Chandra X-Ray Center

CXC Analysis Requirements

Towards a CIAO 3 Interactive Analysis and Rapid Prototyping Architecture Definition

Rev 1.0 — 13 Dec 2001

David Huenemoerder, John Houck
with help from the Spectroscopy 'n More *ad hoc* working group
(Randall Smith, Nancy Brickhouse, John Davis, Michael Noble, Aneta Siemiginowska, Peter Freeman, Stephen Doe)

Abstract

This "white paper" [1] defines what we mean by an interactive analysis architecture, describes a prototype based upon ISIS[2], and lists some issues, design trades, and lessons learned from ISIS with the hope of aiding design of a CIAO 3 Slang-based interactive analysis system. While ISIS was especially designed for Chandra transmission grating data analysis, the scope of applicability is much broader. Such an interactive architecture serves needs for rapid prototyping as well as productive analysis.

---

[1] Current version can be found at: `http://space.mit.edu/CXC/docs/CIAO_3_Arch.ps.gz`

[2] `http://space.mit.edu/CXC/ISIS/`

1. What is Architecture?

The term "architecture" is used in several slightly different ways. We do *not* mean it in the commonly used senses of programming language, file formats, operating system, or machine hardware. We instead use the term in the software project management sense of the design of the system and not the systems components. The system of relevance in the CIAO 3 context is the the high-level interactive data analysis environment as seen by a user. At this level, what an architecture must define are the functions of data analysis, their arguments, and their relationships.

These requirements can, in turn, can lead to an implementation. Some of this is implicit in the general CXC acceptance of the fact that a functionally based architecture is desired, and that Slang provides a means of implementation. Adoption of Slang, however, is no more sufficient to define an analysis architecture than is the use of C as a programming language, or of FITS for a file format. This is the important distinction between the *design of a component* (C, slang, or FITS, for examples), vs. the *design of the system* (what you actually do with those components).

We won't argue the case here, but will assume that a full Slang interpreter is the basis of the interactive analysis environment (complete with "( )", ";", and multi-line parser), and that there is only one parser running. *(This does* not *mean that there isn't a command interpreter or compiled shell tools available as an alternative! Namely, today's CIAO 2.2 Sherpa-Chips-Slang parsers and shell tools.)*

2. Why Architecture?

Platitudes and pedantry... from: Royce, W., *Software Project Management: A Unified Framework* (Addison-Wesley Longman, Inc., 1998):

> *The most critical technical product of a software project is its architecture.*
>
> *... architecture-first development is crucial to success... Poor architectures and immature processes are often given as reasons for project failures.*
>
> *In iterative development an architecture baseline is a partial realization of the architecture description that is sufficient to provide tangible evidence that the architecture is valid in the context of the requirements and plans.*

> *Substantiation of this state typically requires not only briefings and documents, but also executable prototypes that demonstrate evolving capabilities. These demonstrations provide far more tangible feedback on the maturity of the solution. The more standard components are used, the simpler this state is to achieve. The more custom components are used, the harder it is to achieve and the harder it is to estimate construction costs.*
>
> *Achieving a stable software architecture represents a significant project milestone . . . represents a transition from the engineering stage of a project, . . . to the production stage.*

3. The ISIS Architecture

ISIS was conceived and undertaken about three years ago, and it has been used fairly steadily and productively for about the past two years, and has about 13 refereed publications to its credit. As a slang-based data analysis environment, it provides a tangible baseline which should be considered in detail before major CIAO 3 design and development for an interactive layer proceeds. ISIS applications are primarily Chandra grating spectroscopy, but the openness and flexibility of a scriptable interactive environment has made it amenable to spatial-spectral imaging analysis and to imaging-mode pileup spectral analysis (which required a user-definable fitting kernel). As an example of rapid prototyping, RMF regridding (from PHA to PI, in the same was as events are transformed) was implemented via a C module for trial use within the interactive fitting environment (no RMF is written to disk).

The analysis architecture can be defined in terms of several groups (e.g, PHA spectra and responses, atomic database, fitting) which deal with related operations or data (and which are implemented with associated libraries and wrappers). These generally break down into sub-categories like:

Load — Load a file from disk into an internal table.

Delete — Remove a data structure from the internal table.

Get — Read from an internal table into a variable.

Put — Put data from a variable into an internal table.

Set — Set miscellaneous attributes of the tables.

Define — Define a functional form as an alternative to built-in or file-based entities.

List — List information in tables on the terminal.

Plot — Plot, overplot, residual plot from the internal tables or from variables.

Miscellaneous — ad hoc functions.

A block diagram of ISIS (not current, but representative) is shown in Figure 1; the analysis architecture is contained in the "ISIS Interface Module" box, and is the only part relevant here.

3.1. Scope of Analysis Architecture

To convey some idea of the scope of a the interactive slang functional analysis system, we list global-namespace[3] ISIS functions by category, with a very brief description. The functions are listed without elaboration. Often, their purpose is obvious from the name and is analogous to Sherpa, Xspec, or Chips commands. (For detailed information on arguments and return values, see the manual or built-in help.) A subsequent section will isolate a few classes for high-priority CIAO-3 development, and will describe them in more detail.

Spectral Data: Load, read, write, define, associate counts arrays and responses

load — Load a file from disk into an internal table.

`load_arf` `load_data` `load_rmf`

put — Put a variable into the internal table.

`put_data_counts` `put_data_flux` `put_arf`

define — Define a functional form for the internal data structure.

`define_arf` `define_bgd` `define_counts` `define_flux`

delete — Remove a data structure from the internal table.

`delete_data` `delete_arf` `delete_rmf`

get — Read the internal table into a variable.

`get_arf` `get_arf_exposure` `get_arf_info`
`get_data_counts` `get_data_flux` `get_data_info`

[3]The global namespace defines the top-level interface; there are more functions, but they are lower-level, and more relevant to implementation of the components, rather than the analysis architecture.

`get_frame_time`
`get_hist_backscale` `get_hist_exposure`

assign — Associate responses with counts histograms.

`assign_arf` `assign_rmf` `assign_rsp`
`unassign_arf` `unassign_rmf`

list — List the tables on the terminal.

`list_arf` `list_data` `list_rmf`

plot — Plot, overplot, residual plot from the internal tables.

`plot_data_counts` `oplot_data_counts` `rplot_counts`
`plot_data_flux` `oplot_data_flux` `rplot_flux`
`plot_bin_density` `plot_bin_integral`

set — Set miscellaneous attributes of the tables.

`set_arf_exposure` `set_arf_info`
`set_data_color` `set_data_info`

miscellaneous: — group, bin, ignore, notice, region operations, array arithmetic on internal tables.

`group_data` `rebin_data` `rebin_rmf`
`notice` `ignore` `xnotice`
`flux_corr`
`region_counts` `region_flux` `region_sum`

Atomic Data: The atomic data functions support specification of a database, retrieving atomic data (energy levels, wavelengths, transition probabilities, spectral designations) into arrays, given search criteria such as element, ion, and wavelength, definition of line groups, and display of the quantities.

database specification —

`atoms` `list_db`

line selection —

`wl` `trans` `el_ion`

group manipulation —

`define_group` `save_group` `name_group` `page_group`
`delete_group`

list atomic data —

`line_info` `list_branch` `list_elev` `list_group`

plot atomic data —

`plot_group` `plot_elev` `oplot_lines`
`plot_elev_subset` `oplot_elev_subset`

Plasma Emissivity: These functions deal primarily with the emission for a given abundance and ionization mix, as well as for temperature and density components and their weights.

Set Database, get/set state —

`plasma` `default_plasma_state` `set_plasma_state`
`set_abund` `list_abund`
`load_alt_ioniz` `ion_bal` `ion_frac`

emissivity functions —

`line_em` `ratio_em` `brightest` `get_contin`

Spectral Models: Define/list models, manage parameters, fold through response, read models, counts into arrays.

Model/parameter management — Define function, set parameters, list, read, save.

`fit_fun` `list_functions` `add_compiled_function`
`add_slang_function` `del_function` `iback_fun`
`load_model` `save_model` `edit_model` `define_model`
`list_model`
`set_par` `save_par` `load_par` `list_par` `edit_par`
`get_par` `get_num_pars` `get_par_info`
`freeze` `thaw` `tie` `untie` `randomize`

Model evaluation — Bin, fold, simulate, read models into arrays.

`model_spectrum` `use_delta_profile`
`use_thermal_profile`
`eval_counts` `eval_flux` `eval_fun`
`get_model_counts` `get_model_flux`
`fakeit`

Plot model — Display models, flux or counts.

`plot_model_counts` `oplot_model_counts`
`plot_model_flux` `oplot_model_flux`

Fitting: Define, fit models to the data, apply responses, statistic, optimization method. Provide functional forms for data and responses. Read/write model parameters. Compute confidences.

Fit a model —

`fit_counts` `fit_flux`
`renorm_counts` `renorm_flux`
`conf`

Fit Control and Information —

`list_fit_methods` `load_fit_method` `set_fit_method`
`set_fit_statistic` `load_fit_statistic`
`add_slang_statistic`
`set_kernel` `load_kernel` `print_kernel`

Plot: Display data, models, responses, captions, labels, to screen or file. Generic plots of slang variables, or interact with the plot device.

Device control —

`plot_device` `open_plot` `dup_plot` `close_plot`
`erase` `multiplot` `resize` `window`

Drawing/Window Attributes —

`color` `plot_auto_color` `connect_points`
`cursor` `cursor_box` `cursor_counts` `cursor_flux`
`cursor_sum`
`errorbars` `limits` `pointstyle` `title` `xrange`
`yrange` `xlin` `xlog` `ylin` `ylog`

Histogram/Line/Point/Label Drawing —

`hplot` `ohplot`
`plot` `oplot`
`label` `xylabel`

Utility: generic functions for manipulating data, managing the environment, or performing mathematical calculations.

Environment management —

`add_to_isis_load_path` `add_to_isis_module_path`
`get_isis_load_path` `get_isis_module_path`
`apropos` `help` `alias`
`provide` `require`
`reset` `print` `who`

Math. utils —

`max` `min` `moment` `sum`
`binup` `regrid` `reverse` `fft1d`
`grand` `prand` `urand` `ks_diff` `ks_prob`

3.2. A Note on Functions vs.Commands

A list of function names, as given above, does not indicate the fundamental difference between a simple command-interpreter and a Slang functional implementation. To the casual reader, they look the same. Some functions may even act the same or be used interactively in ways equivalent to a simple command interpreter. The differences (and power) comes in when variables are used for arguments, functions are used inside program control structures, and return values are used.

This simple example treats the file names as variables and loads them in a loop. The files are named explicitly here, but they could be obtained by a function which takes a regular expression and matches names in a directory.

```
                              % Define pha files:
variable
  f6= "Pha/acism00006_004N000_pha2.fits",  % phi=0.14-0.34, low,rising;
  f7= "Pha/acisf00007_005N001_pha2.fits",  %     0.48-0.52, big flare;
  f8= "Pha/acisf00008_005N001_pha2.fits",  %     0.98-0.02, sml flare;
  f9= "Pha/acisf00009_003N001_pha2.fits",  %     0.15-0.35, tail flare;
  f10="Pha/acisf00010_003N001_pha2.fits",  %     0.42-0.46, low;
  f11="Pha/acisf00011_003N001_pha2.fits";  %     0.02-0.06, low;

variable dfiles=[f6,f7,f8,f9,f10,f11];
variable i, num_hist, num_rows ;

variable rows=[3,4];                       % Load HEG orders:
num_rows = length(rows);
num_hist = num_rows * length(dfiles);

variable hist_idx = Int_Type[num_hist];    % histogram indices

for (i=0; i<length(dfiles); i++)
 {
 vmessage ("loading %s...", dfiles[i]);
 hist_idx[ [num_rows*i:num_rows*i+num_rows-1] ] = load_data( dfiles[i], rows);
 }

% Do some analysis, then clean up loaded histograms without touching
% any other data which might be loaded by using the load_data return value:
source("My_Analysis_Script.sl");
delete_data(hist_idx);
```

See the CIAO web pages for more slang examples and tips. [4]

[4] http://asc.harvard.edu/ciao/ahelp/slang.html

4. CIAO 3 Priorities

Some work is already done or in progress on slang modules (chips plotting, param i/o), and library re-work is underway (sherpa). Randall has identified some sherpa "`get_`" functions as high priority. These have by-and-large been attacked in a sensible order, but it is time to survey the system and attempt to define the entire scope and priorities.

`get_` Retrieve data from internal structures. This is necessarily high priority so that the arrays can be used in custom analysis. These functions should at a minimum get spectra, images, ARFs, RMFS, models, attributes, and model parameters. Each will require a definition of a data-structure. (e.g., varmm structures may be appropriate; sometimes simple structures of arrays w/out headers are sufficient).

`plot_` Also logically high priority, since once data are "gotten" and undergo computations, they need to be visualized. Plot must support histograms (bin_{lo}, bin_{hi}) pairs on integrated values as well as differential x, y data. Plot functions should be defined which read and display internal structures, which have stored attributes (ignored regions, histogram values).

`load_` Loading data and responses can be prototyped (maybe) via the `sherpa_eval()` function. But in the long term, they need to be written as intrinsic functions with parameters and error checking.

`assign_` Comments on "load_" apply here.

`put_` Putting data back into internal tables is important for customizing responses or data, if internal tables are the basis for fitting and modeling operations. "Put" must also be available for attributes of table entries (e.g., exposure times).

`define_` Defining via functions is important for fine user control over responses, kernels, and models. We already have a user-definable source model; user-definable responses, statistics, and optimization methods are also needed.

`miscellaneous:` Some miscellaneous operations are invaluable for analysis, and preclude the need for disk-based tools. For example, grouping and rebinning of internal histogram tables; regridding of arbitrary slang arrays. Math libraries, system utilities, searching, sorting, binning all fall into this category.

4.1. Adding a Little Detail

It would be possible to traverse the sherpa commands and define a slang function for each command. Randall has looked into this, and come up with a draft of "`sherpa_get_`" functions. Care should be taken to consider the function arguments, return values, and data structures. Instead of mapping sherpa commands directly, functions should be defined as they are desired to be used, with appropriate data structures. Mapping should be done at the library level, not command level (using commands post facto as a guide to the library contents). It is possible that there could be one-to-many or many-to-one mapping, and useful data types in addition to structure tables (essentially analogous to FITS binary tables).

Here is Randall's draft list (red items (maybe), prefixed with "`sherpa_`)", and some commentary intended to indicate the kind of additional information needed to define slang functions based on ISIS' related counterparts (in blue (maybe), with "`isis> `" prompt). This is a very terse listing; see the ISIS manual or help for longer descriptions.

```
sherpa_get_data(dataset)
sherpa_get_data_errors(dataset);
```

Is there modal behavior on flux and counts? What is the return value? Should errors and counts be returned as part of the same structure?

```
isis> Usage:  Struct_Type = get_data_counts (hist_index)
isis> Usage:  Struct_Type = get_data_flux (hist_index)
isis> print(get_data_counts(1));
   bin_lo = NULL
   bin_hi = NULL
   value = NULL
   err = NULL
```

Also in ISIS are the related functions:

```
   Usage:  id = load_data ("pha_filename", [row_num])
   Usage:  Struct_Type = get_data_info (index_list)
   Usage:  put_data (hist_index, struct -OR- binlo, binhi, value, uncert)
```

```
sherpa_get_filtered_data(dataset)
sherpa_get_data_filter(dataset);
```

Does the first function apply notice or ignore regions? Could the filters be part of the returned data structure?

```
 isis> y=get_data_info(3);
```

```
 isis> print(y);
     spec_num = 9
     order = -1
     part = 2
     srcid = 1
     notice_list = Integer_Type[200]
```

```
sherpa_get_instrument(dataset)
sherpa_get_rmf
sherpa_get_arf
```

What is the instrument model? ARF, RMF assignments? Should the instrument include the assigned responses? (See ISIS get, load, assign, put functions.)

```
sherpa_get_source(dataset)
```

No ISIS counterpart. ISIS only has one source model for all noticed histograms. (There will be more sherpa-only cases, especially in imaging applications.)

```
sherpa_get_model_parms
```

What is the returned data structure?

```
isis> Usage:  struct = get_par_info (idx)
isis> print(get_par_info(1));
    name = gauss[1].area
    value = 1
    min = -1.79769e+308
    max = 1.79769e+308
    freeze = 0
    tie = 0
    is_a_norm = 1
```

This can be used in a loop to build a structure array. I don't know about nested models. If user-defined (the only way in isis), then it is transparent - they just become new parameters (any "nesting" is left to the user function). Also in isis: see "_par" class of functions.

```
sherpa_get_dataunits()
```

Not supported in isis. Units can be set for a plot.

```
sherpa_eval_model("string", energy||wavelength||channel);
```

What is the result of a model evaluation? Is it mode dependent, parameter dependent, or will there be explicit multiple functions?

```
isis> s = eval_counts([response_type] [,\&info_struct])
isis> s = eval_flux([response_type][, \&info_struct])
```

```
isis> flux = model_spectrum (binlo, binhi [,line_list]);
```

(`eval_` works on noticed region for noticed histograms. `model_` evaluates on grid.)

5. Design Issues

The preceding material was written from the ISIS point of view. ISIS has made some trade decisions, which are not necessarily either right or wrong, but which do fundamentally affect the way one interacts with data. For instance, there are "`_data`" and "`_counts`" versions of some functions in order to avoid modal behavior. Data are loaded into internal tables first, but if wanted as variables they have to be "gotten". Here is a list of issues in no particular order:

1. Should plots be formatted by function calls? e.g.

   ```
   connect_points(1);
   color(blue);
   xrange(1,10);
   ```

   or by setting fields in a global structure?

   ```
   this_plot.color = blue;
   this_plot.connect = 1;
   this_plot.xmin = 1;
   this_plot.xmax = 10;
   ```

2. What global structures should be defined? Should functions read them implicitly or accept them as arguments? E.g., as per a plot structure:

   ```
   plot( x, y, this_plot );
   ```

3. Should data be managed by the user?

   ```
   d1 = readfits ("d1.fits");
   d2 = readfits ("d2.fits");
   d3 = readfits ("d3.fits");
   info = fit ([d1,d2]);
   ```

   or by the application?

```
() = load_data ("d1.fits");
() = load_data ("d2.fits");
() = load_data ("d3.fits");
ignore(3);
fit_counts (&info);
```

4. Which functions return error codes? Only those which which are likely to fail (loading data)? What is done on error? Exception handling?

5. Which functions return structures? Should functions be allowed to return multiple values?

```
isis> (conf_lo_flux, conf_hi_flux) = conf(1) ;
```

   vs.

```
isis> conf_flux = conf(1);
isis> print(conf_flux);
    low = 1.234e-3
    high = 2.345e-3
    interval = 0.9
```

6. What other data types should we provide? (linked lists; relational databases)

7. Which common functions should be intrinsics? (e.g. how much should be left to the user?)

8. What's the best way to organize libraries of scripts so that we don't have the problem of "where's the entry point?"

9. What slang coding conventions should be enforced? recommended? (usage, help, return values, error handling)

10. Should modal behavior be supported, or excluded in lieu of explicit command names or arguments?

11. Should arguments be positional, w/ null values allowed and parsed? e.g.

```
foo(a,  ,  , d);  % b & c are not given, but parsed as NULL.
```

   vs.

```
foo(a, b, c, d);  % to get to d, b & c must be specified.
```

12. When should we use parameter i/o on slang functions?
13. How are journaling and logging to be done?
14. What save/restore, do/undo capabilities are desired?
15. When/how is header manipulation to be done? A functional interface definition is needed.
16. How are large files to be handled? For some operations, buffering is possible (via `CFITSIO`).
17. What debugging tools are needed? Slang has a traceback function; scripts are easier to debug via print statements, and are not prone to memory corruption. A debugging tool is probably not required, as for compiled code.
18. What parser modes are permitted? What are precedence rules to resolve ambiguous arguments (if any)?

6. More Examples

Several problems have arisen in which Slang provided a rapid solution:

Inserting columns read from an ASCII table into a FITS table
(`http://icxc.harvard.edu/pipe/ascds_help/2001/2406.html`).

Data-type conversion
(`http://icxc.harvard.edu/pipe/ascds_help/2001/2582.html`)

File conversion: ASCII to FITS or QPOE to FITS (see "`ahelp writefits`" examples).

Clearing event `STATUS` bits
(`http://space.mit.edu/CXC/ISIS/examples/clear_bits`).

Adding orders: see the `add_grating_orders` Slang script, in comparison to the `sh` version (covered elsewhere in detail by Mike Noble).

To make these more than "one-off" utilities, some work may need to be done to modify headers or to handle any special cases. The main point is that

Slang allows a quick solution, and provides an iterative path of prototype to "production" solution.

Some other problems can be addressed by some compiled code which is then provided as a slang intrinsic. E.g, `rebin_rmf` in ISIS. This provides a prototype for experimentation with alternate analysis products. It reads a PHA RMF, and re-grids it to a linear (PI) RMF. This works in internal memory space (no output file), and can be used to study differences from FeF PI-based RMFS.

7. Conclusions

This document hopefully provides a baseline for discussion of a CIAO 3 interactive analysis and rapid prototyping architecture and gives some indication of the scope. Forethought is needed to decide on some important design features, and to define fundamental *classes* of functions. That is, some generic interfaces. For instance, the need to solve the pileup equation led to a generalization of the integral being solved (it's non-linear). Hence, there is not just one pileup model in ISIS but a new class of functions, in which the kernel of the integral can be user-defined (as opposed to independent source model and response hooks).

Moving from prototypes to production tools can be done in a variety of ways. Slang functions can be simply wrapped for the shell (such as `add_grating_orders` or `clear_bits`, polished for interactive use, or coded into compiled tools if there is need for efficiency or infrastructure not available via slang intrinsics.

The scope may seem large, if based on a function list. However, many of the functions fall into classes of similar functions, and many are also fairly simple wrappers to already existing code. A little interface definition and library organization can provide a lot of interactive analysis power.

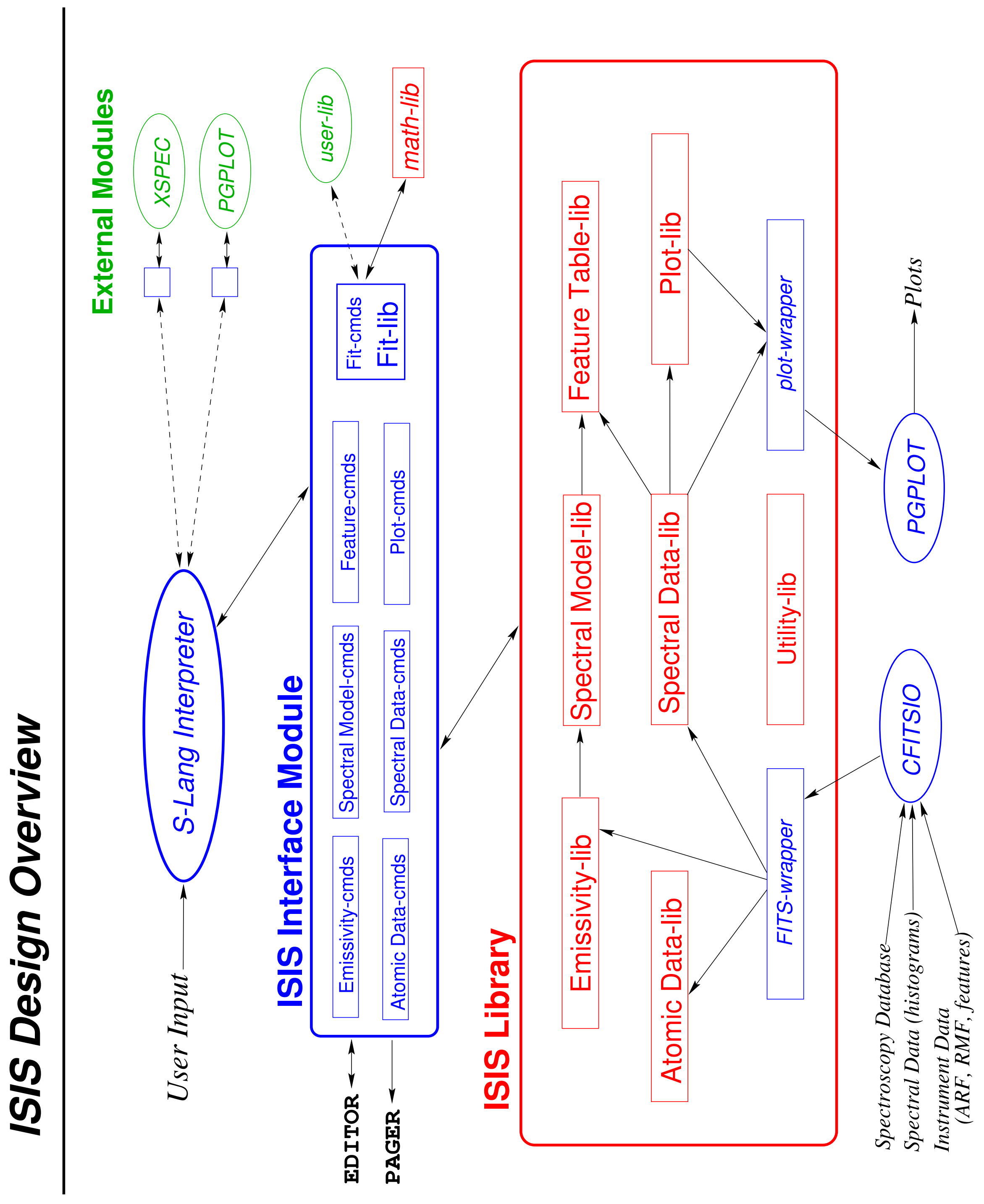

Fig. 1.— *For reference, here is a sketch of the ISIS design. Everything to do with the data analysis architecture is in the "ISIS Interface Module" box.*