AAS 20-103

AVIONICS HARDWARE MODELING AND EMBEDDED FLIGHT SOFTWARE TESTING IN AN EMULATED FLAT-SAT

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

This manuscript describes the end-to-end flight software (FSW) development process for an interplanetary spacecraft mission in which the University of Colorado Boulder and the Laboratory for Atmospheric and Space Physics are collaborating. As the term "end-to-end" indicates, the entire FSW development cycle is covered: starting from a preliminary desktop design and analysis all the way to testing on the flight hardware. For desktop prototyping, the strategy of using Python as a wrapper for C/C++ flight algorithm code is employed. The Basilisk software testbed is presented as a specific incarnation of this desktop development strategy. For embedded development, this work uses the Core Flight System (CFS) middleware and the same C flight algorithm developed in the desktop environment. Regarding flight hardware, a flat-sat emulation is utilized to perform embedded testing of the resulting CFS-FSW application. The flat-sat is emulated in the sense that all the different mission components are software models replicating its hardware counterparts. Here, the CFS-FSW application runs within a processor board emulator and interacts with external applications like the spacecraft physical simulation and a ground system model. Numerical simulation results showcase the closed-loop performance of the embedded CFS-FSW application in an interplanetary mission scenario.

INTRODUCTION

The complete engineering cycle to develop a flight software system (FSW) for an interplanetary spacecraft mission encompasses an involved path of deploying and running the flight algorithms within different testbed environments. As in many of the upcoming Mars exploration missions, in a standard interplanetary spacecraft mission there are mainly three FSW testbed environments to consider: desktop computer (for algorithm design, prototyping and rapid iteration), hardware flight processor (for flat-sat testing and eventually flying) and emulated flight processor in a virtual machine (for emulated flat-sat testing). These environments are illustrated in Fig. 1 where the term single board computer (SBC) is used to refer to the flight processor. The two latter environments, hardware or emulated flight processor, are considered to be embedded. 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.

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

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

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

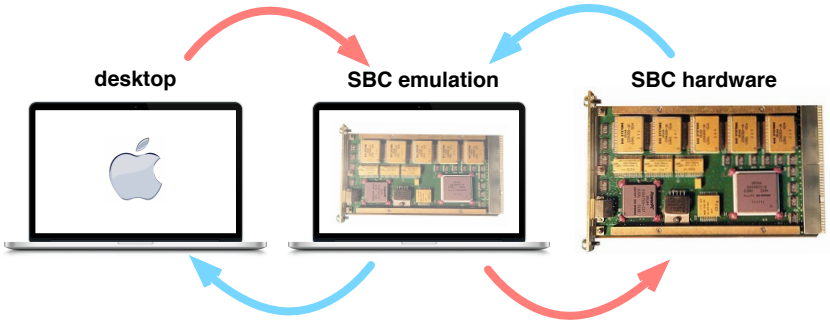


Figure 1. FSW Analysis and Testbeds Platforms

This paper discusses an end-to-end FSW development strategy and working implementation that supports having desktop and embedded environments separately while ensuring, firstly, transparent migration of the flight application and, secondly, consistent testing throughout the different testbeds. 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. Regarding consistent and high-fidelity testing throughout environments, such endeavor can be effectively achieved by means of a distributed communication architecture. The purpose of a distributed communication framework is to enable integration of heterogeneous and independent mission components into a single simulation run, which is agnostic to: 1) mission components being hardware or software models and 2) FSW executing from desktop or embedded environment.

The end-to-end FSW development strategy presented in this manuscript is currently being applied to an interplanetary spacecraft mission in which the Autonomous Vehicle Systems (AVS) laboratory at the University of Colorado Boulder and the Laboratory for Atmospheric and Space Physics (LASP) are collaborating. The term *end-to-end*, as used in this manuscript, implies that the entire FSW development cycle is covered: starting from a preliminary desktop design and analysis all the way to testing on the flight hardware. The mission flight algorithms are designed in the desktop environment through the Basilisk* simulation framework. Basilisk seeks to capitalize on the potential of using Python as a wrapper for flight algorithm and simulation code that, on its core, is written exclusively in C/C++. The advantage of this approach is that desktop users can take advantage of Python's flexibility and, at the same time, the flight source-code does not need to change for migration into embedded systems; the Python wrapper is simply removed for embedding.

Once flight algorithms are verified to satisfy mission requirements in the desktop environment, the next step is to migrate them into the mission flight target of choice. The chosen flight target is, for this mission, the Core Flight System (CFS) middleware.[1,2] Middleware layers are characterized by its portability among different boards and real-time operating systems (RTOS). Hence, they are useful in the long run in order to promote flight code reusability across different missions. The result of migrating Basilisk-developed flight algorithms into the CFS middleware is a CFS-FSW application that can be readily embedded into a radiation-hardened flight processor or its emulated counterpart. This manuscript will focus, in particular, on emulated flat-sat testing of the embedded FSW. The emulation of embedded systems is particularly interesting because it provides

*https://hanspeterschaub.info/basilisk

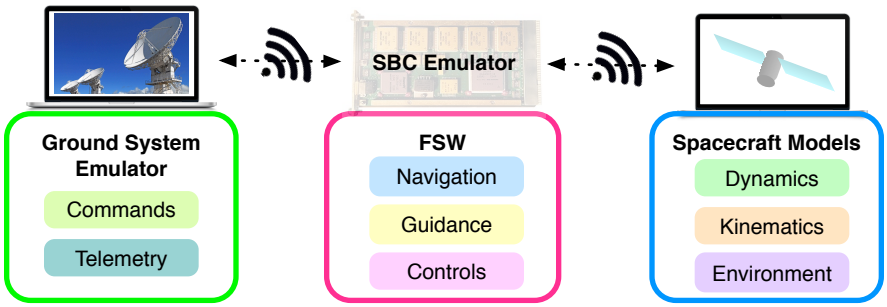


Figure 2. Concept of emulated flat-sat

pure software substitutions for expensive hardware components of limited quantity that might be needed simultaneously for testing by different mission groups.[3–5] While SW-sim testing does not replace hardware flat-sat tests, it can reduce bottlenecks and alleviate schedule constraints by using software models only –hence, providing a flexible and cost-effective means of performing early-on mission testing. The general concept of an emulated flat-sat for embedded FSW testing is illustrated in Fig. 2. The flat-sat is emulated in the sense that all the different components are actually software models replicating its hardware counterparts.

This manuscript is outlined as follows: first, the Basilisk desktop testbed is introduced as the framework in which algorithm design and prototyping is performed. Next, the CFS middleware is presented and the migration of Basilisk developed flight algorithms into CFS is described. Following, the concept of testing the CFS-FSW application in an emulated flat-sat is explained. Additional software implementations required to achieve distributed closed-loop testing of the embedded FSW are also discussed, including: the underlying communication framework and avionics hardware modelling. Finally, numerical simulations are shown and brief conclusions are provided.

DESKTOP ALGORITHM DESIGN: THE BASILISK SOFTWARE TESTBED

Basilisk is an open-source and cross-platform desktop testbed for designing flight algorithms and testing them in closed-loop dynamics simulations. Basilisk leverages the use of Python as a testbed for FSW development provided that the flight algorithm and simulation code are written exclusively in C/C++ and then wrapped into Python for: setup, desktop execution and post-processing. Advantages of using Python as a user-interface language include but are not restricted to:

1. Ease of data analysis through Python's built-in libraries (e.g. *numpy*, *matplotlib*, *pandas*...)
2. Rapid Monte Carlo handling: setting up different scenarios and initial conditions in Python does not require recompiling the C/C++ source code
3. Automated regression tests (via *pytest*)

Because the underlying code is written in C and C++, speed of execution is still guaranteed. Further, in contrast to other strategies like model-based-development,[6] the use of Python as a wrapper ensures migration transparency: the C/C++ source-code does not change for migration into embedded systems –the Python wrapper is simply removed.

Going back to Basilisk, the nominal, but not necessarily required, setup of a Basilisk desktop simulation is illustrated Fig. 3. This setup is decomposed into two main processes: a high-fidelity simulation of the physical spacecraft written in C++ (*SC Models* in Fig. 3) and a suite of GN&C flight algorithms written in C (*FSW Algs* in Fig. 3). The idea is that each of these processes is created

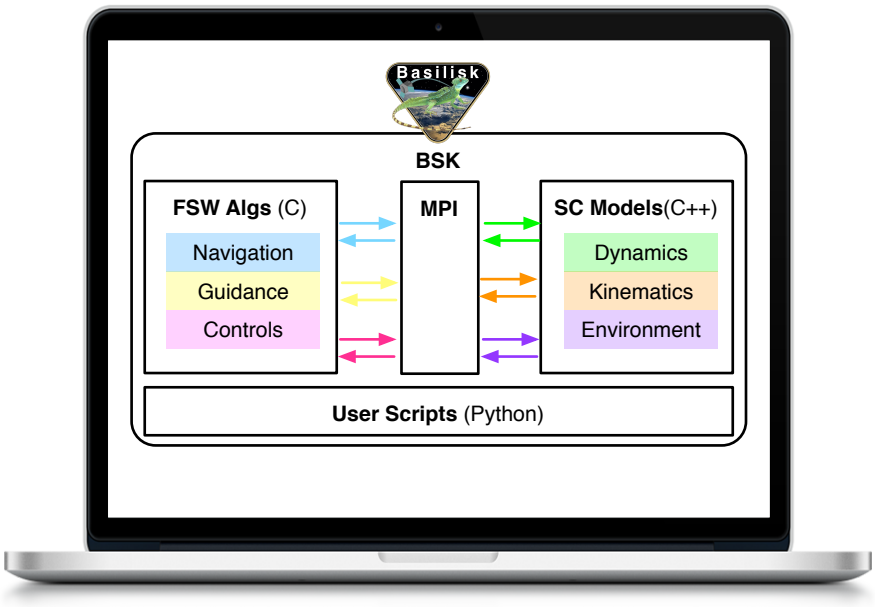


Figure 3. Basilisk (BSK) desktop environment

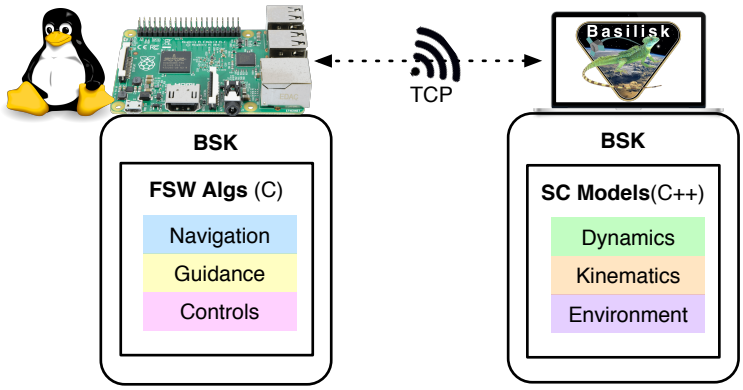
at the Python level. The processes are further decomposed into tasks, which in turn are containers for C/C++ modules that run together at certain rates. During a simulation run, all the different C and C++ modules communicate with each other through a custom message passing interface (*MPI* in Fig. 3) that is based on a publish-subscribe pattern. To be clear, the Basilisk MPI is not based on the well-known Open-MPI standard*; rather, it a is customized messaging bus tailored to Basilisk specific needs.

The beauty of the Basilisk MPI is that it marks a clear separation between the different processes. This clear delineation facilitates, later on, the migration of the flight application (i.e. *FSW Algs*) into a different processor (the flight target of choice), while preserving the capability of closing the loop with a dynamics simulation that remains on the desktop environment. As a matter of fact, Basilisk developed flight algorithms have been ported into several targets: commercial processors like the Raspberry Pi[7] as well as embeddable middleware layers like the Core Flight System (CFS) or MicroPython.[8] Figure 4 shows the use of Basilisk in a multi-processor environment for realistic testing of FSW on the flight target. This manuscript focuses in the port of Basilisk flight algorithms into CFS and, also, in the integration of the resulting CFS-FSW application into an emulated single board processor, as in Fig. 4(b).

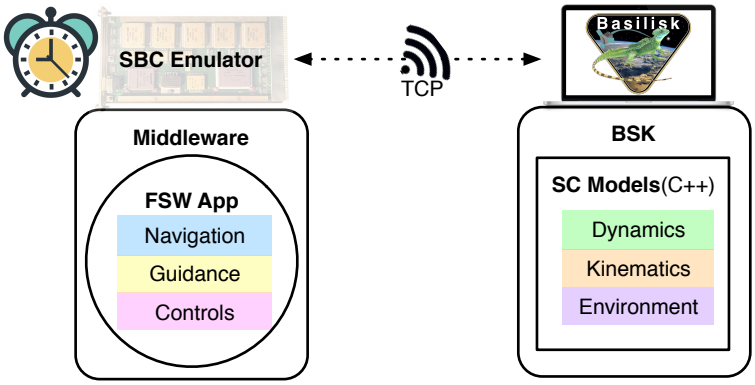
MIGRATION INTO THE CORE FLIGHT SYSTEM

Because a regular desktop computer environment and a flight processor environment operate differently, migrating the flight application from one to another demands a significant migration effort. Furthermore, this effort is intrinsically linked to the specific processor board and RTOS chosen, becoming then mission-specific. An alternative target for mission flight algorithms is a middleware layer. Middleware can be regarded as an abstraction layer or "glue-code" that ensures portability of the flight application among different processors and real-time operation systems. The

*https://www.open-mpi.org



(a) BSK flight algorithms on the Raspberry Pi



(b) BSK flight algorithms on middleware

Figure 4. Port into different flight targets

Core Flight System (CFS), in particular, is an open-source middleware layer provided by NASA Goddard Spaceflight Center that has inherited software from flight missions over the previous 20 years or more.[1,2] The CFS code-base is written in C and its layered architecture is depicted in Fig. 5. The application layer, which lies on top, is mission-specific and, therefore, it is meant to be provided by the user.

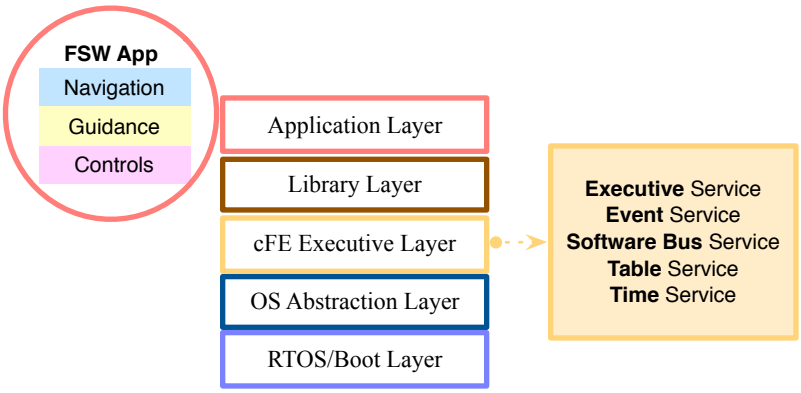


Figure 5. Architecture of the Core Flight System

The question that remains is what does it take to migrate Basilisk-developed flight algorithms into a CFS application that is actually embeddable. Recall that Basilisk leverages the use of Python for FSW (C/C++) setup, desktop execution and post-processing. There is one of these Python functionalities that needs to be translated into C for migration: the setup for the C flight algorithms. Once the flight application is written entirely in C, it can be readily integrated into CFS. Next, the meaning of "setup" code is explained concisely. In the Basilisk framework, the Python setup encompasses:

1. Variable initialization of each individual C module
2. Grouping of modules in tasks that run at certain task rates

The idea for migration is the following: firstly, all the module variables that in the desktop environment are initialized from Python, in the CFS embedded environment must be initialized directly in C (and with the same values). Therefore, out of each Python scenario script, there needs to be an equivalent initialization script written in C that is capable of configuring all the flight algorithms in the given mode. Secondly, the tasks and task rates defined at the Python level for a given FSW process scenario also need to be coded into C such that, during a simulation run, the calling order and organization are preserved.

Interestingly, the translation of setup code from Python to C is handled automatically via an independent script implemented by the authors in Python: the *AutoSetter.py*. The beauty of the *AutoSetter.py* is that is not a black box but, rather, an open and transparent template that maps Python variable types/values into their C counterparts. The resulting C setup code is minimal and completely human readable. 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 while, in contrast, Python is excellent at it. Figure 6 illustrates graphically the translation process that has just been described. A key remark here is that the flight algorithm source code remains unchanged: the pure-C application is conformed by the unchanged algorithm source-code (*FSW Algs.*) plus one additional header (*setup.h*) and source file (*setup.c*) containing the setup code written in C. The resulting pure-C application can be integrated into CFS and then embedded into a flight target.

EMBEDDED TESTING OF CFS-FSW IN AN EMULATED FLAT-SAT

The embedded CFS-FSW application can be tested in an emulated flat-sat configuration, where all the different hardware components are replaced by their software counterparts, as in Fig. 2. In such case, the CFS-FSW application runs within an SBC emulator, which is responsible for enabling interaction between FSW and the external world. Having said that, once FSW is integrated into CFS and embedded into the SBC emulator, it is no longer simple to access the FSW states for reading and writing. One possible strategy to enable interaction between FSW and the external world is to emulate the FPGA registers on the SBC. These registers can be modelled, for instance, as a memory map for input and output of raw binary data. The layout of the combined CFS-FSW application and modelled registers within the SBC emulator is represented in Fig. 7.

For the interplanetary spacecraft mission in which the AVS Laboratory and LASP are collaborating, the general concept of an emulated flat-sat has actually materialized in the configuration displayed in Fig. 8. The illustrated software components are: a flight processor emulator, a ground system model, the spacecraft physical simulation and a visualization interface.

It is important to mention that all these models are independent, stand-alone applications that

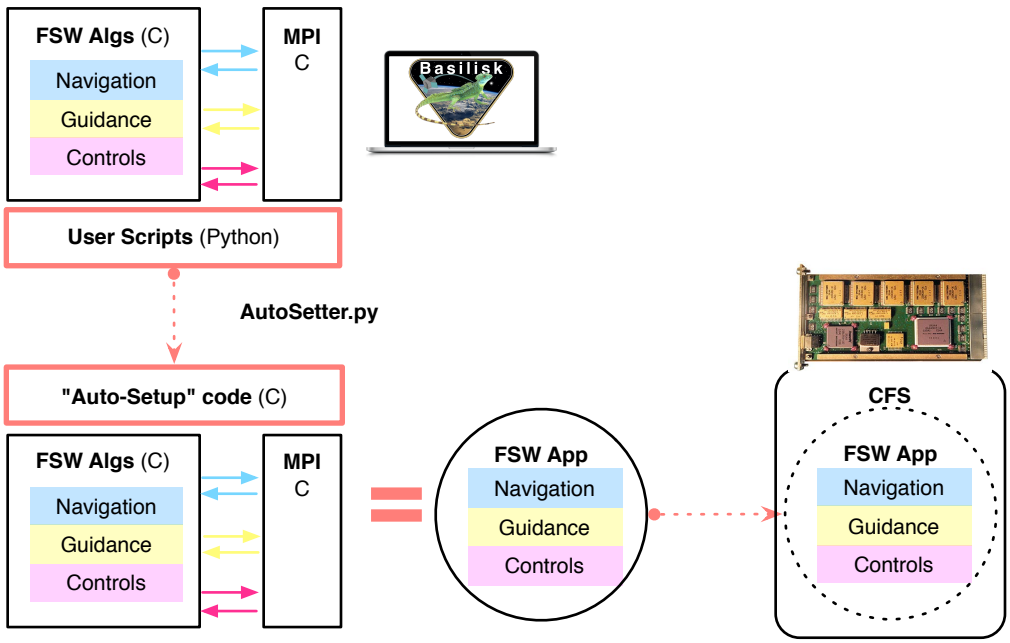


Figure 6. Flight algorithm migration into CFS through the "AutoSetter.py"

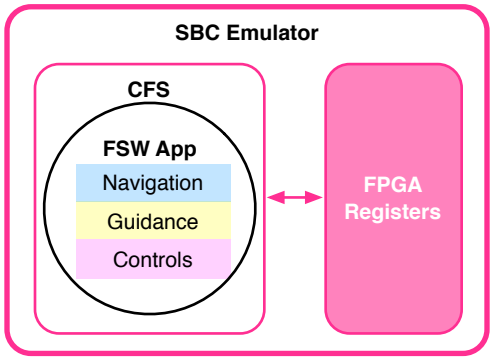


Figure 7. FPGA Register Emulation

were never designed to work together. Therefore, they are inherently heterogeneous and their integration into a single, closed-loop simulation run demands for a solid communication architecture underneath. On these lines, Black Lion is presented as a purely software-based distributed communication architecture that has been developed to support the aforementioned interplanetary mission.[3] For the flat-sat configuration in Fig. 8, the use of Black Lion allows enabling communication between models/nodes whose heterogeneity spans from: multithreaded vs. single-threaded nodes, asynchronous vs. synchronous nodes, little-endian vs. big endian nodes, as well as for a variety of programming languages: Python, C, C++ and C# nodes. The communication tasks handled by Black Lion encompass: data transport, data serialization and synchronization between the components. For further details on the Black Lion communication architecture the reader is referred to Ref. 3. In this manuscript, each Black Lion communication pipeline is marked through a "TCP" sign, as in Fig. 8.

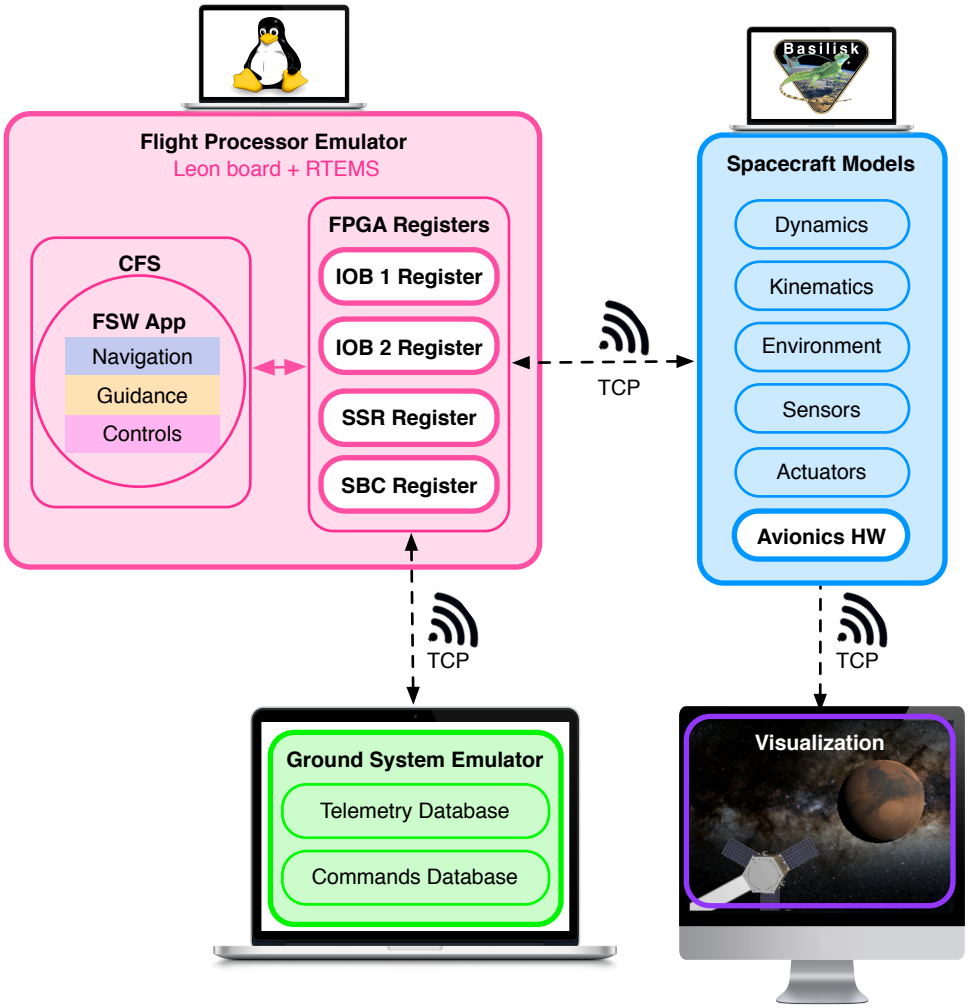


Figure 8. Mission flat-sat components

Now that the emulated flat-sat components have been introduced and the underlying communication architecture has been briefly described, let us focus again on the flight processor emulator and, in particular, on the interaction between the CFS-FSW application and the external world through the modelled FPGA registers.

CFS-FSW interaction

Flight processor emulators can effectively be used to test the on-board FSW executable, which is a complete binary image that includes:

1. The FSW algorithms/application
2. The real-time operating system (RTOS)
3. The board boot-software and the board support package

The present work uses an emulation of a LEON board to run and test the CFS-FSW application on top of the RTEMS real-time operation system.

In order to access the CFS-FSW states, a total of four FPGA registers have been modelled within

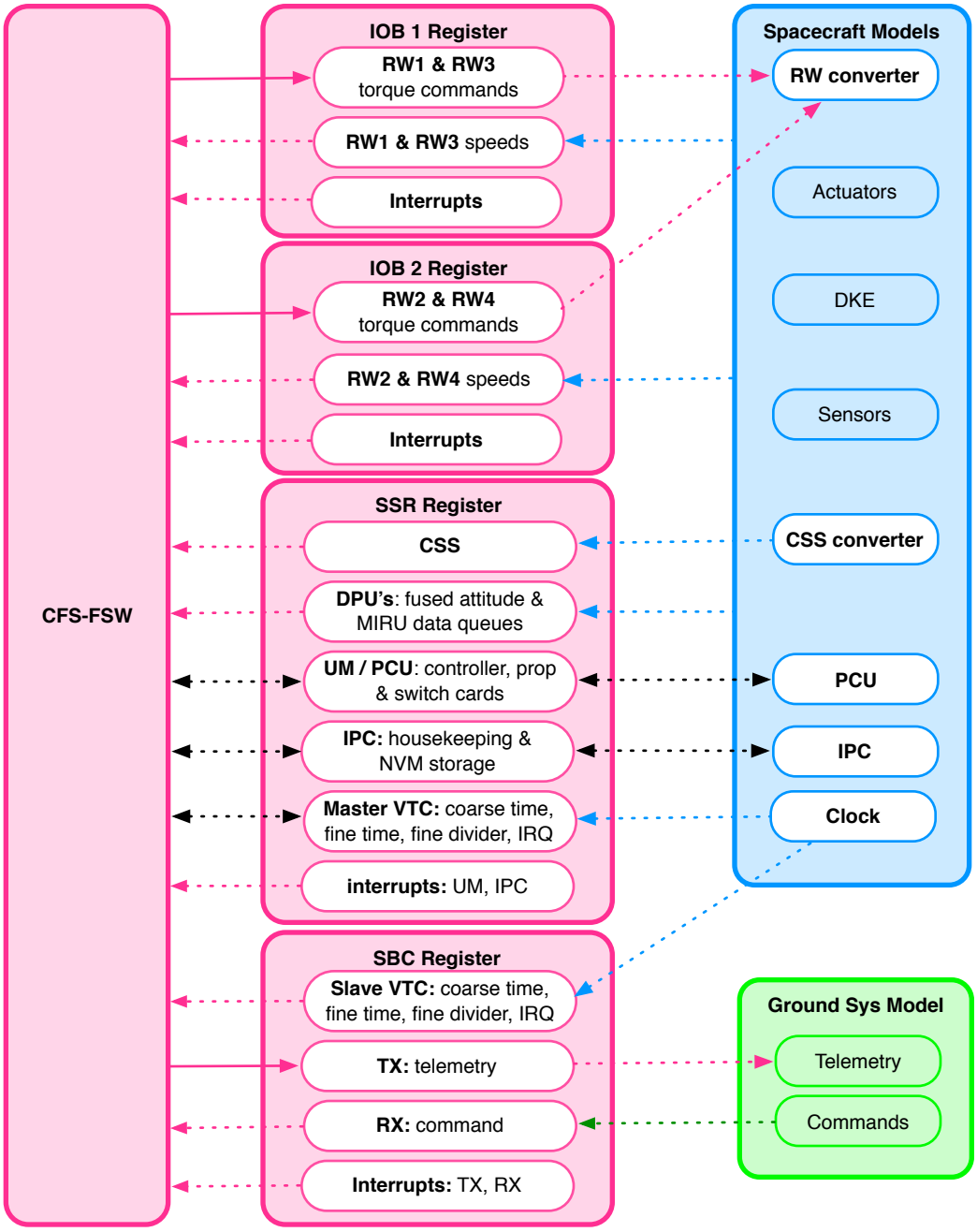


Figure 9. Registers' snorkels and their connections to the external world

the processor board emulator: two input/output board registers (IOB), the solid state recorder register (SSR) and the single board computer register (SBC). Through the FPGA registers, FSW reads and writes in a hardware-like fashion that also replicates interrupts. Further, the modelling of registers has been accompanied with an expansion of spacecraft physical models, which now include complex avionics hardware. These avionics models leverage complex functionality that would otherwise have to be implemented within the registers themselves. The modelled registers and corresponding avionic models are highlighted in Fig. 8 with white boxes.

Regarding registers, the idea is that each of them has an associated memory buffer and specific FSW states are mapped to specific addresses within these buffers. Not all the internal FSW states are mapped to the register's buffers, but only those that require interaction with the external world. These shared states will be referred to as snorkels, in the sense that they are direct connection pipes to the internals of the CFS-FSW application. The different snorkels that have been implemented within the register space, as well as the specific connections of these snorkels to the external world, are illustrated in Fig. 9. Each snorkel is pictured within a white box with pink border.

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

The avionics hardware models that have been integrated into the spacecraft physical simulation in conjunction to the registers' snorkels are also illustrated in Fig. 9. Each avionics model is pictured within a white box with blue border. As shown, these models are: a reaction wheel (RW) converter, a coarse sun sensor (CSS) converter, a clock model, the power computing unit (PCU) and the peripheral component interconnect (PCI). The PCU and PCI are particularly complex models responsible of managing avionics cards (controller, propulsion and switch cards), storing/retrieving non-volatile-memory commands, and producing house-keeping packets. While the specific snorkels and avionic hardware models described here are mission-specific, the register space with its readers and writers constitutes a generic framework that can be applied to any FSW application. Reference 8 shows the use of the same register framework to test a different FSW application that is not integrated within CFS but within MicroPython instead.

NUMERICAL SIMULATIONS

This section describes an early numerical simulation showcasing the application of the modelled registers, snorkels and avionic models for the purposes of testing the CFS-FSW application in distributed closed-loop. The flat-sat configuration in which the simulation runs corresponds to the one shown earlier in Fig. 8, where there is: the flight processor emulator, the ground system model, the spacecraft physical simulation and the visualization interface. The integrated simulation encompasses a Mars science scenario in which the spacecraft is commanded to perform a series of pointing guidance maneuvers. These pointing commands are dynamically triggered by the user through the ground system interface.

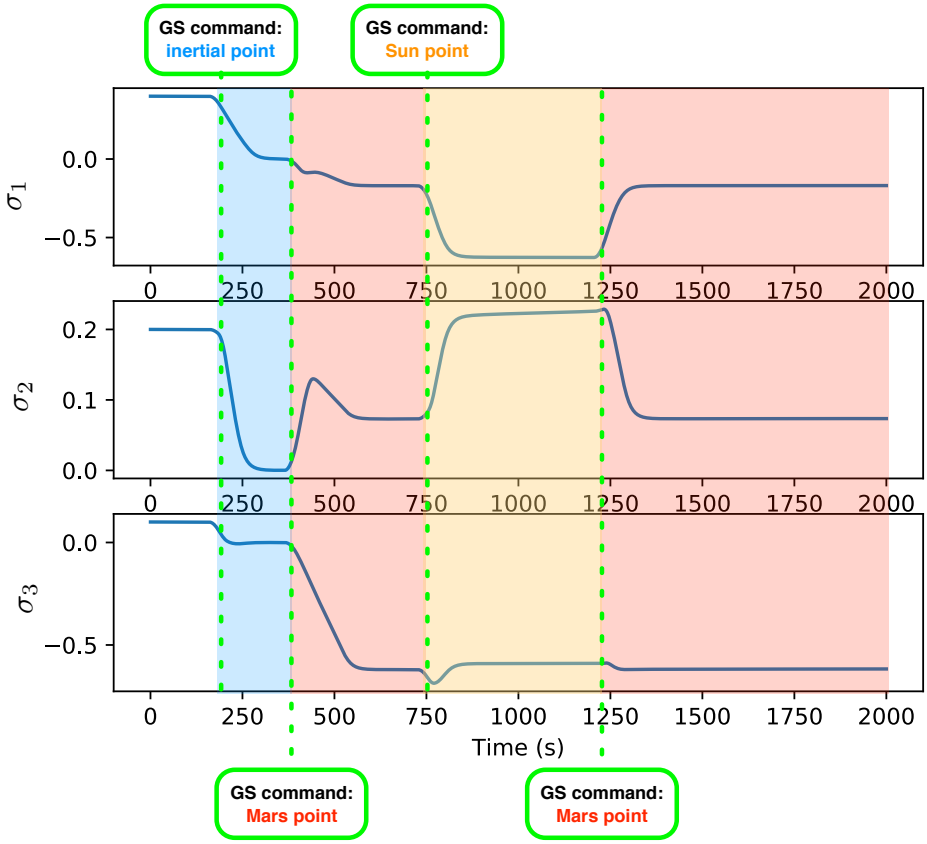


Figure 10. Closed-loop simulation: evolution of the spacecraft main body attitude

The different steps of the maneuver series are described next: after starting the simulation and bringing up all the flat-sat models, the user enables FSW monitoring command in order to acquire attitude knowledge lock. Once attitude is locked, which can be observed by the telemetry reported from FSW to the ground system, the user commands the spacecraft into inertial pointing. After achieving inertial pointing convergence (which can also be seen by the reported telemetry), the user commands FSW to correlate the onboard ephemeris and switches the pointing mode to Mars pointing, followed by Sun pointing, and back to Mars pointing. The convergence of the spacecraft attitude into each of the commanded modes can be observed in Fig. 10. The plots in Fig. 10 correspond to the Modified Rodrigues Parameters (MRP)[9]attitude of the spacecraft main body frame, as evolved in the spacecraft physical simulation.

The four pointing commands issued by the user from the ground system model are also indicated in Fig. 10. These commands are stored in the FPGA registers and picked up by the CFS-FSW application in order to reconfigure the onboard pointing mode. Sensor data from the spacecraft physical simulation is also being continuously updated within the registers. With this data, FSW estimates the spacecraft attitude, derives the associated tracking errors and computes the control torques required to drive the spacecraft into the desired attitude. The control torques are stored in the registers and sent to the spacecraft physical simulation, where a set of four reaction wheels applies the commanded control torque.

CONCLUSIONS

This manuscript discusses the end-to-end FSW development strategy and working implementation for an interplanetary spacecraft mission. This strategy supports having both desktop and embedded environments separately while ensuring, firstly, transparent migration of the flight application and, secondly, consistent testing throughout the different testbeds. The FSW development cycle described in this work encompasses: flight algorithm design through the Basilisk simulation framework, migration into the Core Flight System middleware and embedded testing in an emulated flat-sat. The flat-sat is emulated in the sense that all the different mission components are software models replicating its hardware counterparts. Interestingly, all these flat-sat models are completely heterogeneous and they run from different platforms in a distributed manner. Of specific interest is the interaction between the embedded FSW, which runs on a processor board emulator, and the external flat-sat models. In order to access the FSW states for reading and writing, it has been necessary to emulate the FPGA registers within the processor board emulator as well. In addition, the emulation of registers has been accompanied with the modeling of complex avionics hardware. A sample numerical simulation is employed to illustrate the workings and capabilities of the emulated flat-sat for purposes of embedded FSW testing. The simulation shown consists on commanding the spacecraft into a series of pointing guidance maneuvers. More complex simulation runs performed with the same flat-sat configuration involve: delta-V burns (for Mars orbit insertion), off-nominal star-tracker acquisition, jamming the spacecraft clock time or issuing FSW resets.

REFERENCES

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

[2] A. Cudmore, "NASA/GSFC's Flight Software Architecture: Core Flight Executive and Core Flight System," *Flight Software Workshop*, Johns Hopkins University Applied Physics Laboratory, MD, 2011.

[3] M. Cols Margenet, P. Kenneally, H. Schaub, and S. Piggott, "Distributed Simulation of Heterogeneous Mission Subsystems through the Black Lion Framework," *Submitted to AIAA Journal of Aerospace Information Systems*, 2019.

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

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

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

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

[8] M. Cols Margenet, H. Schaub, and S. Piggott, "Flight Software Development, Migration and Testing in Desktop and Embedded Environments," *Submitted to AIAA Journal of Aerospace Information Systems*, 2019.

[9] H. Schaub and J. L. Junkins, *Analytical Mechanics of Space Systems*. Reston, VA: AIAA Education Series, 4th ed., 2018.