The High Level Data Reduction Library

Pascal Ballester[1], Armin Gabasch[1], Yves Jung[1], Andrea Modigliani[1], Julian Taylor[2], Lodovico Coccato[1], Wolfram Freudling[1], Mark J. Neeser[1], and Enrico Marchetti[1].

1: ESO
2: Informate
Contact: pballest@eso.org, agabasch@eso.org

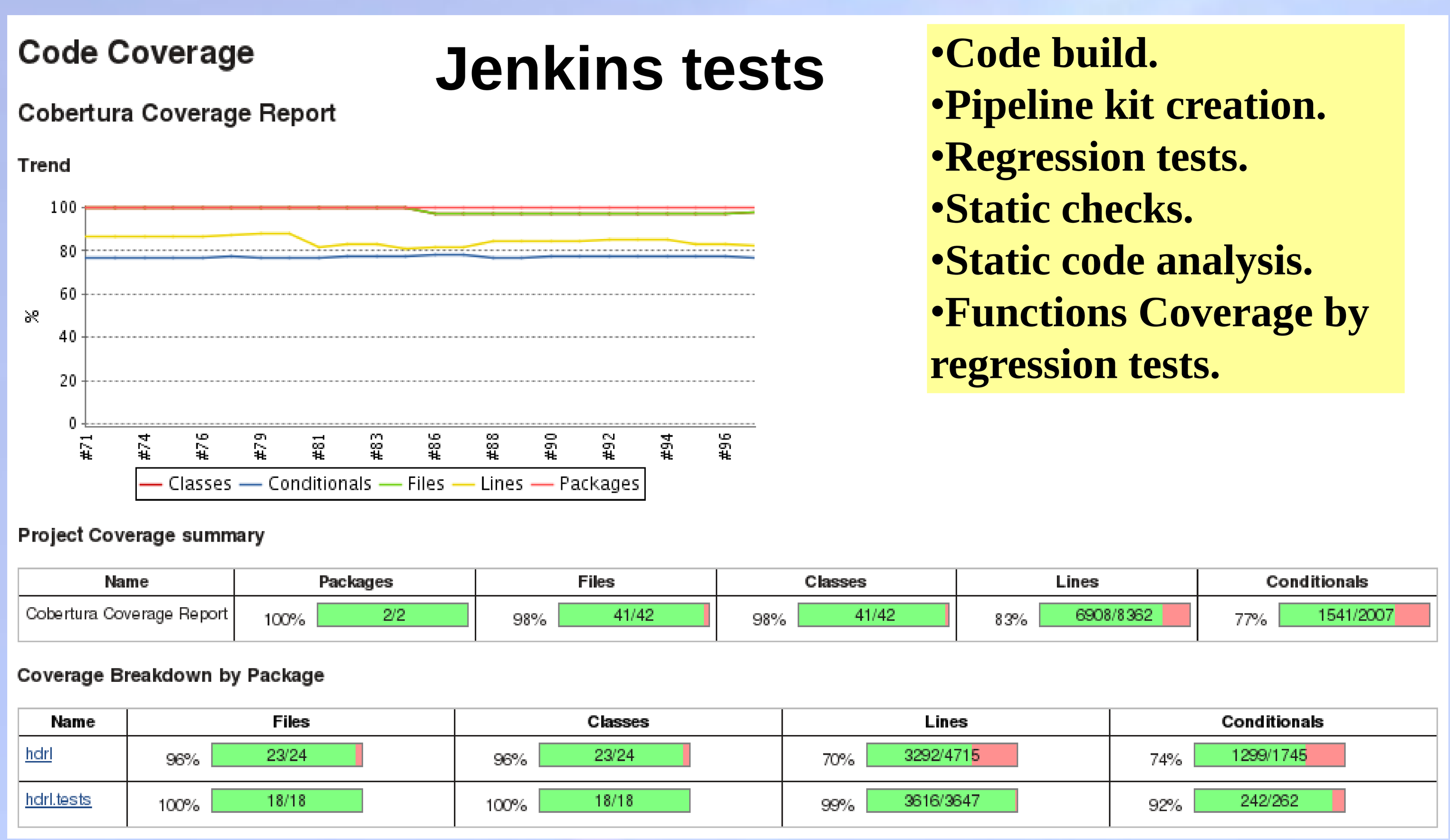
Abstract

The European Southern Observatory (ESO) provides pipelines to reduce data for most of the instruments at its Very Large Telescope (VLT). These pipelines are written as part of the development of VLT instruments, and are used both in the ESO's operational environment and by science users who receive VLT data. All the pipelines are highly specific geared toward instruments. However, experience showed that the independently developed pipelines include significant overlap, duplication and slight variations of similar algorithms. In order to reduce the cost of development, verification and maintenance of ESO pipelines, and at the same time improve the scientific quality of pipelines data products, ESO decided to develop a limited set of versatile high-level scientific functions that are to be used in all future pipelines. The routines are provided by the High-level Data Reduction Library (HDRL).
To reach this goal, we first compare several candidate algorithms and verify them during a prototype phase using data sets from several instruments. Once the best algorithm and error model have been chosen, we start a design and implementation phase. The coding of HDRL is done in plain C and using the Common Pipeline Library functionality. HDRL adopts consistent function naming conventions and a well defined API to minimise future maintenance costs, implements error propagation, uses pixel quality information, employs OpenMP to take advantage of multi-core processors, and is verified with extensive unit and regression tests.
This poster describes the status of the project and the lessons learned during the development of reusable code implementing algorithms of high scientific quality.

HDRL main features

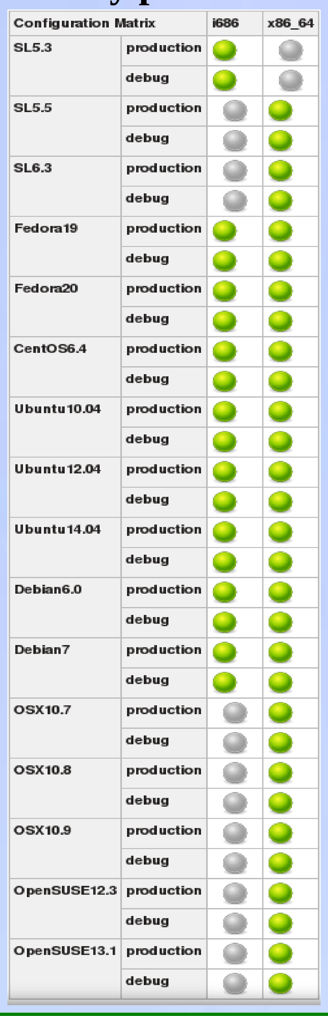
- HDRL is meant to share core across pipelines.
- Careful algorithms evaluation and verification.
- Users may propose new algorithms if needed.
- Clear and detailed algorithm requirements definition with iterative refinements.
- Attentive design to allow code sharing and easy maintenance.
- Initial prototype implementation.
- Extended unit and regression tests to verify results.
- Algorithm implementation and consolidation.
- Code profiling, speed-up, and use of OpenMP to take advantage of multi-core architectures.
- Doxygen and user/developer manual documentation.
- Continuous integrations tests to monitor code builds, portability, compiler warnings, documentation, static checks, and coverage.

Jenkins tests

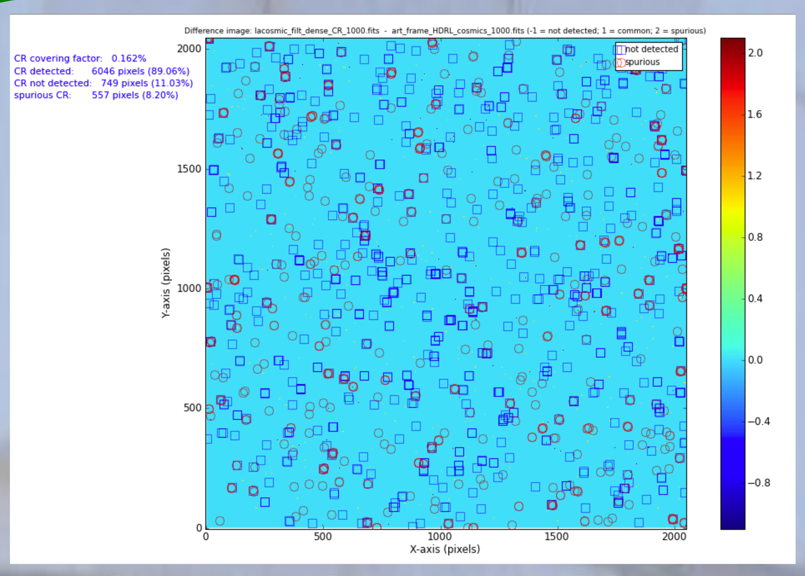


- Code build.
- Pipeline kit creation.
- Regression tests.
- Static checks.
- Static code analysis.
- Functions Coverage by regression tests.

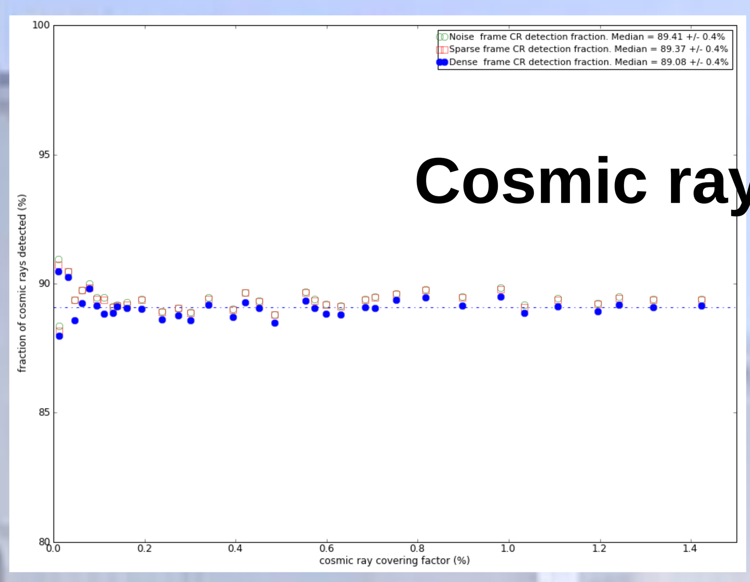
Verified on many platforms.



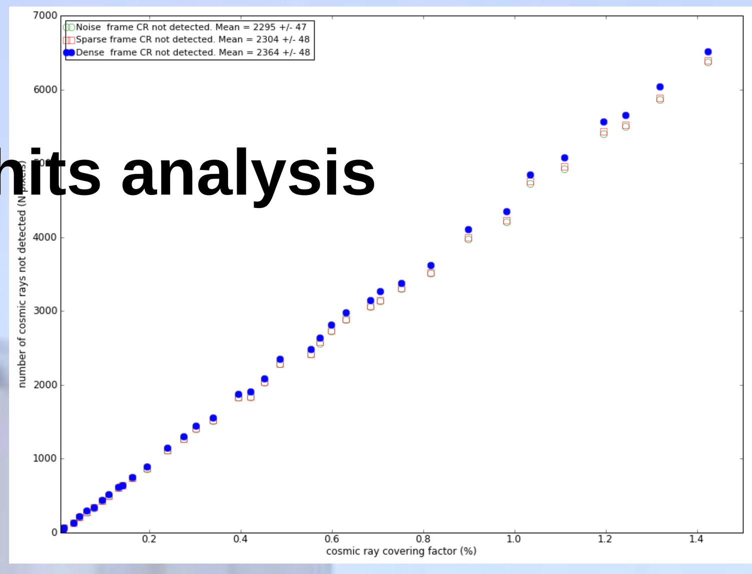
Cosmic ray hits analysis



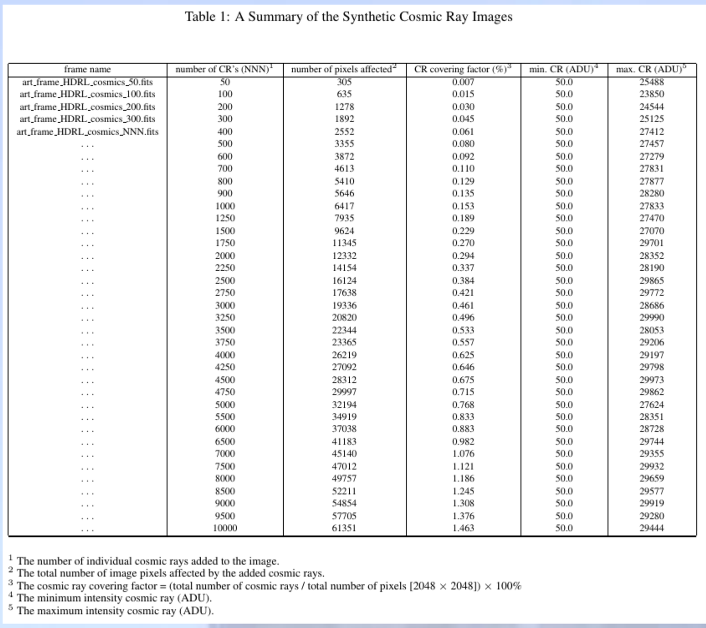
CRH analysis: a) synthetic image with background noise and added cosmics. Not detected CRHs are in blue, spurious detections in red.



CRH detection efficiency measured on pure noise (open green circles), sparsely populated (red squares) and dense (solid blue) frames.

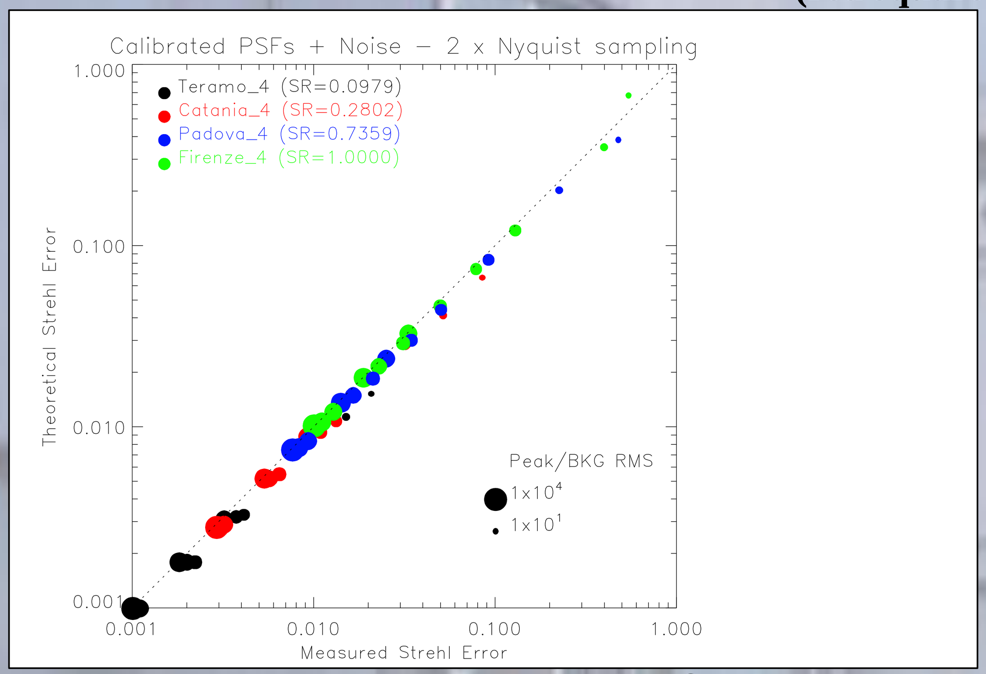


Number of CRHs not detected versus the CRH relative covering factor.

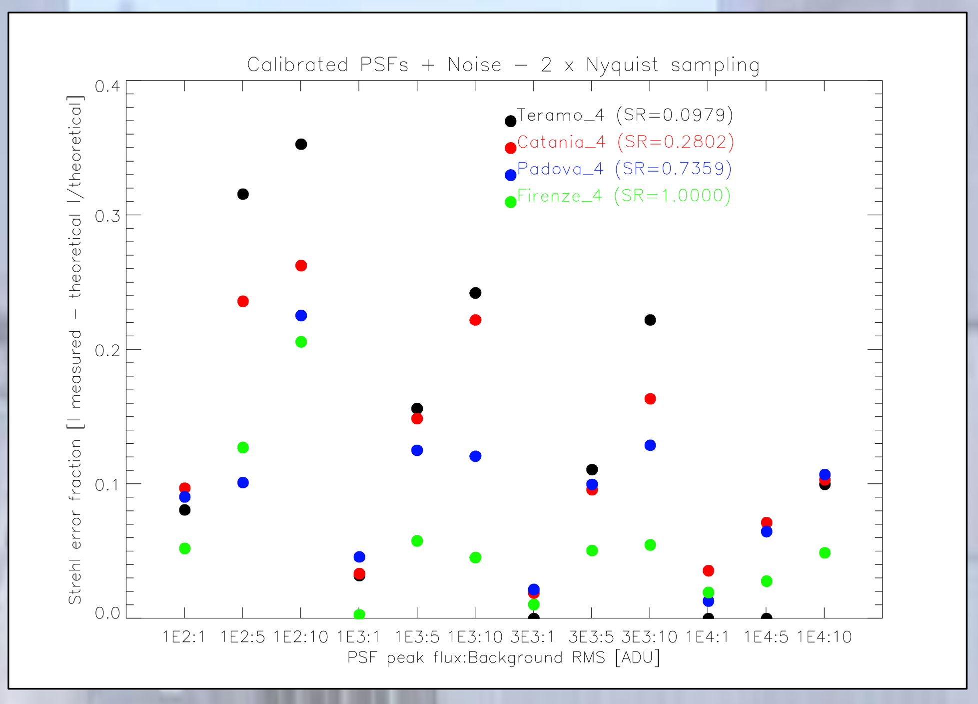


Tests performed on a large variety of simulated data.

Strehl ratio analysis



Strehl error measured versus theoretical for the selected subset of 2xNyquist sampling PSFs and noise configurations.



Fractional Strehl error difference between test results and theoretical values for the selected subset of 2xNyquist sampling PSFs and different noise configurations.

Provided algorithms

- Overscan computation and subtraction.
- Frames combination into a master.
- Pixel quality determination.
- Cosmic ray hit detection.
- Strehl ratio computation.

General helper functionalities

- Statistics based on robust clipping.
- Use of error and pixel quality information.
- Functionalities to deal with large data sets.
- 1D fitting.

It is simple to use

Implementing a function requires usually only two lines of C code.

Generic function example.

```
hdrl_parameter * params = hdrl_function_parameter_create(
        int                 param1,
        double              param2,
        const char*         param3)
```

```
hdrl_value function = hdrl_function_compute(hdrl_object, params);
```

Cosmic hit detection.

```
hdrl_parameter * hdrl_lacosmic_parameter_create(
        double              sigma_lim,
        double              f_lim,
        int                 max_iter);
```

```
cpl_mask * bpm_lacosmic = hdrl_lacosmic_edgedetect(
        const hdrl_image        *    img_in,
        const hdrl_parameter    *    params)
```

Strehl ratio computation.

```
hdrl_parameter * params = hdrl_strehl_parameter_create(
        double              wavelength,
        double              m1_radius,
        double              m2_radius,
        double              pixel_scale_x,
        double              pixel_scale_y,
        double              flux_radius,
        double              bkg_radius_low,
        double              bkg_radius_high);
```

```
hdrl_strehl_result hdrl_value strehl = hdrl_strehl_compute(
        const hdrl_image * himg,
        hdrl_parameter* params);
```

Summary

- HDRL provides several common astronomical data reduction algorithms to ease development, verification, maintenance of data reduction pipelines.
- Interested users may propose algorithms for implementation. They are carefully selected to obtain the best possible results on a large variety of data.
- The code is designed, implemented and documented aiming at reuse across different data reduction pipelines to ease long term maintenance.
- HDRL is easy to use. The user has to define the relevant algorithm parameter structure and pass it together with the input data to the HDR function.
- HDRL is available to instrument consortia and users as a svn extern library based on the ESO Common Pipeline Library.