Evaluation of MicroPython as Application Layer Programming Language on CubeSats

Sebastian Plamauer
Institute of Astronautics
Technical University of Munich
Munich, Germany
sebastian.plamauer@tum.de

Martin Langer
Institute of Astronautics
Technical University of Munich
Munich, Germany
martin.langer@tum.de

Abstract—**Since the dawn of the space age, software has always been a critical aspect for any space mission launched. Over the decades, more complexity, autonomy and functionality was added to both unmanned and manned missions, yielding in an exponential growth of the lines of codes used in space projects over the years. Although a lot of effort was put into ensuring reliable software on those missions, some of them failed. Still, as the space industry is a risk-averse business, testing of novel approaches in space programs cannot be done on large scale. To overcome this limitation, this paper investigates the potential use of MicroPython, an implementation of Python for constrained systems, for use on CubeSats by analyzing the language and tools in practical examples from the MOVE-II CubeSat project.**

Index Terms—**Satellites, SmallSat, CubeSat, Software, Computer languages, Python, MicroPython**

1. Introduction

For decades, satellite design philosophy was dominated by highly reliable components and conservative designs to achieve long lifetimes in the harsh space environment. Since the dawn of the space age, autonomy and control of the spacecraft made software one of the critical aspects of success or failure. Multi-million dollar losses like Mars Climate Orbiter [1], Ariane 5 flight 501 [2] and Mars Polar Lander [3] can be traced back directly to software flaws. As the scope of space systems, similar to terrestrial ones, broadens, the complexity of the on-board software historically increased over the years. An exponential growth rate of a factor of 10 approximately every 10 years was observed for software code in unmanned NASA spacecraft over the last 40 years [4]. As most large scale space programs are risk-averse, choice of the programming language and other formal methods during software development within the programs are heavy influenced by heritage and other organizational criteria. CubeSats attempted to choose a different philosophy, utilizing suitable state-of the art, commercial-off-the shelf products. Novel computer architectures as well as programming and scripting languages can be researched with less resources and risk than in traditional space programs. Within the MOVE-II satellite project [5] of the Technical University of Munich, several novel computing concepts are investigated. Amongst others, this includes a fault-tolerant, radiation-robust filesystem [6], autonomous Chip Level debugging [7], dependable data storage on miniaturized satellites [8] and a novel communication protocol for miniaturized satellites [9]. This paper will investigate the ongoing research on using MicroPython [10] as application layer programming language on CubeSats. First, the motivation for the research is given in Chapter 2. Chapter 3 deals with the background of Python [11], programming language evaluation and our project based evaluation approach. In Chapter 4, the methods used for evaluation are introduced. Results and first conclusions are later given in Chapter 5. Finally, in Chapter 6, we conclude with the implications of this work and give an outlook.

2. Motivation

Programming languages are tools to solve problems. Different problems require different tools and the large number of existing, and also used, languages provide ample resource to find one that fits the task at hand.

However, often the choice of language is severely constricted and the choice is not made by project based criteria, but organizational ones. The chosen languages are the ones that “have always been used”, that “everyone else uses”, that are already “available for the development system” or that are required in order to “satisfy contractual obligations” [12].

The special requirements for space systems, as well as the conservative approach that is common in the industry, result in a very small set of languages being actively used in this domain. For expensive and critical projects, developers default to proven languages, leaving little room for experimentation and thereby progress is slow. This work tries to broaden the scope of suitable languages by evaluating the use of MicroPython in a CubeSat project. CubeSats are small satellites with standardized dimensions that can be launched as secondary payloads on bigger missions, thus providing a low cost option of getting a satellite in orbit. The low cost make them an ideal platform for pushing new technologies in space.

First steps towards the use of MicroPython in space have already been undertaken by an ESA project, motivated by a desire to write the high-level application software in a language more suited to the application layer, meaning a high-level language like Python. Compared to C, Python enables higher productivity, more expressiveness, higher-level language constructs and inherent language safety [13].

Python is a language explicitly designed to aid readability. It has the potential to address the rising complexity of space software system simply by being easy to use. With the novel MicroPython implementation, Python can be used on constrained systems, like those common in space computing. These potential benefits and possibilities call for a detailed evaluation designed to find strengths and weaknesses and to establish use cases where space system developers can profit from using MicroPython. An evaluation also sometimes requires comparisons. Where applicable, C and C++ where chosen as languages to compare against because they are used in the MOVE-II CubeSat project that provides the examples used.

3. Background

3.1. The Python Programming Language and the MicroPython Implementation

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics created by Guido van Rossum. It features high-level data structures and dynamic typing, which makes it attractive for rapid development and as scripting or glue language to connect existing software components. Python's syntax is designed to aid readability, making it easy to learn and reduces the cost of software maintenance. It supports modules and packages, which encourages program modularity and code reuse [11]. The Python interpreter and the extensive standard library are released under the Python Software Foundation License, a BSD-style permissive free software license compatible with the GNU General Public License [14].

The reference implementation of this language is called CPython. It is an interpreter and itself written in C. Other implementations with different goals exist, for example Jython, written in Java to target the Java virtual machine, or PyPy, written in a subset of Python and aimed at improving performance. MicroPython is an implementation of Python for microcontrollers and constrained systems, created by Damien George. It aims to be lean and efficient and includes only a small subset of the standard library [10]. The source code is published under the permissive MIT license.

The CPython interpreter for the UNIX platform has a size of about 4.7 MB, the MicroPython equivalent has 0.5 MB. CPython's start-up memory usage is approximately 100 kB, MicroPythons is 20 kB. Similarly, in CPython object size is large – a simple integer takes 24 bytes in comparison to 4 bytes for 32-bit architectures in MicroPython. Some of this size savings come from the reduced subset of the Python standard library, which also shows a path for further size reduction. MicroPython can be configured at compile time, making it possible to strip out unused parts to reduce the size. Furthermore, the byte-code compiler and the byte-code virtual machine can be separated, so only the size of the byte-code interpreter is relevant to the system. For space systems, the split of byte-code compiler and byte-code interpreter also reduces the amount of software that has to be flight approved, as only the interpreter would run on the spacecraft.

The MicroPython port for microcontroller architectures has an even lower storage and memory footprint. 256 kB of storage and 32 kB of memory are sufficient to run non-trivial programs.

3.2. Programming Language Evaluation

In order to evaluate and compare programming languages, a set of criteria is needed by which to judge them. A canonical set of such is described by Sebesta [15] and reproduced in Table 1 and the following section.

The language criteria influence three main traits of a programming language: readability, writability and reliability.

Readability describes the ease with which a program can be *read* and understood. In the software life cycle, maintenance is the costliest factor and outranks design and development. Readability is a key factor in improving a software's maintainability and also makes it easier to spot errors in the code. In assessing readability, the problem domain has to be acknowledged, as different domains lend themselves to different notations.

Writability describes the ease with which a programming language can be used to create, or *write*, a program that solves a specific problem. Lesser cognitive load inflicted on the developer by getting the syntax right allows to concentrate on the correctness of the program logic. Abstraction and expressiveness also lessen the amount of code to be written and reviewed.

Reliability describes a programs ability to perform its function under all conditions. Exception handling helps to create programs that can recover from unforeseen occurrences, type checking ensures the validity of the input and interfaces.

This programming language evaluation scheme allows to assess the general quality and usability of a language. Suitability for specific domains can be deducted by weighing the relative importance of the criteria, but the focus of the method is clearly on evaluating the language, not its fitness for specific tasks.

TABLE 1. PROGRAMMING LANGUAGE EVALUATION CRITERIA

Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	✓	✓	✓
Orthogonality	✓	✓	✓
Data types	✓	✓	✓
Syntax design	✓	✓	✓
Support for abstraction		✓	✓
Expressivity		✓	✓
Type checking			✓
Exception handling			✓
Restricted aliasing			✓

3.3. Project-Based Programming Language Evaluation

Programming language evaluation criteria, like the ones described above, are based solely on characteristics inherent in a language, but the specific needs of a project are not represented. Therefore, in addition to the classic language evaluation, an evaluation specific to the project is needed [12]. Howatt proposed such an evaluation scheme, but it was never expanded beyond a basic description of the idea.

The criteria for the project-based evaluation would be defined by the software developers during the specification or architectural phase of a project. These criteria would describe the demands of the specific project on a programming language. The format would include:

- the criterion: a description of the quality to be measured
- the importance of the criterion to the specific project
- the degree to which a language satisfies the criterion

This approach allows to consider the practical details of a project and thereby appends the more theoretical approach of the classic Language Evaluation. The defined format also forces to reevaluate languages for each project, helping to provide a rational to find the suitable languages, instead of always using the familiar ones. Language familiarity however is explicitly not disregarded as decisive factor: the benefits of a new language always have to exceed the cost of switching.

4. Method

In order to evaluate MicroPython for CubeSats, different strategies are applied: a language evaluation based on Sebesta [15], a project-based evaluation based on Howatt [12], an analysis of the available programming tools and the comparison of example implementations. The evaluation happens within the MOVE-II CubeSat project, ensuring that the examples are realistic and within the domain of satellite development.

4.1. Programming Language Evaluation

This evaluation focuses on the classic language evaluation criteria described by Sebesta [15] by performing a survey to compare the readability of the Python programming language with C.

The survey consists of nine examples, each with an implementation in Python and C. The participants are asked to answer questions regarding each implementation and the time spent on each question is measured. The comparison of times spent on the implementations of each example can then be compared to judge the readability of the code snippet. As each participant sees each example twice, once for each language, two versions of the survey were produced, with the order of the example implementations switched. A self assessment of the proficiency in the languages is asked beforehand, alongside some demographic data.

4.2. Project-Based Evaluation

The project-based evaluation tries to implement the proposed strategy described by Howatt [12]. To do so, a set of criteria originating from the needs of the development of the Attitude Determination and Control (ADCS) subsystem daemon for the MOVE-II CubeSat is established. The ADCS uses magnetometers, gyroscopes and sun sensors to determine the

CubeSats attitude. Magnetorquers are used to create magnetic fields acting against the earths magnetic field, allowing to stabilize the satellite and point the antenna towards the ground. The ADCS consists of a mainpanel with the main microcontroller, four sidepanels and a toppanel. Each panel has a coil to generate the magnetic field, sensors and a secondary microcontroller. The mainpanel microcontroller controls the other panels and is itself controlled by the Command and Data Handling Unit (CDH). The CDH has a microprocessor running a Linux based operating system. The ADCS subsystem daemon runs on the CDH and enables controlling the functions of the ADCS subsystem by exposing D-bus methods. D-bus is an interprocess communication (IPC) system, providing a mechanism allowing applications to transfer information and request services [16]. These are either called by the on-board control program or remotely via the S-band communications link. The communication between the ADCS daemon and the ADCS subsystem is done via SPI.

4.3. Toolchain Analysis

Programming tools can support developers in writing software and help to enforce quality standards, especially in complex projects with multiple developers. The availability and quality of such tools and resources for MicroPython is analyzed.

4.4. Example Implementations

Different examples are taken from the MOVE-II source code and are re-implemented in Python. The two implementations are then compared in terms of complexity, error rate and performance. In cases where a missing library has to be implemented, the workload of doing so is also analyzed. Furthermore, the creation of performance optimized libraries for Python which are written in C is explored.

5. Results

5.1. Programming Language Evaluation

A first run of the survey was conducted with nine participants. Due to the small sample size, no definite conclusions can be drawn, still the results show a pattern worth investigating.

A bar plot can be seen in Figure 1. The mean times spent on each example per language show that the Python examples where generally processed quicker. A big standard deviation is present for which two reasons are already known: Firstly, the prior knowledge is not yet considered in the analysis. Secondly, the survey participants where not told that the time spent on each example is measured, as to not induce stress. However, their smartphones were also not taken from

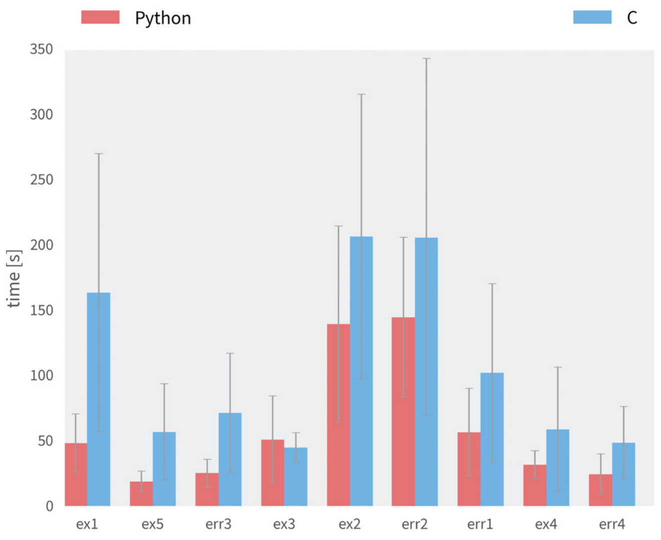

Figure 1. Mean of times particpants spent on answering questions for each example implementation with standard deviation.

them and thus posed a distraction. Participants would pause work on the example to response to messages, which renders the time measurement invalid. Both issues can be addressed by advanced data analysis, but only fixed by increasing the sample size.

5.2. Project-Based Evaluation

MicroPython, Python, C and C++ were compared according to the criteria in Table 2. While Python shows it strengths in usability, it falls short in efficiency, where C shines. MicroPython may fix those shortcomings, but for a clear statement there is not enough information yet, which is why the comparison of example implementations are needed. MicroPython also comes with its own caveats in the form of missing libraries for D-bus and SPI on the Linux platform. Implementation of the missing parts is possible and relatively straightforward, but may have to be done in C. C++ addresses many of the shortcomings of C and is the clear leader in this comparison.

5.3. Toolchain Analysis

A MicroPython interpreter exists for Linux on ARM as well as for microcontrollers using ARM Cortex M series or Tensilica cores. A full implementation and documentation of the Hardware API only exists for the STM32F4 family of microcontrollers from STMicroelectronics and the ESP8266 microcontroller. Currently, API implementations for Atmel SAMD21 as well as Kinetics MK20DX are being developed. Porting is generally possible to all platforms for which a C compiler is available and that have sufficient storage and memory, the main challenge being the implementation of the Hardware API.

TABLE 2. PROJECT-BASED EVALUATION CRITERIA

Criterion	Importance	MicroPython	Python	C	C++
The language enables memory safety.	++	++	++	– –	✓
The language enables creation of programs maintainable by changing developers.	++	++	++	–	✓
The language enables splitting tasks between multiple programmers.	++	++	++	–	+
The language enables creation of programs with a small storage footprint.	+	+[a]	– –	++	++
The language enables creation of programs with a small memory footprint.	+	+[a]	– –	++	++
The language enables creation of efficient programs.	+	+[a]	✓	++	++
The language enables creation of programs that can be updated using small diffs.	++	+	+	+	+
An implementation of the language for the platform in use exists or can be created.	++	++	++	++	++
A library to use dbus on the platform in use exists or can be created.	++	✓	++	++	++
A library to use SPI on the platform in use exists or can be created.	++	✓	++	++	++
The language enables quick and easy file system access to read and write files.	+	++	++	+	+
The language provides quick and easy means to parse data.	+	++	++	✓	✓
The language provides quick and easy means to handle strings.	+	++	++	✓	✓
The language provides quick and easy means to handle C-structs.	+	++	++	++	++

+ marks strengths, - weaknesses and ✓neutral fullfilment of the criterion relative to the other languages as perceived by the team

. [a]trait under investigation in this paper

MicroPythons project structure and build system allow for easy modification of the interpreter, as well as addition of custom modules, and can be compiled with a fully Open Source toolchain. However the internal C API is not officially documented. Some examples and information are provided by the community, but the missing documentation is a clear weakpoint, as the possibility to customize and extend and the simplicity with which this can be done is a key factor.

Because of the different internal structure of MicroPython compared to CPython, Python modules developed for CPython in other languages than Python do not work with MicroPython. Pure Python modules work as long as their own dependencies are met.

As MicroPython adheres to the Python syntax, all Python tools can be used. This includes syntax highlighting in editors as well as code linting tools like Pylint. Pylint allows to enforce coding standards, for example naming conventions, line length, dead-code detection, and thereby aids readability and maintainability of code. It also provides error checking which helps addressing the problem of runtime errors in interpreted languages.

Python is a dynamically typed language, which also creates the risk of runtime errors. Mypy is a tool providing static type checking using type hints that are allowed in the Python syntax. In MicroPython, these type hints can even be used to compile functions to native assembler code, providing better performance.

For practical working with MicroPython on a microcontroller only the most basic tools are needed: a text editor and a serial terminal. Source code can be directly copied onto the microcontroller storage, which, when connected to a computer with a USB cable, acts as a Mass Storage Device

(MSD). The code copied onto the storage then gets compiled to bytecode on the microcontroller itself and is executed. Using a serial terminal application a read–eval–print loop (REPL) can be accessed, allowing to interactively type in code that gets executed by the microcontroller. The usual debugging cycle of microcontroller programming (write $\rightarrow$ compile $\rightarrow$ flash $\rightarrow$ run) is drastically shortened to just write and run. Note that when separating the byte-code compiler from the byte-code interpreter the REPL can not be used, therefore during development a version with integrated compiler and interpreter should be used, switching to the split system when testing begins.

5.4. Example Implementations

The example implementations were developed using a Raspberry Pi 1 Model B acting as CDH and a Pyboard acting as the ADCS subsystem. Figure 2 shows a simplified overview of the system. The UNIX part of the software developed on the Raspberry Pi also works on the actual CDH hardware, the microcontroller counterpart on the Pyboard however does not run on the real ADCS boards. The ADCS subsystem uses ATXMEGA microcontrollers with an 8-bit architecture which is not suitable to run MicroPython.

As the ADCS subsystem daemon uses D-Bus and SPI for communication and those libraries are not available for the targeted platform, the first step is to implemented those.

5.4.1. D-bus Library. A minimal D-bus library for the UNIX port of MicroPython was implemented in C, allowing MicroPython functions to be exposed as methods on the user bus. The underlying C library is sd-bus [20], which is also used in the C implementation of the daemon on the MOVE-II CubeSat. The library can thus be shared and the added size is only the size of the MicroPython bindings.

The following code samples compare the C and Python implementation of a D-bus interface. There is no function logic implemented, the input is simply passed back as output, to concentrate on the boilerplate code needed to create the D-bus interface.

When using sd-bus in C, to expose functionality on the D-bus, it has to be wrapped in a function like shown in Listing 1. The function logic can either be directly implemented in this interface function, or the interface can be separated from the logic by moving the logic into a external function that is only called from the interface function. The interface function parses the input data from the D-bus message and replies with with another message, containing the data created by the function logic.

Using the MicroPython D-bus library, as shown in Listing 2, no boilerplate code is needed. Only the function providing the functionality has to be defined. Type hints are used in the example, but can be omitted.

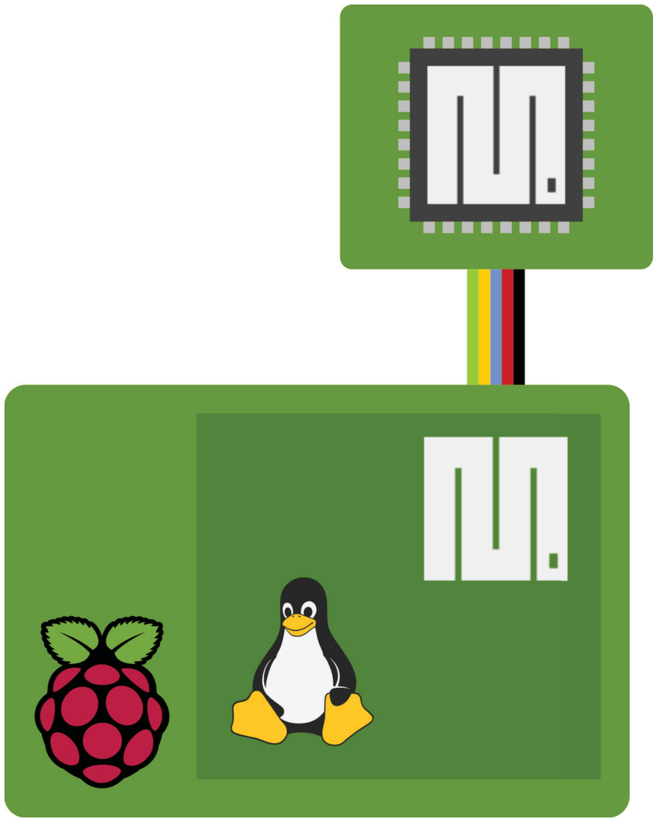
Figure 2. Schematic overview of the system. A Raspberry Pi running Linux an the UNIX port of MicroPython is connected to a Pyboard via SPI [17][18][19].

After having defined a function to be exposed as method, they have to be registered to the bus. In C, this means including the method in a `vtable` that is later passed to the object registered on the bus. Listing 3 shows the vtable. The first argument to `SD_BUS_METHOD` is the method name, followed by the type of the input ("y" means byte in the D-bus convention), type of output, the function pointer and the permission flag. The first and last entry of the table are standardized and provided by the library.

Listing 4 shows how this is done in Python by passing the function to the libraries `register` method. The two other arguments again are input and output data type.

In Python there is only one integer data type - `int` - as opposed to the more fine grained options C provides. This means that internally the MicroPython D-bus library handles all versions of integers with the Python `int` type.

Finally, the daemon has to connect to the bus and listen for calls. The snippet in Listing 5 has been shortened by removing the error handling to make it easier to grasp. The user bus is opened, the object added with the object name and object path and the `vtable` is passed. The name is requested on the bus and then the daemon starts listening on the bus in a loop.

The Python equivalent is shown in Listing 6. The two

Listing 1. Creating a D-bus method in C.

```
static int foo(sd_bus_message *m, void
  ↪ *userdata, sd_bus_error *ret_error)
  ↪ {
    uint8_t input;
    uint8_t output;
    int r;

    /* Read the input parameters */
    r = sd_bus_message_read(m, "y",
      ↪ &input);
    if (r < 0) {
        fprintf(stderr, "Failed to parse
          ↪ parameters: %s\n",
          ↪ strerror(-r));
        return r;
    }

    /* function logic */
    output = input;

    /* Reply with the response */
    return sd_bus_reply_method_return(m,
      ↪ "y", output);
}
```

Listing 2. Creating a D-bus method in Python.

```
def spam(inp: int) -> int:
    output = inp
    return(output)
```

arguments to `init` are the object name and path, the argument to `process` is the time in seconds the daemon should wait for requests on the bus.

The Python implementation is a total of eigth lines long, the C implementation about 10 times that. However, this is in part caused by the higher level of abstraction the MicroPython D-bus library offers. This library is implemented in C and the source code looks very similar to the code of the C program. For example, the `dbus.register` method does nothing more than dynamically inject the function pointer in the `vtable`. Similarly, `dbus.run` merely wraps what can be seen in the C `main` function.

5.4.2. SPI Library. Currently, the UNIX version of MicroPython lacks a hardware API. On UNIX systems, hardware access via device drivers is abstracted by a device file, allowing to interface by simple input/output system calls. Python-periphery is a pure Python library providing hardware access by using this device files and can be used with MicroPython after slight modifications [21]. Further changes were made to mimic the MicroPython hardware API, allowing all code written using this API to be portable between the implementation running on microcontrollers and on the UNIX platform.

Listing 3. Register a method in C.

```
static const sd_bus_vtable
  ↪ daemon_vtable[] = {
    SD_BUS_VTABLE_START(0),
    SD_BUS_METHOD("foo", "y", "y", foo,
      ↪ SD_BUS_VTABLE_UNPRIVILEGED),
    SD_BUS_VTABLE_END
};
```

Listing 4. Register a method in Python.

```
dbus.register(foo, 'y', 'y')
```

When the SPI communication is used between systems programmed in Python and C, the ctypes library allows native usage of native C types in Python.

5.4.3. ADCS Daemon. Using the MicroPython D-bus and SPI libraries, a re-implementation of the ADCS subsystem daemon is possible. Because the original C version of this daemon is currently under heavy development and unfinished, a stable subset of its functionality was extracted to define a target for the re-implementation.

In a first step, the C data structures used in the original daemon are recreated in Python. There are two structures:

- `control` is used by the daemon to send data to the subsystem which controls its functions. For example, the `mode` byte controls the operating mode of the subsystem.
- `data` is populated with sensor data and state information from the subsystem.

The structure definitions can not be shared across the languages, which means that two versions of the definition have to be maintained. Any differences in the definitions lead to broken data, making this a serious source of error for all systems where MicroPython has to communicate with C via such data structures. This problem can be addressed in two ways: Either by creating a tool that can translate the definitions between the languages or a tool that can automatically test the two structures against each other.

In the next step, the D-Bus interface is defined by creating the function bodies and register them to the bus. Then the actual functionality can be implemented. For this, the SPI library is used to transfer the data structures between the CDH and the ADCS subsystem.

In it's current form the C implementation of the daemon results in a binary with a size of 100 kB. The MicroPython implementation depends on the MicroPython interpreter, including the D-bus and SPI library, with a size of 350 kB in total. The Python code itself is 3 kB in size.

Comparing the length of the implementations, the C version with about 160 lines of code is significantly longer than the 80 line Python version. The Python version provides

Listing 5. Run the daemon in in C.

```c
int main() {
    sd_bus_slot *slot = NULL;
    sd_bus *bus = NULL;
    int r;

    r = sd_bus_open_user(&bus);

    r = sd_bus_add_object_vtable(bus,
            &slot,
            "/space/test",
            "space.test",
            daemon_vtable,
            NULL);

    r = sd_bus_request_name(bus,
     ↪  ADCS_OBJECT_NAME, 0);

    for (;;) {
        r = sd_bus_process(bus, NULL);
        if (r > 0)
            continue;
        r = sd_bus_wait(bus, (uint64_t)
         ↪  -1);
    }
    sd_bus_slot_unref(slot);
    sd_bus_unref(bus);
    return r < 0 ? EXIT_FAILURE :
     ↪  EXIT_SUCCESS;
}
```

Listing 6. Run the daemon in Python.

```python
dbus.init('space.test', '/space/test')

while True:
    dbus.process(1)
```

the same functionality as the original while being shorter, requiring less boilerplate code and being more readable.

6. Conclusion

MicroPython enables the use of Python in constrained systems and thereby can potentially bring the ease of use of this language to space computing. The benefits of using Python in space are tied to its focus on readability, which can help avoiding programming errors by increasing the code quality. The faster development speed the language enables can also help to more effectively use development resources. Especially in early stages, when systems are prototyped, this speed can help to create proof-of-concepts earlier and free time for additional design iterations.

The example implementations showed that MicroPython can meet the needs of software project for a CubeSat. The possibility to write libraries for MicroPython in C, and the facilities to interface with C data structures allow to integrate both languages in a common system, enabling the developer to facilitate each languages strengths and compensate their respective weaknesses. However, this interoperability still has some problems, not in function, but in usability: the lack of documentation of the internal C API aggravates the development of custom modules. Furthermore, the need for two separate definitions of data structures to be shared across the languages poses a possible error source.

This evaluations shows that usage of MicroPython for space projects is both beneficial and possible. In order to actually make it happen, further steps are needed by finding paths to build the needed familiarity with MicroPython and further progress the project without risking a mission. The first step is acknowledging that using MicroPython needs a commitment to work on the MicroPython project itself. After that, the development of Ground Support Equipment (GSE) and testing tools are an effective way to start building up know-how immediately. The language can also be used as a rapid prototyping tool during the proof of concept phase. Lastly, a test on a CubeSat in orbit where MicroPython would not act as mission critical infrastructure but act as a software scientific payload can quickly prove the concept.

Going beyond MicroPython, this evaluation hopes to enable bringing the language diversity that we enjoy in other programming domains to the world of space computing, while at the same time providing rationales to act against the rank growth this diversity may spur. Creating proper systems requires proper tools and a transparent way to chose those.

7. Acknowledgments

The authors acknowledge the funding of MOVE-II by the Federal Ministry of Economics and Energy, following a decision of the German Bundestag, via the German Aerospace Center (DLR) with funding grant number 50 RM 1509.

Bibliography

[1] JPL Special Review Board. “Report on the Loss of the Mars Polar Lander and Deep Space 2 Missions”. In: (March 2000).

[2] J.L. Lions. “Ariane 5 Flight 501 Report of the Inquiry Board”. In: (July 1996).

[3] E.E. Euler et. al. “The Failures of the Mars Climate Orbiter and Mars Polar Lander: A Perspective from the People Involved”. In: *Proceedings of Guidance and Control 2001* paper AAS 01-074 (2001).

[4] Daniel Dvorak. “NASA study on flight software complexity”. In: *AIAA Infotech@ Aerospace Conference and AIAA Unmanned... Unlimited Conference* (6 - 9 April 2009).

[5] M. Langer et. al. “MOVE-II – der zweite Kleinsatellit der Technischen Universität München”. In: *Deutscher Luft- und Raumfahrtkongress (DLRK)* (2015).

[6] Fuchs C. et. al. "A Fault-Tolerant Radiation-Robust Filesystem for Space Use". In: *Lecture Notes in Computer Science Volume 9017* (2015), pp. 96–107.

[7] Fuchs C. et. al. "Enhancing Nanosatellite Dependability Through Autonomous Chip-Level Debug Capabilities". In: *ARCS 2016; 29th International Conference on Architecture of Computing Systems* (2016), pp. 1–4.

[8] Fuchs C. et. al. "Enabling Dependable Data Storage for Miniaturized Satellites". In: *Proceedings of the AIAA/USU Conference on Small Satellites, Student Competition, SSC15-VIII-6* (2015).

[9] N. Appel et. al. "Nanolink: A Robust and Efficient Protocol for Small Satellite Radio Links". In: *Proceedings of the Small Satellites Systems and Services – The 4S Symposium 2016* (2016).

[10] MicroPython. *MicroPython Project Homepage*. URL: http://micropython.org/.

[11] G van Rossum. *Python Executive Summary*. 2004. URL: http://www.python.org/doc/essays/blurb.html.

[12] James Howatt. "A project-based approach to programming language evaluation". In: *ACM SIGPLAn Notices* 30.7 (1995), pp. 37–40.

[13] George Robotics Limited. *Porting of MicroPython to LEON platforms*. 2016.

[14] Python Software Foundation. *Python Software Foundation License*. 2016. URL: https://docs.python.org/3/license.html.

[15] Robert W Sebesta and Soumen Mukherjee. *Concepts of programming languages*. Vol. 281. Addison-Wesley Reading, 2002.

[16] Robert Love. "Get on the D-BUS". In: *Linux Journal* 2005.130 (2005), p. 3.

[17] Raspberry Pi Foundation. *Raspberry Pi is a trademark of the Raspberry Pi Foundation.* URL: http://www.raspberrypi.org.

[18] Andrew McGown and Josh Bush. *Tux is licensed CC BY-SA*.

[19] MicroPython. *MicroPython Project Logo*.

[20] Lennart Poettering. *The new sd-bus API of systemd*. URL: http://0pointer.net/blog/the-new-sd-bus-api-of-systemd.html.

[21] Vanya Sergeev. *python-periphery*. URL: https://github.com/vsergeev/python-periphery.