party Documentation

Release 0.1.0

Guillaume Florent

Jun 14, 2017

Contents

Plain text (JSON) parts libraries for open hardware projects.

WARNING : **party** is currently in *work in progress* status

party aims at creating and handling parts libraries defined in a single text (JSON) file where the metadata, the data, the geometry creation logic and the *anchors* creation logic are defined.

The *anchors* concept is a set of vectors attached to a part geometry that can be used to place the part in an assembly.

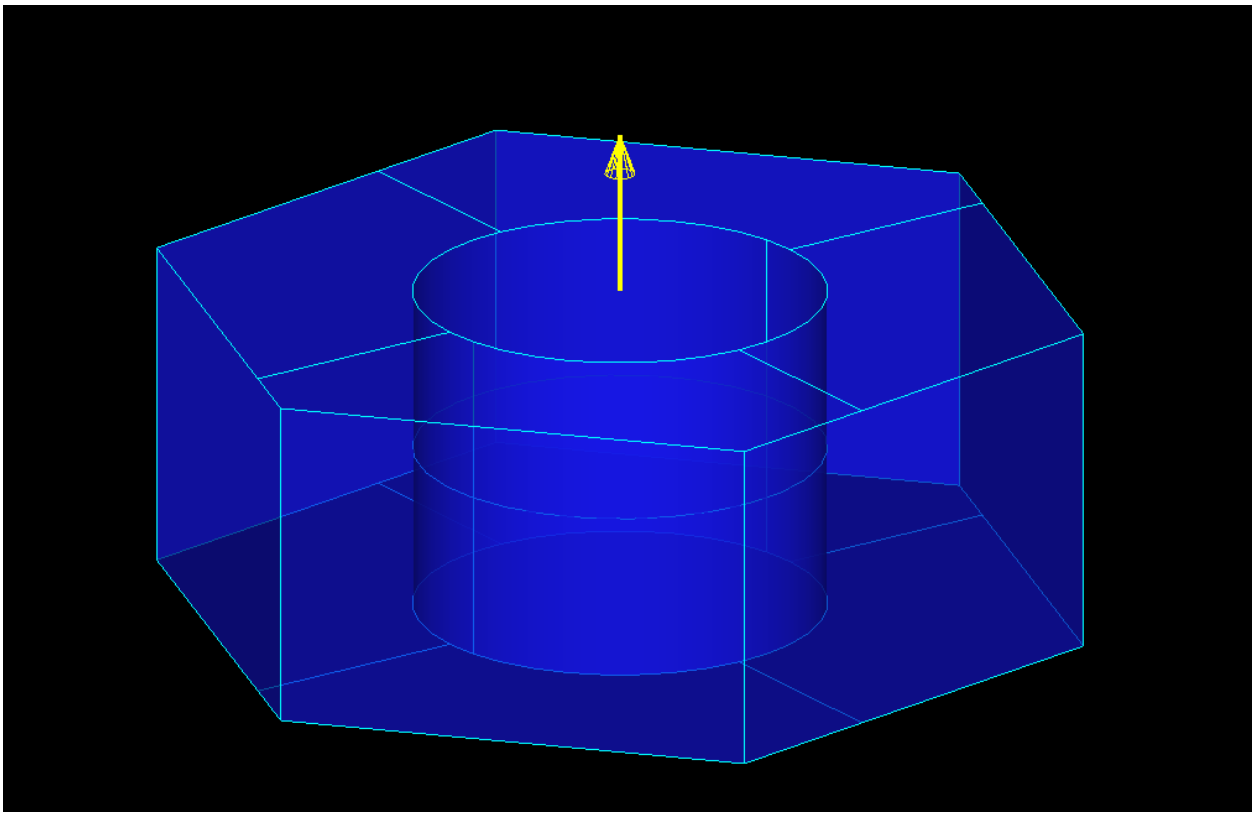

Fig. 1: ISO 4032 M2.5 Nut from a parts library file (the small yellow arrow represents the anchor)

Contents:

CHAPTER 1

Part libraries

The need for part libraries

Any hardware design project is expected to use an important number of standardized parts (screws, bolts, washers, bearings ...) and of catalog/off-the-self components. The designer should not have to redesign these kinds of components that should be available and chosen from part libraries. Industrial CAD softwares include such libraries but unfortunately they are very expensive and not open source.

The problems

Part libraries, when available, are offered under a variety of neutral formats (STEP, IGES ...) and proprietary formats. These files are usually heavyweight and not always in the format expected by the designer. Moreover, these files contain no design intent information nor any means to functionally link them to other parts, assemblies or systems in the project.

Lightweight and self-contained part libraries

The **party** package aims at providing a way to define parts libraries in text files, as well as the tools to use these text files to generate the geometry and the means to attach the parts (anchors/constraints) in a hardware design project. **party** can also generate the documentation in various formats for the parts libraries. The parts library JSON files created and used by **party** are self-contained: all the required information, including the geometry and 'anchoring' logic is defined in the JSON file.

Definition of a parts library

In the context of the **party** Python package, a parts library is a JSON file where every part can be defined by using the same set of fields.

CHAPTER 2

party

What is party ?

party is a Python package that handles parts libraries creation and use.

Installing

Listing 2.1: Installing from the git repo:

```
git clone https://github.com/guillaume-florent/party
cd party
python setup.py install
```

Parts libraries creation

A parts library might contain a lot of duplicated information (e.g. all the M3 screws will have the data defined for their threading). **party** provides standard mechanisms to avoid this duplication and to create the parts libraries JSON files from **template** files where no data is duplicated.

The parts library files must also contain the part geometry creation logic. **party** deals with inserting the geometry creation defined in a Python script into the parts library JSON file. As we will see, the geometry creation logic can also define **anchors** that allow to attach the parts from the library in a wider project.

Parts libraries use

party can generate CAD files from the parts library JSON file (PLJSON) and documentation in several formats for the parts library (HTML, PDF, ePub, Latex..) using Sphinx.

Dependencies

party depends on:

- ccad (which itself depends on PythonOCC)
- jinja2

CHAPTER 3

Parts libraries JSON format

We must distinguish:

- the final parts library JSON files, that are to be publicly released for use in open hardware projects (PLJSON)
- the 'template' files (T-PLJSON) that are used to create PLJSON files. These 'template' files are only precursors for the PLJSON files.

The 'template' T-PLJSON files (see *Assembling the parts library JSON (PLJSON) files*) are used to make maintenance easier by:

- avoiding duplicated information
- allowing to keep the geometry and anchors creation logic in a separate Python file (more readable and easier to debug)

This paragraph describes the PLJSON file format.

General

A PLJSON file is made of 4 sections:

- the **metadata** section;
- the **generators** section;
- the **rules** section;
- the **data** section.

```
{
  "metadata": {},
  "generators": {},
  "rules": {},
  "data": {}
}
```

The metadata section

The metadata section must contain the following entries:

- **name** : the name of the parts library, without spaces (i.e. 'iso4014-screws-library' is ok, 'iso4014 screws library' is not). The characters used must be valid file name characters for the OS.
- **description** : a plain text description of the library. Spaces, punctuation marks and special characters (outside of JSON specific characters) are allowed.
- **units** : for each type of unit, the unit and the variables in that unit are defined:

For example:

```
{
  "metadata": {
    "units": {
      "length": ["mm", ["p", "b_ref_b", "b_ref_c", "b_ref_d", "c_max"....]],
      "force": ["N", []],
      "weight": ["g", []],
      "dimensionless": ["", ["threading", "generics", "grade_specifics"]]
    },
  ....
  }
}
```

defines 'p' as a length in mm (millimeters). 'threading' is data without a dimension nor a unit. A forces would be in newtons if one was defined, but none is.

- **authors** defines the list of library authors.

```
{
  "metadata": {
    ...
    "authors": ["Guillaume Florent", "Thomas Paviot", "Bernard Uguen"],
    ...
  }
}
```

- **url** : url is the well ... the url from which the file was downloaded
- **license** : license is the short name of the license of the library (e.g. GPLv3)

The metadata section may contain the following entries:

- **nomenclature** : the naming convention for the parts defined in the **data** section. The naming convention is defined as a Python expression that evaluates to a string.

For example (threading and l_max are parts data definition identifiers):

```
{
  "metadata": {
    ...
    "nomenclature": "'ISO4014_' + threading + 'x' + str(int(l_max))",
    ...
    }
    ...
}
```

The nomenclature can be used to check that every entry in the **data** section has an id that respects the nomenclature.

The generators section

The **generators** section contains one entry per potential geometry and anchors generator (every entry in the data section is linked to a generator by its 'generator' field).

The value of each entry is a list a Python instructions using the ccad package. Library creators are not expected to directly write the Python instructions in the PLJSON file. Instead, the templates mechanism allows including a Python script in the final PLJSON file)

Here is an example generators section:

```
{
  ...
  "generators": {
    "iso4014_screw": [
      "r'''Generation script for ISO 4014 screw'''",
      "",
      "from ccad.model import prism, filling, ngon, cylinder, translated",
      "",
      "k_max = {{ k_max }}",
      "s_max = {{ s_max }}",
      "l_g_max = {{ l_g_max }}",
      "d_s_max = {{ d_s_max }}",
      "d_s_min = {{ d_s_min }}",
      "l_max = {{ l_max }}",
      "",
      "head = translated(prism(filling(ngon(2 / 3**.5 * s_max / 2., 6)), (0, 0, k_
↪max)), (0., 0., -k_max))",
      "",
      "threaded = cylinder(d_s_min / 2., l_max)",
      "unthreaded = cylinder(d_s_max / 2., l_g_max)",
      "",
      "part = head + threaded + unthreaded",
      "anchors = {1: {'position': (0., 0., 0.),",
      "               'direction': (0., 0., -1.),",
      "               'dimension': d_s_max,",
      "               'description': 'screw head on plane'}}"
    ]
  },
...
}
```

Values in double curly bracket (e.g. {{ k_max }}) are placeholders for the values defined in each entry of the **data** section.

The rules section

The **rules** section contain a list of Python expressions that must all evaluate to True for each entry of the **data** section for the PLJSON file to be considered correct.

Here is an example **rules** section:

```
{
  ...
  "rules": ["c_max > c_min", "d_a_max > d_a_min", "m_max > m_min", "s_max > s_min"],
```

```
  ...
}
```

The first rule in this example states that c_max must be strictly superior to c_min in each entry of the **data** section.

Any Python expression that evaluates to a boolean can be used.

The data section

The **data** section defines the possible values for each part contained in the parts library.

Every entry must contain the same set of fields. If this is not possible, another parts library must be created.

2 fields are compulsory : 'generator' (and its value must be in the ids defined in the **generators** section) and description.

The other fields are application/standard dependant.

```
{
  ...
  "data": {
    "ISO4014_M1.6_grade_Ax12": {
      "description": "M1.6 ISO 4014 screw, 12 mm, grade A",
      "generator": "iso4014_screw",
      "l_min": 11.65,
      "l_max": 12.35,
      "l_s_min": 1.2,
      "l_g_max": 3.0,
      "threading": "M1.6_grade_A",
      "generics": "M1.6_generics",
      "grade_specifics": "M1.6_grade_A_specifics",
      "p": 0.35,
      "b_ref_b": 9.0,
      "b_ref_c": 15.0,
      "b_ref_d": 28.0,
      "c_max": 0.25,
      "c_min": 0.1,
      "d_a": 2.0,
      "d_s_max": 1.6,
      "l_f_max": 0.6,
      "k_nominal": 1.1,
      "r_min": 0.1,
      "s_max": 3.2,
      "d_s_min": 1.46,
      "d_w_min": 2.27,
      "e": 3.41,
      "k_max": 1.225,
      "k_min": 0.975,
      "k_w_e_min": 0.68,
      "s_min": 3.02
    },
    "ISO4014_M1.6_grade_Ax16": {
      "description": "M1.6 ISO 4014 screw, 16 mm, grade A",
      "generator": "iso4014_screw",
      "l_min": 15.65,
      "l_max": 16.35,
      "l_s_min": 5.2,
      "l_g_max": 7.0,
```

```
      "threading": "M1.6_grade_A",
      "generics": "M1.6_generics",
      "grade_specifics": "M1.6_grade_A_specifics",
      "p": 0.35,
      "b_ref_b": 9.0,
      "b_ref_c": 15.0,
      "b_ref_d": 28.0,
      "c_max": 0.25,
      "c_min": 0.1,
      "d_a": 2.0,
      "d_s_max": 1.6,
      "l_f_max": 0.6,
      "k_nominal": 1.1,
      "r_min": 0.1,
      "s_max": 3.2,
      "d_s_min": 1.46,
      "d_w_min": 2.27,
      "e": 3.41,
      "k_max": 1.225,
      "k_min": 0.975,
      "k_w_e_min": 0.68,
      "s_min": 3.02
    },
    ...
  }
}
```

CHAPTER 4

Assembling the parts library JSON (PLJSON) files

From the previous chapter (*Parts libraries JSON format*), it is obvious that a PLJSON file contains duplicated information. This is intentional as it greatly simplifies the exploitation of the PLJSON files.

Though, if we have 200 screws using the same threading, correcting an error for one of the threading definition values will be a very repetitive task as it will have to be corrected for each screw.

To simplify the creation of PLJSON files, **party** provides a set of so-called 'standard-mechanisms' (see *Standard mechanisms*).

Creating a PLJSON file might use any kind of logic and information sources as long as the resulting PLJSON file conforms to the PLJSON file format (*Parts libraries JSON format*) but these standard mechanisms are offered as a guided way to create the PLJSON files.

Roles

Library creators

Parts library creators are expected to know and use the T-PLJSON template files possibilities and the accompanying *Standard mechanisms*

Library users

If your intention is only to use a parts library (PLJSON) file for a project, you do not need to know about what follows.

Template files (T-PLJSON)

Parts library templates skeleton creation

The `create_skeleton()` function of the `party.library_creation` module creates a starting point for a parts library project based on template files.

Standard mechanisms

party provides the following mechanisms to create parts library (PLJSON) files from part library template files:

- geometry and anchors code inclusion
- alias mechanism
- include mechanism (in development)

Geometry and anchors code inclusion

This mechanism is handled by the `template_handle_generators()` function of the `party.library_creation` module.

It expects:

- a *{{ generators }}* tag as the value of the 'generators' id in the template file

```
{
  "metadata": {...},
  "generators": { {{ generators }} },
  "rules": {...},
  "data": {...}
}
```

- the geometry and anchors generation scripts in a *generators* subfolder of the folder containing the template file.

Alias mechanism

This mechanism is handled by the `template_handle_aliases()` function of the `party.library_creation` module.

It expects:

- an *alias* entry in the template file where the aliases are defined

```
{
  "metadata": {...},
  "generators": { {{ generators }} },
  "rules": {...},
  "aliases": {
      "M1.6_grade_A": {
        "generics": "__alias__M1.6_generics",
        "grade_specifics": "__alias__M1.6_grade_A_specifics"
      },
      "M1.6_grade_B":{
        "generics": "__alias__M1.6_generics",
        "grade_specifics": "__alias__M1.6_grade_B_specifics"
```

```
    },
    "M1.6_generics": {
      "p": 0.35,
      "b_ref_b": 9.00,
      "b_ref_c": 15.00,
      "b_ref_d": 28.00,
      "c_max": 0.25,
      "c_min": 0.10,
      "d_a": 2.00,
      "d_s_max": 1.60,
      "l_f_max": 0.60,
      "k_nominal": 1.10,
      "r_min": 0.10,
      "s_max": 3.20
    },
    "M1.6_grade_A_specifics": {
      "d_s_min": 1.46,
      "d_w_min": 2.27,
      "e": 3.41,
      "k_max": 1.225,
      "k_min": 0.975,
      "k_w_e_min": 0.68,
      "s_min": 3.02
    },
    "M1.6_grade_B_specifics": {
      "d_s_min": 1.35,
      "d_w_min": 2.30,
      "e": 3.28,
      "k_max": 1.30,
      "k_min": 0.9,
      "k_w_e_min": 0.63,
      "s_min": 2.90
    },

    "M2_grade_A": {
      "generics": "__alias__M2_generics",
      "grade_specifics": "__alias__M2_grade_A_specifics"
    },
    "M2_grade_B":{
      "generics": "__alias__M2_generics",
      "grade_specifics": "__alias__M2_grade_B_specifics"
    },
    "M2_generics": {
      "p": 0.40,
      "b_ref_b": 10.00,
      "b_ref_c": 16.00,
      "b_ref_d": 29.00,
      "c_max": 0.25,
      "c_min": 0.10,
      "d_a": 2.60,
      "d_s_max": 2.00,
      "l_f_max": 0.80,
      "k_nominal": 1.40,
      "r_min": 0.10,
      "s_max": 4.00
    },
    "M2_grade_A_specifics": {
      "d_s_min": 1.86,
```

```
        "d_w_min": 3.07,
        "e": 4.32,
        "k_max": 1.525,
        "k_min": 1.275,
        "k_w_e_min": 0.89,
        "s_min": 3.82
      },
      "M2_grade_B_specifics": {
        "d_s_min": 1.75,
        "d_w_min": 2.95,
        "e": 4.18,
        "k_max": 1.6,
        "k_min": 1.2,
        "k_w_e_min": 0.84,
        "s_min": 3.70
      },
  "data": {...}
}
```

As illustrated by the example, aliases can be nested.

- the use of the defined aliases in the **data** section with an '__alias__' prefix.

```
{
  "metadata": {...},
  "generators": { {{ generators }} },
  "rules": {...},
  "data": {
      "M1.6x12_A": {
        "description": "M1.6 ISO 4014 screw, 12 mm, grade A",
        "generator": "iso4014_screw",
        "l_min": 11.65,
        "l_max": 12.35,
        "l_s_min": 1.20,
        "l_g_max": 3.00,
        "threading": "__alias__M1.6_grade_A"
    },
    ...
  }
}
```

Include mechanism

This mechanism is handled by the `template_handle_includes()` function of the `party.library_creation` module.

The include mechanism is currently in development.

Automation

The `template_handle_includes()` function of the `party.library_creation` module analyzes the template file and uses the standard mechanisms in the right order to create the parts library (PLJSON) file.

Checks

When the parts library JSON (PLJSON) file has been assembled, the `party.library_checking` module provides functions which goal is to check that the PLJSON file complies with the PLJSON file specification and that the data is correct.

Constant data schema

The `check_library_fields()` function checks that the metadata for each entry in the **data** section is the same.

Rules

The `check_library_json_rules()` function checks that the rules defined in the **rules** section are respected in the **data** section of the PLJSON file.

Units definition

The `check_library_units_definition()` function checks that the units defined in the **metadata/units** are properly defined and that each fields describing a part characteristic are present in the units definition.

CHAPTER 5

Using parts library JSON (PLJSON) files

Now that we have a proper PLJSON file, what can we do with it?

Create library CAD files

The `generate()` function of the `party.library_use` module will create ccad scripts and CAD files (depending on the optional arguments values) in subfolders of the folder containing the parts library JSON (PLJSON) file.

Formats

From the parts library JSON (PLJSON) file, it is possible to create one CAD file per entry in the **data** section of the PLJSON file in the following formats:

- STEP (AP203)
- STL
- HTML (to view the part in the browser using X3DOM)

Example

See the `examples/ISO_4014/use_library_json.py` script for an example of how to generate the CAD files from the parts library (PLJSON) file.

```
#!/usr/bin/python
# coding: utf-8

r"""Example use of the library.json file to create geometry_scripts and
cad files

"""
```

```
import logging
from os.path import join, dirname

from party.library_use import generate


logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s :: %(levelname)6s :: '
                           '%(module)20s :: %(lineno)3d :: %(message)s')

generate(json_library_filepath=join(dirname(__file__), "library.json"),
         generate_steps=True,
         generate_stls=True,
         generate_htmls=True,
         generate_svgs=True)
```

Create the library documentation

The `generate()` function of the `party.library_documentation` module will create HTML documentation of the parts library JSON (PLJSON) file in the folder passed as a parameter.

Formats

Currently, only HTML documentation generation is implemented.

Example

See the `examples/create_documentation_example.py` script for an example of how to generate the documentation of the parts library (PLJSON) file.

```
#!/usr/bin/python
# coding: utf-8

r"""Parts library documentation creation example"""

from os.path import dirname, join
import logging

from party.library_documentation import create_libraries_sphinx_sources


def main():
    r"""Main function for the parts library documentation example"""
    create_libraries_sphinx_sources(join(dirname(__file__), "../examples"),
                                    join(dirname(__file__), "../examples/doc"))


if __name__ == "__main__":
    logging.basicConfig(level=logging.DEBUG,
                        format='%(asctime)s :: %(levelname)6s :: '
                               '%(module)20s :: %(lineno)3d :: %(message)s')

    main()
```

Use the parts in other projects

Work in progress to use the parts library JSON (PLJSON) files in:

- a new (in development) CAD project definition structure based on acyclic directed graphs.
- other CAD softwares that offer scripting/plugin capabilities

CHAPTER 6

Indices and tables

- genindex
- modindex
- search