Rubin Observatory

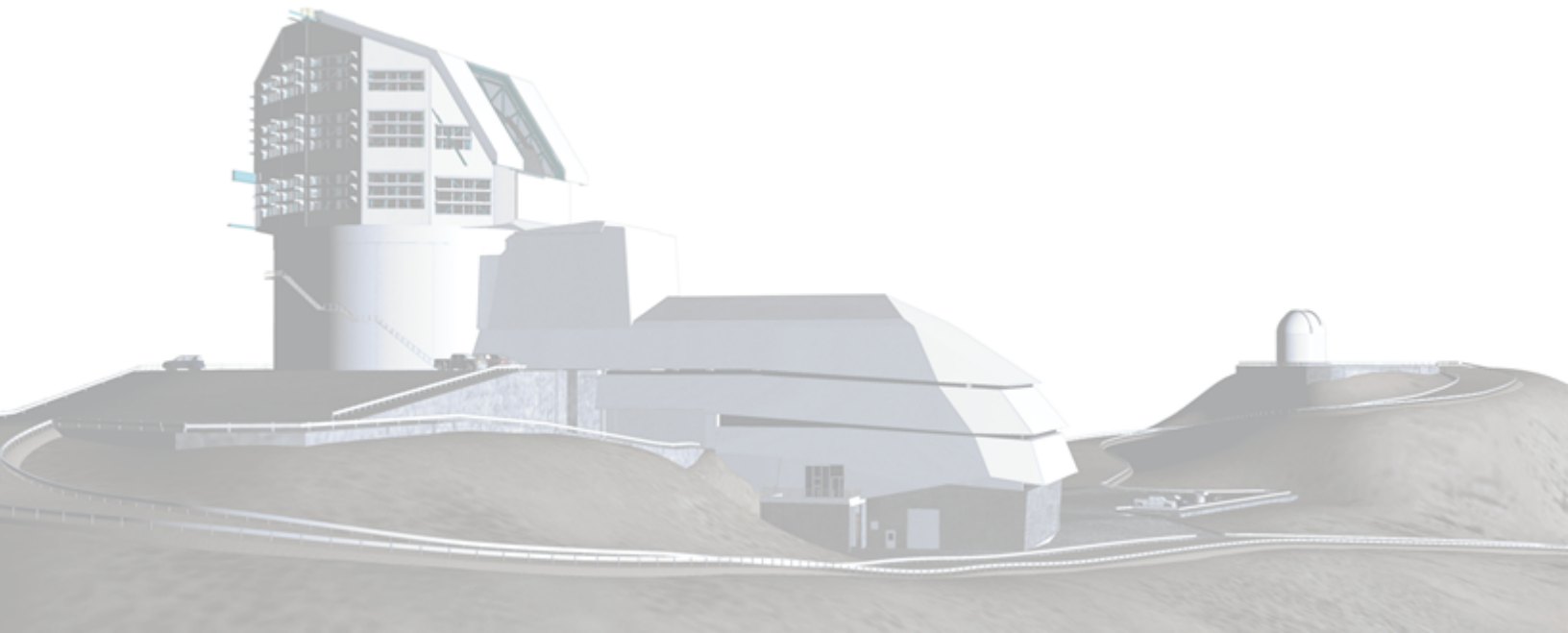
Vera C. Rubin Observatory
Data Management

Rubin-Env Integration with DM Build Tools

Gabriele Comoretto

DMTN-174

Latest Revision: 2020-12-22



Rubin Observatory

Abstract

Impacts on using the rubin-env environment definition from conda-forge in lsstsw and newinstall.

Rubin Observatory

Change Record

Version	Date	Description	Owner name
1	YYYY-MM-DD	Unreleased.	Gabriele Comoretto

Document source location: `https://github.com/lsst-dm/dmtn-174`

Contents

Rubin-Env Integration with DM Build Tools

1 Introduction

This technical note has the main scope to document how we are integrating the new introduced **rubin-env** environment definition from conda-forge in the tooling currently used in DM.

In order to have a clear vision, in this topic, a few historic information are given in section 2, Section 3 describes the implementation and management of the rubin-env definition in conda-forge, while section 4 describes its integration in the DM build tools. Finally a few words on possible future improvements.

2 Historic Scipipe_Conda_Env

The Science Pipelines is requiring a set of libraries to be available in the underlying conda environment. The definition of this environment has been handled until now, end 2020, using the github project scipipe_conda_env.

Many of these libraries were managed as Eups packages as part of the Science Pipelines. They have been moved out of the science pipelines, as proposed in DMTN-110. This has increased the size of the environment, bringing a few stability and management issues into the loop.

In order to overcome this instabilities, we moved away from a fully semantic versioning approach, and defined the environment an exact list of libraries, obtained using:

```
conda list --explicit
```

This has the advantage to always relay to a fixed environment but makes it more difficult to extend it, by the downstream user of the Science Pipelines. Therefore the environment definition has bene implemented in the **rubin-env** meta-package in conda-forge, as described in section 3.

2.1 Semantic Versioning

Before moving to the description of conda the new rubin-env meta-package, it is important to clarify the use of Semantic Versioning in DM.

Due to the collaborative nature of the project, it is very important to be able to add packages to our distributions. This implies the need of using semantic versioning to reduce conflicts.

Semantic versioning should guarantee that no breaking changes are introduced when only minor or patch releases are done. This permits the environment to include updated fixes on the resolved libraries, without the need to redefine the environment.

3 Rubin-Env Feedstock

Defining the environment required by the Science Pipelines in a conda-forge meta-package has multiple benefits. For example, all changes need to follow the conda-forge workflow and will be tested before making a new version available.

You can refer to the Github repository rubinenv-feedstock for all information and procedures to follow.

4 DM Build Tools

The tools to be updated in DM in order to switch from scipipe_conda_env to rubin-env are:

- lsstsw
- newinstall
- jenkins-dm-jobs
- ci-scripts

The tool most impacted is lsstsw.

4.1 Semantic Versioning Limitation

In some cases, a minor change in a library resolved by rubin-env, is breaking the usability of the Science Pipelines code. This implies that given a fixed reference or tag of the Science Pipelines, build is giving a error and it is not possible to deploy a specific distribution from source anymore.

The main reason that this may happen is that in some libraries resolved by rubin-env, the semantic versioning is not followed as it should be. In this case, in order to be able to deploy from source a specific version version of the Science Pipelines, we need to fix a dependency in rubin-env, avoiding to semantic versioning for one or more provided libraries.

In other cases, the problem can be in the Science Pipelines code itself. In tis case, it is required to patch the Science Pipelines

4.2 Ensuring Reproducibility

In order to be able to debut the cases described in the previous section 4.1, it is important to have available the exact pinned version of the environment used for a specific build.

Also, from an operational point of view, we don't need the flexibility provided by the semantic versioning, but it is preferable to stick to the same configuration that has been used for development and validation.

Therefore we need to store the pinned version of the environment used in each build that we persist.

4.3 Implementation

The adoption of the rubin-env meta-package is implemented in the DM-27005 ticket branches, in all four DM build tools.

In the general case, it is sufficient to switch the creation of the environment from scipipe_conda_env to the rubin-env definition, and update the reference, that will become the rubin-env version.

However, in order to be able to persist the exact pinned version used for each build, some additional changes have to be added.

4.3.1 Lsstsw

Lsstsw provide a few scripts to perform the deployment, setup and build of the Science Pipelines. In addition there is a script that push to the EUPS_PKGROOT the source Eups packages, for a specific build.

In order to store and persist also the pinned version of the used environment, a specific env file is added to the distribution, and pushed together with the source packages.

Source packages are stored in the Eups packages repository in an unstructured way, in the folder:

```
\$EUPS\_PKGROOT/stack/src
```

Under the *src* folder there are a few subfolders, for example to provide the Eups tags information. A new folder **env** is added under *src*, where the pinned environment for each build is stores.

Since the environment may change with the architecture where the build is performed, we need also to make it persistent for each binary distribution that we persist. See following subsection 4.3.2.

4.3.2 Jenkins-Dm-Jobs

The tooling used to run continuous integration jobs in principle could be affected just in a very marginal way by the introduction of the rubin-env meta-package.

However, this set of tooling, includes a large amount of logic that do no belong strictly to CI. This is the case of the generation and storage of the Eups binaries distributions in the EUPS_PKGROOT pakages repository.

Eups binary packages are stored in the EUPS_PKGROOT following a folder structure that den-

des on a few factors, like for example, platform, compiler, or environment reference. For example:

```
\$EUPS\_PKGROOT/stack/redhat/el8/conda-system/miniconda-py37_4.8.2-cb4e2dc/
```

All packages produced with the same setup, will be stored in an unstructured way in a EUPS_PKGROOT as described above. Under this root folder, there are a few subfolder, for example, for the manifests obtained in a build.

In the same way, as previously described for the source packages in the previous subsection 4.3.1, the pinned version of the environment is saved in a specific **env** subfolder and pushed to the package repository together with the binary distribution.

5 Future Work

As described in this technical note, the tools have been updated in order to be able to store the information. However no use of it is done at the moment in a programmatic way.

We should update the deployment tools to be able to resolve a specific environment, when requested, as an alternative to the default environment.

Also, it should be possible to analyze the environment changes. Making this visible may help anticipating problems.

A References

[DMTN-110], Comoretto, G., 2019, *Conda Environment Proposal for Science Pipelines*, DMTN-110, URL http://dmtn-110.lsst.io

B Acronyms

Acronym	Description
CI	Continuous Integration
DM	Data Management
DMTN	DM Technical Note