Experiences in Building Python Automation Framework for Verification and Data Collections

Juki Wirawan Tantra
jw.tantra@gmail.com

Abstract
This paper describes our experiences in building a Python automation framework. Specifically, the automation framework is used to support verification and data collection scripts. The scripts control various test equipments in addition to the device under test (DUT) to characterize a specific performance with a specific configuration or to evaluate the correctness of the behaviour of the DUT. The specific focus on this paper is on documenting our experiences in building an automation framework using Python: on the purposes, goals and the benefits, rather than on a tutorial of how to build such a framework.

1. Introduction
I work in a growing IC design company that focuses on wireless solutions. The project described in this paper was developed when we were developing the fifth generation of our product. We were facing huge challenges for this project: our product's feature sets have grown so much that manual verification of the features require more than 2 months (and this would involve most of the engineering team doing manual tests).

Recent economic situation prohibited any further investment, both for manpowers and tools. However, the project needs to be completed on time, or we would risked losing a significant potential business. We basically faces a risk of taping out our Application Specific Integrated Circuit (ASIC) with potential major defects due to untested features.1

Understanding this risk, we did an evaluation on how much can we gain by automating a number of crucial test cases. It was obvious that the benefits were significant: we would have 16 more hours each working day2 and two extra days on weekends to run tests and data collections---an extra of 128 hours a week to execute tests. The extra hours are more than the actual employee's working hours in one week. Our existing software was developed with

---

1 Following the common inclinations to cut verification time when development time is tight. Not advisable, not recommended, and not suggested...

2 Assuming 8 working hours per day. Disclaimer: assumptions might reflect real-life poorly.
C/C++ (not surprisingly); this basically limits what we can achieve in such a short time period. Here Python comes to the rescue.

This paper focuses on documenting the experiences that we have on developing an automation framework with Python. The contents of this paper focus more on how the project was developed and brought to a fruitful end, rather than providing a tutorial on how to build the automation framework.

2. Purpose and Goals of Our Automation Project
With this automation project, we are heavily constrained by time and resources. We knew early on that we need something that other engineers can pick up very quickly, so that more people can contribute to the progress of the automation project by various means. We have at that time a few hundreds test cases that need to be executed across a (not insignificant) number of configurations. We argued that if anyone (with minimum explanations) can use the framework and start automating the test cases, we would have better progress due to more contributions.

The ultimate goal of our automation project was quite simple: reduce the verification time and potential defects by automating as many test cases as possible. To achieve this, we realized that we needed to develop an easy-to-use framework so that many people can contribute in the development of the scripts.

What did we consider as the goals of the automation project? We put several items of priority as the project goals:

a) Easy-to-use framework based on Python that many engineers can start using immediately.

b) Reduce the required test time by automating all test cases.

c) Extensibility of the framework to support various use: scheduled run, control of various equipment, and remote control/execution of the script.

At the end, what matters most will be whether we can achieve a significant time saving and verification coverage through this project.

3. Why Python?
To us, Python was an obvious choice due to several reasons:
3.1. No compilation necessary
No compilation means that we can use source code repository system such as CVS or SVN to store our test scripts, ready to run any time. We will not have problem identifying whether the script that we run is the latest and the greatest. No one likes to manage the compilation and the repository for hundreds of small executable. And having one big executable will hinder our development as there is a need for someone to glue the small pieces together, and to enforce safety and quality rules.

No compilation also means that we can set up development and test machines equally and easily. This doesn't sound like a significant advantage, but to us, this is of great importance. We can set a PC up just by executing several installers; and that machine is ready for rapid debug-fix cycles often necessary. And note also that test equipments are usually attached to specific PCs due to difficulties in moving them around.

3.2. Python is easy to read and is very flexible
We need to get as many people as possible to start writing test scripts. Easy to read means that it is easy to learn from examples and existing scripts. From this project, we learnt that once the momentum started, the scripts were adding up pretty fast.

Python is also very flexible, not limiting anyone to specific programming paradigm, e.g. object oriented. Frankly speaking, object oriented paradigm is not easy to pick-up by non-programmer; and forcing object oriented paradigm on test scripting, where most of the operations consist of a) do this, b) and do this after, is just plain silly.

3.3. Python is mature with much supports behind it
We need to have something ready fast. Python with so many libraries built for it really helps. For example, to interface with test equipments, we need to use VXIplug&play I/O software language (VISA) protocol [6]; conveniently, there is PyVISA [4] ready for our use.

4. Automation Project: Description and Challenges
In our automation project, we built an infrastructure to make it easy to script test and data collections. We need to work with various test equipments, for the purpose of various measurements and verification. We also need to perform various controls on our Device Under Test (DUT).

Thus our automation project deals with three main components:

1. Equipments control using PyVISA.
2. Connectivity to the existing C++ software to control the DUT.
3. How to make scripting easy.
Our automation project produced an infrastructure that glues these three components. And Python is the main ingredient to each of these three components, and also is the glue that brings the three components together.

4.1. Equipment control using PyVISA
VISA [6] is the interfacing protocol adopted by most of the test equipment vendors. Using VISA protocol, communicating and controlling various test equipment become much simpler as we need not deal with the lower layer protocol. With VISA, we can focus on the control and the results processing.

PyVISA [4], written by Torsten Bronger, provides the necessary interfacing of VISA driver with Python. Thanks to Torsten Bronger, when we want to write a data collection script in Python, we just need to install a VISA driver and PyVISA. After we connect the equipment (through one of the many supported VISA interfaces: GPIB, USB, LAN, etc.), we can immediately take control of the equipment using Python.

In our project, we need to develop a higher layer abstraction for each test equipment that we use so that it is easy to use even for non-software programmers. The following snippet shows a minimized example of the wrapper for Agilent mixed-signal oscilloscope (MSO), and how to use it in Python.

```python
class AgilentScope(VisaDevice):
    LARGE_TIME_OUT = 50
    NORMAL_TIME_OUT = 10

    def read_setup(self):
        """
        Read the MSO setup and return it as a string.
        """
        self.set_timeout(AgilentScopeConnector.NORMAL_TIME_OUT)
        setup = self.ask_raw(":SYSTEM:SETUP?")
        setup = self.strip_term_chars(setup)
        return setup

    def write_setup(self, setup_settings):
        """
        Write the previously saved MSO setup.
        """
        self.set_timeout(AgilentScopeConnector.NORMAL_TIME_OUT)
        self.write(":SYSTEM:SETUP %s" % (setup_settings,))

    def __read_image(self, setting):
        self.set_timeout(AgilentScopeConnector.LARGE_TIME_OUT)
        img = self.ask(":DISPLAY:DATA? %s, SCREEN, COLOR" % (setting,))
        self.set_timeout(AgilentScopeConnector.NORMAL_TIME_OUT)
```

3 VISA drivers are provided by many of the test equipment vendors. Agilent and NI are two of the vendors providing VISA drivers and tools for free download [1], [2].
return img[10:]

def read_bmp_image(self):
    """
    PNG is faster.
    """
    return self.__read_image("BMP8bit")

def read_png_image(self):
    return self.__read_image("PNG")

def enable_persistence(self):
    self.write (":DISPlay:PERSistence INFinite")

def disable_persistence(self):
    self.write (":DISPlay:PERSistence MINimum")

And example of how to use the above class:

scope = AgilentScope()
#settings_variable is loaded from a file
scope.write_setup(settings_variable)
image1 = scope.read_png_image()
scope.enable_persistence()
image2 = scope.read_png_image()

In the above example, I load the scope configuration from a file (previously made by the read_setup). After that, I read out the MSO's display image two times. The first is with default setting (with persistence off), and the second one is with persistence on. This is a very simple example of what you can do with PyVISA controlling a test equipment. We can do quite complex computation here. We can basically read out all the measurement data from the scope, and process them point-by-point in Python, or we can use the MSO's functions, e.g., to measure delay/distance between two edges. An example of the saved image is in the following figure.

![Figure 1: Image of sine-wave audio captured using Agilent MSO.](image-url)
It looks easy; in real-life it couldn’t be easier. We have our test engineers, without real programming background, starting to write data collection and test scripts with very little explanations. What they need is only a decent easy-to-use text editor (Notepad++ [3] comes to mind).

At this point of time, we have built abstractions for a number of test equipments: mixed-signal oscilloscope (MSO), spectrum analyser, signal generator, digital multimeter, and DC power supply. I think that you can pretty much imagine what we can do with all these equipment controls. Add ease of scripting, and we have a very usable platform, capable to fulfil all your test and data collection dreams.

4.2. Connectivity to the existing software
Other than controlling the test equipments, we also need to control the DUT. Our existing software is implemented in C++. We need a simple mean to control the C++ software from Python.

We decided to use socket for the communication between the C++ software and the Python automation framework. This socket will be used to pass commands and information, supporting the required capabilities of controlling multiple DUTs from one script.

In the Python implementation itself, I decided to use the basic socket library rather than the very useful Twisted framework [5]. We have two reasons for this decision:

1. One less installation to care about.
2. Event driven programming is harder to pick up by developers used to procedural programming.

In the hindsight, these reasons do not really matters in our project. One less installer does not matter much. And the implementation of our communication protocol's lower layer (i.e., socket) was so short (less than 200 lines of code) written in Python that it wouldn't matter whether we use socket or Twisted in our project.

4.3. How to make scripting easy?
We knew early on that the scripting must be easy for this project to be successful. So the real question was how easy can scripting in Python be?

In the early stage, the basic necessity was to control DUT(s), test equipment(s), and to save results to a file. We developed a number of higher layer abstractions to make these tasks...
easy to script. The end result is a highly abstracted operations such as in the following snippet.

dut1 = Dut(0)
dut2 = Dut(1)
scope = AgilentScope()
scope.write_setup(settings)
dut1.write_register(REGISTER_NAME, value1)
dut2.write_register(REGISTER_NAME, value1)
image1 = scope.read_png_image()
dut1.write_register(REGISTER_NAME, value2)
dut2.write_register(REGISTER_NAME, value2)
image2 = scope.read_png_image()

The above shows a simple capture of the MSO display after each change to the DUTs. This example was written just to show the basic operations possible with several lines of code. There are several imports, logging and debugging facilities provided in the framework. In most use cases, the script developer copies from one of the existing templates (which include the main, imports, and output file functionalities), and modifies the content accordingly.

From my experience, this facilities are good enough for most people to start working with. Many of the easier scripts can be done by anyone. And the more difficult scripts are developed with assistance from the software developers. Some of you might wonder why not just have the software developers write the scripts. The burden on the developments of test scripts are not only on the scripting itself, but with all the set ups, testing, equipment controls, and the understanding of the actual system behaviour. And the domains that need to be covered are much wider than just actual software behaviour. Thus, it's either that the respective engineer writes a specification for the software developer, or write the scripts directly themselves with some assistance from the software developer. I believe that the second approach requires much less time in some environments (such as ours), as it cuts down misunderstanding on the specification or the expected behaviour and the requirements of a more rigid procedure for test case development.

5. Automation Project: Results and Experiences

At the time this paper is written, I am happy to say that the automation framework is extensively used for various uses in the whole of our engineering department. The time saved is just too significant to ignore. I admit that at the beginning many if not most of the engineers are sceptical of how the automation framework can help their work. However, after witnessing how efficient or how much faster that many things can be done using the framework, many people just started to use. And they started to expect more and more measurements or evaluations to be automated.

Just to give a rough idea on the time spent to develop the automation framework that we have. The basic framework supporting DUT controls and some basic libraries to make it
easy to develop scripts took less than 2 weeks of one engineer. (Actually, it was quite usable after the first week). The development time for the equipment supports varies depending on the complexities and the number of operations required. However, most of the equipments require less than three days to develop for, including example scripts and testing with the equipments themselves.

In my opinion, the 2 weeks required to build the basic framework is a time well spent. Having the basic framework up, the rests build up pretty fast with more people working on them. The time saved from having this automation framework is substantial. Just to give an example, the current measurements using digital multimeter take one engineer 1 week just to collect all the data with the numerous possible configurations. With the automation framework, the same measurements took 24 hours (1 day and 1 night---another benefit of having the automation framework).

The following table illustrates the difference in efficiency with some examples (assuming 8 working hours/day). Note the differences in the measurement units. Automated test can be executed at night time and weekends.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Manual</th>
<th>Automated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Audio defect test</td>
<td>5 man days</td>
<td>40 hours (&lt; 2 days)</td>
</tr>
<tr>
<td>Power consumption measurement</td>
<td>3 man days</td>
<td>24 hours (1 day)</td>
</tr>
<tr>
<td>Modem sensitivity test</td>
<td>5 man days</td>
<td>40 hours (&lt; 2 days)</td>
</tr>
<tr>
<td>Interference robustness test</td>
<td>5 man days</td>
<td>40 hours (&lt; 2 days)</td>
</tr>
</tbody>
</table>

The automation framework also opens up a number of new possibilities. Some measurements that in the past were thought of to be too time-consuming and too tedious to do are now easily done with some (arguably more complex) scripts. However, the benefits of this cannot be measured easily; one can argue that by having these new possibilities, we actually open up more opportunities for better design and engineering.

Fairly speaking, all of the capabilities of the automation framework described in this paper can be implemented with other tools: C, C++, Visual Basic, .NET, etc. However, Python offers a mixture of advantages that are crucial for this automation framework purpose: ease of use, lightweight, doesn't require compilation, and free to use. The readability of a Python script makes it easy for any engineer to learn by examples. I believe Python will play a key role in this field not far in the future. And I would heartily recommend Python for similar use in your organization.
6. References