Concept of a PDS4 product writer library and a python demonstrator

D. Fraga
ESA/ESAC

Introduction

Presented here is both the concept of a generic and configurable library to write PDS4 compliant products and a **prototype** of a specific implementation as **a Python 3 package, which is now available as open source**. The proposed tool is a PDS4 product writer meaning that **it generates both the data file and the PDS4 label file**, as opposed to other tools that help in the generation of the label only.

How does it work?

The tool is intended to be used inside a processing pipeline or other data handling software. It writes the data that is in the computer memory into a PDS4 compliant data file and also writes its PDS4 label file, ensuring the consistency between the two. Only the essential information is passed to the tool and the tool fills in many attributes that can be automatically determined (e.g. the *offset*, *records* or *record_delimiter* etc).

A PDS4 label is an XML file with two very different parts: The so called *File Area* contains a self contained definition of the data format and other file related metadata. The rest of the label contains scientific or technical metadata about the data (the *Identification Area*, *Observation Area* etc.) The tool generates automatically and transparently to the data producer the *File Area,* without requiring a template. The rest of the label is more case specific and cannot be generated transparently to the user. Nevertheless, the tool provides a mechanism to assist in the rest of the label's generation: it receives as input a template (without the *File Area*) and the user can set the attribute values calling one method.

Its use has two steps:

1. Configure the tool (define a product type):
   1.1 Tell the tool if products will have ASCII tables, arrays etc. and define all details that cannot be determined automatically (e.g. name of the table).
   1.2 Provide a template for the label not including the *File Area*.
2. Generate the products:
   2.1 Pass to the tool the data you want to write. The tool will write it in a PDS4 compliant data file, generate automatically the *File Area* of the label and ensure consistency between the two.
   2.2 Fill in the metadata out of the *File Area* using the provided and simple mechanism.

What problems does it intend to solve?

As a Rosetta Archive Scientist at ESA/ESAC the author has spent a lot of time helping the data producers debug their software and templates. I frequently see the same errors that could have been prevented through the use of a common and validated piece of software. The specific goals are:

- **Less PDS4 standards learning effort**: Why should a scientist learn all the details of the PDS4 representation of the data and the metadata in the files to produce a PDS4 compliant product? Do you have to learn these things to produce a Word or JPEG file?
- **Less validation effort**: Before going through an exhausting validation effort PDS products typically have many errors that have to be fixed.
- **More reliable products**: Often not all errors are detected and solved before releasing the products to the public. The tool aims to prevent them instead of detecting and fixing them.
- **More homogeneous products**: Similar data is always stored in the same way to make products more consistent.

Limitations of XML, schemas and schematrons

Validating the consistency of a label with its schema and schematron does not reveal anything about the consistency of the label with its data file.

Limitations of validation tools

It is not possible to build a validation tool that detects all errors. Some errors cannot be detected because there is nothing to check them against. See example in the figure.

Left: An image. Right: The same image file with an error in its label. The *sample_type* is incorrectly set to *msb_unsigned_integer* instead of *lsb_unsigned_integer*. These type of errors are undetectable in a fundamental way by current and future validators.

FAQ

What happens when the information model or the standard changes?

All changes affecting the part of the label outside the *File Area* do not affect the tool. All backwards compatible changes inside the *File Area* do not affect the tool. Only in the event of backwards incompatible changes in the *File Area* could the generated products potentially be incompatible with the new version of the standard.

Who are the potential users?

Future missions or missions in development from any agency. Users who intend to produce new products (e.g. high level products) to be submitted to archiving authorities such as the PSA or PDS. Users who do not mind foregoing some control on data format so as to gain production simplicity. Users who want to adopt PDS4 as their day to day format and not restrict it to archiving.

Who are NOT potential users?

People who have already produced the data files or have a pipeline to produce them and only want to produce label files. People who want to retain full control of all data format details at the expense of more production and validation complexity.

What abut FITS, CDFs and other third party standards?

It might not be the most straight-forward way to produce PDS4 compliant FITS products. However, since many FITS files comply PDS4 restrictions, it should be possible to produce a FITS writer that relays on the PDS4 writer for the physical writing. The role of the FITS writer would be to produce the FITS header and call the PDS4 writer.

The EasyPDS4writer prototype

EasyPDS4writer is a newly released open source Python3 prototype of the described concept. Currently it can write observational products with an arbitrary number of fixed width ASCII tables with any number of columns. Other PDS4 objects are not supported yet. It is being developed by Diego Fraga Agudo in a best efforts basis only. Under these circumstances, it is beyond the reach of the author to develop the entire tool, documentation and testing. **Do you want to contribute with code, testing or simply give your opinion or show your interest as a potential user? Contact the author at dfraga@sciops.esa.int.**

Available at the ESDC Github account (https://github.com/esdc-esac-esa-int).

EasyPDS4writer is an experimental piece of software and is not an official ESA tool.

Example of use

```
from pathlib import Path
import os
from easypds4writer.product_observational import ProductObservational

# Create a PDS4 product observational object and set a label template for it. The label template must not contain
# the File_Area_Observational which is generated automatically and transparently by the tool including all
# necessary tags and its values. If an empty template is used the products will still be readable by readPDS python
# package but will not be fully PDS4 compliant.
test_path = os.getcwd() # The test path is the path where this file is located.
pds4_product= ProductObservational(Path(test_path + "/example_templates/example_template.xml"))

# Now we will configure the tool to write a specific product type.

# Tell the tool that the product type has one fixed ASCII table called table_test
table_character= pds4_product.declare_table_character("table_test")

# Define (declare) the fields in the table. For each field you must specify its format, name, units and a
# description. Currently it is also necessary to provide the data type but the intention is to determine this
# automatically from the format except for data types of string type where it is not possible.
table_character.declare_field("%-23s",
                              "ASCII_Date_Time_YMD",
                              "PARTICLE DETECTION IN UTC",
                              "N/A",
                              "TIME OF GIADA DETECTION IN UTC")
table_character.declare_field("%6s",
                              "UTF8_String",
                              "DETECTING SUB-SYSTEM",
                              "N/A",
                              "THE SUB-SYSTEM OR SUBSYSTEMS WHICH DETECTED THE PARTICLE")
table_character.declare_field("%8.3f",
                              "ASCII_Real",
                              "PARTICLE SPEED BY GDS+IS",
                              "METER/SECOND",
                              "PARTICLE SPEED MEASURED BY GDS+IS")

# Define second table in the product
table_character2= pds4_product.declare_table_character("table_test2")

table_character2.declare_field("%-6.6s",
                               "UTF8_String",
                               "DETECTING SUB-SYSTEM",
                               "N/A",
                               "other")

table_character2.declare_field("%5d",
                               "ASCII_Real",
                               "PARTICLE SPEED BY GDS+IS",
                               "METER/SECOND",
                               "PARTICLE SPEED MEASURED BY GDS+IS")

# End of product type definition.

# Now create the first product...

pds4_product.new_product(Path(test_path + "/outputs/phys20160421t000000m_v1_0.tab"))

# Now fill in the first table with data. The inputs you pass will be validated against the definition given
# in declare_field and exceptions will be raised if they do not match.
# Typically this will be in a loop with one iteration per record (row) in the table.
table_character.add_record(("2016-04-21T03:31:34.958", "GDS-IS", 1.35))
table_character.add_record(("2016-04-21T08:39:07.969", "GDS-IS", 10.83))
table_character.add_record(("2016-05-22T08:39:07.969", "GDS-I", 123.56))
table_character.add_record(("2017-05-22T08:00:00.000", "GDS-I", 123.5678))
table_character.add_record(("2017-05-22T08:00:00.000", "GDS-I", 123.67))

# Fill in the second table
table_character2.add_record(("GDS-IS", 5))
table_character2.add_record(("GDS-IS", 12345))

# All metadata outside the File_Area of the label cannot be filled in automatically. You must provide a template for
# this part (done when creating the ProductObservational object) and now replace the place holders by the desired
# values.
pds4_product.set_metadata("$year", "2017")
pds4_product.set_metadata("$mission_name", "imaginary mission")

# Close_product method writes the label file (the data file was written already in each add_record call), closes data
# files etc and leaves the pds4_product ready to call new_product.
pds4_product.close_product()

# We can reuse the product type definition and write more products by doing the same again...
pds4_product.new_product(Path(test_path + "/outputs/phys20160421t000022m_v1_0.tab"))
[...]
# Close the product
pds4_product.close_product()

# You can reuse pds4_product object to write as many products as you want but you have to create a new object to
# write products of a different type.

print ("You should have two products in the output directory that in this example is " + str(Path(test_path +
"/outputs/")))
```

The author would like to thank Mark Bentley for his useful feedback and Eleni Ravanis for the poster proofreading.