IDIS Tutorial

Document version: 0.4
Covers IDIS version: 3.0.0pre

written by
Jörg Rachen (MPAC), Jörg Knoche (MPAC)

with contributions from
Giovanna Giardino (RSSD)

The MPAC IDIS group:
Uwe Dörl (ProC GUI; udoerl@mpa-garching.mpg.de)
Torsten Enßlin (group manager; ensslin@mpa-garching.mpg.de)
Reinhard Hell (DMC initial design and development)†
Wolfgang Hovest (ProC chief developer; woho@mpa-garching.mpg.de)
Jörg Knoche (DMC chief developer; knoche@mpa-garching.mpg.de)
Frank "Theo" Matthai (DMC GUI)†
Jörg Rachen (science application / quality management;)†
Martin Reinecke (Level S / module integration; martin@mpa-garching.mpg.de)
Thomas Riller (ProC Editor / Sampling;)†
Georg Robbers (DPC Level2/3 module integration)†

The ESTEC IDIS group
Martin Bremer (Federation Layer; mbremer@rssd.esa.int)
Giovanna Giardino (Federation Layer; ggiardin@rssd.esa.int)

August 12, 2014

†Former group member

Abstract

This document explains the usage of the IDIS components used for Planck data analysis, i.e, the Process Coordinator (ProC) and the Data Management Component (DMC). It is meant to introduce the software, explain its capabilities, how it works, how it can be installed and configured, and how the various components are used. The user is guided to build a simple pipeline and execute it, but also more complex applications are explained. Nevertheless, this tutorial is not meant as a reference manual explaining every menu item and button on the GUI. For this, the user is referred to the help system delivered with the software, and we believe that, after the first steps are taken, the ProC/DMC software is sufficiently self explanatory to be used by a user who has read the essential parts of this tutorial.

About this document

This document is supposed to give a comprehensive introduction into the operation of the ProC workflow engine. It is a combination of tutorial and a task oriented manual. That means that it is not, like many other manuals, just a collection of tautologies attached to a list of anyway self explaining menu items (i.e., "**File** → **Save**: Save the file"). Rather it explains the usage scenarios of the software, and describes tools and menus of the software in this context. It uses screenshots to a reasonable extent, otherwise describes functions in text or item lists.

Items calling for special attention are highlighted like this throughout the document.

The ProC package is under permanent development. There are some major changes and extensions planned for the near-to-mid future, which makes some sections of this document soon obsolete.

We may give an outlook on the things to come in text boxes like this.

The development of this tutorial is notoriously lacking behind the development of the ProC. Even some fundamental sections are yet missing. This gives rise to the first warning box:

This document is under permanent construction. The current status of its chapters and the current schedule for their completion is found below.

Chapter 1, Quickstart: a road map through the ProC package — new and fairly complete, no plans for major updates soon.

Chapter 2, Introduction to IDIS — fairly complete, although quite old. May need updates, but this has low priority.

Chapter 3, Installation and Setup — somewhat behind development, should be up-to-date in the next version of this document.

Chapter 4, ProC manual — Sections on the Editor are out of date, will be updated after the new Editor, currently in development, is available. Sections on the Session Manager and Pipeline Coordinater should be completed and up-to-date.

Chapter 5, DMC manual — Severely out of date as significant development of the DMC has taken place. Should be completed and up-to-date in the next version.

Chapter ??, Integration of code — Currently non-existent, a first incarnation is now written with highest priority and should be included in the next version.

The schedule given here is according to current planning, but not carved in stone. If you need information currently missing in this document for your work with or evaluation of the ProC, please write an email!

For questions regarding this document or ProC usage in general, don't hesitate to contact the responsible people in the MPA development group. A list of people with their expertise area and email-addresses is given on the cover page.

Contents

Chapter 1

Quickstart: A road map through the ProC package

The ProC package is a database supported scientific workflow engine, which has emerged from parts of the Integrated Data and Information System (IDIS) developed for the Planck mission. In the following chapters of this tutorial, we provide a description of the concept and philosophy of IDIS, the functionalities and use of all its components for scientific data handling and processing. As an introductory overview, we want to perform here a solution of a simple task: We construct a small cosmological simulation pipeline from the Planck "Level S" simulation package provided with the ProC and explain with every step we perform, the actions performed by the ProC package. Through references to the following chapters, this introduction may be regarded as a guide to read this tutorial, and where to find the information needed next in practical applications.

1.1 Launching the ProC

1.1.1 Installation

We assume that the ProC package (IDIS) has been installed on a proper database system following the instructions given in Chapter 3, or that an IDIS-in-a-box tarball has been obtained and unpacked (Section 3.6). However, the ProC can also be run in a file based mode, without database - in this case, ignore all references to databases in this Quickstart.

If you use the ProC with a database (i.e. the Derby database from the Tarball itself), please pay attention to information in these boxes.

We further assume that we reside in a directory `<ProC-Dir>/IDIS`, where `<ProC-Dir>` is the directory in which the package has been unpacked and installed.

1.1.2 Start and Login

The ProC GUI is started with the command

```
IDIS> ProC/bin/startProC.sh
```

A GUI opens and asks for a login. For this release the so called Federation Layer has been disabled, therefore you don't need to enter anything and just click on "local". For projects which require access control to databases, the Federation Layer can be activated again. But this requires certain changes in the source code, which are documented.

The GUI which remains after the login dialog disappears is called the ProC Session Manager (SeMa). It controls all actions performed in the ProC for constructing, configuring and executing pipelines. It also acts as an entrance to the GUI of the Data Management Component (DMC), which allows to query and access the data products used and generated in our pipeline, when they are stored in a database. By pulling the mouse pointer over the buttons of the SeMa, tool-tips give some brief idea what they are good for, if they are not self explanatory anyway. More information on the Session Manager functionality is found in Sections 4.1.1 and 4.4.

1.1.3 Starting the Editor

The leftmost button in the top panel starts the ProC Editor to construct a new pipeline (the same effect is obtained by clicking the **Definition: Edit** button on the right panel). The editor has two windows. On the left, it shows a directory tree containing the essential pipeline constituents, which are modules, subpipelines or control elements (see Sections 4.1.2 and 4.2 for more information). The window on the right is used to combine these elements graphically to a pipeline, which is described next. Before continuing, however, make sure that all constituents needed are available. For this, please unfold the modules folder in the left window, and verify that the module **syn_alm_cxx** is contained. If not, please read Section 3.7.3 and perform the described steps to include Level S modules

1.2 Constructing the Pipeline

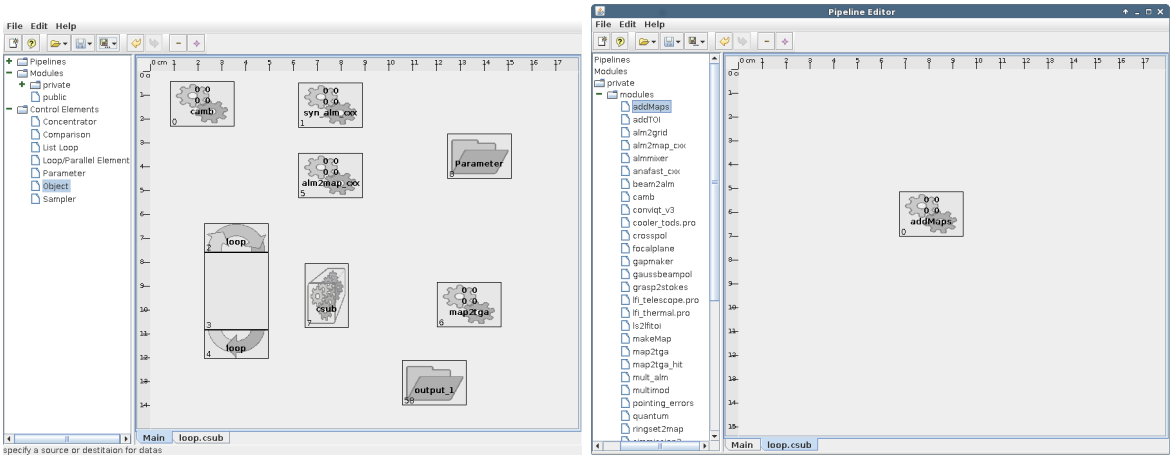


Figure 1.1: Editor view on the quickstart pipeline with all pipeline elements inserted, but yet unstructured and without data flow.

Constructing a pipeline means to place and connect modules and control elements (we do not use user-defined subpipelines in our exercise, to learn more about them read Section 4.2.2). Therefore we define four terms which will be used from now on. **Insert**ing a module means to click on the name of the module in the tree on the left side, and click and drag it onto an empty space

on the canvas. **Connect**ing two modules means to move the mouse on the inside border of the first module until a yellow frame appears inside the module, click and hold the left mouse button, move the mouse to the target module and release the mouse button. You will notice a directed line appearing, following the mouse and finally connecting both modules. The first module is therefore the data source, while the second module is the data sink. **Moving** a module is done by clicking somewhere inside the border of the module, keeping the mouse button pressed and dragging the module to the desired position. And finally **Edit**ing a module means to right click on a module and select **Config Module** from the appearing menu.

The pipeline we are constructing is of no real scientific use, but it should explain most techniques used within the ProC. What the pipeline does, is to calculate a realization of the CMB for a set of cosmological parameters, create a map from this, sum up this map repeatedly, store it in the database (or in a file) as a Healpix map and display it on the screen. For this we need five modules, **camb**, **syn_alm_cxx**, **alm2map_cxx**, **addMaps** and **map2tga** provided by Level S, and some control elements provided by the ProC.

1.2.1 Inserting and moving Modules and Control elements

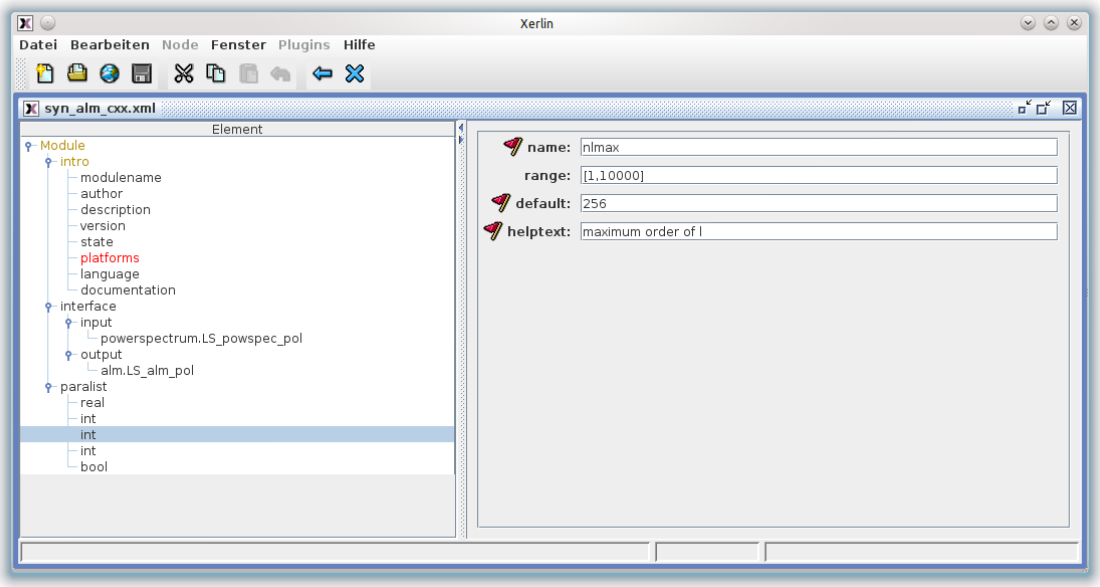


Figure 1.2: The xml description of the module **syn_alm_cxx**, viewed in the Xerlin xml editor. The left panel shows the xml structures, clearly seen are the input and output definitions for **syn_alm_cxx** (see Section 1.2.2). The right panel shown here allows to define properties of module parameters, in our example of the parameter `nlmax` (see Section 1.3).

First, we will need the **Loop/Parallel Element** from the **Control Elements** section. You will notice that actually two elements will be inserted on the canvas. The first and bigger one is the representation of the Loop Element which provides also some configuration parameters. The second object is a subpipeline which is associated with this Control Element (it cannot be deleted by itself, but will be deleted when the associated Control Element is deleted). Inserting a subpipeline will

create a new tab window at the bottom of the Editor (labeled "loop.csub" in our case). Click on this tab to switch to the contents of the subpipeline, which of course is empty at the moment.

On this empty canvas, insert the module **addMaps** as described above. Switch back to the 'Main' pipeline (again a tab at the bottom of the editor). Insert here the modules **camb**, **syn_alm_cxx**, **alm2map_cxx** and **map2tga**. Additionally insert one **Data Object** and one **Parameter** box from the **Control Elements** section on the canvas. Move the modules around as you want. Peek at Figure 1.1 to get an idea on a possible arrangement.

What has happened so far? When constructing a pipeline, the graphical editor builds an xml-file which describes the pipeline structure (see Section 4.1.3). So far, we have added to this xml file references to other xml files, which describe the input, output and parameters of the required modules. Figure 1.2 shows the xml-description of the module **syn_alm_cxx** in the view of the Xerlin[1] xml-editor, look at Section 6.2 for more details on module-xmls.

Every "module box" on the canvas represents the information contained in these module xmls - not more and not less. In particular, there is no executable code involved so far. In principle, if we are just up to construct a pipeline, we do not need access to this code, it doesn't even need to exist yet!

1.2.2 Building a data flow

The next step is to pass data between the modules. For this, connect module **camb** with **syn_alm_cxx**. Do the same between **syn_alm_cxx** and **alm2map_cxx**. Now, connect the **alm2map_cxx** with the top segment of the **Loop**, then connect the top segment with the inner box (the subpipeline) - a window will appear asking you, what you want to connect (see Figure 1.4). Select the topmost option to connect a 'map.LS_map_pol' datatype to 'inner_output_1'. Then connect the subpipeline with the bottom part of the **Loop**, and the bottom part with **map2tga**. Again a dialog will appear where you should select the first option to connect the 'map.LS_map' as 'output_1'. Now everything is connected, except the **Data Object** and the **Parameter**. Since we want the map to be displayed **and** to be stored, we need to split the connection between the **Loop** and the **map2tga**. Therefore click on the existing connection and draw from there another connection to the **Data Object**. Leave the **Parameter** box unconnected for now, we will explain its presence in a minute.

What have we done? Every arrow connects an output of a module with the input of another module. Obviously, the output and input data must be of the same format, which is described by the *datatype*. When you hover with the cursor over a connection (e.g. between **camb** and **syn_alm_cxx**), a popup will appear with information about the connection and the datatype passed between these modules. The available datatypes are, like everything, defined in xml files, which form the so-called *Data Definition Layer* (DDL), part of the DMC (see Section 5.1.1). Datatypes are usually complex constructions, containing various columns of data, as well as metadata, which are numbers or text strings used to characterize the data. Metadata are usually set by the modules, but some of them are set by the ProC.

The menu displayed when right-clicking on a connection, allows the user to set some properties as well. To illustrate this, please choose the property `persistent` *for the powerspectrum datatype passed by* **camb**.

When connecting modules, the ProC checks whether the output and input datatypes of the module descriptions match. If only one match is found, the connection is done without request for confirmation. If more than one match is found, the user is asked which output to connect to

[1] Xerlin is an open source xml-editor written in JAVA, available at www.xerlin.org

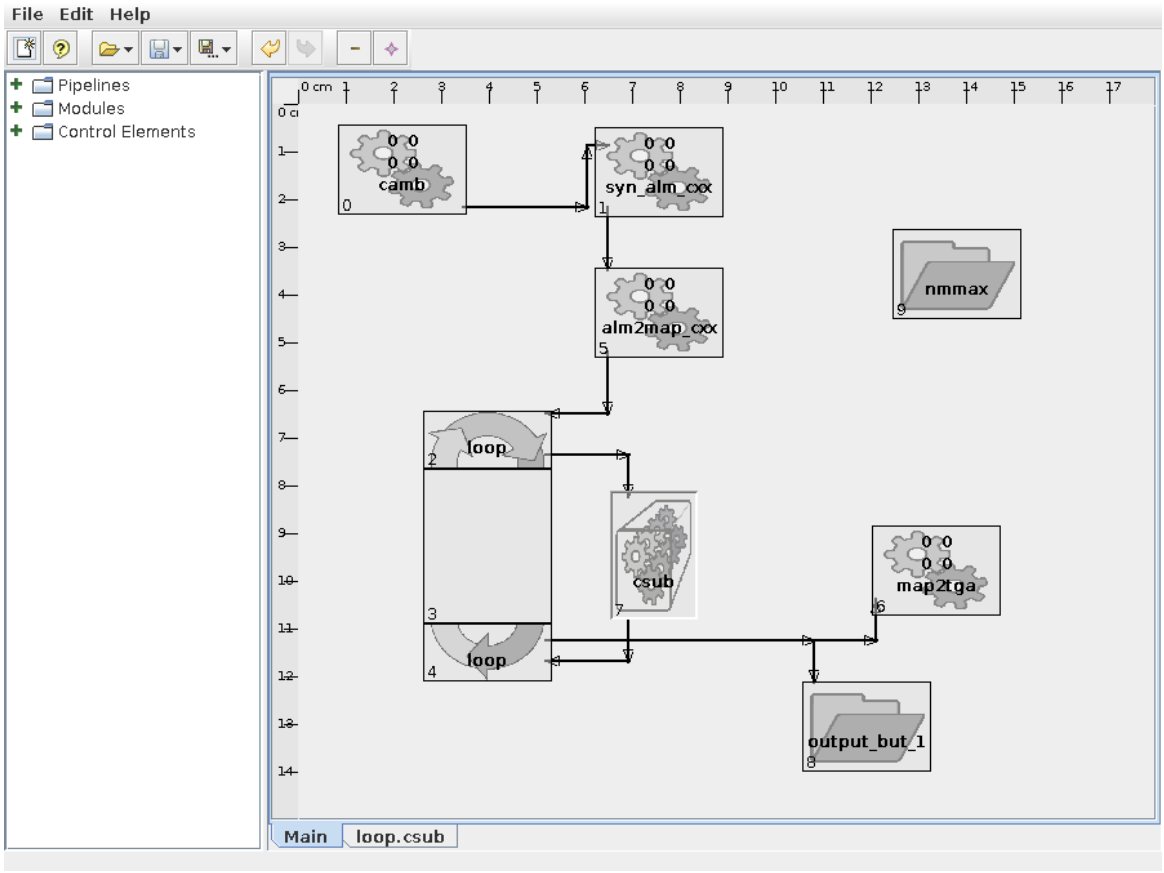


Figure 1.3: Editor view on the quickstart pipeline with data flow.

which input (of course, it is assumed that the pipeline constructor knows about the modules and therefore also knows what belongs where). If no match is found, an error message is displayed. To see this, simply draw an extra **alm2map_cxx** module into the pipeline and place it below the connection from **camb** with **syn_alm_cxx**. Click on the line connecting **camb** with **syn_alm_cxx**, and pull down a "branch" to **alm2map_cxx**. The connection is refused with the message **Modules have no common datatypes!**. To delete the extra module, click on it and press the delete key on your keyboard. When you are done, your pipeline fragment should look like in Figure 1.3).

We recommend to save the pipeline fragment now by clicking the disk button on the top panel beside the folder button. This saves the pipeline-definition xml-file to disk, in a dialog you will be asked for a file name. The name you type will be automatically extended by `.pdef.xml`. There will be an error message stating that there is an isolated Parameter, since the ProC checks for the completeness of the pipeline. This is ok, as we are not done yet. We have not set all parameters for the modules and control elements. This will be done next.

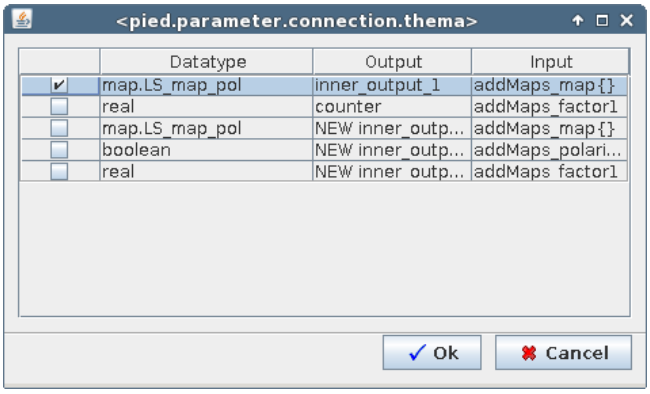


Figure 1.4: The dialog for selecting the datatype to connect.

1.3 Setting the pipeline parameters

1.3.1 Fixing default module parameters

Beside input and output, modules may (and mostly do) also have parameters. These are numbers or strings used by the code. To define them, right click on the camb module and choose **Config Module** from the menu. A panel opens and shows a long list of parameter values, representing the myriads degrees of freedom cosmologists have conceived to describe possible universes. Other modules in our pipeline, like **alm2map_cxx**, have much more profane parameters, like integers determining the map resolution. Most of these parameters have default values, and in many cases we don't want to touch them. Such parameters should be fixed in the pipeline definition (see Section 4.3.1 for more sermon on why to fix parameters), which is done as follows.

Open the parameter menu for all modules in the pipeline, one by one, and click the checkbox **fixed** for those parameters which should be fixed. For modules with many parameters, the opposite way might be easier: click the big **fix** button below the parameter list, and then deselect those parameters which should remain configurable. For our little exercise, please fix all parameters except the ones shown in Figure 1.5. Note that for the integer parameters `nside` in **alm2map_cxx**, although it is fixed, we have changed the default to `128`.

1.3.2 Combining parameters

Quite often we find the situation that two modules depend on the same parameters, and the scientific problem requires that the same values are chosen for them. In our case, this applies to the parameters `nlmax` and `nmmax`, respectively, which are used by both **syn_alm_cxx** and **alm2map_cxx**. To ensure that only consistent values can be chosen, the parameters can be connected to parameter elements, which we have wisely inserted into our pipeline just for this reason.

Connect the parameter element to **alm2map_cxx**. You are now presented a list of the parameters – choose `nlmax`. Now click on this connection and branch it again into **alm2map_cxx**, this time select `nmmax`. Repeat the same procedure for module **syn_alm_cxx**. The final result should look like in Figure 1.6. For more information on using parameter elements, see Section 4.3.3.

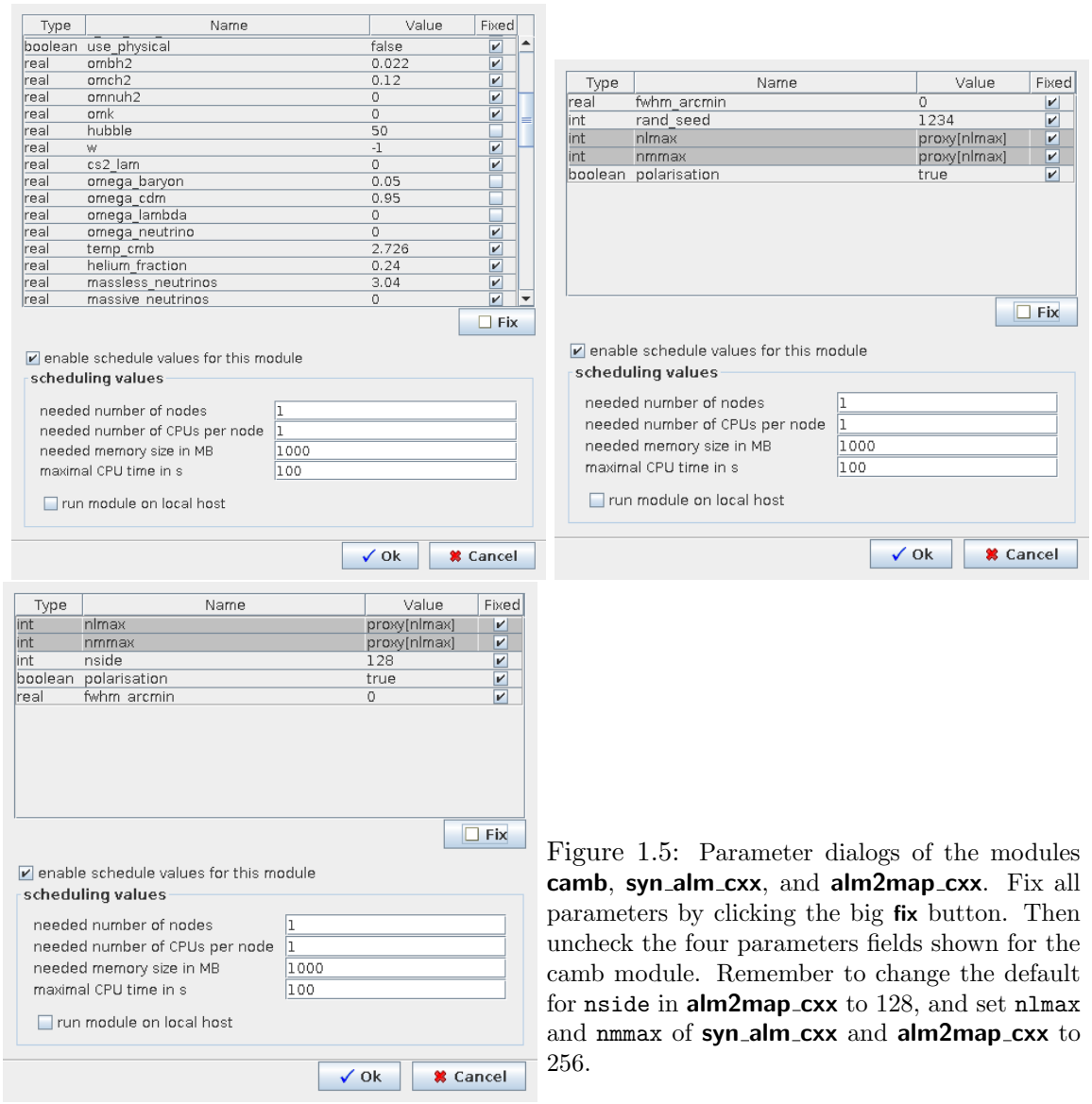


Figure 1.5: Parameter dialogs of the modules **camb**, **syn_alm_cxx**, and **alm2map_cxx**. Fix all parameters by clicking the big **fix** button. Then uncheck the four parameters fields shown for the camb module. Remember to change the default for `nside` in **alm2map_cxx** to 128, and set `nlmax` and `nmmax` of **syn_alm_cxx** and **alm2map_cxx** to 256.

1.3.3 Parameter passing and control flow

Combining parameters is a static construction which helps to guarantee consistent configurations. The ProC allows also dynamic passing of simple data types (int, real, string or bool), which are determined either by module output, or by control elements, into module parameters. Unfortunately, the Level S package does not contain any module which provides simple data types as output – how to write such modules is explained in Section 6.3. Here, we just use parameter passing within control structures.

Until now, we didn't touch the **Loop** itself. To demonstrate this, we will include some scientific nonsense now. The **addMaps** normally takes an arbitrary amount of maps and adds them up using definable factors. Combined with the **Loop** this process is repeated for a definable number of iterations. In our example, it is just adding up one map and to add insult to injury, we will now use the counter, which is used to track the number of iterations, as input for the factor parameter.

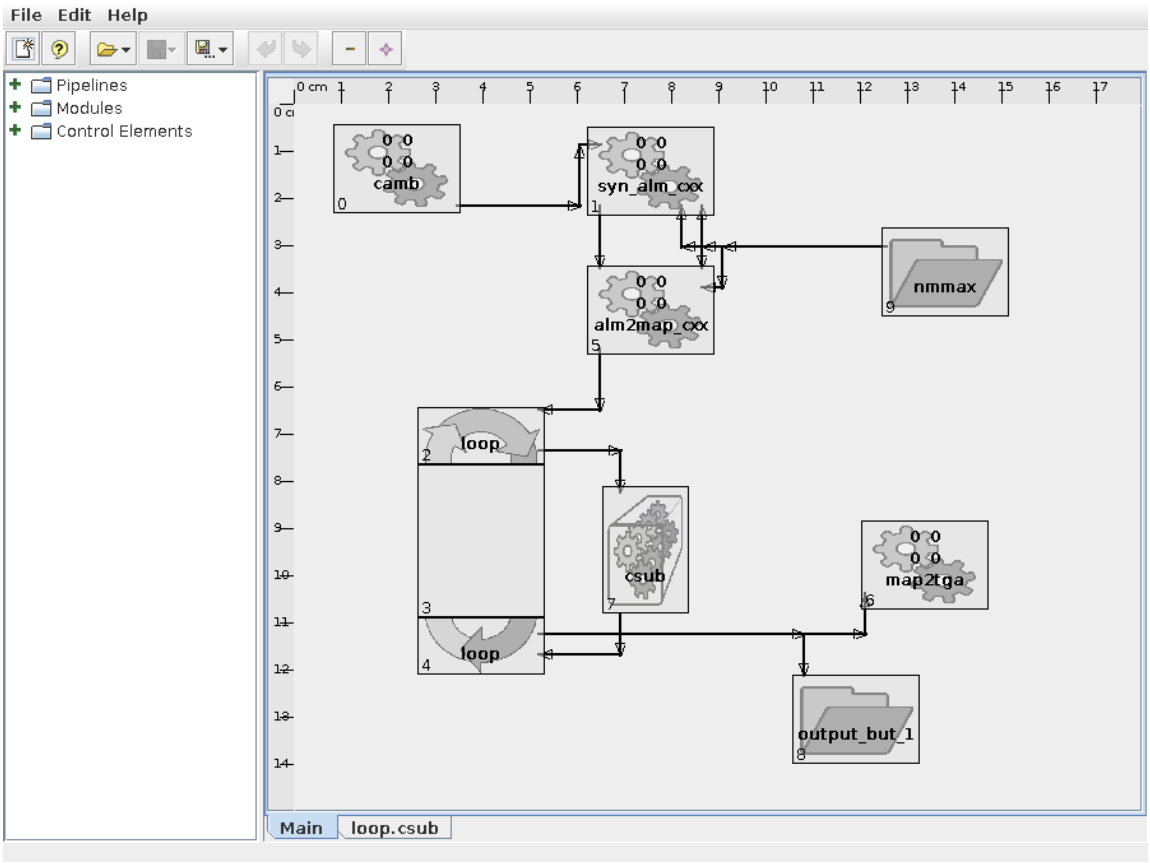


Figure 1.6: Editor view on the complete quickstart pipeline.

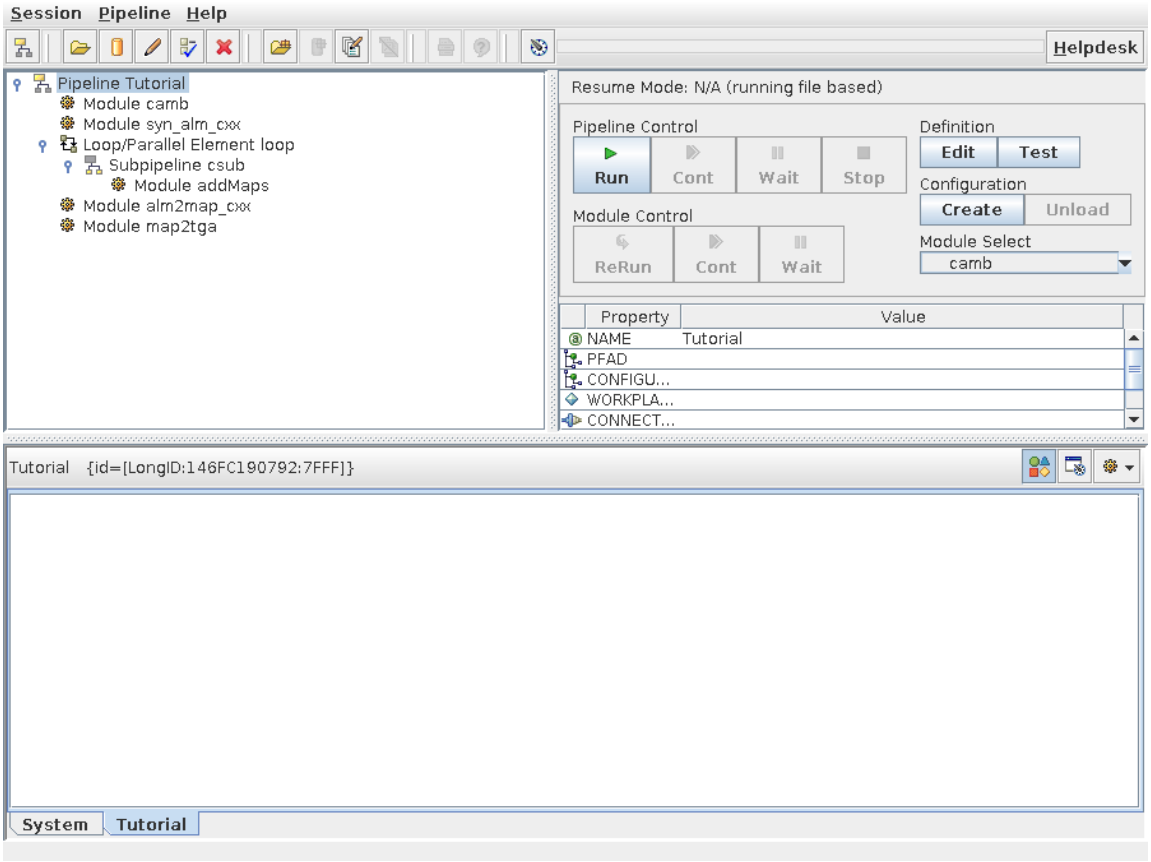


Figure 1.7: Session Manager view on the complete quickstart pipeline. The tabbing indicates that the **addMaps** module belongs to the subpipeline controlled by the **Loop** element.

To do so, switch back to the "main" tab of the Editor, where we placed the **Loop** element. Connect the **Loop** element again into the subpipeline. You are asked for the parameter you want to connect. Choose the `real` parameter `counter`.

When you rightclick on the **Loop** and select **Config Loop** you can configure, as with every module or control element, this specific element. Ignore the upper part of the dialog, only the 'limits' section is important for now. Here you can define how many iterations the **Loop** shall run and in case of parellel computation, how many of these iterations should be run in parallel at the same time. Specify the number of iterations as 5 and the number of parallel runs as 1 (for more information about Loops, read section 4.2.1).

Close the dialog and save the pipeline again (as you have named the xml file already, this goes by one click now), and close the editor window. You return to the Session Manager, which now represents a simplified control flow in its upper left window. The pipeline construction is completed now, but the pipeline is still not executable, as we have not set the configurable parameters, but the way it is displayed in the SeMa already shows whether it is built correctly.

1.3.4 Configuring pipeline parameters

The next step after the definition of the pipeline is its configuration. In the pipeline configuration, parameters are set which may change for different runs. Every pipeline definition can have many pipeline configurations, which are also saved in xml-files. In its usual setup, the ProC requires that a pipeline configuration is loaded. Moreover, it is required that every configurable parameter is confirmed to be set, even if for some configuration the defaults taken over from the pipeline definition are not changed. This is why we recommend to fix all parameters which are likely not to be ever changed.

On the control panel of the SeMa, click **Configuration: Create**. The Pipeline Configurator opens, and you will notice that it pretty much looks like the Editor. The difference is, that all operations affecting the pipeline structure are disabled. Unfold all expandable items of the list on the left side. You can now see all configurable parameters which are left to be defined. For **camb** you see four parameters. Here parameters can be set by double-clicking in the specific entry and entering the desired values, followed by a return. Try it, set `omega_baryon` to 0.05, `Hubble` to 72, `omega_cdm` to 0.68, and `omega_lambda` to, e.g., 0.27,,

What remains is to set the values in the parameter element for `nlmax` and `nmmax`: Set it to 256 (if you like COBE type cmb-maps, try 16). The **Loop** we already configured before, so we can ignore it now, the same is true for **addMaps** which gets its parameter from the **Loop** counter. But we completely ignored the **map2tga** module. Set `outfile` to 'Test.tga' and `viewer` to 'xv' (if it is available under your linux system, otherwise leave this empty).

Not required, but possible at this point is to provide a self-chosen name to the output object. Right-click on the **outfile** element, and select **Determine Data**. Enter there the name `quickstart_map`, and click **OK**.

Now save the pipeline configuration to a file (same button as in the Editor). You need to give a name for it, recommended is a similar one (the same) as for the pipeline definition. The filename is automatically extended with `.pconf.xml`. Then close the Pipeline Configurator window to return to the SeMa. The pipeline is now executable.

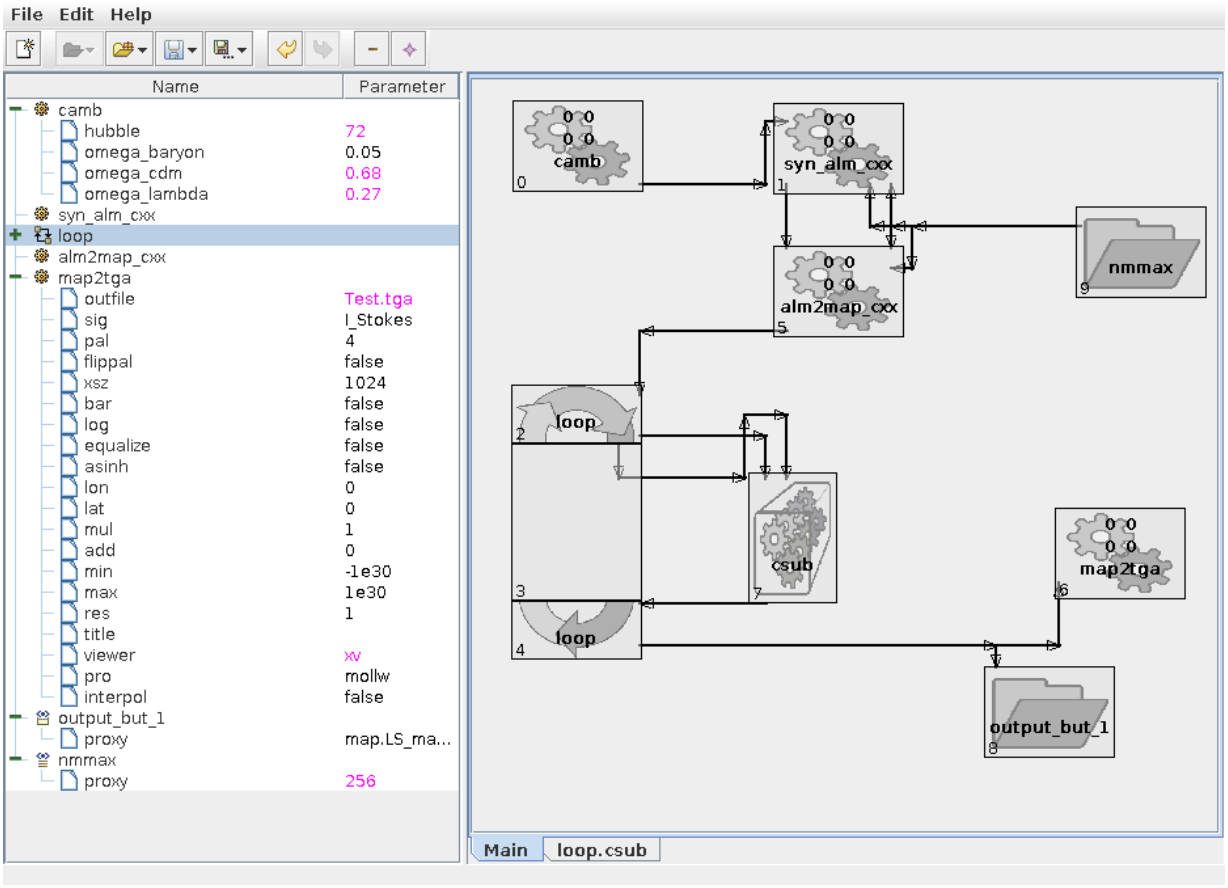


Figure 1.8: Configurator view on the quickstart pipeline, with list to set the value of all parameter elements.

1.4 Running the pipeline

1.4.1 Execution preferences

Starting a pipeline in the SeMa is more or less self explanatory due to its "CD Player" design. However, before we start, we should take care of some settings.

Click the leftmost button on top of the logging window (**Tooltip: Set execution preferences for current pipeline**). The dialog you see first manages the logging level: please verify that all messages up to **Module** are selected (see Section 4.4.1 for details).

If you plan to use the example with a database, please follow the next steps. Switch to the Resume mode dialog and verify that None is selected (you find more details on this further below, or in Section 4.4.5). Although probably done during the installation procedure, double-check that the DMC is enabled and the right database chosen in the dialog DMC; if not, check again Section 3.7.2. Finally, move to the Coordinator dialog and uncheck the option Delete temporary objects if pipeline fails — this is important for our example to work as described! For more information on execution preferences, see Section 4.4.

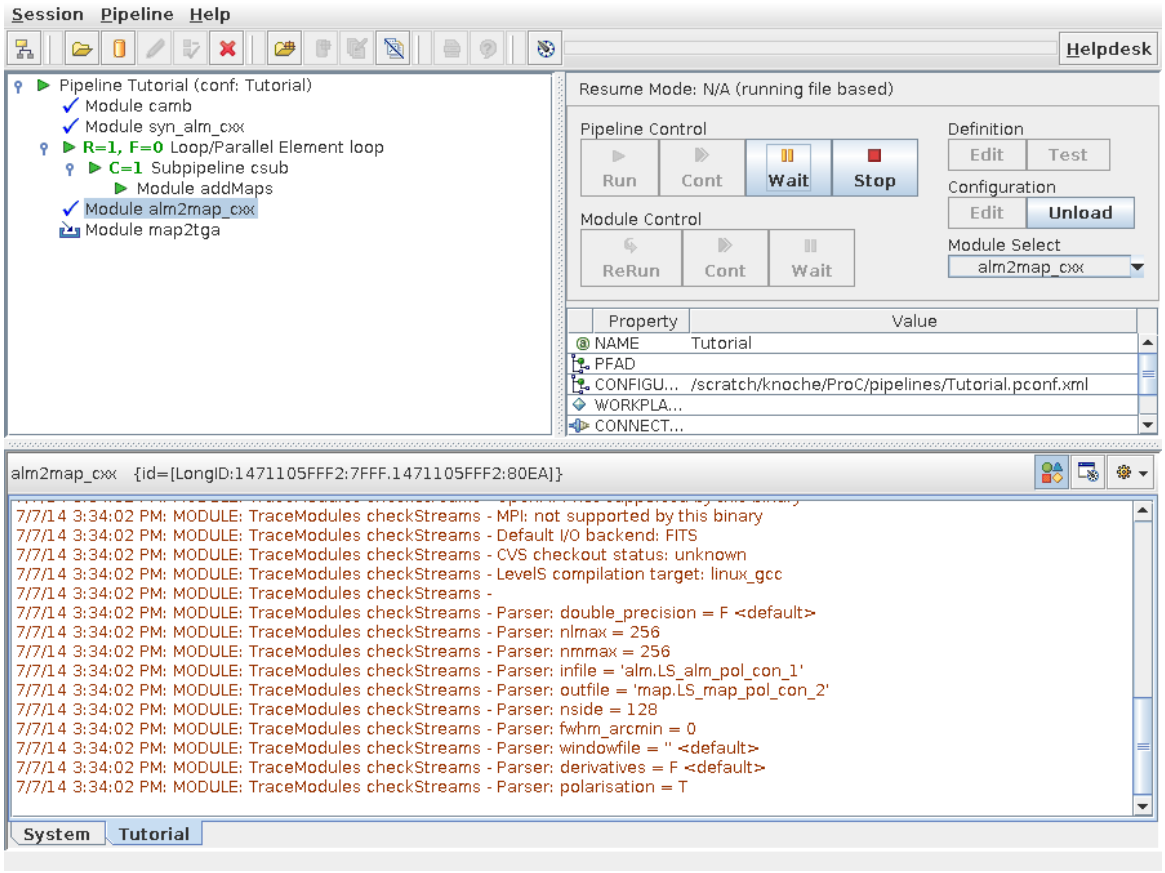


Figure 1.9: The quickstart pipeline during execution. The numbers and signs in front of each module indicate the execution status: Modules **camb**, **syn_alm_cxx** and **alm2map_cxx** are finished succesfully. Currently the **Loop** is being executed with one running module ($R = 1$) and none finished yet ($F = 0$). Module **map2tga** is still waiting for inputs. The logging panel shows the logging output of **alm2map_cxx**, which is currently selected.

1.4.2 Pipeline execution and control

Click the **Pipeline Control: Run** button, and the pipeline starts executing. The symbols in front of the control element and module names (left panel of the GUI) change. Eventually, a list of numbers appears in front of each of them, one by one as the execution passes on. While the pipeline is executed, please click on **Module alm2map_cxx**, and then **Module Control: Wait** on the control panel. A red exclamation mark appears in front of the line, indicating that the execution flow will stop as soon it reaches this point. In the lower panel of the GUI, you see logging messages returned by the modules and the PiCo, if you click and select the respective modules. The color scheme helps to keep them apart.

What is going on now, behind the scenes? The PiCo has started the execution flow, which it takes from the pipeline definition. For each module to start the PiCo writes a parameter file, which is read by the module code on startup, and parsed. For this, a common interface is delivered with Level S, for C, C++ and Fortran modules. The ProC also supports modules written in Java and IDL, and inclusion of modules written in shell script is also trivial; read Section **??** for details. We show here as the example the C++ code of **syn_alm_cxx**:

```
int syn_alm_cxx_module (int argc, const char **argv)
  {
  module_startup ("syn_alm_cxx", argc, argv, 2, "<init object>");
  iohandle_current::Manager mng (argv[1]);
  paramfile params (mng.getParams());

  bool dp = params.find<bool> ("double_precision",false);
  dp ? syn_alm_cxx<double>(params) : syn_alm_cxx<float>(params);
  return 0;
  }
```

The routine `syn_alm_cxx_module` is called from the main routine in `syn_alm_cxx`, and contains some of the typical code segments used for starting up ProC modules: `module_startup` writes a startup message to the logging screen, `paramfile params (mng.getParams())` reads the parameterfile provided by the ProC into a C++ object `params`, using an I/O handler also provided with the Level S package. The parameter file contains all information not only on parameters, but also on the input and output of the module (i.e., which DMC objects to read or to write to). The C++ class `paramfile` provides a set of methods to scan and retrieve the available parameters; here, `params.find` is used to retrieve a parameter specifying the precision of calculation in the module. After this, the object `params` is passed to a templated subroutine `syn_alm_cxx` which reads the rest of the information needed. You can find the entire code involved here under `IDIS/LevelS/Healpix_cxx` to check on more details. General information on how to integrate code into the ProC is found in Section 6.3.

To decide when to start up a module, the ProC employs a forward chaining logic: A parameter file is written, and the executable started, as soon as all information needed is available. Usually this depends on the results of modules placed "upward" in the data flow. However, if two modules do not depend on each other, they may be started in parallel – this we call *intrinsic-parallel* execution, and it depends on the pipeline topology to which extent it can be used. Additionally, as we are using it in our example pipeline, explicit parallel execution of pipeline parts can be arranged by control elements. In Section 4.4.2 you can find more on parallel execution, also in highly parallel environments.

To start a module, the PiCo submits the launch command for the executable to the shell, which is assumed to reside in a directory listed the `PATH` environment variable (for special setup requirements to launch Java or IDL modules, see Section 4.4). The PiCo considers a module finished when it receives a return status, either through the operating system or through a scheduler system (Section 4.4.3). The ProC analyses then the status of execution, and launches the next module in the pipeline chain whenever all information is present. If a module is marked as a breakpoint or the **Pipeline Control: Wait** button is clicked, the execution is halted (letting all running modules finish and leave the pipeline in a defined state). If the **Pipeline Control: Stop** button is clicked, the execution control is terminated immediately, which often leads to an error in the actually running modules.

If you are running with a database, you can try the following experiment. Otherwise just skip over the next sections and continue in Section 1.6.

In order to continue with the exercise as planned, please let the pipeline continue until it reaches the paused module **alm2map_cxx** and then click **Stop**.

1.4.3 Avoiding repetitions by using **resume**

When the PiCo executed a pipeline, it writes all information on module runs into the database. These `MRun` objects are associated to a `PRun` object containing all information of the pipeline run (see next section). This complete tracking enables a nice feature of the ProC: the ability to *resume* pipelines. Since this data is only stored in the database, this also means that this functionality is not available in file-based mode.

Open the pipeline preferences menu of the SeMa (as above), select **resume** and check resume mode **optimistic**. Close the dialog, and restart the pipeline. You will see now that the pipeline proceeds very fast, and the modules **camb** and **syn_alm_cxx** are now skipped without ever being executed. Only the module **alm2map_cxx** of the pipeline which has not been executed in the first run will now be executed plus every module depending on the output of it.

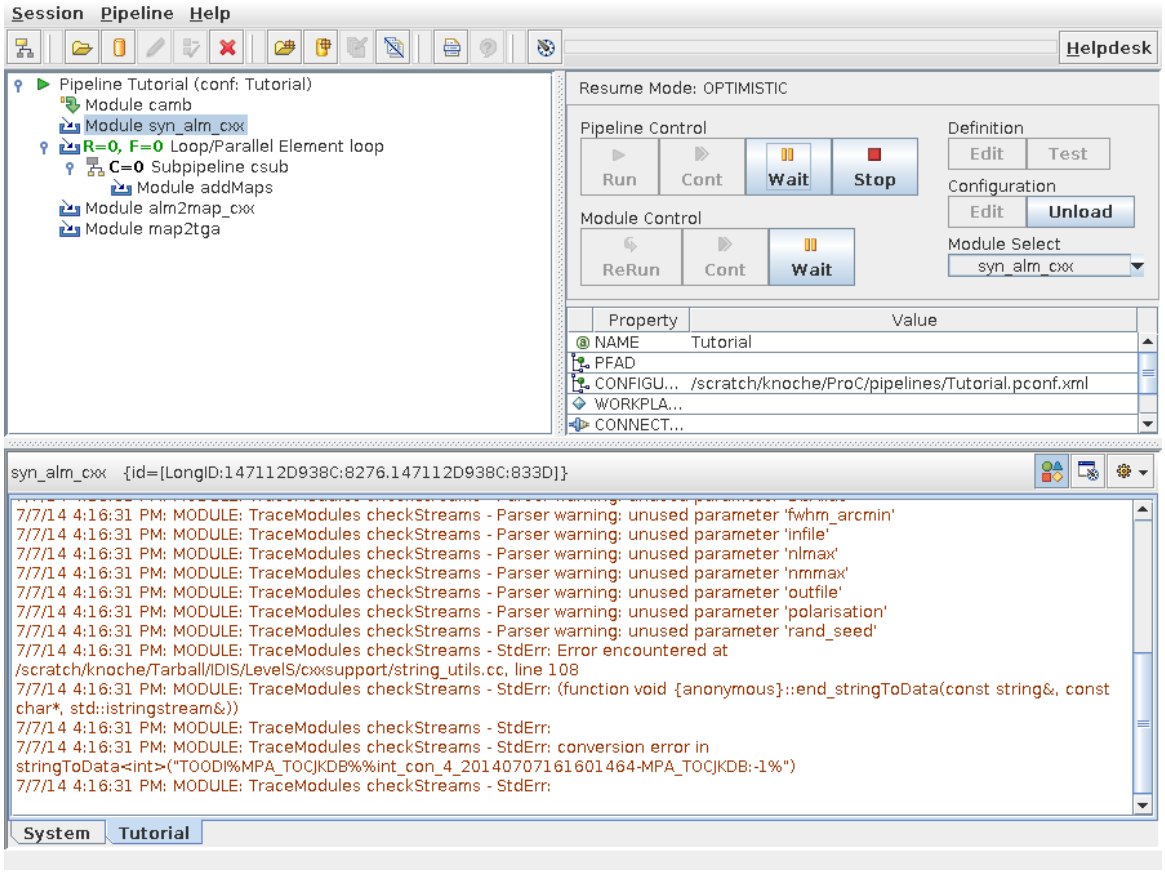


Figure 1.10: Second run of the quickstart pipeline in resume mode. **camb** has been resumed, i.e., the values of a previous run have been loaded from the DMC instead of executing it again.

In serial execution, required if using IDIS-in-a-box, resume is indicated by a kinked arrow in front of the module. Resume will then work up to the point where the pipeline has been stopped, all following modules will be executed.

What is going on? In the resume mode, the PiCo checks for every module whether a result

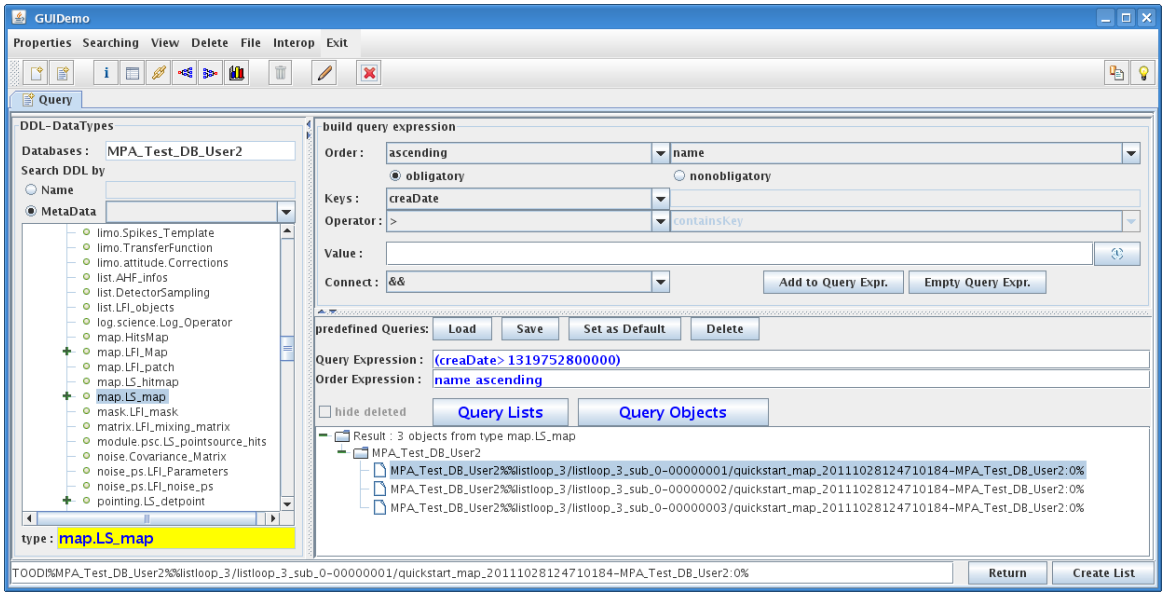


Figure 1.11: Result of a query for `map.LS_map` objects created by our example pipeline, showing the three maps created for different sets of cosmological parameters. The details of building the query are found in the text.

of this module has been obtained in previous runs for *exactly the same input data and module parameters*. If specified, the ProC even checks whether the md5-sum of the module executable is unchanged, which guarantees that the same binary is used. If all these conditions are fulfilled, the PiCo does not execute the module again and just pipes this previous result of the module run to the next module in the pipeline. Here, it checks again whether the resume conditions are fulfilled. If not all conditions are fulfilled, the module *and all subsequent modules of the pipeline* are executed in the actual run. More information on resume is found in Section 4.4.5.

Let the pipeline continue to run. When all modules are reported in o.k. status and the Start button light up again, the pipeline execution is finished. We can now check for the results.

1.5 Looking at the results

All data management in the ProC is done by the Data Management Component (DMC)[2]. The DMC handles all data objects passed from one module to another, and also the init-objects used for module startup.

1.5.1 Querying and viewing data

To look at the data we produced, open the DMC GUI by clicking the button marked with the database symbol (looks rather like a can) in the top panel of the SeMa. A GUI appears as shown in Figure 1.11. As shown in the figure, select `map.LS_map` (listed under `Common.GenericCore`) in the panel of available data types on the left; this selects the data type for queries acting on the database. In the top right panel, you find a number of tools to help you building the query. First we define a sort order by selecting **Order: ascending — name** from the drop-down panels. Then we define

[2]Although it is possible to run the ProC without it, see Section **??**

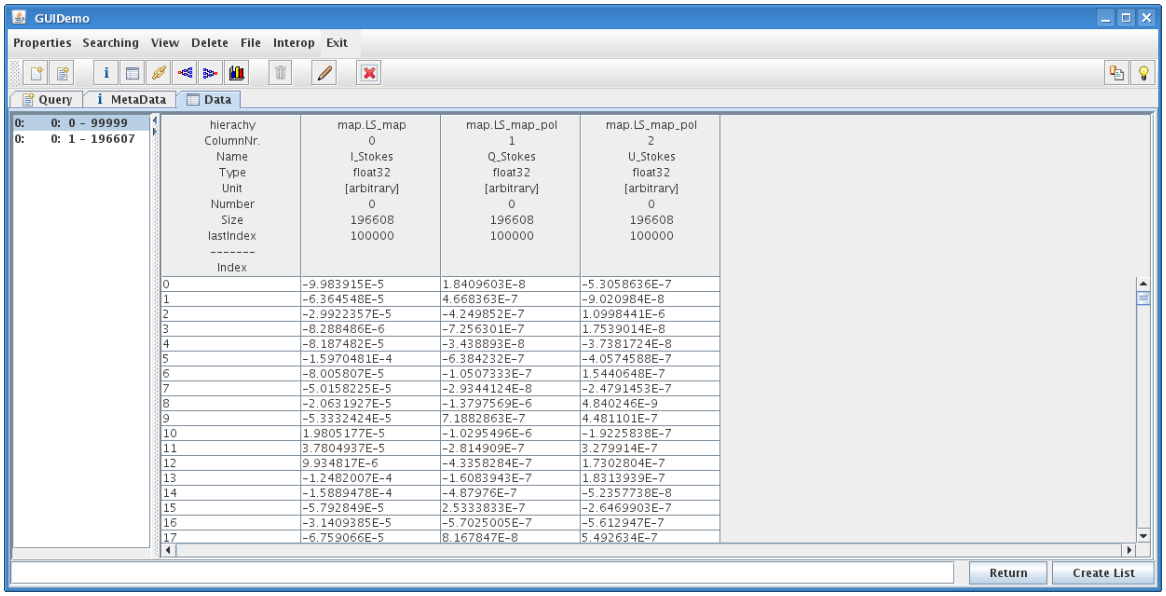


Figure 1.12: Data columns of a selected map produced by the quickstart pipeline. The columns show the Stokes parameters for each pixel in the (polarized) map.

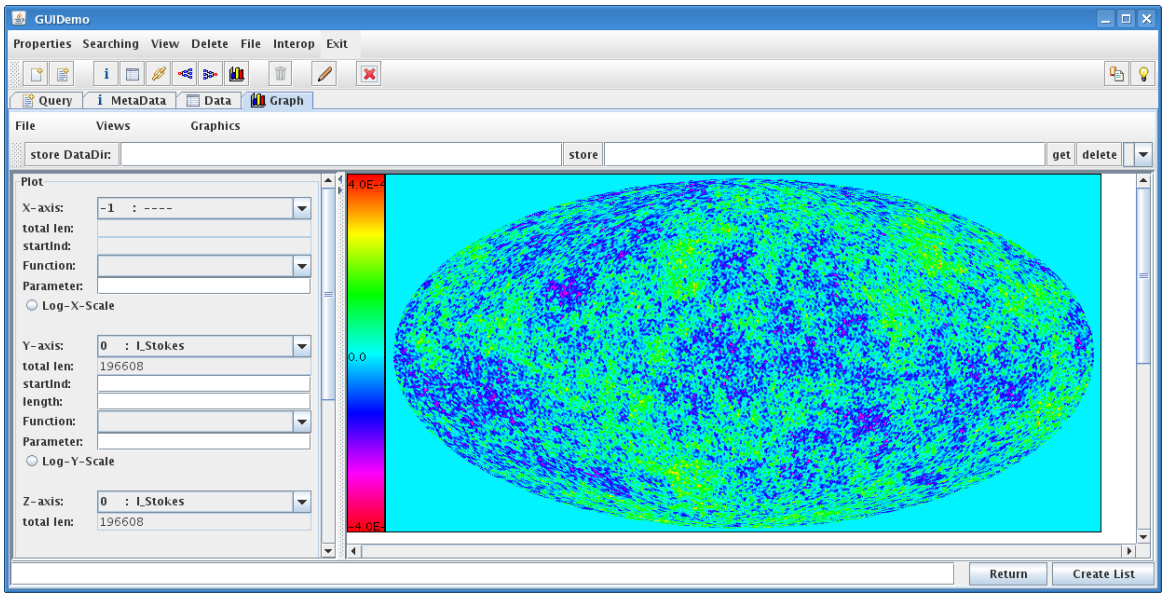


Figure 1.13: The intensity map of the simulated CMB in a graphical view. If a different y-column in the left panel had bee chosen, we would have obtained the Stokes Q or U polarization map.

a selection criterion which makes sure that we retrieve only the data of our pipeline run: assuming that no other pipeline has produced maps today, we check **obligatory** (default), enter **Keys: creaDate**, then **Operator:** >. As the value of a creation date must be given in Unix timestamps, which are quite obscure for any brain not made of silicon, the DMC GUI offers some help here: click the button to the very right of the **Value:** field, then a window opens which allows you to select a date/time in the human-readable format. Enter an appropriate data/time here, and click **OK**. The converted time stamp appears, take it over into the **Value:** field by clicking OK, and click **Add to Query Expr.**. Finally, in the panel right below, you find the selection and order criteria for the query in JPAQL/JDOQL statements.[3]

Now click **Query objects**, and (ideally!) the map we have just produced appears in the lower right panel of the GUI, as shown in Figure 1.11. You can identify them by the name segment "quickstart-map" we have chosen for the object element in the ProC; however, it has been embedded into the unique ProC naming scheme, which makes it difficult to use it in queries. Note that our order criterion has lead to an output sorted after the loop number in which the map has been produced. You can select an object by clicking on it, and by a following right-click you open a menu showing possible actions on the data objects. Choose **Data** from this menu, to retrieve a table showing the data columns of the selected Healpix map (see Figure 1.12. The DMC GUI also offers tools for a graphical display of data: If you return to the Query-tab in the DMC GUI, now right-click on the same map and select **Graph**, then inside the Graph-tab select **Graphics** → **Contour Plot** → **Healpix Map**, you see a picture of the map as shown in Figure 1.13.

So, how does all this work? In order to maintain maximum I/O performance, the DMC stores all data in binary files (i.e., as serialized Java objects) on the file system. To provide easy access to the data, information about all data produced, called *metadata*, is stored in a database. Metadata can be used in queries to select and order data objects. Once a data object is found, the DMC manages access to the data files, and provides a selection of tools to look at the data in a convenient way (like the Graphics tool we have used), or to export them to other data formats like FITS or VOTables (including broadcast functionalities to other applications, see Section 5.2.2). Which data columns and which metadata a data object has, is defined by the *data type*. All available data types are contained in the Data Definition Layer (DDL), which is provided to the DMC as a set of xml files. For more information on the functionality of the DMC, see Section 2.3 and Chapter 5.

1.5.2 Looking at the data history

In complex scientific projects, it is often relevant to reconstruct the generation history of a data product, e.g., to check which parameters, initial data or scientific code has been used to produce it. It is one of the essential features of the ProC/DMC package to provide this information for all data products from pipelines running in it (see Section 2.3.4 for more discussion).

We want to retrieve the history of one of our generated CMB maps. In the Query-tab, select — for a change — the second one, right-click and select **History**. A new tab is opened, containing a tree view of all data objects generated by the pipeline producing the maps (Figure 1.14). If you double-click on the `powerspectrum` object, the right panel shows all cosmological parameters used to generate it. Also other information can be retrieved: for example, the ProC stores the md5 checksum for the executable code used to produce each data product, which allows in our case to check the code version of **camb** used to produce the powerspectrum. The DMC GUI provides the same functionality on all data objects shown in the history view, as it does for query results. If

[3] If you are proficient in JPAQL/JDOQL, you could enter them here right away, instead of using the query builder.

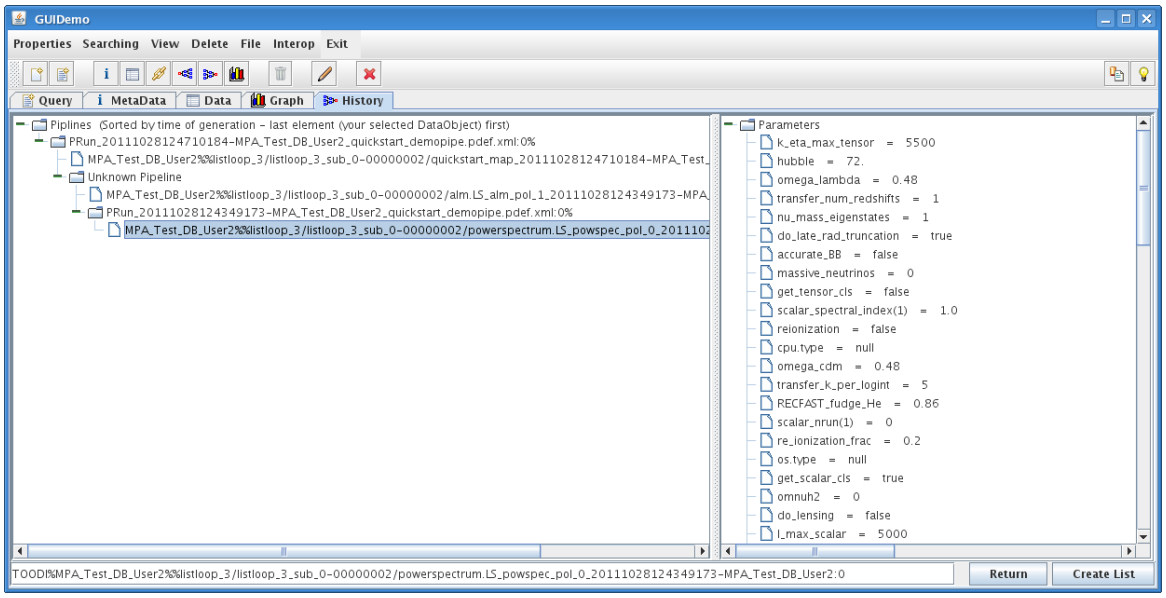


Figure 1.14: The history of data product generation as shown in the DMC GUI. The left panel shows all data products on which the chosen product depends in a tree representation. In the right panel, the parameters used to produce the powerspectrum are shown.

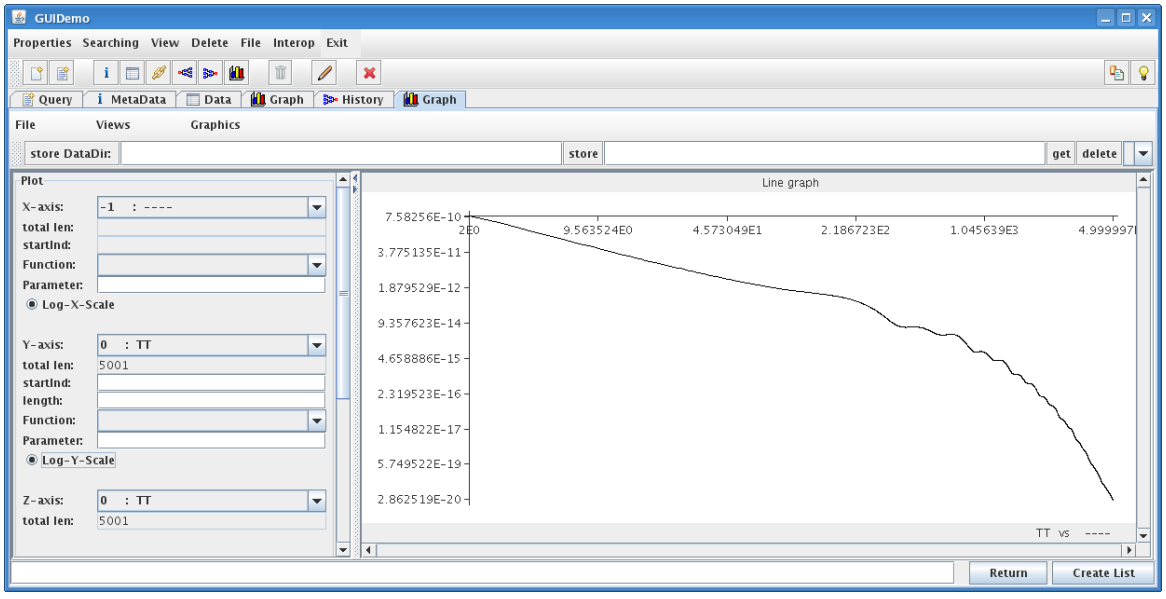


Figure 1.15: This plot has been generated from the history view by selecting **Graph**, an then **Graphics** → **LinePlot**. Please note that log-scale has been selected for both x and y axes.

you right-click on the powerspectrum object and select **Graph**, you can also visualize it, e.g., as a Log-Log plot shown in Figure 1.15.

The history functionality demonstrates that the ProC does more than just starting modules in the right order. When a pipeline runs in the ProC, not only the final and intermediate data products are stored in the DMC. Additionally, the ProC writes objects collecting all the information needed to re-enact the pipeline run. This information is needed, e.g., for providing the resume functionality we have used above. The DMC GUI uses this information for its history functionality – for more information how to use it, see Section 5.2.4.

1.5.3 Deleting data

Lots of data objects in a data base provide lots of information, but are also a nuisance, as the performance of the database deteriorates with an increasing number of objects in it. It is therefore advisable to keep only those objects in database which are really needed, and to delete all others. To attain this goal in a way requiring as little as possible user interaction, the ProC distinguishes between *persistent* and *temporary* objects; the former are retained in the database, while the latter are deleted after each pipeline run. Deleting all temporary objects right away after every pipeline run, however, has the disadvantage that it inhibits the use of the resume functionality – obviously, data from previous module runs can be used for resume only if they are still in the database.

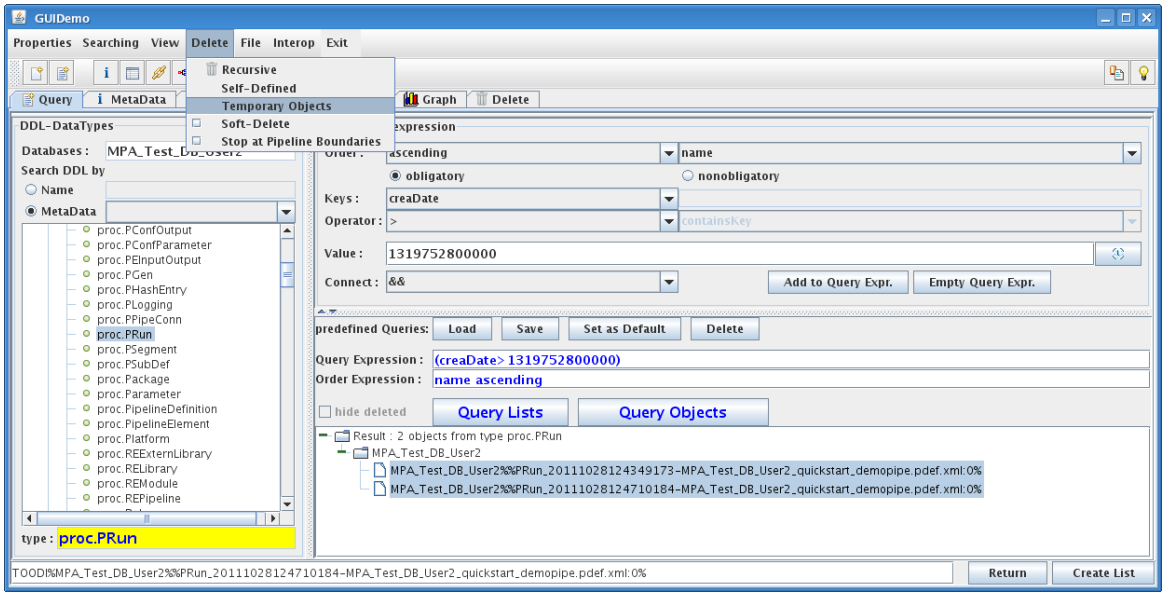


Figure 1.16: `PRun` objects selected to delete all temporary objects produced in the corresponding pipeline runs. With clicking **Delete** → **Temporary Objects**, the DMC starts to select the objects for deletion.

In our example, we have configured the ProC to delete temporary objects only after successful pipeline runs, but not after failed ones. This allows to use resume in the important special case of unexpected failures, to avoid unnecessary repetitions of successfully performed module runs before the failure occurred (imagine that in usual scientific applications, module runs may take a lot longer than in our example). We have then simulated a failure by stopping the pipeline after pausing it (the ProC treats this like a failed run), and then used resume. Therefore, the `alm.LS_alm` objects from the first pipeline run are still in the database, which you can easily check by going to the

history tab and look (successfully!) at the data of the `alm.LS_alm` objects. After our pipeline has now finished successfully, we want to get rid of them by using the DMC GUI.

Go back to the query tab and select in the left panel the datatype `proc.PRun` (listed under `Common.GenericMeta`). Luckily, the query settings for selecting and ordering the results remain unchanged, as we need them again. Click **Query Objects**, and you retrieve the `PRun` Objects corresponding to the two pipeline runs we have executed. Select both of them (as usual by holding the Control-Key when selecting), and chose **Delete** → **Temporary Objects** from the top panel (see Figure 1.16). The DMC takes a while until it found all objects selected for deletion, and displays them in an extra delete tab (see Figure 1.17). Clicking the yellow **Delete** button removes them from the database for good! You can check by returning to the history tab and try to display the `alm.LS_alm` data again – you will not succeed this time!

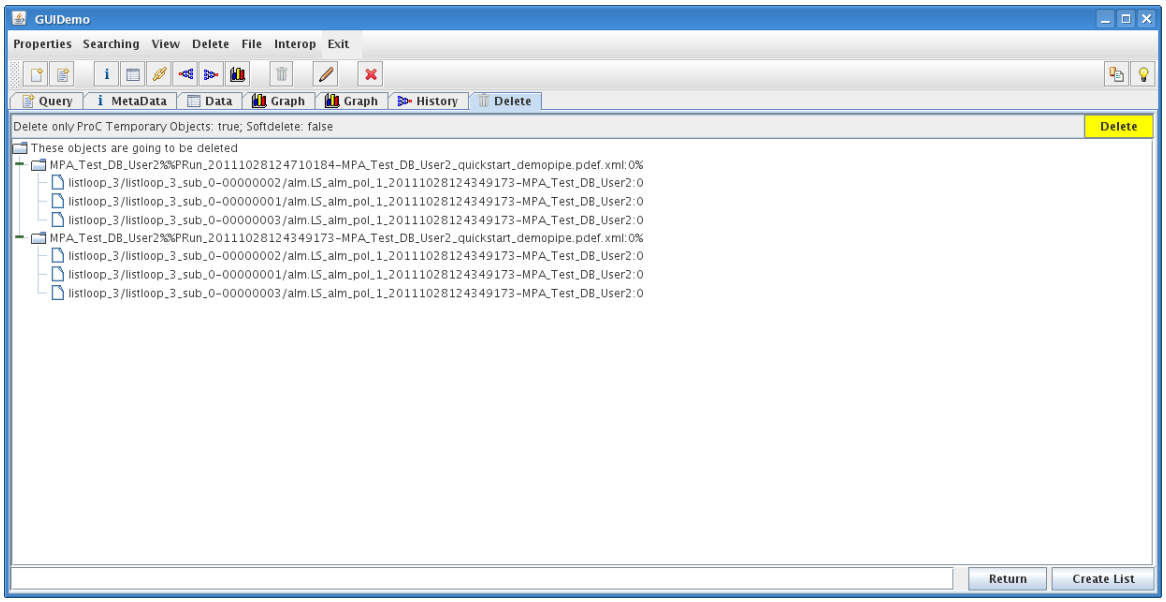


Figure 1.17: The temporary objects found for our two pipeline runs are selected for deletion. The reason that objects for both pipeline runs are found – although the second one just resumed the objects from the first — is that the DMC stores links in the resumed run, which are shown here. In fact, each alm-object exists only once.

This just demonstrates one function the DMC GUI provides to support data deletion. Essentially, any data objects returned form arbitrary queries can be selected for deletion. The DMC also distinguishes between soft-delete, in which only the data are removed but the metadata remain in the database, and hard-delete, which removes everything. More about the options of data deletion can be found in Section 5.2.6.

1.6 How to read on

Our exercise offered a panoramic view over the functioning of the ProC workflow package. It covered many typical, simple applications, and provided links for further reading on special topics. However, simplicity and clarity has a price: complex control structures like iterative loops, support of parameter sampling, the concepts of object lists and query modules, work on parallel environment

and GRID support are examples for important features of the ProC/DMC which had to be left out. In the following chapters, this tutorial will give a complete introduction into all details of the ProC/DMC package, and is structured as follows.

Chapter 2 gives an outline of the history and idea of the ProC/DMC with respect to their role in the Planck Integrated Data and Information System (IDIS). It describes many aspects arguing for the use of the system on the project management level.

Chapter 3 describes the installation options of the ProC/DMC package, both for users inside and outside the Planck collaboration. It also covers legal issues like licenses etc.

Chapter 4 offers a task oriented manual of the ProC, including all usage features of the Editor, Configurator and Session Manager.

Chapter 5 offers a task oriented manual of the DMC, including the usage of the DMC GUI, some relevant background information on data storage and how to define datatypes in the DDL.

Chapter ?? describes how to integrate code into the ProC and DMC. It covers module descriptions (xml) and libraries to be used in the code for module startup, as well as I/O libraries as the Planck CLLIO, the native DMC API called TOODI and the Planck Level S interface.

If your specific question or problem is not covered in this tutorial (which is currently not unlikely as the document is incomplete), please don't hesitate to contact the people mentioned on page 1 (after reading that page!).

Chapter 2

Introduction to IDIS

IDIS is an acronym for the *Integrated Data and Information System* of the Planck Surveyor Mission. It consists of many components, among them commercial tools for document and software management. The part of IDIS covered in this tutorial are those tools which are designed to support the data processing for the Planck Mission. These are, in particular:

- *The Process Coordinator (ProC)*, a workflow engine which allows the graphical construction of data processing , so-called "pipelines", from existing software modules
- *The Data Management Component (DMC)*, which handles and stores intermediate and final data products of these pipelines.

All these components can in principle be used stand-alone, and all have certain functionality by themselves. However, the full capability of IDIS is revealed only when all components are used together. This chapter will introduce into the functionality of IDIS, explain how it works, what it does, and how it can be installed. The following two chapters will explain how to use the ProC and DMC. Finally, the last chapter will explain how to write scientific code so that it can be used with IDIS.

Although IDIS has been developed mainly follwing the requirements set by the LFI-DPC, it is important to understand that the IDIS components introduced here are not "something running at the DPC". ProC, DMC and Federation Layer can be used by anybody inside the Planck project, and installed in any Planck institute, on any workstation or Linux-PC. For evaluating and practicing the use of the IDIS software, the MPAC provides an IDIS-in-a-box package, which can be installed with a few steps on any laptop.

2.1 IDIS Components

In this section we give a brief description of the IDIS components covered in this tutorial, explaining a prospective user what they are good for and what functionality can be expected.

This section is written particularly for people who want to know what IDIS is, and whether it provides the functionality they need for their research project. People who already know what IDIS is, or just want to get into practice without bothering about background information, can skip this section

2.1.1 The Process Coordinator

The IDIS Process Coordinator, short *ProC*, is a graphical tool to design, configure and execute data production workflows, in the astronomical context often called *pipelines*. A pipeline is a structure of well defined data processing units, called *modules*, which are connected by *data objects* passed in between them. The ProC, therefore, acts similarly to a script in starting these modules, but it provides significant additional functionality, in particular:

- Graphical design of pipelines in a *Pipeline Editor*, with automatic detection of data types and the parameters accepted and delivered by the modules of the pipeline. It thus provides an automatic detection of syntactical correctness and completeness of a pipeline;
- Control elements for standard processing flows, like loops and conditions, as well as (independent) parallel execution of parts of a pipeline. One particular feature of the scientific application of the ProC is the *Sampler Control Element*, supporting sampling problems in N-dimensional parameter spaces, such as Monte Carlo, minimization or integration.
- Convenient handling of pipeline configurations (i.e. different parameter sets for the same pipeline design), which can be viewed and edited in a graphical *Pipeline Configurator*;
- GUI based control of pipeline execution in the *Session Manager*, allowing to stop, pause and continue pipeline execution (similar to a tape recorder).
- A standardized logging of pipeline execution, together with all logging information written by the modules themselves, which is displayed in a designated window of the Session Manager. Different logging levels can be configured, and distinguished by different colors
- Convenient and flexible handling of resources, such as parallel processing capacities, scheduling queues, and remote execution hosts.
- Readiness for use of the computational Grid through the generic Grid Application Toolkit (GAT).

If run in combination with the DMC, the ProC additionally provides

- Resume functionality, allowing the efficient re-use of results from modules calculated for the same inputs and configuration in previous pipeline executions. This can significantly speed up repeated executions of workflows, and avoids loss of calculation time in case of pipeline crashes due to network or system failure.

- Storage of the data history, i.e., a detailed log of the way a data product was generated, providing reference to the data products which have been used, and an identification of the executed code by md5 checksums.

If a central Federation Layer Server is used, the ProC further allows to

- Control the execution of pipelines from a remote resource (e.g., from the user's laptop at home).
- Central "manager" control over pipelines running on an execution resource, allowing the administrator to stop and restart pipelines, if necessary with different execution settings.

The Process Coordinator is implemented in Java and XML and therefore platform independent. It is described in detail in Chapter 4.

2.1.2 The Data Management Component

The IDIS Data Management Component *(DMC)*, was developed to allow the secure and convenient handling of large scientific data sets, providing the high I/O performance of file systems and the data management capabilities of databases at the same time. There are essentially two interfaces to access the DMC: the DMC GUI, which allows interactive searching and displaying of data and metadata; and the DMC-API, available to the module programmer. The main features of the DMC are summarized as follows:

- GUI based construction and reuse of queries on data products, which are identified by a set of metadata
- Graphical tool to view data objects (e.g. plotting columns of data, showing HEALPix maps)
- Versatile data model which permits user-defined data type definitions with object oriented inheritance.
- Ability to connect to diverse database types, like Oracle, PostgreSQL and Apache Derby.
- Generic API for different programming languages (C, C++, Fortran, Java, IDL)

If used with the ProC, the DMC also stores all information on pipeline runs written by the ProC. In this case the DMC also allows to

- Retrieve information on the data generation history, conveniently linked to every data object through the DMC GUI.

The DMC GUI is described in more detail in Chapter 5, the DMC API and other programming interfaces based on it in Chapter **??**

2.2 Operation Modes of the ProC

2.2.1 How the ProC works

The ProC has two major tasks:

1. Designing and configuring data processing pipelines in a standardized way
2. Controlling the execution of pipelines and the generation of data products

The first task is served by the pipeline editor and configurator, which allow the graphical construction of pipelines. Modules are selected from a list, arranged, and connected through data flows. Finally, the results are saved in two XML files: a pipeline definition, `<pipename>.pdef.xml`, which contains the structural information of the workflow (which module depends on which result), and a pipeline configuration, `<pipename>.pconf.xml`, which contains information on all parameter settings for a pipeline. There can be many different pipeline configurations for a particular pipeline definition, and the ProC is able to detect which configurations match the definition. Alternatively to saving in a file, pipeline configurations can also be saved in the DMC as pipeline configuration objects; this is currently not working for pipeline definitions.

Although a "Save to DMC" button is implemented also for pipeline definitions, this functionality should not be used, and should be disabled in the ProC settings

Another structure the editor can produce are subpipeline definitions, which contain pieces of workflows which can be reused in bigger workflows. Subpipeline definitions are stored in files named `<subpipename>.subdef.xml`; they cannot be stored in the DMC.

Being coded in simple XML-files, it is obvious that workflows designed with the ProC can easily be communicated: A pipeline definition and/or configuration produced by one member of a project, can easily be sent to some other place and executed there on a local IDIS setup — provided all modules and data types used are present there as well.

Another aspect of XML files may be that people feel invited to save time in creating similar pipelines or configurations by editing the XML files directly. We strongly discourage this, as the proper execution of manipulated XMLs cannot be guaranteed!

Fulfilling the second task is the responsibility of the *Pipeline Coordinator (PiCo)*, which may be considered the background engine of the ProC. The PiCo analyses dependencies of modules in the workflow, and executes the individual modules of the workflow as soon as all inputs needed by the module are available. The PiCo has a fundamentally parallel design, which means that modules which do not depend on each other will be executed in parallel. The PiCo controls available computational resources through the *Grid Application Toolkit (GAT)*, a generic interface to external scheduling systems like the Portable Batch System (PBS) or the Sun Grid Engine (SGE), which are used on many cluster systems or supercomputers. Its main function, however, is to support the connection to the computational grid through tools like *Globus* or *G-lite*. For operation on a single computer, GAT also provides a local mode which just submits the jobs to the local operating system.

Data flows between modules are generally I/O operations on the file system, either managed by the DMC if this is enabled, or via direct file exchange between the modules. This way of

communication has been chosen to keep as much freedom as possible for the PiCo to distribute modules to available resources, without requiring a common file system or common memory. The price of this flexibility is, of course, that communication of data through file I/O is relatively slow, which has to be considered in pipeline design: The modularization of a workflow should always be such that the calculation time of a single module is comparable or larger than the time needed for I/O of the data!

It is clear that under certain circumstances it may be unavoidable to write modules which do virtually nothing in terms of calculation time, for example, reformatting data in order to be used by the next module. It has long been discussed to support memory resident data objects by the ProC/DMC software for this reason, but this has not been implemented so far. If you see a strong need for this, please communicate this to the IDIS development team. Before you do this, however, please consider the possibility that your performance requirements may be solved by an appropriate module design.

But not only the data itself, also all information needed by a module to start up are communicated by the ProC through the file system, either by *parameter files* (in so-called file-based operation) or by DMC init objects (in DMC-based operation). Obviously, this also defines the standard interface of a ProC module: as its only argument, it accepts the name of a parameter file or init object, which it then needs to read and parse in order to obtain all information to operate (for example, where the data object passed to the module by the previous one in the workflow can be found). The Planck Level S package, a compilation of modules designed to simulate Planck data within cosmological models (included in the IDIS distribution), provides a standard interface for integration into the ProC which is meanwhile widely used in Planck.

Another option to easily integrate code into the ProC/DMC engine is the *module wrapper*, which allows to run programs as ProC modules without modifications. The module wrapper will be a standard part of the distribution after release of IDIS 2.8

Another important function of the PiCo is to generate and store logging and history information on the pipeline execution. How and in which detail this is done, depends on whether the ProC is used with the DMC or without, as explained in the following.

2.2.2 DMC-based vs. file based operation

Depending on the preferences set by the user, the ProC can run with or without the DMC. If DMC operation is enabled, the ProC communicates all data between modules through *DMC objects*. Clearly, this requires that the module has been written to read and write data from and to the DMC, using the DMC API. If the DMC is disabled, the ProC assumes that data objects have been written to files by the modules, and communicates the names and paths of these files to the subsequent modules. If the Level S standard interface is used to integrate code, it can be decided at compile time of the module, whether the DMC API is used, or whether all data are written to and read from FITS files.

The Level S module interface is compiled either for DMC use or file based use, and the operation mode of the ProC must be consistent with this choice. A mismatch will lead to an error. Note that this affects not only Level S modules, but all modules using the Level S module interface. Modules which use the native DMC interface and have not programmed any FITS or other file alternative cannot be used in file based ProC operation (and vice versa).

When the ProC operates with the DMC, all information on the pipeline run is stored in special DMC objects, called *Prun* (for *P*ipeline *run*) and *Mrun* (for *M*odule *run*) objects. They contain all logging and parameter information, and links to all produced data objects stored in the DMC. As these objects cannot be easily manipulated by the user, they can be linked to the generated data objects, in order to provide a log of the generation history of the data product. Within this history, not only the data products used and all module parameters can be tracked; the ProC also stores the MD5 checksums of the executables of all modules used, which allows to exactly determine which code versions have been used to generate a data product.

In the normal case, intermediate data produced by a pipeline are deleted after termination of the pipeline (but still some information on them, the metadata, are kept, see section **??**). However, the user can specify in the ProC preferences to keep them. This allows another functionality of the ProC only provided when used with the DMC, called the *resume mode*. In resume mode, the ProC can load the result of a previous execution of a module with the same input and parameters, instead of starting it again. The result is passed to the next module, and again it is checked whether the conditions for resume are fulfilled. In particular with modules which run over a considerable time, this can mean a significant speedup when executing a pipeline again and again, for example for debugging purposes, or, of course, also in regular operations when similar pipelines are executed.

If the ProC works in file-based operation, all data and parameters exchanged can be found in a directory named <`pdefname`>, if <`pdefname`>`.pdef.xml` is the pipeline definition, and under the same path as the latter. Under the main directory, every run of the pipeline is assigned a directory carrying its timestamp, below which subdirectories are found containing the results, parameters and logs written by the ProC. As this file structure provides no protection against being changed or deleted by the user at any time, the ProC cannot rely on this information for its operation; using the information written by the ProC during a file-based pipeline run in a consistent way is in the responsibility of the user.

This means, that when the ProC is operated without the DMC, functions like data generation history or resume are not available. However, depending on the application, it may still make a lot of sense to use the ProC in this way. File based operation is generally faster, and also more robust in heavy parallel operations, in which the DMC would act a as a serial bottleneck, causing performance loss or even pipeline crashes. File based operation is also less demanding on the environment and works also for reduced operating systems sometimes found on the compute nodes of supercomputers. Moreover, using the ProC in stand-alone mode requires virtually no effort in installation and configuration, which makes this mode interesting for users who just want to play around with the ProC to figure out how to work with it.

Obviously, big science projects may require both massive parallel applications on supercomputers, which may be only possible in file-based operation of the ProC, and the data structuring provided by the DMC. For later releases, a mixed operation is considered, with file-based operation at the compute nodes and a separate collection of all result and logging files into the database.

2.2.3 GUI and command line operation

Normally, all operation of the ProC is controlled by a central GUI, the session manager. From the session manager, pipeline definitions and configurations can be loaded, the editor or configurator can be started to manipulate them, and their execution can be controlled. The session manager displays all logging information, sorted by modules, and allows to pause, continue or stop a pipeline run at any time. The detailed function of the session manager is explained in sections **??** and 4.4.

If pipelines and configurations have already been constructed, it is possible to execute them without invoking the ProC GUI. For this, the pipeline coordinator, called *PiCo*, can be called from command line by a the shell script `startCoordinator.sh` contained in the `IDIS/ProC/bin` directory, with the pipeline definition and configuration being passed as command line arguments. A typical call would look like

```
IDIS/mypipelines> ProC/bin/startCoordinator.sh --pipe demopipe.pdef.xml \
 --conf demopipe.pconf.xml --db testdb1
```

In this example, the pipeline definition and configuration for a pipeline *demopipe* are stored in XML-files of a `mypipeline` directory within the IDIS installation, and the pipeline is executed via DMC on an operating database with alias *testdb1*. More details on this operation, including description of other command line options can be found in section 4.4.6.

2.3 Operation Modes of the DMC

2.3.1 How the DMC works

The DMC handles data objects which have two major constituents: *Data* and **Metadata**. *Data* are tables of numbers and units, and represent actual physical data processed in a workflow. They are organized in segments, which can be viewed as a user-defined separation between data sets of the same type. *Metadata* are attributes which can be used to identify the data. In its normal operation mode, the DMC stores data in files and metadata in a database, together with the path to the data file belonging to them. This allows querying for the metadata of data products, and retrieve the data associated to them. The DMC GUI provides a convenient tool for generating queries in the JDOQL language, and the easy retrieval of the data associated to them.

The DMC can be configured to store all data in the database. For Planck, however, this operation mode is not recommended as it tends to be slow for large amounts of data. The data handling configuration of the DMC is defined in its installation and completely transparent to the user.

As the DMC works with relational databases, but uses an object description internally, it requires a mapping from data objects to relational tables. This is provided by an implementation of

the Java Persistency API (JPA) technology. Currently, the DMC uses the OpenJPA implementation, which is freely available and distributed together with the DMC.

When the DMC is built, an object-relational mapping is generated and stored in internal files, which are used by the DMC during operations. The object-relational mapping is dependent on (a) the object model, which is expressed in the Data Definition Layer (DDL, see below), and (b) the database type. This implies that, once the DMC has been built, it can work only with one DDL and one database type. If an extension of the DDL is needed or the database type is changed, the DMC must be rebuilt.

The DMC setup described above refers to an operation mode which may be called the *local operation mode* of the DMC. Currently it is the only mode available, and we briefly summarize its restrictions, which the user should always have in mind

- Database server must have a shared file system with the operation machine
- All databases simultaneously connected to one DMC setup must be of the same type
- All databases simultaneously connected to one DMC setup must have the same DDL and schema name.

Obviously this is a severe restriction for acessing data in distributed systems of large projects, but certainly it is the safest operation mode for projects who want maximum access control of their data.

2.3.2 DMC and databases

The DMC is a Java application acting on an external database. As there are quite considerable differences between the capabilities of available database systems, the features provided by the DMC depend to some extent on the quality of the database management system used. Currently the sophisticated database management systems *Oracle* and *PostgreSQL* are supported. For an easy out-of-the-box solution, we also support the Apache Derby database, although some restrictions to the functionality apply in this case. There are drivers for Sun's MySQL delivered with the DMC, but this implementation has never been tested.

The Oracle DBMS is currently the only one which supports the full functionality of the DMC. Otherwise, the free database PostgreSQL is regularly tested with new IDIS distributions and recommended to be used. The Apache Derby database is recommended for easy installation, but the user may experience problems in pipeline executions when parallel computation is involved.

2.3.3 The Data Definition Layer

The DMC operates on data types defined in the *Data Definition Layer (DDL)*. For each data type, the DDL describes the structure of the scientific data (i.e. columns with elementary data types) and the *metadata* available to identify them. Metadata are elementary data (i.e. numbers or strings) which characterize a data product and which can be used to search for it. Therefore, metadata are stored in the database which allows their use in data queries, while the real scientific data are stored in files, which improves the I/O performance.

The DMC does not allow to query for real data, just for metadata defined to identify them. It is in the responsibility of the user to define enough metadata for each data type that it can be identified within typical operations of his/her science project.

The DDL is delivered with the DMC and consists of a number of XML files, which are loaded by a central file `ddl.xml`. The grammar of the DDL is defined in `TOODIddl.dtd`, and cannot be changed. It specifies how datatypes can be defined. Besides the attributes of metadata keywords, and numeric and string columns to store data, it is specified here that data types can be inherited, or associated to each other.

Inheritance has the same meaning as in object-oriented programming: All properties of the mother-type are given to the child, which can add additional properties. For example, Planck Level S defines a data type "polarized map" which is inherited from the datatype "map". When a module accepts an input of type "map", it can also be fed with a polarized map (but not vice versa).

Associated data objects are data objects which are, in a way, contained on a mother-object. Their constituents (data and metadata) can be accessed directly from the mother-object, but their metadata cannot be used for querying for the mother object.

The DDL is an object-oriented structure. As the DMC works with usual, relational databases, this structure is mapped on database tables by using the Java Persistency API (JPA) implementation (as already mentioned). How it is done in detail, cannot be specified, and depends on the JPA-implementation and database used. In principle, however, one can assume that every property of a data type (metadata and column names) coresponds to a column in a database table. When the DMC is built on a newly created database schema, all tables are generated accordingly. When it is built on an existing database, the JPA will try to match the existing data structure. If objects or properties are found for which no corresponding tables or columns are found, they are created (or existing tables extended). However, if a data column is found for which there is no object in the DDL, the DMC throws a "Class not found" exception at runtime.

This has one important implication: when the DMC is built on an existing database, but with a DDL incompatible to the one used to create this database, the database may become unusable for the DMC. *Incompatible* in this context means that datatypes, or properties of datatypes (like columns or metadata) have been *deleted* or *renamed*. In principle, there is no need to ever do this: if a data column is no longer used, there is no reason to remove it from the database. The same applies to data types which are no longer used. If somebody feels that a column or metadatum should have a different name, it is possible to add a new column or meta-datum with the new name, and abandon the old one without removing it from the DDL (probably, the contents of the old column would have to be copied to the new one with a small program). Additions to the DDL are no problem in general, as the new mapping will then just add columns to the database tables without affecting the existing ones.

During the evolution of a science project, the DDL should always be kept consistent. This means that data types and data type properties can be added, but should never be deleted or renamed. Inconsistent DDL changes will make existing databases inaccessible by the DMC. Regarding the DDL contained in the IDIS distributions, the MPA IDIS team takes responsibility for keeping the DDL consistent with respect to earlier releases, but of course not with respect to earlier DDL changes the user may have applied!

If the DMC has been messed up by an incompatible DDL change, the database can be made accessible again by putting all datatype properties, which have been removed from the old DDL (possibly by renaming a property) back into the *new* DDL. This means, a DDL must be created which contains *all old and all new* properties. When rebuilding the DMC with this DDL, the database will be accessible again.

2.3.4 Operation with the ProC vs. other clients

Obviously, the DMC can be accessed through its API by any code, and its usage does not require usage of the ProC. Also the DMC GUI can be started in stand-alone mode, and be used to submit queries interactively. Nevertheless, the DMC unfolds its full usage only when used with the ProC, and we shall briefly explain why this is the case.

In addition to its function as a coordinator of data processing workflows, the ProC logs all its actions in the DMC. For this purpose, it generates for every pipeline run a *Prun* object, and for every module started an *Mrun* object. The Prun-objects contain the Mrun objects, and the sequence in which they have been called in the pipeline. The Mrun objects contain all the parameters for which a module has been configured, an MD5 checksum to identify uniquely the executable code, thus the true version of the module, and references to the data objects which have been used and which have been produced. Even if the data themselves have been deleted from the DMC, this structure remains active as the metadata of the deleted objects usually remain in the DMC (unless overruled by the data administrator, see chapter 5), so it is still possible to determine which properties the data objects had that were used to generate another data product.

Even if a processing code cannot be modularized, and the whole processing chain consists only of one module which reads some data and generates other data from it, this relationship between initial and final products can be recovered later from its Mrun object. If a chain of modules is used, the whole *history* of the generation of this data product is stored in the DMC, if the ProC has been used to run this sequence. In many mid-size to large scientific projects, knowledge of the data generation history has turned out crucial in order to guarantee the correctness of a published result. The generation and secure storage of the data production history has therefore been the main driver for developing the ProC / DMC software for the Planck project.

Generation of the data production history is a function of the ProC and DMC in combination. Neither the ProC nor the DMC as stand-alone applications provide this functionality.

To summarize this paragraph: Without the ProC, the DMC acts as a data storage optimized for scientific use, combining the advantages of databases with respect to organizing the data, and of file systems with respect to I/O speed. But only with the ProC, the DMC becomes control and tracking system for scientific data, which contains also all dependencies of data products on each other and on code which has been used to generate them.

Chapter 3

IDIS Installation and Setup

This section describes how to get and install IDIS. If you just intend to use a centrally installed IDIS system (e.g. at the LFI DPC), you can skip this section and just ask the local administrator which steps to perform in order to get started.

Users who want to install the IDIS-in-a-box distribution, only need to read section ??

3.1 System requirements

3.1.1 Environment

IDIS is currently supported for several Unix or Linux distributions and on Mac OS. In order to install IDIS without problems, the following tools should be available:

- ant (version 1.6 or above required)
- gcc (version 4.2 or above including gfortran recommended)
- bash (recommended)
- java SDK (1.6 at least required, 1.7 and above recommended)

If Linux is used, a kernel version 2.6.12 or higher should be installed to run the DMC without problems.

IDIS does currently not support Microsoft Windows operating systems. Sorry, Bill!

3.1.2 Supported databases

Note that the ProC can be used without a DMC altogether, in which case all data products are stored in simple files on the file system. If this operation mode is chosen, no database installation is required, but we emphasize that the full functionality of IDIS is not available in this operation mode.

In order to run the DMC, a database must be available. Available means that a database management system (DBMS) is installed and a database has been created, on a machine *which has a shared file system with the host running the IDIS installation*. Currently, IDIS functionality is tested on Oracle and PostgreSQL databases, their usage is recommended. The Oracle DBMS is currently the only database system which supports all DMC functionality, and it is used in the official setup of Planck LFI. Recent versions of IDIS are tested for Oracle 10g.

For users who do not have access to an IDIS database connected to their file system, but still want to use a full IDIS installation, we note that the ProstgreSQL DBMS supports almost all operations of the ProC and DMC as well. PostgreSQL is an open source DBMS, and quite straightforward to install. It does not have tremendous demands in system resources and can also be installed, e.g., on a regular laptop. However, as the setup for the database requires some administrative tasks which are difficult to automate, we cannot include an "out of the box" PostgreSQL database into the IDIS distribution, and refer to the installation manual available on the PostgreSQL webpage. The recent releases of IDIS are tested for PostgreSQL 9.2.

Even simpler to install is the free database system Apache Derby. Essentially, all you need to do is to download a tar-file from the Derby webpage, unpack it, set a couple of environment variables and start the database network server via a shell script contained in the distribution. It is not even necessary to create a database, as for Apache Derby this is done automatically when building the DMC. However, we note that not all IDIS functionality is supported with this DBMS. Problems may be expected in particular with all kinds of parallel processes involving concurrent access to the database. During this tutorial, we note all restrictions which might be expected with this database, and also give tips for possible workarounds.

To support the full capabilities of IDIS, use of either Oracle or PostgreSQL databases is strongly recommended, but for just trying out the system and getting used to it, installation with an Apache Derby database might be your choice. In the IDIS-in-a-box package, a ready-configured Apache Derby database is contained.

3.2 Getting IDIS

3.2.1 Retrieve tarball

If you have installed IDIS already and need an update, please proceed to Section ??

The complete IDIS package including Planck LevelS is distributed as a tarball. There are essentially two ways to obtain this tarball:

1. Through the ESTEC release repository
2. Directly from MPAC by http/ftp or wget from
   http://planck.mpa-garching.mpg.de/idis.tbz

The ESTEC release repository contains the official IDIS releases, and is used by the LFI DPC. However, the MPAC keeps its idis.tbz tarball always consistent with the latest ESTEC release. Additionally, MPAC provides access to bug-fix releases, which are usually not verified in an independent system test (but may be urgently needed to continue work), and also to the newest development tarballs (which are *not* recommended to be used for science operations). See section **??** for details.

Save the tarball in whatever directory you want to install the IDIS software (we call it `Planck` here for instance), and unpack it using the command

```
myhomedir/Planck> tar jxpf idis.tbz
```

Note that the top directory unpacked from the tarball is named `IDIS`, so it is not necessary to specify this name in the directory used to unpack the tarball. In our example all IDIS packages are found in a directory `Planck/IDIS`. Additionally, a script named `updateIDIS.sh` is stored in the unpack directory, which can be used later to install future versions or patches of IDIS in a very simple way (see Section **??**).

3.2.2 Updating IDIS

If IDIS has been installed once, updates with future versions are almost automatic. Whenever a new tarball has been provided on the MPA web server, just type

```
Planck> ./updateIDIS.sh
```

The script downloads the newest tarball, unpacks it and executes all necessary build and configuration commands described in this section. The MPAC will keep the main IDIS tarball always consistent with the official release available in the ESTEC release repository. As the latter contains only tested releases, quick fixes cannot be provided this way. To allow the distribution of quick fixes also without an extensive system test phase, MPAC provides an additional tarball called `idis_fix.tbz`, in order to update with this tarball rather than the main one, type

```
Planck> ./updateIDIS.sh --fix
```

Additionally, a `--dev` option is available for very bold users or testers: it accesses the current development tarball at MPAC! Users who want to do reliable science with IDIS are strongly discouraged to use this option!

3.3 Configuring the Setup

3.3.1 The `.proc` directory

When operating IDIS, a directory named `.proc` in the home directory of the user is used to save all automatically generated files storing preferences and configurations of the IDIS software. The preferences stored there are continued to be used also after updates of the software. If the .proc directory is not present in the home directory of the user, it is created upon installation.

A safe way to avoid incompatible settings when some new version of IDIS is installed, i.e., to create a "virgin" setup, simply move your existing .proc directory to some other name. This is recommended in particular if you install IDIS-in-a-box after previously using a regular installation, see section ??.

The .proc directory is also a good place to store non-automatic files which contain general setup properties which you want to use in all installations. One important file of this kind is the `build.properties` file needed to build the DMC, described in the following. Note, however, that this file needs to be copied to the `IDIS/DMC` directory of your current installation before it can be used.

3.3.2 Setting up the build properties for the DMC

After unpacking the tarball for the first time, you find a file named `build.properties.template` in the directory `IDIS/DMC`. In order to configure your setup, this file has to be edited and renamed into `build.properties`. If an IDIS installation was made before (e.g. with a previous version), the `build.properties` file of the previous installation will still be there, and unless changes have been made to the database setup or other essential properties described below, it does not need to be changed.

If several databases are available, we recommend to keep all the `build.properties` files for them under special names (e.g. `build.porperties.apache`, `build.properties.postgres`) and rename them to build.properties according to the desired, current setup. If you install IDIS for the first time, you may simplify your work by copying the `build.properties` file from a colleague who has done the installation already, and just change the properties containing personal information. In the following, we explain all of them:

`JDOTYPE=JPA` the JPA/JDO implementation which should be used by Java. Currently OpenJPA and Kodo are supported.

`DBTYPE=` the database management system used. Supported values are: `Oracle`, `PostgreSQL`, `MySQL`, `ApacheDerby` - there are more databases which are supported by JDBC (see the end of bin/questions.subsh), but only for the databases mentioned above drivers are delivered with the DMC. If you intend to use another database system, you need to supply the connection URL and the drivers to bin/questions.subsh.

`JDO_RUNTIME_KEY=` the 20-digit license key needed to run the DMC. This key can freely be distributed by MPAC or any Kodo license holder. This parameter is only necessary when using Kodo as JDOTYPE.

JDO_DEVELOPER_KEY= the 20-digit license key needed to create a database mapping, which is necessary when installing the DMC or updating the DDL. To get this key, you must be a Kodo license holder. This parameter is only necessary when using Kodo as JDOTYPE.

DBNAME= The name of the database which should be used. If you use e.g. PostgreSQL it is the name you specify when doing a CREATEDB. For Apache Derby, a database of the name you insert here is created with the DMC installation.

SERVERNAME= The hostname of the server on which the database is running. This will be needed to build the JDBC-ConnectionURL.

HOST_ALIAS=any Obsolete, just contained for historic reasons. Don't change the the default.

DBUSERNAME= the user name ...

DBUSERPWD= and password used by the DMC to access the database. This username and password are introduced when installing and setting up the database; for Apache Derby, you can choose these parameters here. Database username and password are needed for the build process during which the schema and mapping information of the database is prepared, and also if the DMC is run stand-alone without ProC.

SCHEMAS= The name of the database schema the DMC uses. Naming the schema is part of the database setup; for Apache Derby, the schema name is chosen here.

DB_ACCOUNT_ALIAS= In the standard setup, without a Federation Layer, this is just an freely definable alias for the database configuration. If you are using the FL, this is the name for the database as known by the Federation Layer. If the local FL is used, this name is arbitrary, but the same name must be used when setting up the local FL, see Section **??**. This name is also used by the ProC for the database, and important if multiple databases are used.

HOSTING_DBALIAS= DB_ACCOUNT_ALIAS of the first database, if the DMC is set up to use a second one (e.g. read from one, write to another); see below. The second database needs to be configured in the same way as the first. When installing the DMC initially, this line MUST be commented out (#), which is the default.

STORAGE_METHOD='-2' Method of data storage used by the DMC, chosen by a numeric value. Possible methods are

'0' – save all data in the database

'-1' – save data to disk immediately, using Java file streams

'+1' – save data to disk when transaction is completed, using Java file streams

'-2' – save data to disk immediately, using native I/O methods

'+2' – save data to disk when transaction is completed, using native I/O methods

TIME_DIR_INT='-3' Specifies in which way the DMC organizes the directory structure for data on disk, STORAGE_METHOD] other than '0'. Values consist of a sign and a number in single quotes, using the following coding:

'0' – create a directory for every object type, with no time structure

'+*n*' – within each object-type directory, there is another level of directories sorted in time, which contain the objects of this type created in the respective period

'+*n*' – as before, but with the time-structure on the upper directory level, and the object structure on the lower.

The values for n determine the granularity of the time structure: for $n = 1$ a new directory is created every year, for $n = 2$ one per month, and $n = 3$ one per day.

`DATA_PATH=` When `STORAGE_METHOD` is different from `'0'` (meaning data will be written to disk) this value specifies the path to the directory to which the data should be written. Note that this directory path must be mounted to both the database server and the local machine you are using to run IDIS.

`SHELL=sh` The type of the shell you are using to run the ProC; we recommend to leave the default setting, `sh`, and to do all installation within a bourne shell or bash.

`OSNAME=` Your operating system type. Allowed values are `linux`, `solaris`, `macos`, `aix`, `macosx`.

`MACHNAME=` The architecture of the computer running the DMC (e.g. `i386` for 32-bit x86 style processors or `amd64` for 64-bit machines). This value is used e.g. for finding the correct Java libraries.

`JAVA_HOME=` The path to your Java installation. For Linux and most Unix systems, all Java specific libraries should be found under `$JAVA_HOME/jre/lib/MACHNAME`, and there should be either a directory `client` or `server` which contains the `libjvm.so`. So, the easiest way to find the correct setting is to search for `libjvm.so` on your system. If the operating system is `macos` or `macosx`, this tag must be set to `/System/Library/Frameworks/JavaVM.framework/Home` (the path to the libraries is different here, but this is considered in the DMC setup procedure).

`JW_TARGET=` An identifier describing the compiler to use in order to compile the Java Wrapper (e.g. `linux_gcc`, `linux_f95f`, or `macosx_gcc`). It has to be the same with which LevelS will later be compiled. The default setting (and recommended for linux users) is `linux_gcc`, which requires gcc version 4.x or above containing gfortran. See Section **??** for more information.

`JAVACF='3'` This parameter defines for which languages the Java Wrapper will be compiled.

`'1'` – for Java only

`'2'` – for Java and C

`'3'` – for Java, C and Fortran

The following keywords in the build.properties file are used when no Federation Layer is used. Otherwise the identification of the user is done in a different way, as explained in section **??**.

`FRSTNAME=` your first name, ...

`LASTNAME=` your last name, ...

`AUTHORID=` your username (arbitrary, if no FL is used) ...

`AUTHOR_PW=` and password.

The order of the keywords in your `build.properties` file is arbitrary.

3.3.3 Creating the user information

If you are using the FL and if you install the DMC for the first time, you will need to create a file containing your personal identification. This is used to identify you as a creator of data objects. Otherwise, just skip this section.

In IDIS releases before version 2.7.4, this information was stored in the build.properties file, which is part of the IDIS installation (see above). Obviously, this creates problems if more than one person is using this installation - for the DMC, all of them have the same identification. In the new releases, all information concerning the user are stored in a file `.proc/gui/user.id`, thus located in the users home directory tree, and therefore different for every user.

To create this file, go to the `IDIS/DMC` directory call the script

```
IDIS/DMC> bin/createUser
```

The program prompts you to input all the information listed above as "obsolete fields" in the `build.properties` file. After you have done this step, you can continue with installing the DMC.

The information entered here is only the default user information, and overruled if different user settings are used by the Federation Layer, see section ?? and ??

3.4 Building the DMC

3.4.1 Compiling the DMC

The DMC will not compile without the user information. If you forget it, the installation process stops and asks you to call `createUser` first.

After the build.properties file is set up and the user information is created, the DMC can be compiled by

```
IDIS/DMC> ant jar JW
```

During the 'jar' part the DMC is build and the database is set up. The target 'JW' runs several makefiles which compile the JavaWrapper.

Even when `ant` reports 'Build successful', errors may still have occurred. Watch out for some Java Exceptions - if these appear during the jar-part something went wrong. Also if the 'make' process reports errors during compilation of the Java wrapper this indicates some problems (mostly due to java libraries being searched in the wrong places).

You can test if the setup is correct by using 'ant gui' and trying to query the database. For advice on the most common build problems see the Troubleshooting section below.

3.4.2 Troubleshooting

Errors during `ant`

```
kodo.util.DataStoreException: Connection refused.
```

No database is running on the specified machine. Check whether you mistyped the servername and that the database type is correct. This message might occur as well if your database server is not running on the standard ports, or has not been started at all (a common mistake in laptop installations).

```
kodo.util.DataStoreException: A connection error has occurred:
org.postgresql.util.PSQLException: FATAL: no pg_hba.conf entry for host
'w.x.y.z', user 'user', database 'database', SSL off
```

PostgreSQL-specific - a database is running on the specified server but is not accepting connections from the machine you are trying to build the DMC on. Modify the pg_hba.conf to accept connections.

Errors during `make`

```
error: jni.h: No such file or directory
```

most likely you just set the `JAVA_HOME` wrong (or the directory has a modified content) so that the JNI Header file was not found. The file `jni.h` is expected under `$JAVA_HOME/include`.

```
cannot find -ljvm
```

either the MACHNAME is set wrong or you have a broken JAVA installation. `make` is trying to find `libjvm.so` which should be located in one of the following directories:

```
$JAVA_HOME/jre/lib/$MACHNAME/client
$JAVA_HOME/jre/lib/$MACHNAME/server
$JAVA_HOME/jre/lib/$MACHNAME/native_threads
```

3.4.3 Adding databases

It is possible to connect to different databases within one DMC setup; for example, to read a data product from one database, process it and save the result to another. For this, a second database can be added to the current setup by configuring the HOSTING_DBALIAS property in the build.properties file. As an example, if the first database has been setup as:

```
DB_ACCOUNT_ALIAS=TestDB
#HOSTING_DBALIAS=?
```

the corresponding lines in the build.properties file of the second one second one should be:

```
DB_ACCOUNT_ALIAS=DemoDB
HOSTING_DBALIAS=TestDB
```

This setup adds a database called "DemoDB" to the current setup using "TestDB" by executing the command

```
IDIS/DMC> ant jar-add
```

It is possible to connect an arbitrary number of databases this way. In all cases, the property `HOSTING_ALIAS` is set to the alias of the database used for the first setup.

All databases accessed in one DMC setup must be of the same type (either all Oracle, or all PostgreSQL, or all Apache Derby) and must use the same DDL and schema name! This restriction is planned to be removed in future versions.

3.4.4 Switching between databases

If the DMC has been setup for one database (say, a PostgreSQL database called `PGTEST`), and is then compiled for a, say, Apache Derby database called `DerbyDB1`, the connection to the PGTEST database can be reinstated without recompiling the DMC - provided the `build.properties` file of the original setup has been saved under some special name, e.g., `build.properties.PGTEST`. Just replace the current `build.properties` by the original one — but be sure to copy it first to, e.g., `build.properties.derbyDB1` for later use – and then execute the command

```
IDIS/DMC> ant setupInit
```

If any additional databases had been added before to the DMC (say, a database called derbyDB2 by executing `ant jar-add`, they are lost after this step. This is quite meaningful, as when you change to another main database you may change the database type (as in our example), and all previous connections of databases will not work any more. Assuming that you had previously (i.e., before switching to Derby) added another PostgreSQL database (and took care to save the build.properties for this in, e.g., build.properties.PG2), you can add it again by replacing build.properties with build.properties.PG2 and execute.

```
IDIS/DMC> ant setup
```

The conclusion of this is that it is not necessary to build the DMC on each database you have on your system more than once – unless you want to change the DDL or fundamental settings on data storage. After this, the switch between databases is a simple change of setup, requiring about a couple of seconds to be done.

Be sure that `ant setupInit` is used whenever the database type is changed, and that a build.properties file not containing the `HOSTING_DBALIAS` keyword is used for this step.

3.5 Compiling Level S

The Planck Level S package is a collection of modules to simulate the Planck mission, in order to test data analysis code. The complete Level S is included in the IDIS tarball distributed within

the Planck mission. For users outside Planck, a subset of Level S (not containing proprietary code or modules containing mission-sensitive information) is still included for demonstration purposes. Level S uses a standardized interface to integrate modules into the ProC/DMC software, which is widely used within Planck LFI and explained in section **??**. Data between Level S modules can be communicated either via FITS files, or via the DMC; which mode is chosen is decided at compile time through environment variables.

As it is decided at compile time, the user cannot switch in operation which mode of the ProC (file based or DMC based) is used. The ProC has to be used in the mode compatible with the Level S compilation option. The newest version of Level S supports to specify the data input and output mode of each module specifically during runtime, provided the module has to be compiled for the DMC. The ProC does not yet support to specify this when building a pipeline, but this is planned for the IDIS 2.9 release

To create the setup needed to compile LevelS for use with the DMC, go to the `IDIS/DMC` directory and type

```
IDIS/DMC> source sourcefile
```

If Level S should be compiled for FITS used, simply don't perform this step. If you are unsure about your setup status, you can check the content of the DMC environment variable:

```
IDIS/DMC> echo $DMC
TOODI
```

If the value of the variable is `TOODI`, Level S will be compiled for DMC use. If the variable is undefined or empty, Level S will be compiled to exchange FITS files. Another valid value of the variable is `HFIDMC`, which compiles Level S for used of the HFI DMC (which is used by the Planck HFI collaboration, and not controlled by IDIS). For all other settings of this variable, compilation will fail.

The DMC `sourcefile` is designed for bash-style environments - you can change this by setting the SHELL argument within the build.properties file (but this is *not* recommended, better use `bash` or `sh`).

Additionally you need to specify the environment variable `LEVELS_TARGET`, which defines for which system architecture Level S will be built, e.g. by

```
IDIS/DMC> export LEVELS_TARGET=linux_gcc
TOODI
```

This is best done in your `.bashrc` file. The variable specifies the make-target for Level S, essentially a combination of the operating system and the compiler(s) used for various languages. The most common setting for this variable is `linux_gcc` (which supports also gcc installations on most other operating systems!), other important options are `macosx_gcc` and `linux_f95f`. Users experienced in reading makefiles can seek for more options in the directory `LevelS/config`, where you find the Level S makefiles, following the naming convention `config.<targetname>`.

For use with the DMC, Level S needs to be compiled with the same compiler and for the same architecture as the JavaWrapper of the DMC. So the settings of the tag `JW_TARGET` in the `build.properties` file and the environment variable `LEVELS_TARGET` should be identical. Please note that if you use some odd compiler option which you found in the makefiles contained in LevelS/config, make sure that the same configuration exists for the DMC. The appropriate makefiles for the Java Wrapper follow the same naming convention, and are found in the directory `IDIS/DMC/src/main/c/JW/config`.

After all this is prepared, change to the `IDIS/LevelS` directory and simply run

```
IDIS/LevelS> make
```

When the LevelS modules are build you need to make sure that for later use with the ProC they are available in your local `PATH`:

```
export PATH=$PATH:~/Planck/IDIS/LevelS/$LEVELS_TARGET/bin
```

Note that `~/Planck` is our assumed directory in which the tarball has been unpacked - if you have chosen another name and path for this directory, replace this here.

To have this variable set every time you log in you can include this in your .bashrc or equivalent. Be sure that no entry in the old path points to outdated executables of LevelS, because they would be executed first!

To be used by the ProC, every module needs an XML-file containing is module description (see section **??**). For historical reasons, IDIS distinguishes between *user-XML* and *machine-XML* descriptions. The user-XML descriptions for all Level S modules are delivered with the package, and contained in the directory `LevelS/xml`. The `machine-XML` files *must* be kept in a different directory, and for avoidance of confusion we suggest to put them in subdirectories of the `IDIS/ProC` directory named after the module package they belong to. So for Level S we suggest to create a directory `ProC/LevelS`, i.e.

```
IDIS/ProC> mkdir LevelS
```

Then convert the user-XMLs by typing

```
IDIS/ProC> bin/testXML.sh ../LevelS/XML/*.xml LevelS
```

After this, the machine-XMLs for Level S are found in the directory `ProC/LevelS`. Within the initial steps of using IDIS, they need to be imported to the ProC editor, as explained in section **??**.

3.6 Using the IDIS-in-a-box tarball

The IDIS installation process described in this section gives the user the full choice to decide on the operation modes he wants to use, and also on required external software like database management systems. This, however, keeps the installation process still quite complex, in particular, as a database system has to be set up first. Another severe restriction is the need of a Kodo license to do a full installation, as explained in section **??**.

For people interested in evaluating the capacities of the system on their own laptop, or to test the integration of their science modules independently without requiring use of an official IDIS installation, the MPAC team provides a special tarball called *IDIS-in-a-box*, designed for scientists within the Planck collaboration, which contains

- A complete IDIS system with preconfigured setup, requiring only a fraction of the effort to be installed compared to the full IDIS system
- An Apache Derby database, which requires no installation and is automatically started when starting the ProC
- A ready-mapped schema in the database for the current Planck DDL (Level S + Level 2 + Level 3), and a Kodo runtime-key to use it.

In order to keep things that simple (and legal), however, some compromises had to be made

- As already mentioned, the simple Apache derby DBMS places restrictions on parallel execution of modules, so the users of IDIS-in-a-box should pay attention to the workarounds described in section 4.4.2.
- The DDL of the database is fixed, and no new datatypes can be added, as this would require a Kodo development key. Of course, in case of DDL changes MPAC can provide new databases with new mappings, but this requires migrating the data the user has in his old database.

In spite of the existing restriction, we encourage all Planck scientists to install IDIS-in-a-box on their laptops. With some modifications, the IDIS-in-a-box tarball can also be made available to people outside Planck. The capabilities of IDIS-in-a-box are large enough for evaluation and testing purposes. If you are interested to use the system in a more flexible way later, please contact the MPAC team to discuss possible solutions on a case to case base.

To install the IDIS-in-a-box tarball, please read the "System requirements" part of section **??**, and check whether all required software is available on your system (a common problem is that no java SDK, but only a java runtime environment is installed, which will not allow you to install IDIS; it is easy to get and install a java SDK on your system from `http://java.sun.com`).

If you had an IDIS installation on your system before and a `.proc` directory has been created, you should copy it to some other name or place, to allow IDIS-in-a-box to create its own `.proc` with the proper settings in it

After having clarified the prerequisites, execute the following steps (we again assume for instance that you do all installations in a directory called `Planck`)

```
Planck> wget http://planck.mpa-garching.mpg.de/idis-in-a-box.tbz
Planck> tar xpvf idis-in-a-box.tbz
```

In contrast to the normal IDIS distribution, a ready-configured `build.properties` file is included in IDIS-in-a-box, and found as usual in the `IDIS/DMC` directory. The tags in this file are

reordered such that only those you really need to change are shown on top (don't make any changes below the line!). The tags above the line are preconfigured for a Mac OSX setup – if you have a Macbook, don't do anything. Otherwise, you need to edit this file and change the tags above the line to the right settings for your system; read the comments in the file, the `README` file included in the tarball, and the appropriate parts of section **??** for help. The only real problem might be to find the right settings for the `JAVA_HOME` tag, but your local Java guru may help you with this.

After having configured the `build.properties`, continue with

```
Planck> ./finishInstallation.sh
```

Watch out for possible error messages or exceptions in this step! If everything worked, you can continue with starting the ProC, as described below.

Watch out in all steps for special comments on IDIS-in-a-box

3.7 Starting IDIS

3.7.1 Login to the Federation Layer

The usual way to start IDIS is to launch the Process Coordinator by executing

```
myhomedir/Planck> IDIS/ProC/bin/startProC.sh
```

from the command line. The startup screen shown in Figure 3.1 appears. Here you have the choice to login to the official Federation Layer with your ESTEC user name and password, or to use the "Local" mode for which only the user name (the same es for ESTEC-FL) is necessary. In the latter case, access to remote infrastructure controlled by the Federation Layer is not possible.

For IDIS versions 2.7.3 or earlier, only the login according to the build procedure of the DMC will work properly. After version 2.7.4, the decision between using official or local FL is indeed made with login, not during build.

To save one mouse-click, the login sequence can be finished by hitting the return key. If this is done right after entering the user name, the local mode is chosen, or return is hit after entering a password, the ProC connects to the ESTEC FL.

For IDIS-in-a-box, use the local mode (i.e., no password) with user id "id".

3.7.2 Setting the Preferences

The ProC keeps its preferences, i.e., the general settings for its operation, in a file `ProC.pref` kept in the `.proc` directory. Although possible for people who know what they are doing, we recommend

Figure 3.1: Starting up IDIS from the ProC.

not to edit this file, but change the preferences only through the dialog provided by the ProC GUI by selecting **Session** → **Preference Presettings...**. The full configuration options offered in this dialog are explained throughout this tutorial in the respective sections. Here we just describe the essential steps to do in the beginning.

Enabling or disabling the DMC

One of the most relevant things to do before doing anything with the ProC is to enable the DMC – provided the installation procedure has been performed in this spirit, and not to work in file based mode. After first installation, the DMC is not enabled. To enable the DMC, chose **Session** → **Preference Presettings...** → **DMC**, and check the box **Enable DMC**.

The easiest check whether the DMC is enabled or not is the color of the database symbol in the session manager icon panel: if it is grayed out, the DMC is not enabled and the ProC is working in file based mode. If it shines orange, the DMC is enabled and the database successfully connected. Be aware that connection to the database may take a while after starting the ProC.

Then select your proper FL queue and database alias in the drop-down menus **Select your queue** and **Select your database**, respectively. The contents of the drop-down menus are provided by the Estec or local Federation Layer, depending on how the ProC was started.

Note that all databases known by the Federation Layer are displayed, and that the ProC does not check for which of them are accessible by *your* DMC installation. Only databases which have been added to the DMC in the build procedure can be used!

Selecting the scheduler type and queue

By default, the ProC is configured to use the GAT adaptor for local, i.e., to submit all jobs to the local operating system. Users on computer clusters may not be permitted to use this mode,

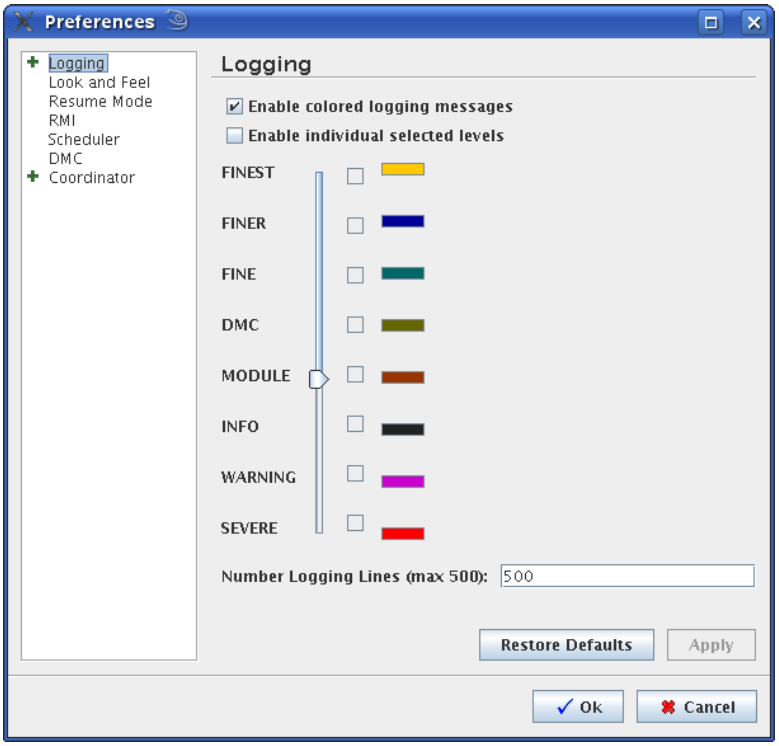


Figure 3.2: The ProC Preference-presetting dialog. In the startup panel, you can enter the settings for logging level, i.e., the level of detail on which the ProC and modules report their actions. For the general user, we recommend the level **MODULE**. For more details on logging and preference settings in general, see section 4.4.1. Specific preferences are discussed throughout section 4.4 in the respective subsections. For the important starting steps described in this section, you need the **DMC** tab, and maybe the **Scheduler** and **Coordinator** tabs (see text).

as it would mean that all pipeline jobs are started on the login node of the cluster. To connect to the batch system, usually PBS, select the **PBS_SCRIPT** setting in the **Scheduler** tab of the preference dialog. After this, you need to specify the directory through which the PBS system communicate the return status of the modules. We recommend to create this directory in your `.proc` directory, e.g.,

```
myhomedir/.proc> mkdir pbs
```

and then enter the full path of this directory into the field **Shared directory to store the return values ...** found in the **Coordinator** tab (you need to enable this field first, see figure **??**). Note that, depending on the setup of your system, several other steps might be needed to be done before the scheduler can really be used (see section 4.4.3)

If you are not using an explicit scheduler system, we recommend to use the scheduler setting LOCAL, which means that the GAT adaptor, inherent to the ProC, is used to distribute jobs on your system. The scheduler setting NONE is not recommended.

Specify deletion of temporary objects

Also in the **Coordinator** tab, you can specify in which cases data of temporary objects in the DMC are deleted after the pipeline run. These objects must be kept if you intend to use the resume mode of the ProC, see section 4.4.5. On the other hand, your database will fill up quickly when you don't delete them, so you would have to clean your database manually yourself in regular intervals (through the DMC GUI, see section 5.2.6). For the beginning, we recommend to check

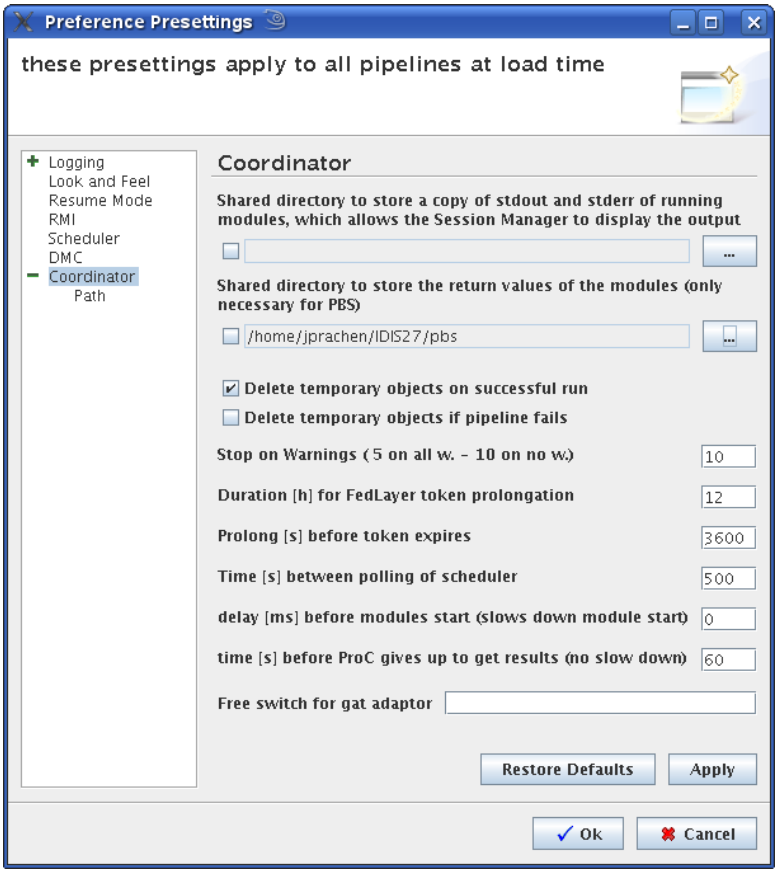


Figure 3.3: The Coordinator tab in the ProC preference dialog. Here you specify how return values of modules run by external scheduler systems are communicated to the ProC, and in which cases data of temporary DMC objects exchanged between modules in a pipeline are deleted by the ProC after the pipeline run. The other settings shown should not be changed, unless there is a specific need for it, as discussed throughout section 4.4.

Delete temporary objects on successful run, but uncheck **Delete temporary objects if pipeline fails** (see figure **??**). With this setting, you will be able to use the resume mode to recover crashed pipelines quickly.

3.7.3 Adding modules to the ProC

Before building the first pipeline, you have to import the module definitions you have created during the setup to the ProC Editor. For this, start the editor by clicking the **Definition** → **Create** button (or select the **Session** → **New Pipeline** menu item). On the left panel of the editor, double-click on **Modules**, and then select and right-click the **Modules** → **private** folder, and select **Add ModuleDef**. In the following file-chooser you select the directory where you saved the converted XML-descriptions (in our example: `IDIS/ProC/LevelS`).

For IDIS-in-a-box, the module descriptions are found in `IDIS/modules`

The directory you just added may not immediately appear in the directory tree on the left editor panel. Just close the editor after this, and start again.

In the same way, you can add more directories with module or subpipeline descriptions (see section **??**). Be aware, however, that module names in the ProC *must* be unique. So, if a directory you add contains a module-XML already contained in another directory, the old XML will be overloaded.

If some module descriptions are missing in a directory you just imported, check whether a module or subpipeline of the same name exists already in one of the other directories. If so, resolve the name conflict be renaming XML and binary of one of the conflicting modules (or renaming a subpipeline XML), then remove all directories from the editor panel and include them again!

Chapter 4

The Process Coordinator (ProC)

The *Process Coordinator (ProC)* is a workflow engine designed to construct, configure and run data processing workflows (in the Planck context also called *pipelines*). Basic entities of such workflows are *modules*, pieces of code which perform distinct, well defined operations on complex data passed between them. Figure **??** shows an example pipeline, which we use here to explain the elements handled by the ProC.

4.1 Introduction

4.1.1 ProC components

The ProC consists of three components: (a) the Session Manager (SeMa), (b) Pipeline Editor and Configurator, and (c) the Pipeline Coordinator (PiCo). Their basic functionality and usage is briefly explained as follows. For a full description, we refer to the respective user manuals.

The Session Manager

The Session Manager is the component which appears immediately after starting the ProC. It is the central control unit which allows to load and run pipelines. It allows to start, stop and pause pipeline runs, it displays the logging written by the modules and the ProC itself, and it controls the run environment (see Section 4.4 for more details.

From the Session Manager, the Pipeline Editor and the Pipeline Configurator can be started, to construct new pipelines or configurations, or to manipulate loaded pipelines or configurations, respectively

The Pipeline Editor and Configurator

The Pipeline Editor is used to build pipelines from a given set of modules. It shows in its left panel a directory list with ProC modules, which can be selected and inserted into the edit field (right panel). Then, connections can be drawn between them, representing the data flows. The Pipeline Editor automatically detects possible connections between output and input data types of the respective modules. In case of ambiguity, the user is asked which connection should be made. Any state of construction — whether complete or incomplete — can be saved into a pipeline definition file (or DMC object).

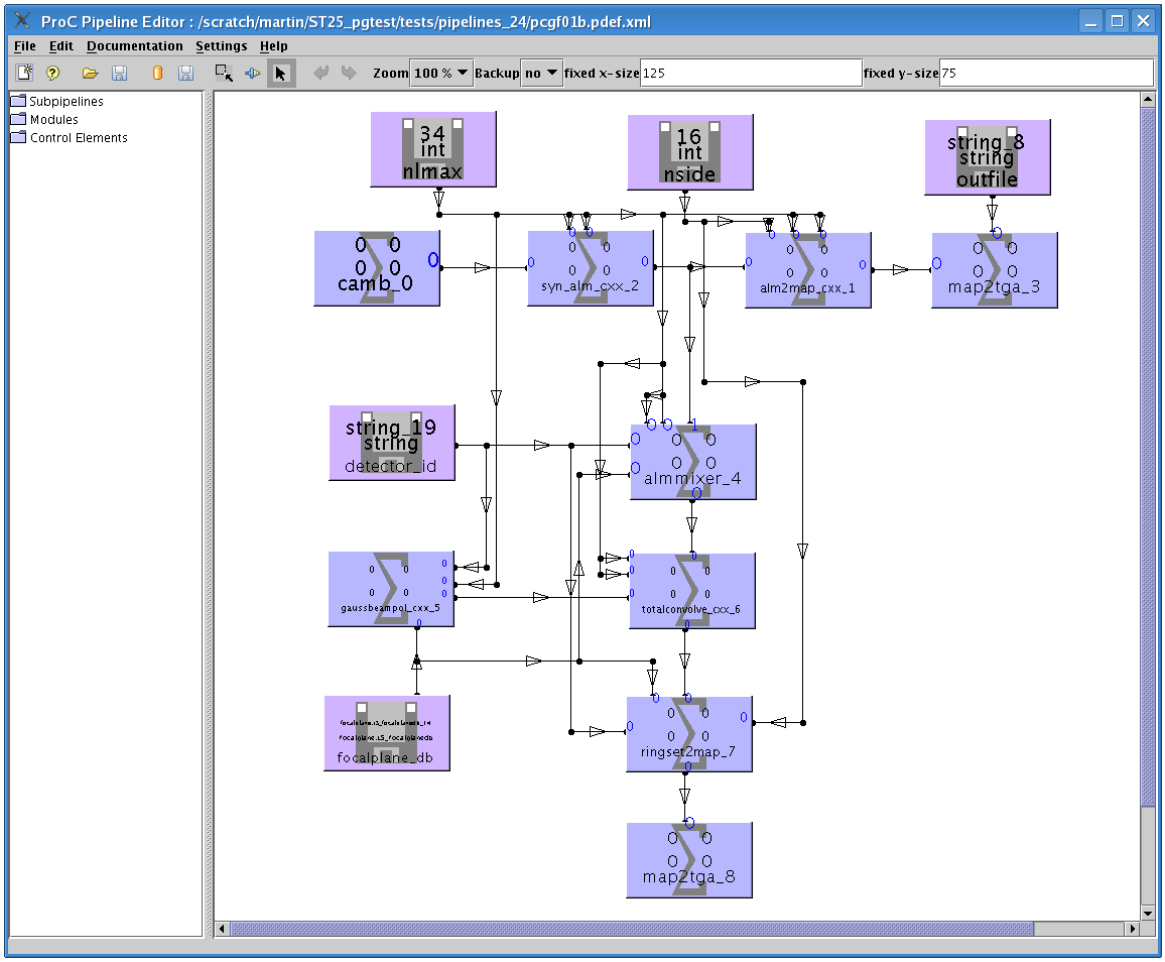


Figure 4.1: An example pipeline, as seen in the ProC Pipeline Editor

The three different modes of the editor, insert, connect and manipulate, is somewhat unusual compared to the largely standardized behavior of modern graphical user interfaces, and requires some time to get used to.

The Pipeline Configurator looks very similar to the pipeline editor, but does not allow the user to manipulate the structure of a pipeline. Instead, it allows the user to set parameters in the modules to specific values, and save them into a pipline configuration file (or DMC object) For more explanation, see also Sections **??** and **??**.

For details see the HTML documentation for the Pipeline Editor and Configurator

The Pipeline Coordinator

The Pipeline Coordinator (PiCo) is the inner engine of the ProC, and responsible for controlling the pipeline run internally. It examines dependencies and the potential for parallel execution; it also writes logs and checksums for all executed modules. If using the graphical ProC interface (i.e. the Session Manager) the user has practically no direct interaction with the Pipeline coordinator. However, the pipeline coordinator can also be started from the command line, by typing, e.g.,

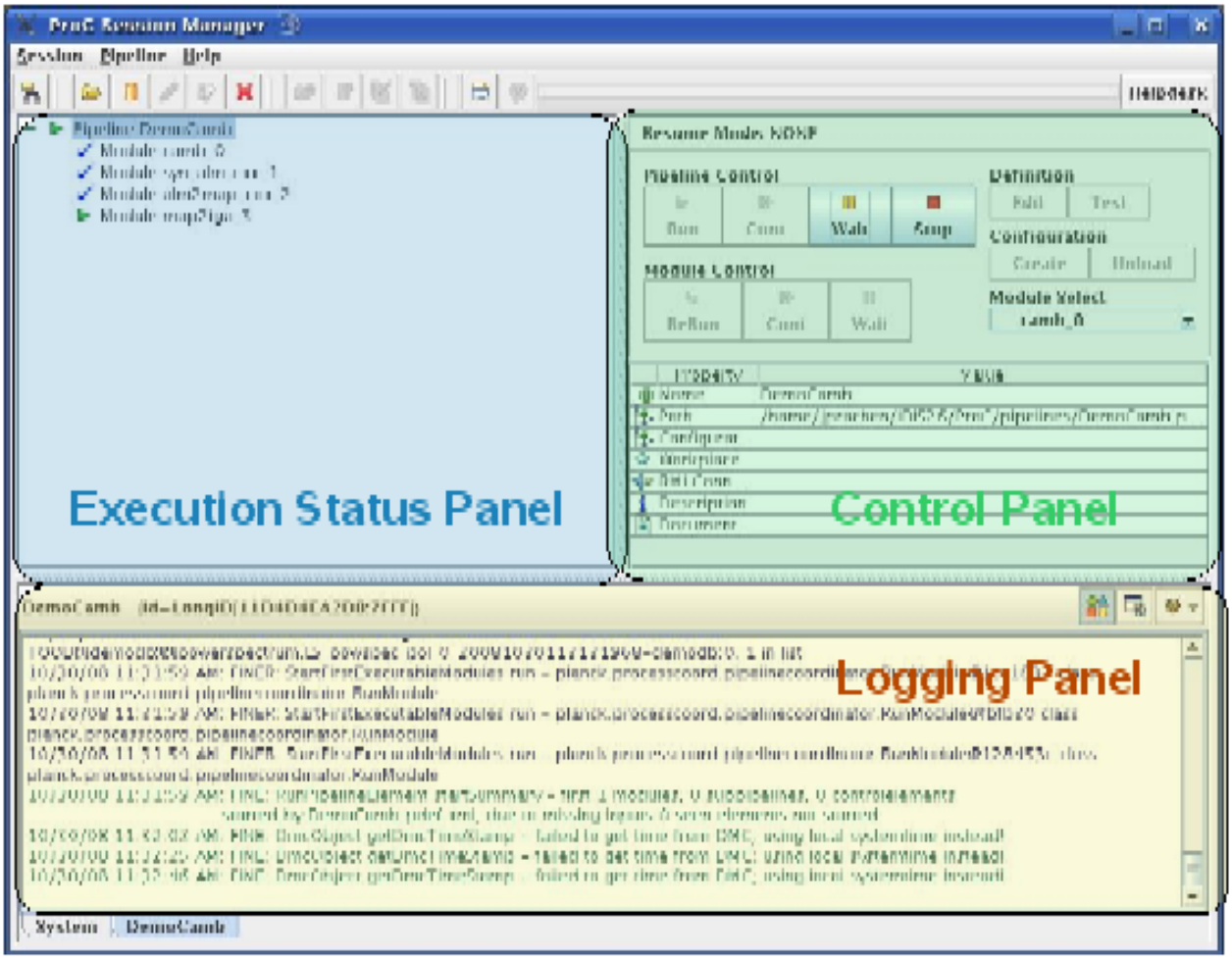


Figure 4.2: Graphical User Interface of the Session Manager. The three main parts are shown in different colors: The upper left panel shows the pipeline execution status, the upper right panel contains control buttons and displays, and the lower panel shows pipeline and module logging.

```
> StartCoordinator.sh --pipe demo.pdef.xml --conf demo.pconf.xml
```

which allows to run pipelines from scripts or in batch mode.

4.1.2 Pipeline Elements

In the following, we summarize the elements from which pipelines are build. These pipeline constituents fall into two classes: user defined elements, like modules and subpipelines, and native ProC elements, like control, parameter or data elements. In the left panel of the ProC pipeline editor, all elements available in an IDIS implementation are shown in a directory tree, as depicted in Figure **??**.

To add directories containing self-written modules or subpipelines to this panel, select **File** → **Add ModuleDef./Pipeline Dir**, and a standard directory selection dialog appears. To remove directories, first left and then right click the directory to delete, and choose **Rem Dir**.

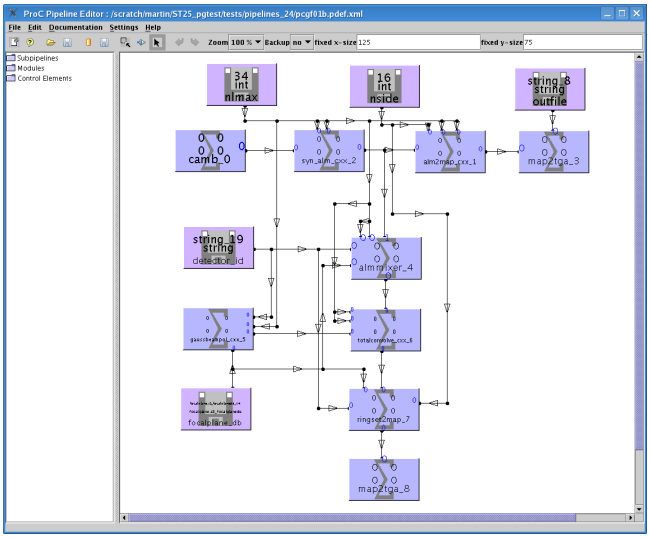


Figure 4.3: Basic function icons of the editor, from left to right: New pipeline, help, load from file, save to file, load from DMC, save to DMC, insert element, connect elements, manipulate elements.

More detailed descriptions are given in the following sections and in the pipeline editor manual.

Modules

Modules are the fundamental processing elements of workflows. They are executables of small programs written in some programming or script language - the ProC actively supports all direct executables, Java .jar and .class files, and IDL (see Chapter **??** for details). Different modules in a workflow can be written in different languages.

The ProC is a Workflow engine, not a graphical programming language! The capabilities of the ProC are best used when workflows are modularized such that defined, flexibly usable, but not to small packages of the processing flow are encapsulated in each module.

For each module, the ProC requires a module description in XML, following a fixed module description grammar. Details how to write and describe modules are explained in Section **??**.

Data Objects

In the context of the ProC, data are generally understood as complex data types. The existing data types of a module package, like Level S, are described in a Data Definition Layer (DDL). Each module has output and input data objects, which are connected by the ProC. Such connections are represented by arrows in the editor. The ProC automatically checks the proper match of the data types, invalid connections are prohibited.

Once a data object has been "drawn" out of a module, it connot be accessed a second time. If one data output object from a module has to be inserted in two other modules, the connection has to be "split", e.g., a new arrow has to be drawn out of the existing one (See Figure ?? and examples below

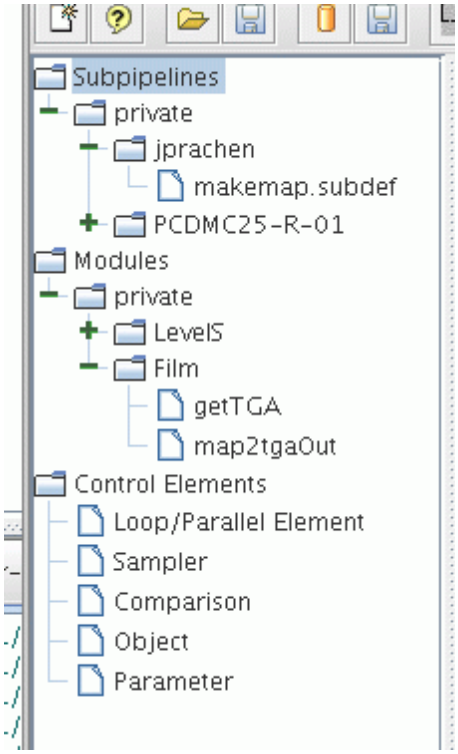


Figure 4.4: Directory tree showing available modules. At the top, available subpipelines constructed by the ProC, are shown. The next tree refers to available modules: The directory "Level S", not unfold here, contains all Level S modules which come with the IDIS distribution, the directory "Film" contains here two modules which are specifically written for some demo pipeline. Directories for "Level 2" or "Level 3" modules may be typically added here for a Planck application. At the bottom, the control elements provided by the ProC itself can be selected.

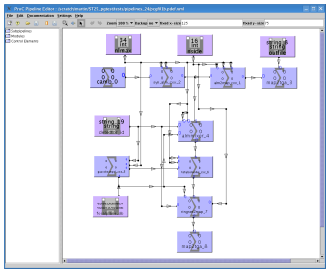


Figure 4.5: The Level S module "camb", which calculates a CMB power spectrum from a set of cosmological parameters

Lists

To be written

Parameter

In contrast to data, parameters are generally simple data types, like integer, real or string. In the simplest case, parameters are set for all modules in a configuration. However the ProC allows the use of parameter elements, which specify a parameter and supply them to different modules. The parameter transfers are depicted by the same arrows as data objects on the editor. The use of data objects allows the pipeline constructor to make sure that the same parameters are used in different modules, if this is required by the nature of the problem. Figure **??** shows an example, in which

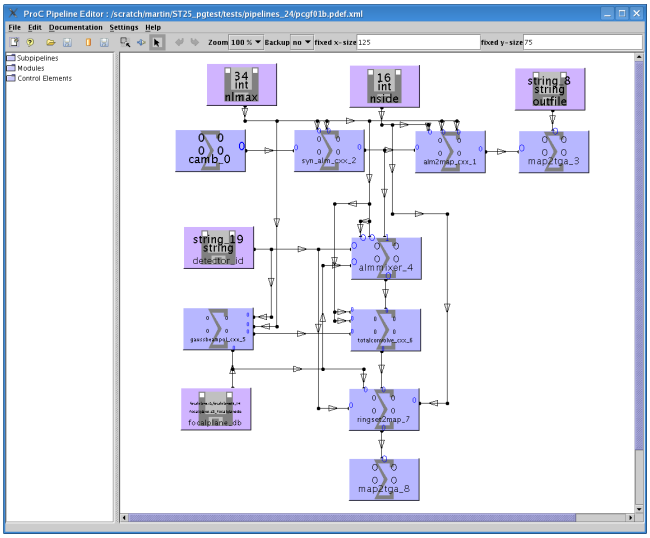


Figure 4.6: A data type `map` is passed between two modules

the same parameters are supplied to a large number of modules.

Parameter elements can also be used to organize the parameter handling in subpipelines, as explained in Section **??**.

From IDIS2.7 onward, parameters of types integer, real and string can also be passed between modules. If a module provides an appropriate output, i.e. a real number, the ProC will allow the connection of this to parameters of type "real" of any other module, or to control elements like the comparison element (see Section **??**)

Input/Output

Similarly to parameter elements, the ProC uses Data Object elements to read or write complex data defined in the DDL either from/to (fits) files, or from/to the DMC. The data type of a Data Object element is determined from the connection to a module, where some input or output must be chosen. Then, the Data Object element is connected either to a file or a DMC object; if the DMC is used, objects of the correct type are preselected, but for files there is no type check. The detailed procedure of using Data is described in Section **??**

Input and output of data is not an action controlled by the ProC, but by the module. Input and output objects just specify file names or DMC object names for the module. Therefore, although the ProC would principally support mixed I/O, i.e. partially to files and partially to DMC, this does not work for module packages which need their I/O mode specified at compile time, like Level S.

As parameter elements, Data Object elements can also be used to organize data flows, which is particularly useful in subpipelines (see Section **??**.

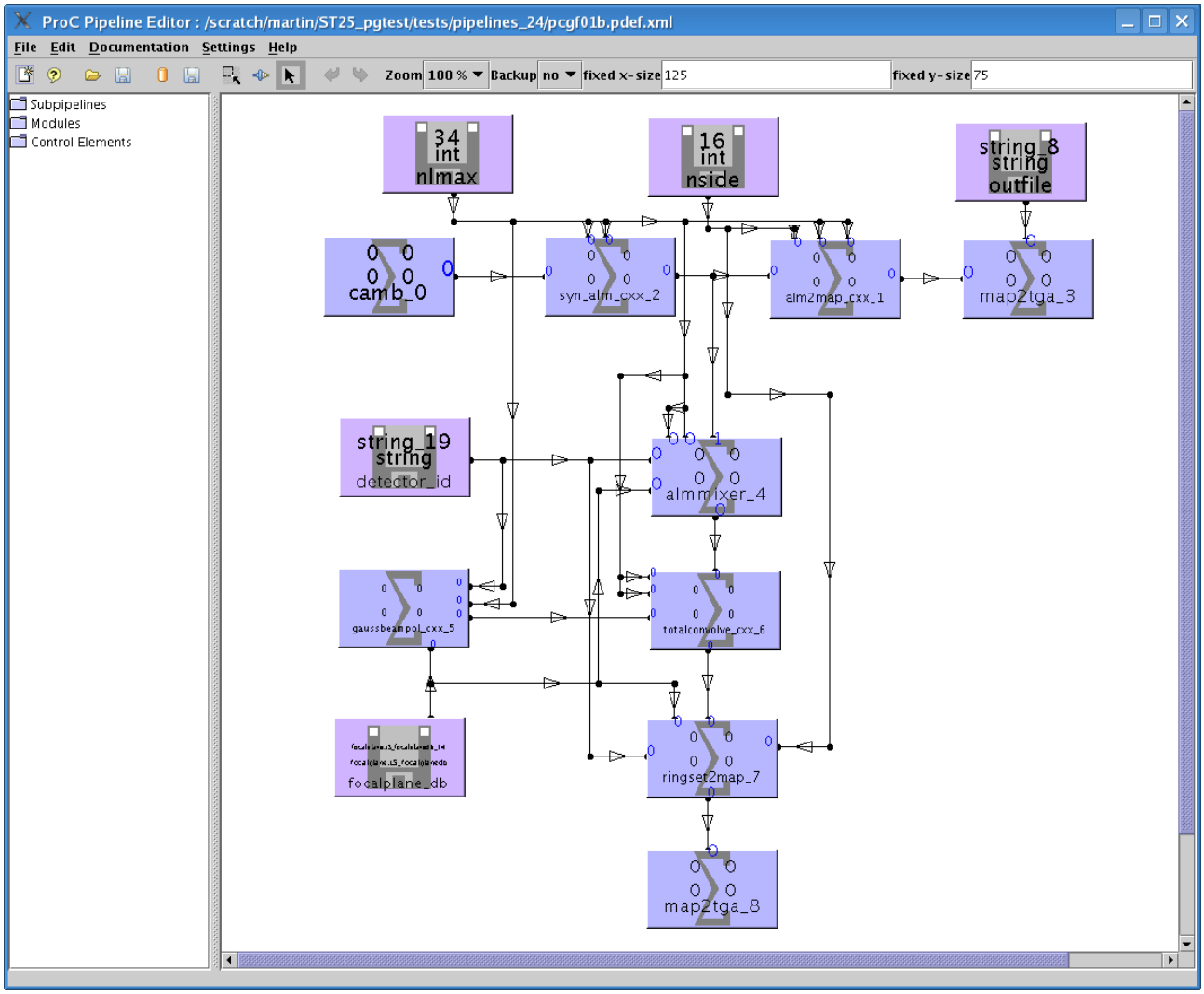


Figure 4.7: Parameters supplied by parameter elements. Note the usage of split connections.

4.1.3 Pipeline definitions and configurations

Pipeline definitions

A pipeline definition is the description of the structure of a pipeline. It is constructed in the Pipeline Editor by connecting available modules to a workflow (see Section **??**), and saved as either XML files or to the DMC. The pipeline definition does not contain any executable code. This means that all module executables must be installed and in the execution path of all users able to run the respective pipeline. On the other hand, a standard setup within a user group (say, Planck LFI) presupposed, this allows a very simple communication of pipeline definitions, as they are stored in a compact, and still human-readable format.

The ProC Editor offers the possibility to save pipeline definitions in the DMC. This function was found to be not functional in all cases, and is no longer officially supported. Therefore, pipeline definitions should *always* be stored in files.

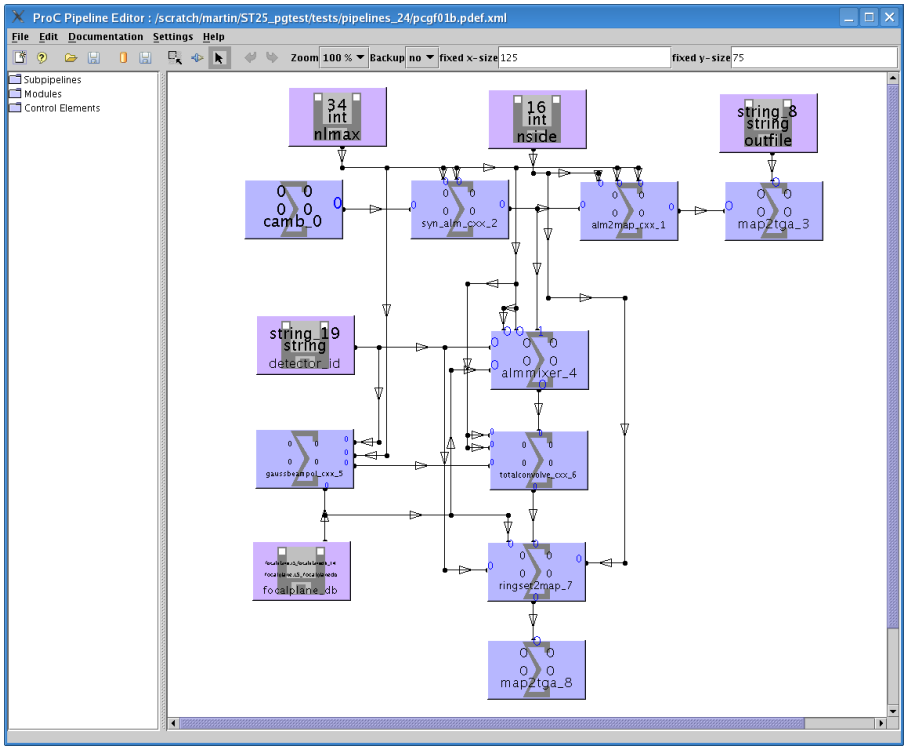


Figure 4.8: Input element reading data out of a file, and piping them into three modules at once, using split connections (the third module is not in the view)

It is also possible to fix the parameters of modules in a pipeline definition, which prevents changing them in a configuration. For this, right-click on a module in the Editor and choose **Parameter**. In the upcoming dialog, all parameters of the module are shown and can be edited. Beside the parameter value, a radio-button "fixed" can be checked.

Parameters don't need to be fixed in the definition: if they are set, but not fixed, the values given in the definition will be used a default in the configuration, unless they are changed there. However, it is recommended to leave only a parameters unfixed which should be configurable according to the problem at hand, and to fix all others in the definition. This is discussed in more detail in section **??**.

Besides the parameters of modules, also parameters of control elements (discussed in the sections below) can be set in the pipeline definition, and then used as defaults for the configuration.

Only in the pipeline definition it is possible to set the properties of data connections. The main properties to be mentioned here are the definition of *Breakpoints* and `Persistence` of a data connection. In order to change the properties of a connection, right-click on it and choose the appropriate tag from the menu.

Pipeline configurations

In contrast to definitions, pipeline configurations do not contain information on the pipeline structure. They are sets of all configurable parameters of all modules or control elements used in a pipeline. The idea behind the distinction between pipeline definitions and configurations is that

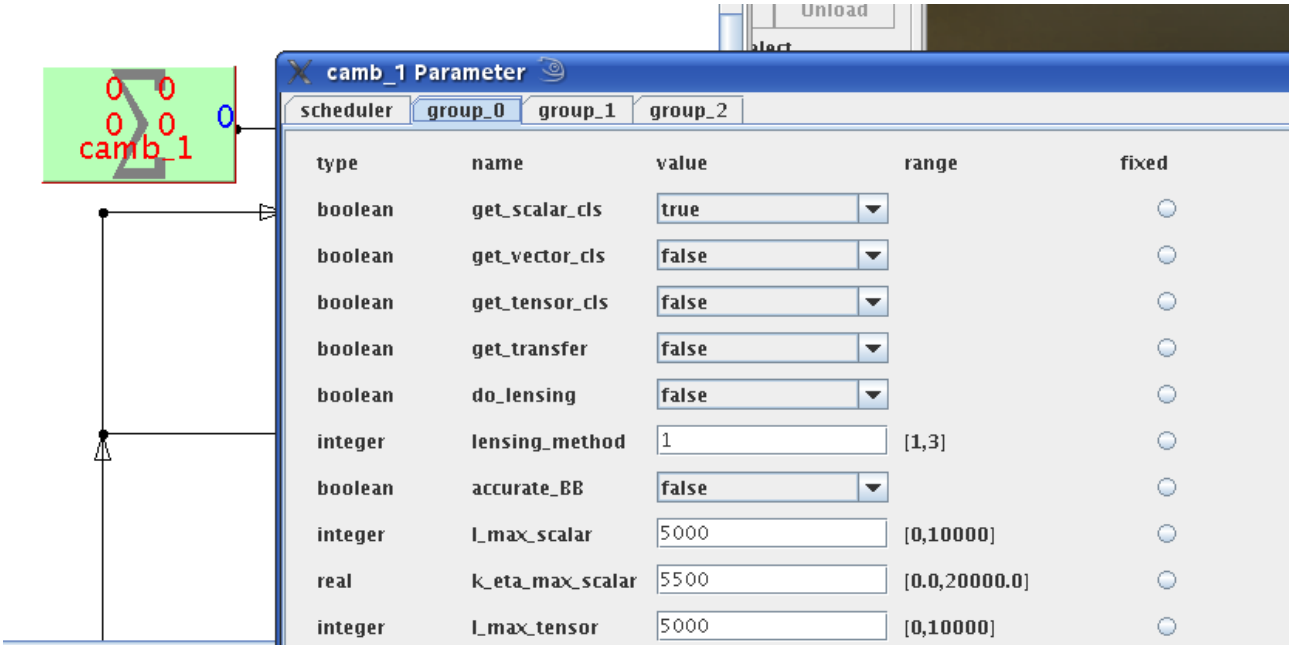


Figure 4.9: Dialog to set and fix parameters of a module in the Pipeline Editor

it provides an easy way to run the same scientific analysis for different parameters (for example, a detector calibration for different detectors, a cosmological simulation for different cosmological parameters).

Pipeline configurations are created in the pipeline configurator, which is started when the **Configuration** → **Create** button is pressed. The handling of module and control parameters is very similar to the editor, except that it is now no longer possible to set parameters to "fixed", or to change fixed parameters. Rather, fixed parameters are shown (not editable) in a separate tab.

In it's standard setup, the ProC requires that all non-fixed parameters are "touched" in the configuration, otherwise the pipeline is not executable. This should keep users from blindly relying on parameter defaults set in the definition.

The ProC requires to configure all non-fixed parameters in it' standard, "strict checking" mode, even if defaults are given in the definition. This can be quite cumbersome if modules with hundreds of parameters are used, and most are kept configurable. As a short-cut, the Pipeline Configurator offers to set the option **Settings** → **Configure All**, in which case given defaults from the definition are taken over into the configuration.

The option Configure All belongs to the ProC preferences, and is kept valid for all piplines until it is deselected by the user. For the dangers of using this option, read section ??

Pipeline configurations can be saved to XML files or to the DMC. The ProC Editor recognizes configurations which match a pipeline definition, and does not allow a non-matching configuration to be loaded.

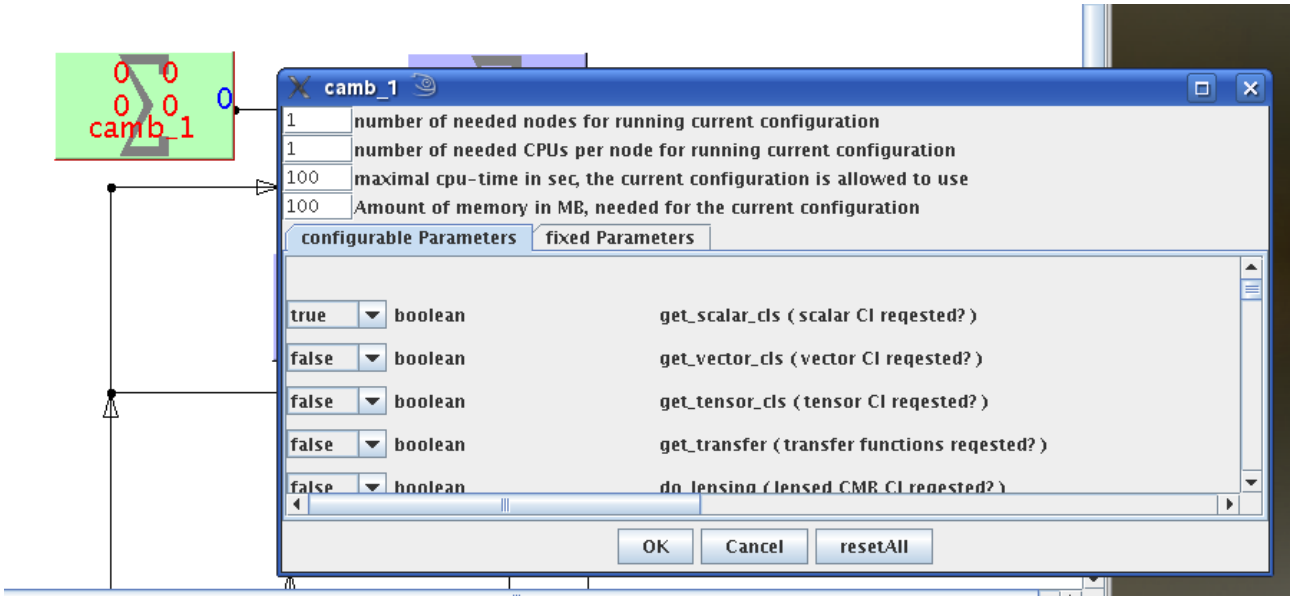


Figure 4.10: Dialog to configure parameters of a module in the Pipeline Configurator

Whenever a pipline definition is changed, old configurations created for it can no longer be used, which can mean a lot of work! It is therefore important to be sure that the construction of a pipeline is finalized, and to protect the corresponding XML file against accidental overwriting, before a large number of configurations are constructed

Besides the parameters of modules, also parameters of control elements (discussed in the sections below) can be set or changed in the pipeline configuration.

In strict checking mode, also all parameters in control elements must be "touched" in the configuration unless the option Configure All is set.

4.2 Pipeline structure and cpntrol elements

4.2.1 Loops and Parallel Pipelines

Parallel Pipelines

The ProC can repeat the execution of certain parts of a pipeline. There are two fundamental types of repetitive structures:

1. Iterative loops, in which a data product is circulated through a sub-pipeline, in which the first element accepts the same data type as input as the last module generates as output.

2. Parallel structures, in which many instances of the same sub-pipeline are executed independently.

The control structures to build loops or parallel pipelines are called the loop and parallel element, respectively. In the editor, they are invoked by the same control element. The graphical depiction are horizontal square brackets, between which the inner part of the loop of parallel structure is placed. The type of the loop is determined by the pipeline structure and the configuration of this control element. Figure **??** shows this structure for the case of an iterative loop.

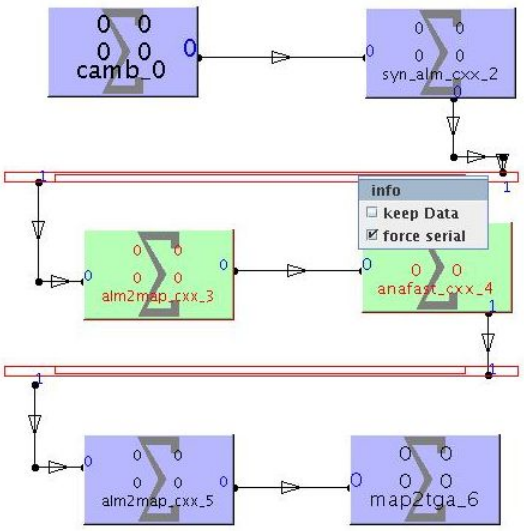


Figure 4.11: A pipeline containing an iterative loop. A data product of type `almLS_alm_pol` is iterated over two modules, shown green here, which transform it into a map and generate it back from it. The modules inside the loop light up green when the loop elements (red structures above and below) are left-clicked. The dropdown menu appears when the loop elements are right-clicked, selecting **info** leads to the configuration menu, **keep data** provides a mode in which all results of single iterations are kept in the DMC, and the **force serial** option switches off parallel operation.

Whether the element controls a parallel pipeline or an iterative loop, is determined by the topology of the pipeline and the configuration of the control element. To build an iterative pipeline, the same data type which is piped from the start control element to the first module, must be piped into the end element by the last module. In this case, the structure is considered an iterative loop. The loop is executed for a specified number of times, or until an exit condition is satisfied. The final result of the iteration can be piped into some post processing pipeline, as shown in Figure **??**. More information can be found in the ProC reference manual section on loops.

In parallel pipelines, no data are piped back into the control element. In order to tell the ProC which modules are inside the structure, a connection is drawn from the last module to the end control element, and then declared a **Virtual Connection** – if the last module does not offer a data type for output which matches the data type piped into the loop, the ProC will automatically chose the connection to be virtual.

Virtual connections are not seen in the editor, but by clicking on the control element, all modules included in the structure will light up green. To remove a virtual connection, right click on the start control element, and all virtual connections will be offered for removal in the drop-down menu.

Finally, both iterative and non-iterative (i.e. parallel) loops are configured by selecting **info** from the drop-down menu which appears on a right click. The configuration dialog allows the user to determine the total number of iterations, and for parallel pipelines also the number of parallel runs. In execution, the ProC will send this amount of subpipelines (i.e., the part of the pipeline enclosed in the control structure) simultaneously to the operating or scheduler system.

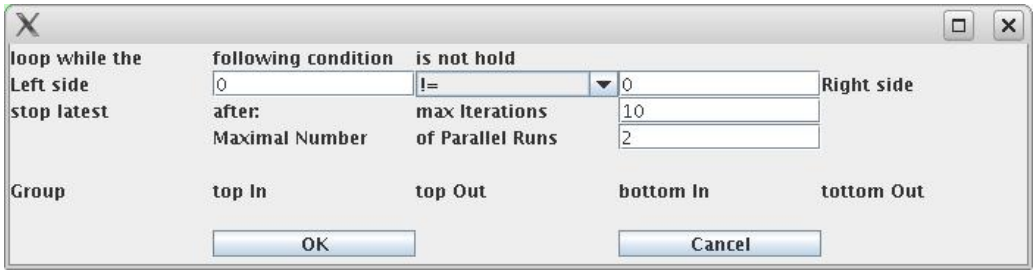


Figure 4.12: Dialog to configure loops and parallel elements. For iterative loops and parallel pipelines for which the **Force Serial** mode is activated, the setting on parallel runs is ignored.

Currently, it is not possible for parallel pipelines to post-process the results of the parallel run in the same pipeline. This is proposed to change with the availability of the list element in IDIS 2.9

Be aware that parallel execution may cause problems in transaction handling in operation with databases, and that the real gain of speed in parallel executions depends on the load of the machine, even if enough processors are present. Read more on potential restrictions on parallel execution in section 4.4.2

Iterative Loops

List Loops

4.2.2 Subpipelines

ProC allows simple parts of a workflow, which are frequently used in bigger pipelines, to be cast into so-called "subpipelines". These are in principle nothing else than incomplete pipeline definitions, which can be used in other pipelines like modules, with their input and output connections given by the missing inputs and outputs of their definition, respectively.

Although subpipelines are thought to be used like modules, their implementation is not quite like this. Therefore, it is currently not possible to use subpipelines inside subpipelines. The same is true for any kind of control elements (except parameter elements). The ProC prohibits to export an incomplete pipeline definition as a subpipeline if such elements are contained in it

A subpipeline has two ways to define inputs: first, all mandatory inputs of modules contained in the subpipeline which are not connected to a module output are considered inputs of the subpipeline; and second, all (input) data object or parameter elements which are not fixed in the definition become inputs of the subpipeline. Outputs of the subpipeline are all unconnected outputs of the modules contained in it. Figure **??** shows a subpipeline defining five inputs and one output.

Empty parameter objects contained in a subpipeline become mandatory inputs and are not configurable as parameters any more.

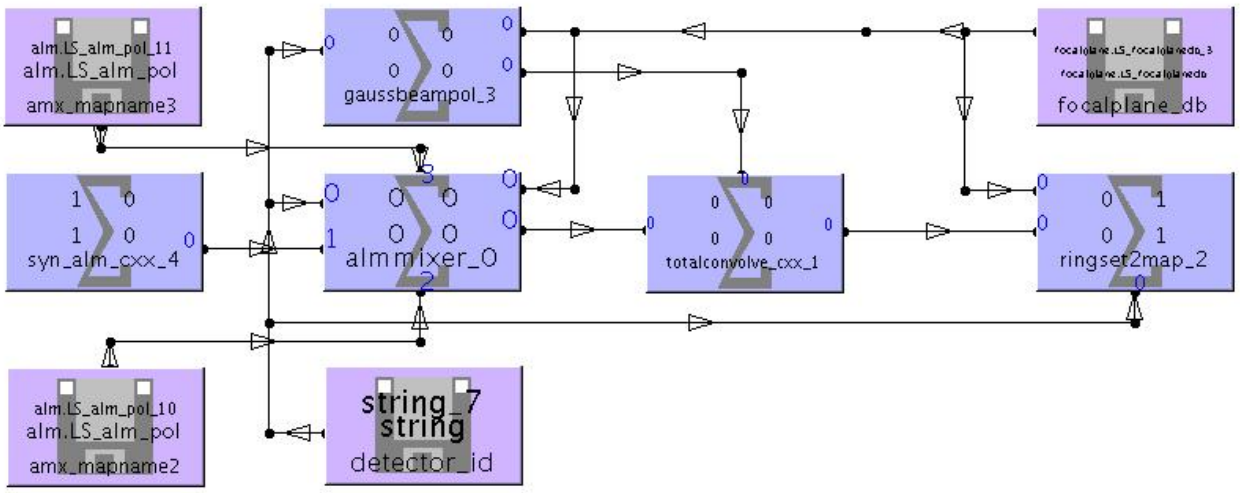


Figure 4.13: Example for a subpipeline with five inputs (one powerspectrum, two of type `LS_alm_pol`, one data object of type `focalplane_db`, and one string for detector_id) and one output (of type `LS_map_pol`).

To export a subpipeline, select **File** → **Export as Subpipeline to File**, and the dialog shown in Figure **??** appears. Fill out the text fields, enter a file name and save. To the file name, the extension ".subdef.xml" is automatically added. Be sure that the directory you save to is in the ProC path for modules and subpipelines, otherwise it has to be added. Currently, it is not possible to save a subpipeline to the DMC.

An exported subpipeline cannot be edited. In order to avoid writing the subpipeline again if a simple change to it is needed, we recommend to save it also as an incomplete pipeline definition (extension `.pdef.xml`). This file can be edited in the ProC, and used to overwrite the `.subdef.xml`.

To use a subpipeline, expand the superfolder "subpipelines" from the left panel in the Editor and select the appropriate name, then draw it into the main editor panel like a module. Inputs and outputs can be connected to it in the usual way, also the configuration is straightforward: In the pipeline configurator, the configuration dialog contains a tab for each module in the subpipeline, on which the parameters of the module can be configured in the usual way (see Section **??**).

Subpipelines can handle optional inputs of modules contained in them, but at least one input must be connected with an (empty) data object. Otherwise the subpipeline cannot be exported. In the example of Figure ??, the two module input connected to input data objects are mandatory, but the other 7 optional inputs of `alm_mixer` are still accessible as optional inputs of the subpipeline.

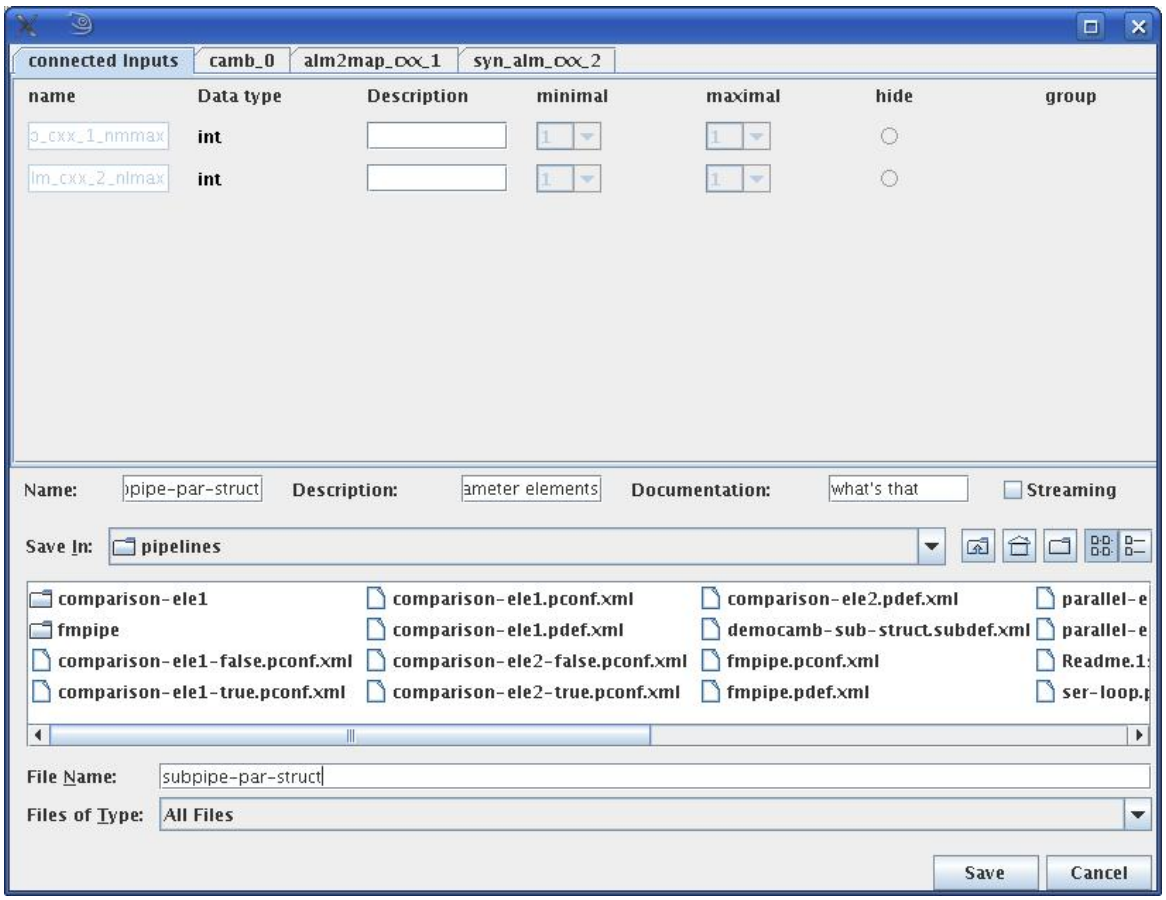


Figure 4.14: Dialog to save a subpipeline. The first text field contains the name, under which the "subpipeline-module" is seen in a larger pipeline using it.

4.2.3 Conditional data flow

Comparison Element

The comparison element is a control element of the ProC which allows conditional execution of parts of a workflow. There are two fundamental applications:

1. Merging two branches of a workflow into one by deciding in a condition which result to use for further processing.

2. Splitting a workflow into two optional branches, and deciding in a condition how to continue.

In principle, a comparison element allows an arbitrary amount of input connections of different data types, and three kinds of output connections: **output for true**, **output for false**, and **true/false selection output**. Every output of a comparison element must have a defined data type, which is chosen from one of the input data types (it is therefore important to always connect all inputs first!). Every input is related to a group, with the group number printed beside it in the pipeline configuration. If a datatype chosen for an output connection matches multiple inputs, the appropriate group has to be chosen. This way, the comparison element can handle for each input a conditional split in two branches.

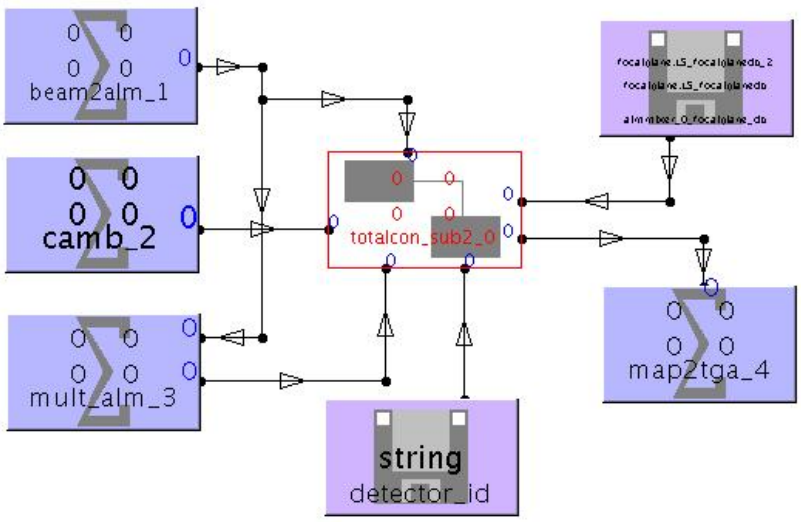


Figure 4.15: A complete pipeline using the subpipeline shown in Figure **??**.

Another option offered by the comparison element is to map two inputs on one output, using the condition to decide which input to take. This option can be configured by right-clicking on the control element and choosing **info** (in fact, it is the only choice!), after which the configuration dialog appears (Figure **??**). Here, for every true/false selection output, the appropriate inputs for true and false can be chosen. Obviously, the data type of both inputs must be the same as that of the output.

Too excessive use of the many configuration options of a comparison element may cause complete confusion for anybody why tries to understand the pipeline without having built it. We recommend to keep the use of this control element simple!

Although we use the phrase "configuration of the comparison element" to fix the relation of it's input and output channels, we should make clear that this is a part of the pipeline definition, and cannot be altered in a pipeline configuration.

Collector Element

4.2.4 The Sampling Control Element

To be written

4.2.5 Input and Output of Data

A module input or output can be directly connected to a file or to the DMC, instead of to another module. This way, pipelines can read complex data which have been produced by other workflows, or write complex data for further processing. This is done by connecting modules to data objects. In principle, this operation is nothing but a formal trick: the module reads/writes to a file or the DMC (dependent on configurations) anyway, the connection to the data object is only to tell the

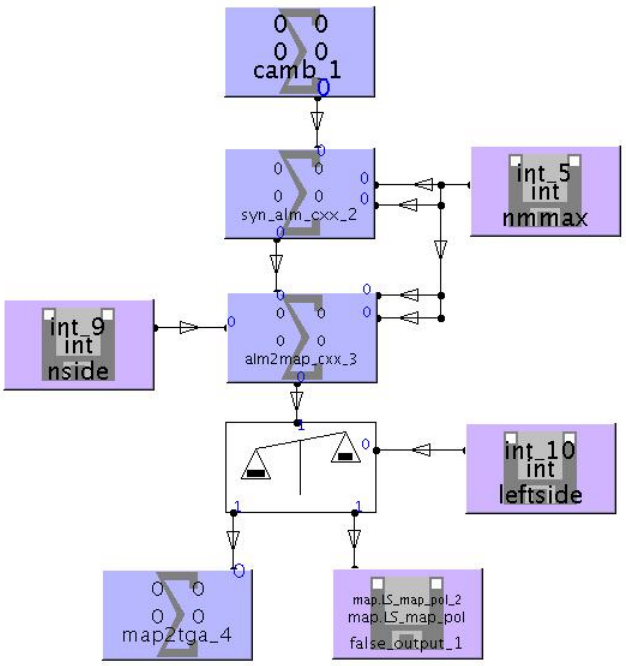


Figure 4.16: A pipeline containing a comparison element which splits the workflow in two branches on a static condition. Depending on the configuration of the pipeline, the generated map is either displayed, or saved into a DMC object. This is a quite meaningful application for a static condition, as the immediate display of the map is good for test purposes, but would cause an error in batch operation on a remote machine.

ProC that the pipeline is completed by the I/O operation (see warning box in Section **??**).

Object elements are used in the editor in a similar way to modules, except that their data type is not predefined; rather it is determined by the selected data slot of the module it is connected to. As soon as a data object is connected, it can be joined either to an output file or a DMC object. The selection is done by a dialog which appears upon a right click on the object element, see Figure **??**

There are essentially three options:

select data opens a standard file selector to determine an input or output file saved on disk.

select from database opens a DMC object selector, which allows the user to choose an input object from query results. For output to the DMC, select an object of the same data type and edit its name appropriately.

determine data opens a freetext field, which allows the user to enter the name of the object directly. If the object is connected to a file or DMC object already, the present connection is shown and can be edited. DMC objects have to be entered in the following form:

```
TOODI\%<db-name>\%\%<object name>:<version number>%
```

If the value -1 is used for the version number, the DMC chooses the next available version number for the object written. If used in an input object, the last version of the object is taken.

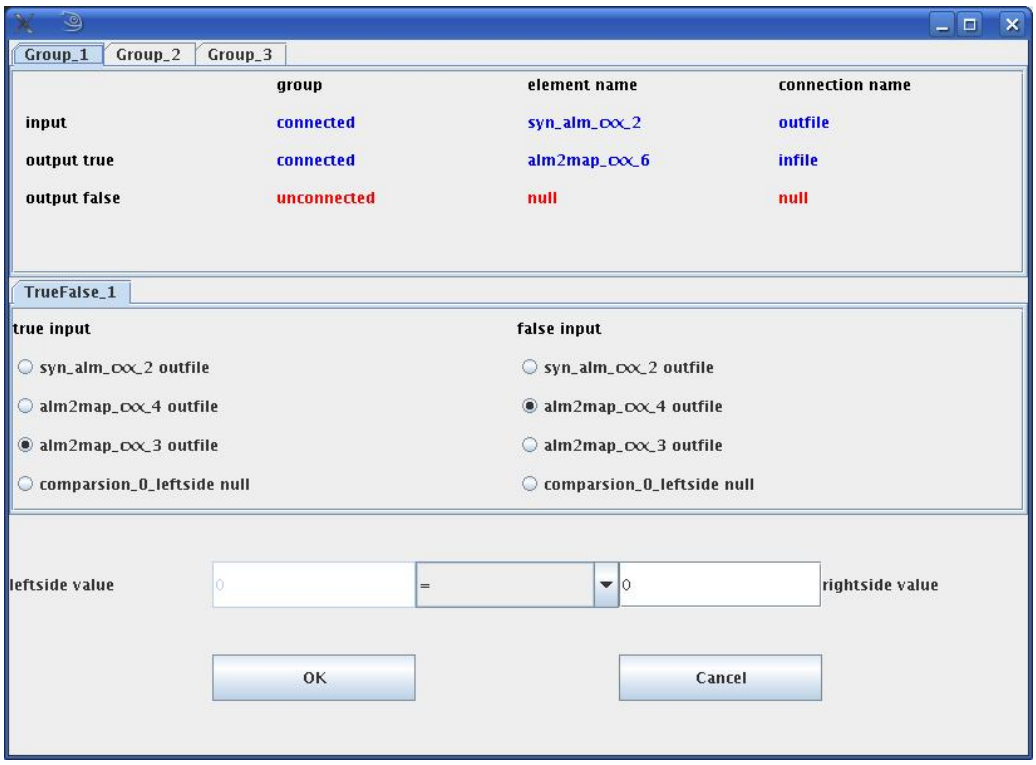


Figure 4.17: The configuration dialog for a comparison element with three inputs, one output for true for the first input group, and one true false/selection output deciding between group 2 and 3, which are of the same data type.

Using determine data is useful in particular if the connection to a data object should be changed to a similarly named object.

4.3 Pipeline Design

In many cases, IDIS offers many ways to solve the same problem - more elegant and less elegant ones, as far programming convenience is concerned, and also more secure and less secure ones, as far as fault prevention is concerned. This section is thought as a guideline who to use the options IDIS provides for an optimal compromise.

4.3.1 What to define and what to configure

Pipeline definitions determine not only the pipeline structure; the parameters of the modules which are necessary for the pipeline operation can all be defined there, too. Moreover, it is possible that the module programmer already defines defaults for all parameters. Therefore, it is possible in principle to run a pipeline without having looked at any of its parameters.

It is obvious that operating pipelines bears the danger that "hidden" defaults, set by the module programmer or the pipeline designer, are used unintentionally by another user. Some modules,

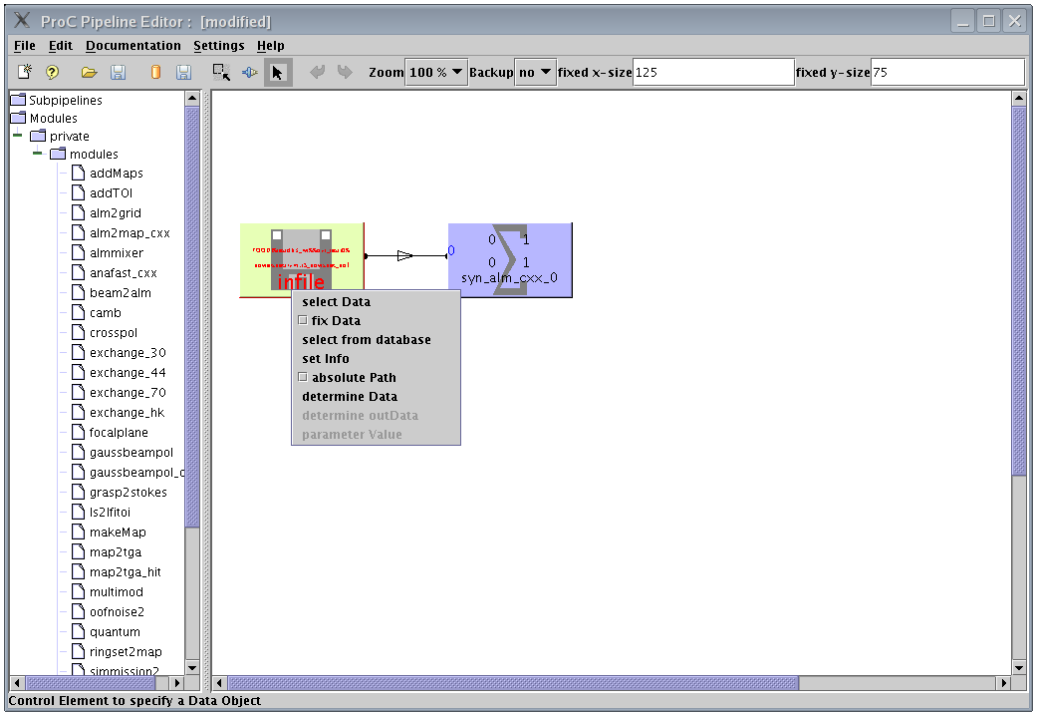


Figure 4.18: Select menu for data input, can be used both in file based and DMC mode

like **camb**, have ~ 50 parameters while a complete, somewhat complex, pipeline may have several hundred: it is quite unlikely that every user in a project knows exactly which of them he can change, which he must change, and which he must not change. However, IDIS offers a solution to this problem: namely, the proper use of parameters in pipeline definitions **and** pipeline configurations.

The idea behind allowing parameters to be set in pipeline definitions is not to introduce hidden defaults, but to **fix** all parameters which should not be changed in any application of the pipeline already in the pipeline definition. This may be seen as an application of the concept of information hiding; in the operation of the pipeline, these parameters are no longer accessible (they are still shown, however, in a separate tab, see Figure **??**). The ProC can be set up such that all parameters which are set in the pipeline definition are fixed automatically: for this, check the **Settings** → **Fix Parameters by Default** option.

All other parameters should be set in the pipeline configuration, which is typically created by the user who runs the pipeline. In order to avoid the accidental use of hidden defaults, we recommend *not* to use the **Configure All** option, and to leave unfixed parameters either empty in the definition (possible for strung parameters), or set them to meaningless defaults. In particularly important cases, empty parameter elements in the pipeline definition can force the pipeline operator to configure the respective parameter.

To avoid the use of hidden defaults, parameters in the pipeline definition should either be fixed, or set to meaningless defaults if possible. Avoid the use of Configure All when running pipelines in operation!

4.3.2 Using Subpipelines

Subpipelines open a way to drive the modularity concept beyond the modules themselves. If some simple workflows are used again and again, it is advisable to save them as subpipelines, and to use these for the construction of larger pipelines containing this sequence. As an example, Figure **??** shows a pipeline which uses the same sequence of modules twice, where the two identical subpipelines encapsulate a simple three module sequence, shown in Figure **??**

If the same subpipeline is used multiple times in one pipeline, its instances can still be configured separately.

Subpipelines implement another aspect of information hiding in the ProC: partial workflows can be independently constructed and tested in a simple context, and then be used in more complex applications, without bearing the danger of introducing errors in reconstructing them on this higher level.

Until IDIS3.0, be aware of the restrictions in the use of subpipelines explained in Section ??. Moreoer, before using subpipelines extensively, read the follwing Section??.

4.3.3 Using parameter and data object elements

To be written

To be written

4.4 Running Pipelines

4.4.1 Execution Control and Logging

To be written

4.4.2 Parallel execution of pipeline elements

On some databases, parallel execution causes a problem. The Apache Derby database, for example, produces all kinds of errors even in simple intrinsically parallel workflows (e.g., with comparison elements): in contrast, more sophisticated databases like PostgreSQL or Oracle can handle some tens of parallel branches in a non-iterative loop. For heavy parallel applications, as for example on supercomputers, we recommend the file-based use of the ProC.

Non-reproducible errors in execution of pipelines involving parallel execution may point to a problem of simultaneous transaction handling of the database. Use the force serial option for parallel elements in this case. So-called "intrinsically parallel" execution of modules in a pipeline which do not depend on each other can be avoided by the usage of breakpoints, unless the intrinsic parallelism occurs at the beginning of the pipeline.

4.4.3 Using Scheduler Systems

To be written

4.4.4 Running pipelines on remote machines

To be written

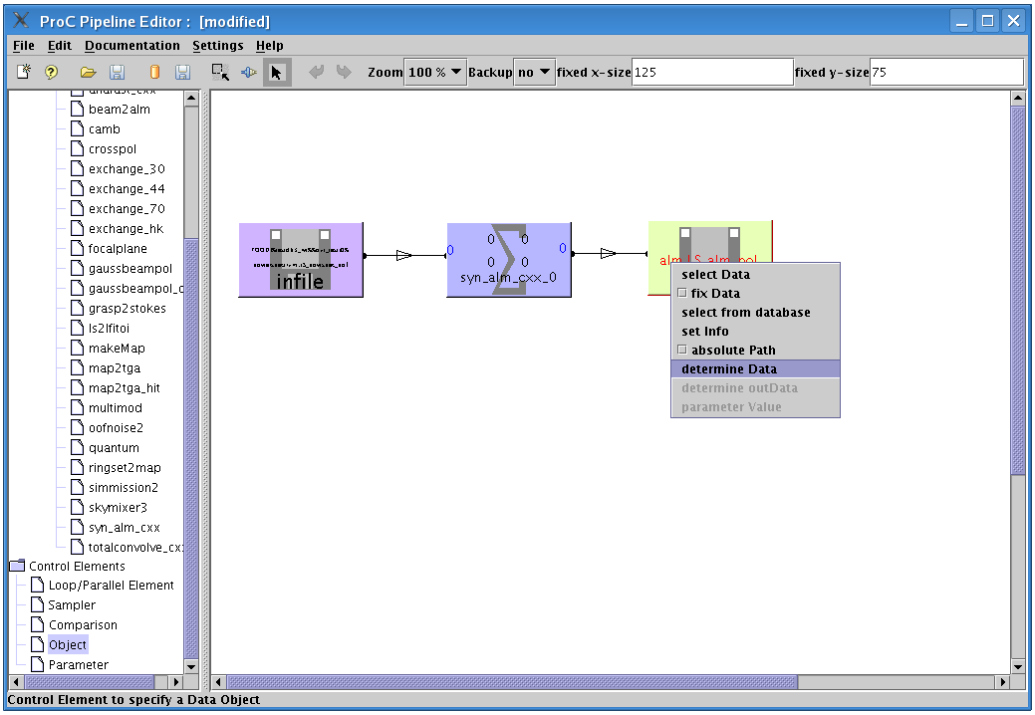
4.4.5 Using the resume functionality

To be written

4.4.6 Running pipelines from the command line

To be written

(1)



(2)

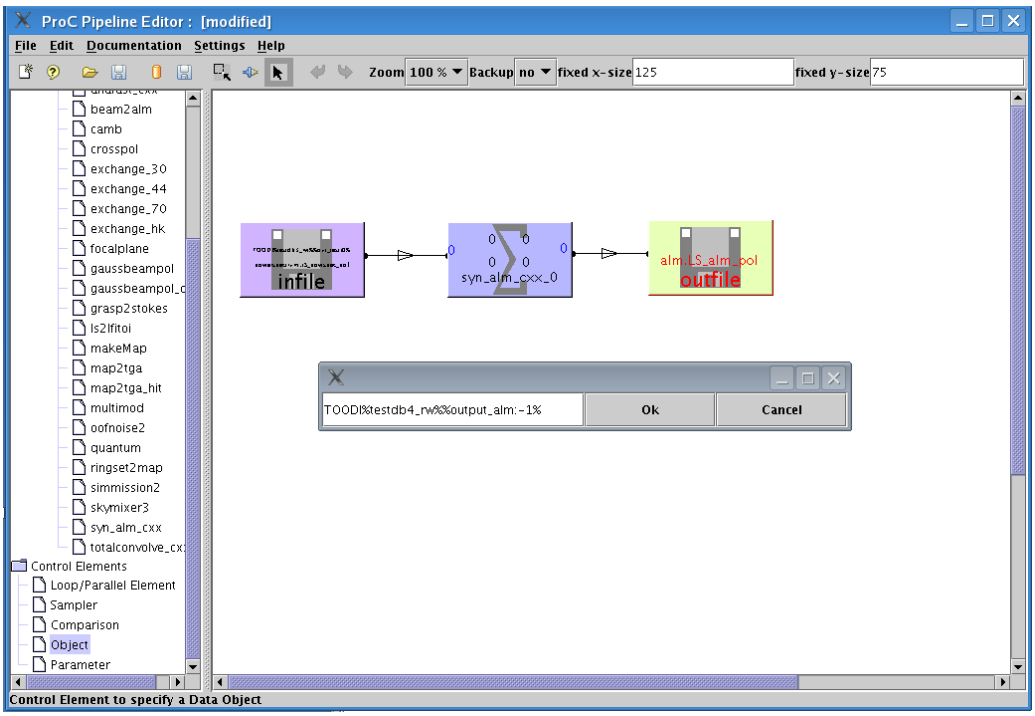


Figure 4.19: An Example for using **determine Data**

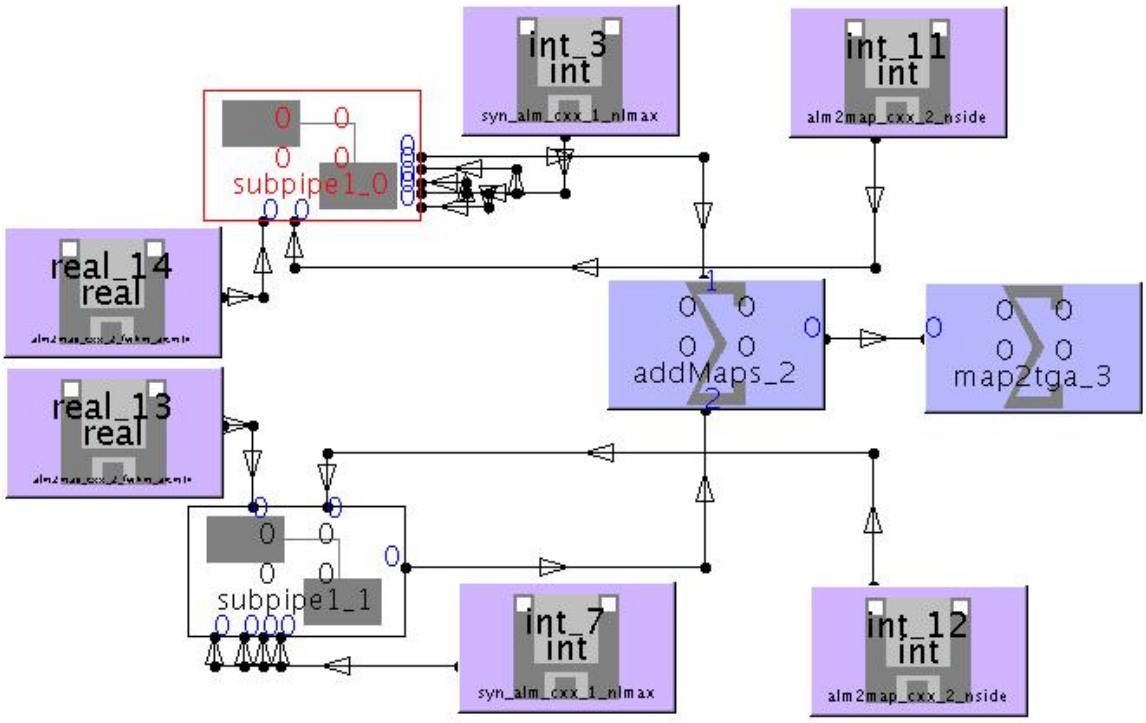


Figure 4.20: A reconstruction of the HFI preFM pipeline in the ProC, making use of subpipelines.

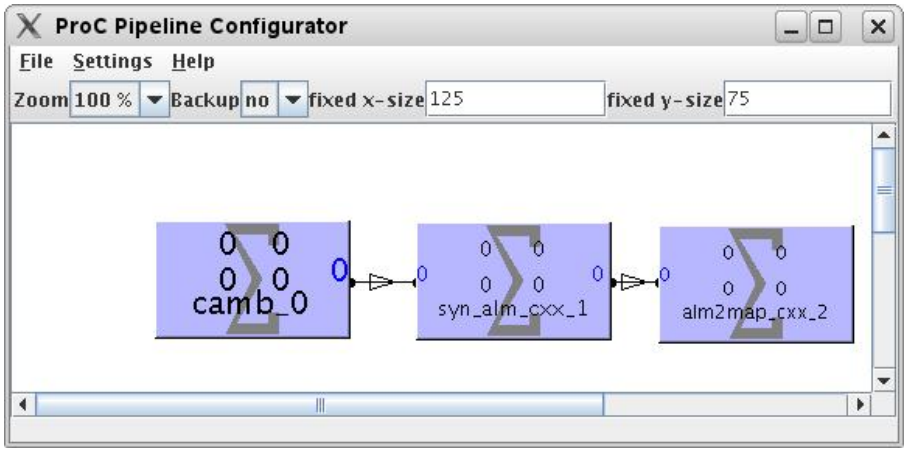


Figure 4.21: The simple subpipeline used in Figure **??**

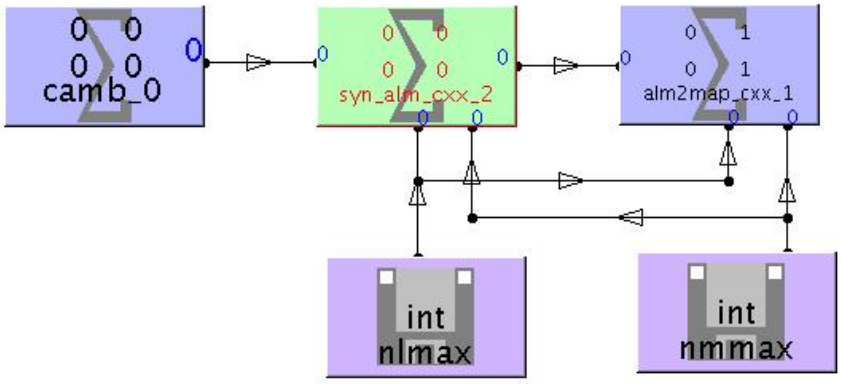


Figure 4.22: Organizing subpipeline input using parameter and data object elements

Chapter 5

The Data Management Component (DMC)

5.1 Introduction

5.1.1 Defining data types

5.1.2 Storage options

Performance considerations

To be written

5.2 The DMC GUI

The DMC GUI is your frontend to the database. It will be presented whenever you need to select objects or pipelines from the database. You can also start it manually from within the DMC directory by calling 'ant gui' - it will then be started in stand-alone mode communicating with the FL and the database based on the parameters set during the configuration. After calling the GUI you will be presented with a window very similar to Figure 5.1. The GUI window is divided into 4 important parts: the menu bar at the top, the DDL-Type listing at the left, the query configuration at the upper right side and the object list at the bottom right.

With the GUI you can:

- Search (Query) for objects
- Look at the (Meta)Data of objects
- Visualize the Data of objects
- View the creation history of an object
- View the dependent objects of an object
- Delete objects

These options are described in the following subsections.

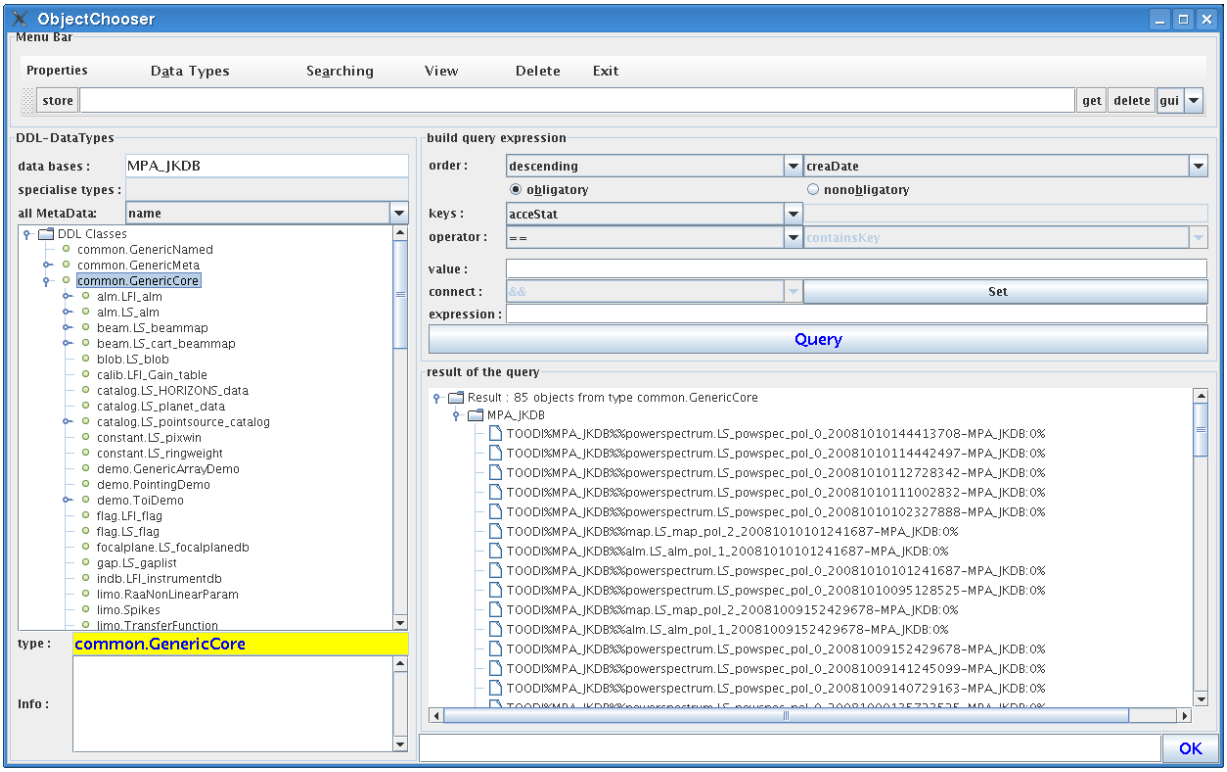


Figure 5.1: The DMC Object chooser

5.2.1 Submitting queries

To be written

5.2.2 Data operations

If you want to look at the Metadata of an object just select the object (it has to be highlighted) and click with the right mouse button on it. A menu is opening which presents you all the options which are also accessible via the 'View'-Menu in the menubar. Select 'MetaData' and a new window will open showing the reserved, obligatory and nonobligatory metadata. When you are done, click on the close button at the bottom.

If you want to see the data of an object, select 'Data' instead of 'MetaData'. If you accidentally request to see the 'Data' for an object which does not contain any data, you will be noticed in the Info-Window at the bottom left. In general 'Data' is only available for objects of the type GenericCore and its descendants.

5.2.3 Visualize the Data

Instead of looking at the data in table format the DMC GUI also provides a simple data viewer when you select 'Graph' instead of 'Data' in the Menu. The graph always shows the content of one data column. To switch to another column you can select the desired one with the drop-down menu at the bottom left.

5.2.4 View the History of an object

To see the creation history of an object(i.e. which modules with which parameters contributed to the data which created the object), select (highlight) the object in question. Then right-click on it and select History from the menu. Another window will open containing two areas. On the left side you see the whole tree of data objects whose data was incorporated in the selected object. On the top of the tree is the object you selected - each object is listed under the pipeline name which created it. Below you now see the data objects which provided data for the creation. In the first indented layer the data was directly used by the module which created the object. Each indentation means that another module processed the data. If you now double-click on a data object you will see on the right side the parameters with which the module was run to produce this object.

5.2.5 View dependent objects

This item shows all dependent objects for an selected data object. There are two options to select: ProC and self-defined - self-defined means that all directly dependent objects (either Associated or MetaData Objects) are regarded. This method should only be used if the data was **not** processed with the ProC. With the option ProC this view shows all dependent objects which are created by the ProC - this includes all MetaData which belongs to the selected dataobject and to further dependent dataobjects. Knowledge of the ProC metadata structure is recommended.

5.2.6 Delete objects

Deleting objects needs to be done with care as these objects are going to be deleted from the database for real. Also all metadata will be deleted.

Deleting objects works nearly the same as viewing the dependent objects since the delete routine takes care of the exact same objects. There are again the two options ProC and self-defined which affect the collection of the data which should be deleted. The option ProC should be used e.g. in the case of corrupted data where a dataobject is corrupted and all data produced by this object is also wrong. The self-defined option should be used if you need to delete a specific object and all its directly dependencies. With this option a check will be done for cross-references i.e. 'external' objects pointing to objects in the list of those which shall be deleted or dataobjects from the deletion list which are used as inputs in other pipelines (this is no direct dependency in words of associated or metadata objects). In this case deletion will not be allowed because history information for those objects might get lost.

You will also need special rights within the Federation Layer in order to delete objects. It will be checked whether the user is manager of the scheduler queue to which the database belongs. In the case that the user is not a manager for this database, deletion will also be prevented.

5.3 The DMC API

Chapter 6

Integrating scientific code into IDIS

6.1 Modular pipeline design

6.1.1 Defining data products

Data products for persistent use

Data products for temporary use

6.1.2 Defining modules

Modules in workflows

In principle, a ProC module is any computer program processing data. To be usable by the ProC, the only requirement is that the input and output of the module is handled through a parameter file (in file based mode) or object (in DMC mode). Thus, every module just takes one argument, which is the name of the file or object which contains all parameters and data used by the module.

Defining subtasks: Transforming data products

Programs which do not comply with this standard can probably still be used with the ProC, if the module wrapper delivered with the ProC package is used. We refer to the documentation of the module wrapper for details on how to use it.

Making parameters accessible

Thinking parallel

6.2 Writing module descriptions

6.2.1 Elementary items

Header

Mandatory Input and Output

Parameters

6.2.2 Groups

Optional Input and Output

Optional parameters

6.3 Integration of code

6.3.1 Overview

Module startup

Data I/O

Supported Languages

To be written

6.3.2 Implementation

C++

Fortran90

IDL

Java

Python

6.4 The module wrapper

To be written