Increase avionics software development productivity using Micropython and Jupyter notebooks
Nicolas Valot, Pierre Vidal, Louis Fabre

To cite this version:
Nicolas Valot, Pierre Vidal, Louis Fabre. Increase avionics software development productivity using Micropython and Jupyter notebooks. ERTS 2018, Jan 2018, Toulouse, France. hal-02156167

HAL Id: hal-02156167
https://hal.archives-ouvertes.fr/hal-02156167
Submitted on 14 Jun 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
1 Summary

Requirement engineering is a key to understand avionics development productivity. In particular, requirement validation accuracy is considered as the main productivity driver.

In this study, we present a typical avionics software development life cycle, and propose a cost model for avionics software development. We provide a method to increase the productivity through early detection of requirement error. Ultimately we present a use case using some cutting edge tools (micropython & jupyter notebook), to achieve requirement validation objective, and increase the overall software life cycle productivity.

2 Introduction to avionics software development life cycle

In the scope of avionics software development, the ED-12C / DO-178C (ref [7]) guideline is used as a mean of compliance to software certification when involved in safety assessment. Some other guideline might be used for mission critical software (ex: military developments).

These guidelines specify software developments objectives, and life cycle data. The life cycle process is not described, but life cycle data dependencies are identified. Therefore, the life cycle process can be deduced from life cycle data dependencies.

A subset of the life cycle data is composed of:

- requirements (functional description)
- design (component breakdown, interfaces, low level requirements, behavior)
- code (source files, control coupling, data coupling)
- tests (integration and verification scenario, cases)

We need to define the error introduction and detection phase:

- error introduction phase: the phase during which the error is introduced.
- error detection phase: the phase during which the error is detected.
Error introduction / detection phase:
- requirement phase
- design phase
- coding phase
- test phase

The guideline also requires formal reviews of life cycle data, which is intended to detect error previously to test phase, and also provides a different angle of view which increases the overall error detection capability.

There are mainly two approaches to conduct reviews.

The first one is to review an item as soon as it is released. This provides the advantage to detect errors as soon as possible. On the other hand, the item coherency with other items is delayed after other items release, which complicates activities scheduling.

The second one is to wait until all other activities have been conducted, so that the overall coherency is enforced and provides a simple activities schedule. On the other hand, error detection is delayed, and review activity provides less value as some errors have previously been triggered by testing activity.

There exist some in between approaches to adjust pros & cons. Instead of considering reviews as an activity, we will consider the error detection phase.

In the case of a review scheduled as soon as possible, we will consider that the error detection phase is equal to the error introduction phase.
In the case of a review scheduled during testing phase, we will consider that the error detection phase is the test phase.

The life cycle data dependencies are expressed here with an arrow (A<-B means B depends on A):
- requirement <- design <- code
- requirement <- tests

### 2.1 Error cost model

This chapter purpose is to setup a cost model to demonstrate the advantage to detect errors as soon as possible in the development life cycle.

**Assumption:**
The number of introduced errors is proportional to the item size.

**Expansion/refinement ratio:**
For each link of impact tree, an expansion/refinement ratio represents the amount of details that are added through each impact tree link.

The workload of a software development is shared among development phase.
Here is an example of workload ratio per development phase, assuming that testing needs the half of the workload, and that requirement/design/code branch has an expansion factor of 2 for each refinement level:

<table>
<thead>
<tr>
<th>Phase</th>
<th>workload</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement</td>
<td>8%</td>
</tr>
</tbody>
</table>
The probability to detect an error during a phase is somehow linked to the time spent in this phase which is linear to the phase workload.

Let us assume that the probability to detect an error during a phase is equal to the phase workload ratio, which seems optimistic: the probability to detect an error should be higher during test phase, because the test phase is intended to detect error.

We also need to define impact outcome of error detection. The impact outcome depends on error introduction phase and error detection phase.

Assumption:
The impact analysis outcome concludes to some possible error recovery:
- one or several items in {requirement; design; tests} are erroneous and shall be updated
- the code is erroneous and the code shall be updated

Therefore we can enumerate for each detection phase, the corresponding impacts linked to any introduction phase, provided that introduction and detection phase respect life cycle data dependencies.

In the following tables, per detection phase, the rows identify the error introduction phase. The columns identify the impacted/updated life cycle data.

For each error introduction phase, the column probability is the probability that an error was introduced in the introduction phase $ip$, provided the error detection phase $dp$: $P(ip/dp)$. $P(ip/dp)$ is ideally set from statistics gathered on previous project developments. In our paper, these data are estimated through engineering judgment to evaluate the model.

The column cumulated workload ratio (CW) quantifies the effort to update impacted life cycle data, provided that 100% means the update of all life cycle data (requirement, design, code and test).

<table>
<thead>
<tr>
<th>Requirement detection phase:</th>
</tr>
</thead>
<tbody>
<tr>
<td>requirement</td>
</tr>
<tr>
<td>Requirement</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Design detection phase:</th>
</tr>
</thead>
<tbody>
<tr>
<td>requirement</td>
</tr>
<tr>
<td>Requirement</td>
</tr>
<tr>
<td>Design</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Coding detection phase:</th>
</tr>
</thead>
<tbody>
<tr>
<td>requirement</td>
</tr>
<tr>
<td>Requirement</td>
</tr>
<tr>
<td>Design</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test detection phase:</th>
</tr>
</thead>
<tbody>
<tr>
<td>requirement</td>
</tr>
<tr>
<td>Requirement</td>
</tr>
<tr>
<td>Design</td>
</tr>
<tr>
<td>Code</td>
</tr>
</tbody>
</table>
This shows clearly that the cost increases with the distance between detection phase and introduction phase. Considering an error introduced in requirement phase, and detected during test phase, which concludes that the requirement shall be updated, the cumulated workload is 72%. If we can detect this erroneous requirement during requirement phase, we save 72 - 8 = 64% of workload.

### 2.2 Requirement validation

The requirement validation activity is enforced in avionics by ARP4754A standard at system level. The validation shall enforce correctness and completeness of requirement definition. It is very unlikely that these properties are satisfied through a system requirement review.

At software level, upper (system) requirements are considered valid, and a refined as software requirements. These requirements are mainly verified through requirement standards and tests.

As we see in the introduction chapter, the probability to detect an error is higher at the end of the life cycle, than at the beginning for the following reasons:

- Testing is the phase that evaluates software requirements
- Reviews are usually performed lately
- Reviews are unable to detect complex defects (cross requirement/code modules interactions, behavioral/dynamic properties, hardware coupling)

If we can test software during system and software requirement phase, we can detect error earlier and foresee some workload reduction.

Some COTS product (example ref [8]) enables the user to perform a symbolic execution of formal requirements with various random input vectors, in order to detect requirement errors.

In this paper, we will go a little further, assuming that errors will happen also in design, coding and test phase. We are studying the effect of using the python programming language to design, implement and test requirements on target system during requirement phase.

In particular, the *micropython* project gives us the ability to meet hard real time constraints. We did not see any limitation to implement any avionics requirement.

According to the study of several aspects of programming languages (ref [5]), we assume a productivity factor of 5 between python and C language.

Therefore, prototyping software in python require $\frac{28\%}{5} = 5,6\%$ of extra coding workload.

We will not address the use of python as the target implementation language.

### 3 Context

In the near future, some UAV manufacturer will develop industrial grade products to address missions that will require navigating in controlled air traffic, and therefore will have to comply with some dedicated regulation and safety requirements.
These UAV target price will not support aircraft development NRC level and RC level, and will require optimized SWaP solutions together with safety standards compliance. As an example, an industrial grade autopilot with redundant channels for UAV is priced around 20K€ (ref [4]), which is the same order of magnitude than the same function for a certified helicopter. The business case of UAV authorized in urban area and controlled air traffic will be several orders of magnitude below helicopter’s one. The same ratio is expected on avionics functions. Therefore, enhanced open sourced solutions are an opportunity to bridge the gap between UAV constraints, air traffic regulation and safety requirements.

Micropython (ref [1]) is an open source crowd founded project released in May, 2014. It has been targeted to tiny microcontroller for IoT embedded systems and is therefore optimized in size to fit into internal SoC flash and RAM. It is developed using C language, and supports a subset of python3 language. It is a ~50 KLOC project with a modular compile time module selection. On one hand, this all makes micropython suitable to be used in constrained embedded systems like UAV. On the other hand, this provides a good opportunity to raise the productivity level of aircraft and rotorcraft avionics.

Jupyter (ref [2]) is a notebook framework that enables to write HTML notes with WYSIWYG technics, and also write code, compile it, execute, and display the results in the same document. It can be viewed as a runnable document. This is widely used in the statistics, and in the deep learning domain. We are using it as a testing & reporting tool in avionics for rapid prototyping & validation purpose. It could be ultimately used to build formal verification artifact. We will evaluate the advantages and drawbacks of both tools in the scope of avionics domain. We will perform integration with derivate equipment from legacy IMA modules, developed in the scope of CORAC AME research project (ref [3]).

### 3.1 Productivity objective

We have the desire to improve the productivity on an A653 IMA target provided by our equipment supplier. The legacy tool suite is composed of IMA platform configuration tools, compilers, A665 FLS load builder, A615A dataloader (ref [6]). The tool chain is designed to enable ED-12C / DO-178C (ref[7]) certification. It is perfectly suited to an ED-12C / DO-178C life cycle for production grade software, which is not optimized for productivity (short development loops).

It is not suited to rapid prototyping and requirement validation on target. Thanks to micropython embedded in A653 partition, we change the productivity order of magnitude in the early development phase.

### 3.2 Micropython

We were looking for a cutting edge technology, with some characteristics suitable to avionics software:

- Low footprint to ease target integration and improve performance thanks to locality
- Low complexity
- Openness to be easily extended
- Design assurance artifacts (test coverage, source readability, documentation)
- Performance
- Robustness
- MIT license
Micropython owns very few external dependencies. Almost all the low level libraries are implemented and optimized in the project, which is a strong advantage in avionics, where the software development kits provided by equipment suppliers are usually quite poor, when they exist. It is easily compiled with any cross GCC toolchain.

Initially, micropython does not support powerpc target, but the porting effort is quite low. We used GCC 4.9 cross compiled for powerpc-unknown-eabi target and newlib standard C library. The total footprint is around 300 Kbytes, with almost all core modules integrated. Smaller footprint might be achieved if some modules are not required.

### 3.3 Jupyter

Jupyter open source notebook is designed as a webserver connected to some kernel. A kernel is a piece of code that executes the notebook code.

A kernel is usually a language virtual machine (python, java, ...). Some kernels are able to compile on the fly C, C++ language. This is used for deep learning GPGPU target.

The user is able to write a notebook in web navigator client, to compile that code, make the code be run by the kernel and browse the result in the web client and store it in an archive file, that might be reopened and rerun later.

This is a very convenient way to share scientific results, but also the process to obtain the results which let other people to tweak some parameters and generate derivate results.

### 4 Project

To enable requirement validation using python language, we need to integrate a python VM in the target system, which is a IMA target module embedding an A653 operating system.

The project will require developing an A653 binding layer, and some micropython built in functions binding to Arinc 653. It will also require adaptation to jupyter kernel to communicate with the remote IMA target computer.
4.1 A653 binding (pyA653)
As our target computer owns an A653 operating system, we need to implement an A653 binding in micropython. We implemented QUEUING and SAMPLING services, TIME services, and PARTITION services, plus the A653 Part 2 file system services.

The threading part was not required yet and would require a little effort due to the A653 static resource creation constraint. The A653 module offers strict A653 binding. We are able to receive/send any data to/from any avionics I/O and perform mass memory storage and retrieval through micropython script.

Here are simple examples of Arinc 653 service call performed with micropython. Each cell contains code that is sent to the target, computed by the target. The results are gathered and displayed below the code snippet.

4.2 Micropython libraries binding to platform
Micropython defines several libraries and built-ins that rely on platform capabilities. For instance, if we want to use the open, read, write built-ins, we need to implement the built-ins binding to A653 Part 2 services.

The micropython builtins offer pythonic way to access A653 resources and provide portability to micropython libraries.
4.3 Jupyter Notebook

In the context of our project, we want the target to run a micropython kernel so that our notebook executes the code on target.

As our target embeds micropython, we need a micropython kernel to be compiled on target. We did not find any of these in the open source community. Therefore, we derived a host python3 kernel to remotely send commands to the remote micropython target on IMA equipment and get back the results. This enabled with a reduced effort to compute notebooks remotely on a target that is not jupyter aware.

It would be convenient to share data between the host kernel and the remote kernel in the same jupyter notebook, which is not possible with jupyter, because it binds a notebook to a single kernel.

4.4 Multi target notebook

We modified the host kernel to get back the remote kernel answers in the host kernel global variables. We then have a multi target jupyter notebook.

Assuming that we have several target equipment embedding a micropython kernel, we could then perform system testing from a single jupyter notebook. We defined specific tokens to switch between host (>py>) and target (>mp>) kernel in the notebook. The communication between the host kernel (python3) and the target kernel (micropython) is performed using a global variable named ‘_’. When we switch from the host to the target, the ‘_’ variable is evaluated and sent to the target as a script: (_ = <value> )

As the target computer does not contain any battery, it cannot maintain a real time clock. In the aircraft, the RTC is provided by a specific computer, or GPS. On bench, we need to synchronize remote target with host computer. This is quite easy using python commands to get local time on host and send a micropython script to set remote target RTC.

5 Requirement engineering and validation use case

To validate the principle, we provide here an example of a high level requirement that leads to a complex breakdown and needs a lot of effort to get it developed right.

5.1 Engineering

The system shall provide a remote rectangle range request capability to a DEM persistent data storage, in O(log(n)) where n is the number of DEM values.

Note: the DEM might be used to display a SVS, DMAP, or perform HTAWS, or any kind of function that require terrain elevation data.

At software level, this involves a lot of refinement to be able to implement and verify this system requirement. In particular, ‘remote’ means network interface, ‘persistent’ involves mass memory and file system, ‘O(log(n))’ means both an optimized data structure, and a bounded time request, ‘rectangle’ means a set of geo referenced values.

The persistency and O(log(n)) access will require a balanced tree indexing of the nodes in a persistent file system. The rectangle range request capability would take benefits of the use of rectangle tree organization, which adds more complexity than balanced tree. As elevation data are
equally distributed on earth, rectangle tree do not bring a true advantage over balanced tree. Balanced trees provide a single key value. We need to compute an \((x,y)\) coordinate to a single key, with the constraint that key ordering respects geo locality. We also want to minimize the database footprint to improve runtime performance. This will require data compression.

5.2 Validation means
To refine and validate software requirements, we have set up the architecture (Figure 1: ). We also need a data source for testing purpose. The SRTM mission provides 3 arcsec resolution data (ref [9]).

The SRTM data is retrieved as an ascii elevation tile files. These files are pre-processed to generate *micropython* scripts that will load the target database.

We will use *micropython* as the request language through a dedicated API and Berkeley DB as the database index, *zlib* component to uncompress data.

We also traced the file system read call in order to monitor contributors of execution time.

5.3 Integration, analysis and outcome
With the legacy tool chain, even with a good automation, it requires several minutes to a couple of hours to add/modify some requirements implementation in an embedded application and run the corresponding test. With *micropython*, it requires several seconds to do so, thanks to REPL.

After executing the tests, we were able to retrieve DEM tiles corresponding to lat/long coordinates, and check their contents using an independent data source. We were able to evaluate the requests execution time on target platform thanks to file system call traces, showing a lot of discrepancy with the initial requirement.

The defect analysis concluded that the root cause was due to the file system unpredictable latency. It led to a decision choice: either relaxing the real time request constraint or changing the file system implementation.

Another outcome was that the translation of the \((x,y)\) coordinate to a single key was erroneous and would not respect geo localisation of database records.

6 Conclusion and future work
The whole requirement engineering and on target validation cycle took one week, to find dozens of errors in requirements, design, coding and tests. It would have taken months of DO178 development to find the same outcome plus a couple of weeks to recover the defects.

Thanks to *jupyter* and *micropython*, we raise the productivity of early requirement validation and verification on any avionics platform with several contributors:

- Early detection of specification errors, design errors, coding errors, computing platform usage limitations
- Programming language productivity (*5*)
- Interactivity: fast loop (what, how, run, check)
- Training: python scripting and *jupyter* web browsing require minimal software knowledge
Tooling: do not require complex IT infrastructure and COTS licensing

Beside productivity topics, we can go further and evaluate the effort to bring design assurance level evidence to open source software, which will enable safety assessment of UAV avionics together with a reduced addition of effort to preexisting well designed open source products. 

*MicroPython* demonstrates lots of qualities to be widely deployed in the avionics domain. Some commercial grade solutions are derived from *micropython* project.

Some static analysis tools might be used to evaluate accuracy and consistency of the code components.

The coding style is self-explanatory to ensure that a wide community of developer is able to maintain the project.

The test coverage achieves 98% instruction coverage, although the product does not provide requirements (required by *DO178* objectives).

In essence, open source are test driven development, and it might be straightforward to reverse engineer requirement from tests and user’s manual and ultimately perform requirements to code traceability.

Configuration management is planned and performed using cutting edge reliable tooling. The change process is defined and controlled by the *github* project owner team.

When we look at existing UAV avionics products with some design assurance level, the hardware looks cheaper than aircraft of rotorcraft avionics: cost drivers are not included in these products (EMI shielding, lightning protections, high end connectors, and robust power supply). Their price is mainly driven by non-recurring development cost, which could be widely reduced by the use of some robust open sourced solutions during development life cycle.
7 References

[1]. https://en.wikipedia.org/wiki/MicroPython
[2]. https://jupyter.org/
[4]. https://products.embention.com/veronte/uav-autopilot/4x-redundant
[5]. A Comparative Study of Programming Languages in Rosetta Code [Sebastian Nanz _ Carlo A. Furia Chair of Software Engineering, Department of Computer Science, ETH Zurich, Switzerland firstname.lastname@inf.ethz.ch]
[7]. https://fr.wikipedia.org/wiki/DO-178

8 Acronyms:

AME extended modular avionics
DEM digital Elevation Model
DMAP digital MAP
FLS field loadable software
GPGPU general purpose graphical processing unit
HTAWS helicopter Terrain Awareness and Warning System
IMA integrated modular avionics
IoT internet of things
KLOC kilo line of code
NRC non-recurring cost
RAM random access memory
RC recurring cost
REPL read evaluate print loop
SoC system on chip
SRTM Shuttle Radar Topography Mission
SVS Synthetic Vision System
SWaP size weight and power
UAV unmanned aircraft vehicle
WYSIWYG what you see is what you get