorithm source code (*FSW Algs* in Fig. 5) remains unchanged. The pure-C application (*FSW App* in Fig. 5) 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.

The translation of setup code from Python to C is 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 a very simple python template that maps Python variable types/values into their C counterparts. The resulting C setup code is straightforward and completely human readable. In the end, as long as recursion is handled properly, C variables always boil down to the same types –and there are not that many ways to initialize, for example, an array of 3 doubles.

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 easily realize that, inside the C/C++ desktop simulation wrapped in Python, there is a FSW process with a list of tasks. And inside

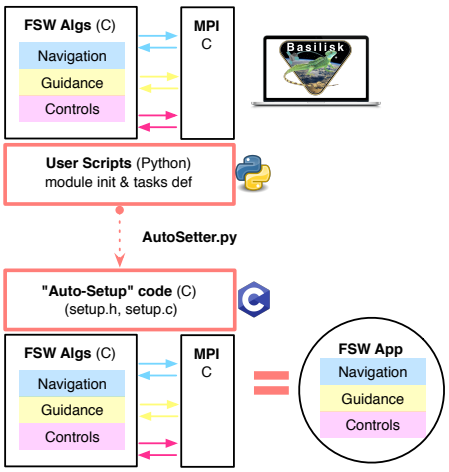


Fig. 5.: Translation of setup code from Python to C

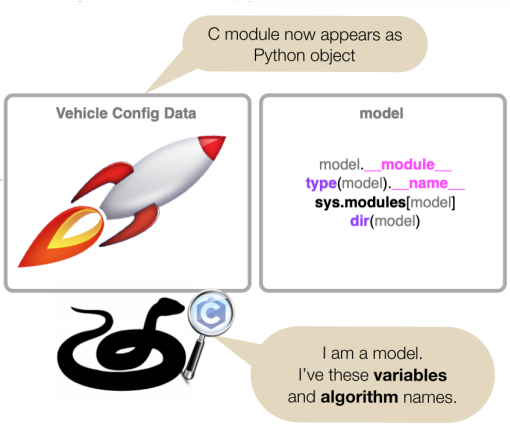


Fig. 6.: Python Introspection

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. This idea is graphically illustrated in Fig. 6.

Listing 3 shows a snipped of code automatically generated by the *AutoSetter.py*. Note that this C setup code (output of the *AutoSetter.py*) corresponds to Python code shown previously in Listing 2 (input of the *AutoSetter.py*). Let us take a closer look, for instance, to the inertia variable (*ISCPntB_T*). In Python, the inertia is initialized as a list of 9 floats, with only 3 of them being actually non-zero values; for the *AutoSetter.py* this unambiguously translates into a C array of 9 doubles, with the same indices filled with-non zero values as in the Python list.

Listing 3: AutoGenerated C Setup Code

```
// Struct containing all FSW modules in a process
typedef struct{
        VehicleConfig veh_config;
        // [...] More modules below
} AllConfig;
// Initialization of all  FSW modules
void AllConfig_DataInit(AllConfig *data){
        memset(data, 0x0, sizeof(AllConfig));
        // VehicleConfig module init
        data->veh_config.ISCPntB_B[0] = 600.0;
        data->veh_config.ISCPntB_B[4] = 600.0;
        data->veh_config.ISCPntB_B[8] = 600.0;
        strcpy(data->veh_config.outputMsgName,
        "adcs_config_data");
        // [...] More modules below
}
```

5.2. Embedded FSW Testing

Figure 7 illustrates that the unmodified FSW algorithms plus the auto-generated C setup code constitute a CFS application that is embeddable. For the aforementioned interplanetary mission, the embedded FSW application is tested in an emulated flat-sat configuration. 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 shown in Fig 8. Here, the CFS-FSW application runs within a processor board emulator (or single-board computer, SBC, emulator) and interacts with external applications like the space-

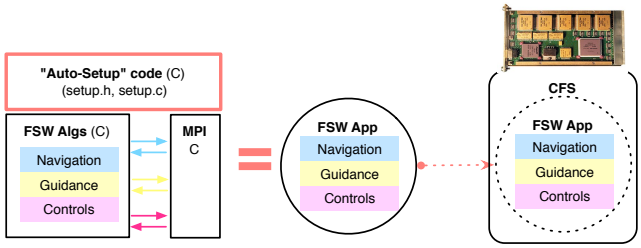


Fig. 7.: Embedded FSW Application

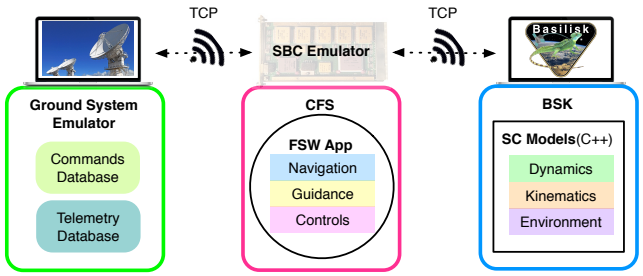


Fig. 8.: Concept of Emulated Flat-Sat

craft physical simulation and a ground system model. Figure 8 showcases the general idea of an emulated flat-sat for the purposes of integrated FSW testing. However, once FSW is integrated within CFS and embedded inside the SBC emulator, it is actually not that straightforward to access the FSW states for reading and writing. In contrast to the desktop environment, here there is no longer a flexible Python layer that allows easy interaction with the C flight algorithm code. In fact, in order to communicate with the embedded FSW application running within the SBC emulator, it has been necessary to emulate the FPGA registers as well. These registers have been simply modeled as a memory map for input and output of raw binary data. The layout of the combined CFS-FSW and modeled registers, within the SBC emulator, is showcased in Fig. 9.

In the very end, for the aforementioned interplanetary mission, the general concept of an emulated flat-sat represented in Fig. 8 has materialized in the configuration displayed in Fig. 10. Note that, within the flight processor emulator in Fig. 10, there are several registers emulated. Through these registers, FSW receives commands and returns telemetry (using CCSDS packets from and to the ground system GS emulator). Also through them, FSW commands the actuators (which are modeled within the spacecraft physical simulation in Basilisk) and receives sensor data from the Basilisk sensor models. The emulated flat-sat configuration in Fig. 10 is used in the aforementioned interplanetary mission for the first phase of integrated testing. While pure software flat-testing (i.e. emulated) does not replace hardware flat-sat tests, it can reduce bottlenecks by providing pure software substitutions for hardware components of limited quantity that might be needed simultaneously for testing

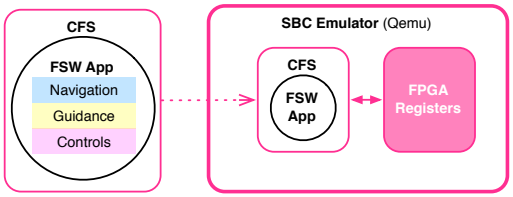


Fig. 9.: FPGA Register Emulation

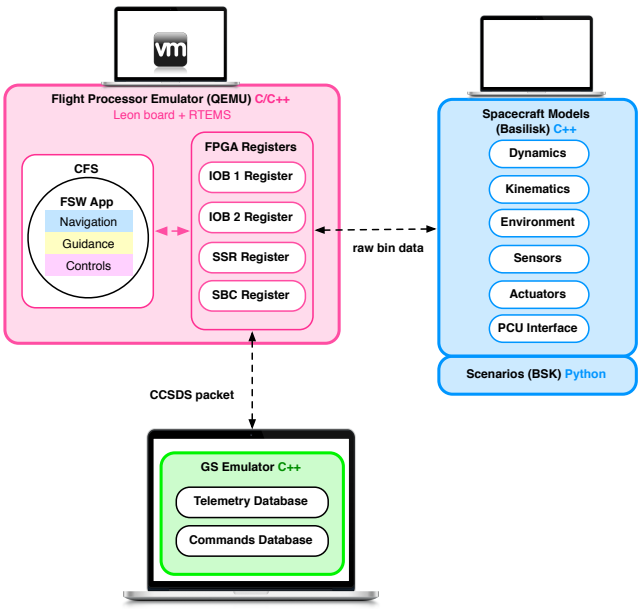


Fig. 10.: Actual Emulated Flat-Sat Configuration

by different mission groups.

5.3. Requirements and Limitations

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 as in Fig. 10, the FPGA registers need to be emulated. This approach has allowed for flexible development of the flight algorithms that can quickly be tested in the embedded system; the migration back and forth desktop and embedded environments is very rapid thanks to the *AutoSetter.py* and integrated testing within the emulated flat-sat has already revealed several problems within the flight application. Having said that, this approach presents three clear cons: migration effort, difficult interaction with FSW and replicated CFS functionality.

Migration effort: even if the migration is pretty transparent, there still is a migration effort involved. 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 to be tested in the embedded environment.

Difficult interaction with FSW: the modeling of FPGA registers, necessary for interacting with FSW in the emulated embedded environment of Fig. 10, is still work in progress. The connections between FSW states and their corresponding memory addresses within the registers are unique, involving additional complex code to work. Further, in the embedded environment of Fig. 10, it is not possible to fully capture all the CFS-FSW states; only the raw binary data that is actually mapped to the register addresses can be accessed and, only some of it, is converted into telemetry for the ground system model to parse. This is specially problematic when corner cases are found during integrated testing in emulated flat-sat configuration: if FSW states cannot be fully captured, then it is not possible to replicate the configuration 100% exactly again.

Replicated CFS functionalities: Last but not least, CFS has revealed some inflexibilities in its design. Recalling Fig. 3, the cFE executive layer provides five core services, which cannot be removed or customized –hence, yielding to the presence of replicated functionality within the embedded flight application. For instance, the mission flight algorithms shown in Fig. 7 already come with their own message passing interface (MPI); therefore, the software bus services within cFE are not used. Similarly, the processor board emulator and RTOS shown in Fig. 10already handle timing; hence the cFE time services become redundant. Finally, the event services within cFE are meant to be used for asynchronous messaging but, in the configuration of Fig. 10, this role is done by the ground system model.

6. Embedded FSW Development: MicroPython

After seeing both the feasibility and the drawbacks of the CFS embedded development approach, there is a question that lingers into the air: Is there a way to minimize the gap further, between desktop and embedded environments, and to reduce the migration effort involved?

Since the desktop development proposal (i.e. using Python as a user-interface language with underlying C/C++ flight source code) has proven, in the authors' experience, to work extremely well, it makes good sense to consider MicroPython for embedded FSW development.

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. MicroPython is packed full of advanced features such as an interactive prompt, list comprehension, generators, exception handling and more. All these advanced features make the MicroPython environment more alike desktop, while still being an embedded system: it is compact enough to fit and run within just 256k of code space and 16k of RAM. Furthermore, MicroPython aims to be as compatible with normal Python as possible, enhancing the transfer of Python 3 code from the desktop to the embedded environment.

As in the desktop development proposal, the idea here is to maintain the unmodified flight algorithm code written in C – which in the future will hopefully be C++– and use MicroPython for embedded setup only. With this purpose in mind, an open-source C++ wrapper tool for MicroPython is presented next.

6.1. MicroPython C++ Wrap

The MicroPython C++ Wrap[††] is a header-only C++ library that provides some interoperability between C/C++ and the MicroPython programming language. The standard way of extending MicroPython with custom C or C++ modules involves a lot of boilerplate code. Using the MicroPython C++ Wrap, the process of integration with MicroPython is drastically reduced.

6.2. Proof of Concept

The goal is to prove that the combination of MicroPython and its C++ wrap can successfully be used for: **setting up** the unmodified C/C++ flight algorithms as developed in the Basilisk desktop environment, and **executing** the simulation within embedded system.

Recall that, in the desktop environment, both setup and desktop execution as well as post-processing are handled by the

[††] `https://github.com/stinos/micropython-wrap`

Python layer. Because MicroPython is only a light version of the Python3 programming language, and most data-analysis libraries for post-processing are not supported, MicroPython will only be used for archiving flight data from an embedded simulation run –rather than fully post-processing it. Thanks to the interoperability between MicroPython and regular Python, the archived results can be straightly loaded into desktop Python for regular post-processing. The key aspect of this idea is that, because the same flight data can be logged in a MicroPython run and a regular Python run, the post-processing scripts of the desktop environment still apply either way.

The technical work required in order to prove the presented concept involves three development items, which could be considered as migration effort. These three items are: **1)** Creating a C++ class for every C FSW module, **2)** Generating MicroPython integration code for every C++ class that needs to be available to the MicroPython layer, and **3)** Adapting existing desktop Python scenario scripts into MicroPython. Before describing each of these three items in detail, let us recall the Python introspection capabilities that have been introduced through the *AutoSetter.py*. This independent script that allows the generation of specific C setup code (for integration within CFS) has been modified to automatically handle items **1)** and **2)** of the MicroPython embedding approach. This new introspection file will be referred to as the *AutoWrapper.py*.

1. **Create a C++ class (.hpp file) for every C FSW module**: the FSW algorithms in Basilisk have been written in C instead of C++ because many space missions still have –or impose– a C code requirement. The space industry tends to be generally conservative and it takes time to adopt new development approaches or coding languages. However, C++, which is an object-oriented evolution of C, is a powerful, efficient and fast language that dovetails perfectly with space FSW if missions are willing to embrace it. The Basilisk desktop testbed totally supports the development of C++ modules. Indeed, as shown in Fig. 1, the spacecraft physical models within Basilisk are already written in C++.

   Since 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. This C++ class, which is automatically generated by the *AutoWrapper.py*, contains:
   (a) The C config struct instantiated as a private member of the C++ class.
   (b) For every variable within the C struct, setters and getters are created in the C++ class. Setter and getter functions are necessary because MicroPython cannot access C++ variables directly, but it can operate on them through function calls.
   (c) Finally, the C++ class also contains public function callbacks to the C module algorithms for self-init, cross-init, update and reset. These callbacks will be externally called from MicroPython for embedded execution.
2. **Generate integration code for every C++ function/type that needs to be available to MicroPython**: For every Basilisk FSW C++ module, its corresponding header .h, source .c and recently created .hpp, need to be copied over inside the MicroPython C++ Wrapper directory. Further, it is necessary to declare/register the C++ classes/function /types of the flight application that need to be available to MicroPython. This integration code, which is generated by the *AutoWrapper.py* consists, more precisely, of the following:
   (a) Registration of each FSW C++ class as a MicroPython object
   (b) Mapping of function names between the C++ class and the MicroPython object.
   (c) Linking of setters and getters (in the C++ class) to properties (in MicroPython).

   At this point, the Basilisk FSW application and the C++ Wrapper are ready to be compiled as a static library and linked to the MicroPython built.
3. **Adapt existing desktop Python scenario scripts into MicroPython**: The desktop setup scripts written in Python do not work seamlessly on MicroPython because not all the Python libraries used for setup are supported in MicroPython. Further, currently, all the Basilisk FSW modules are still written in C. Upcoming work encompasses creating scenario scripts that can run both in desktop Python and embedded MicroPython, and which use the newly created C++ FSW modules. This would allow an apple-to-apple comparison of the results between the desktop and embedded runs, which should match identically.

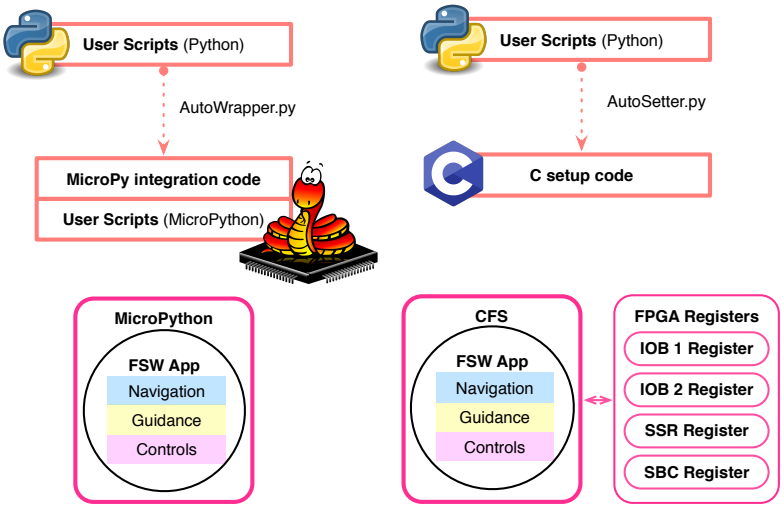
6.3. Advantages of the MicroPython Approach

How the MicroPython embedded approach compares to the CFS development approach is illustrated in Fig. 11 and briefly summarized as follows. The **migration effort is reduced** because the automatically generated code is no longer specific, but reconfigurable: the C setup-code generated by the *AutoSetter.py* for CFS integration was specific to every single FSW configuration as defined in a given Python scenario script; in contrast, the FSW C++ classes and MicroPython integration code generated by *AutoWrapper.py* are only written once. At this point, all the FSW states become fully reconfigurable from MicroPython without recompilation. Because MicroPython has full access to the message passing interface of the FSW application, it is **no longer necessary to replicate the FPGA registers**. Another key advantage of the MicroPython approach is that now it is **possible to fully capture all the FSW states** at any point in a simulation run. Last but not least, MicroPython guarantees the portability of a middleware layer **without** the **replicated functionality** imposed by CFS.

7. Conclusions and Future Work

This paper has presented two different strategies for end-to-end FSW development. Both strategies use the Basilisk testbed as a desktop development environment but they differ on the targeted embedded environment: the Core Flight System (CFS) in one case and the novel MicroPython in the other.

The feasibility of the CFS approach –more conventional than the MicroPython one– is discussed through the experience of its



(a) MicroPython approach (b) CFS approach

Fig. 11.: Embedded FSW Development Approaches

application into an interplanetary spacecraft mission. Of particular interest is the method for integrating the Basilisk-developed flight algorithms into an embeddable CFS application. This integration is achieved by automatically generating a minimal set of C integration code through Python's introspection capabilities. Having said that, the overall challenges in the use of CFS motivate the search for a different strategy that smoothes the migration of the flight application between environments.

On these lines, the feasibility of combining a light-weight 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). While the MicroPython investigation is still ongoing research, the current results are promising. A complete implementation of this strategy would enhance the testing capabilities of FSW 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.

References

1) 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.
2) 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.
3) 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.
4) 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.
5) 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.
6) 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*, No. 7, College Station, TX, May 20–21 2018.
7) Lauretta, D., *OSIRIS-REx Asteroid Sample-Return Mission*, Vol. Handbook of Cosmic Hazards and Planetary Defense, Springer, 2015.
8) Mangieri, M. and Vice, J., "Kedalion: NASA's Adaptable and Agile Hardware/Software Integration and Test Lab," *AIAA SPACE*, 2011.
9) 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.
10) McComas, D., "NASA/GSFC' Flight Software Core Flight System," *Flight Software Workshop*, San Antonio, TX, Nov. 7–9 2012.
11) Cudmore, A., "NASA/GSFC's Flight Software Architecture: Core Flight Executive and Core Flight System," *Flight Software Workshop*, 2011.
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*, 2018.
13) 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.