From: Ed Fomalont
Subject: Note from Ed Fomalont

Dear Brian,

You are getting to the stage where you need good input from people who have met all kings of astronomical data. I am committed to debugging the VLBA and this will occupy most of my time, so I'm sorry that I can't spend much time on the aips++ design. But, I want to say a few things about what you should be thinking about for the design.

The most important point, in my opinion, is that to design a good, flexible system, you need only worry about how to APPLY calibration. This is basically the TELESCOPEMODEL object. This means taking measured data and transforming them to obtain something called calibrated data. These algorithms are general simple matrix transforms of the input data. How to determine the parameters in the APPLY calibration matrices is called SOLVE. The complexities involved in solving from parameters from a set of data is enormous. But, I don't think you have to worry about this. You have to make sure, that the structure of aips++ can handle the APPLY calibration process and transfer sufficient information (data and calibration parameters) to SOLVE, which the gives better calibration parameters. But, what and how SOLVE does is unimportant at this stage of design.

One important assumption is that APPLY calibration need only handle data at one particular time interval. If this is true, then this must say something about the fundamental ordering of the data base. Clearly the APPLY parameters will change with time and the information (tables? algorithms?) must have to be repeated as often as necessary. And the proper interpolation methods must be used.

On polarization calibration. At most you observe four independent polarization data points for any baseline and time. The general calibration procedure is a 4x4 matrix which transforms these four inputs points to four output points. It doesn't matter what type of polarization state is measured (RR, LL, RL, LR or linears or some Westerbork style) or what output is desired (I, Q, U, V or RR, LL, RL, LR), I think that a 4x4 matrix will do it. Of course each matrix element will depend on the style of polarization measurement and calibration. And SOLVING for the parameters for a specific telescope is tough, egs. VLBI polarization mapping.

BUT, in determining the properties of aips++, I think you only have to convince yourself that a 4x4 matrix is all that you need. How to fill in the matrix is not important as long as you know how to send the appropriate data (now over a long period of time) and a description of the parameters to SOLVE and let it do its job, somehow, and return better estimates of the parameters which you stick in the proper matrix.

Some complications: It is possible to measure many more than four input values which are then not all independent. The APPLY would then be a sort of least square solution which obtained the four desired independent polarization parameters. On the other hand, only one or two polarization parmeters can be measured. The output then cannot determine the four polarization states and simplifying assumptions (ie some of the 4x4 matrix elements will be zero) must be made. BUT, this can all be buried in some Matrix (nxM) where n is the number of input polarization measurements and M is the number of output (M<n). I'm not sure if all this is correct, but this shows you the kind of things that have to be designed into aips++ for proper APPLY calibration. Again, DON'T WORRY ABOUT SOLVE. As long as you have defined the APPLY part correctly and can send data and parameter information to SOLVE, it can do something.

In self cal we generally assume that the atmospheric/tropospheric phase correction over the entire primary beam is constant. We, therefore determine one phase for each antenna. In principle, the phase may change over the primary beam and the changes may have a different form between antennas. Close antennas should have the similar phase behavior over the primary beam, distant antennas should be independent. The main question facing the aips++ design is how to structure the parameters needed for this problem in sufficient detail so that you can add on this complicated self-cal in the future. It

will take many years of experience to know how to SOLVE this problem, but that is not of concern now. The present problem is how best to define the APPLY calibration part. Of course, we rely on experience of SOLVING this problem in the past to guess at a good APPLY for aips++.

A calibration model might be: a phase at the center of the primary beam, a wedge of arbitrary direction, and maybe a second order term. This already has about 10 parameters, so that with 27 antennas, you have 260 parameters (one reference antenna). With the VLA with 351 baselines, it would be difficult SOLVING for all parameters. The SOLVE program would have to be very smart. Couple some parameters for close antennas, for example. Of course, you would need a field which contained lots of radio sources so that these parameters could be well sampled in the primary beam. Maybe, a more general calibration model could be thought of so that you don't have to decide now on what is should be. Maybe, just a 2d-powerlaw of refraction across each primary beam. So, I guess what I am saying is that an isoplanicity APPLY system might include some algorithm which has some functional dependence (upto second order) in the primary beam of each antenna. You might want to add in weather data from which you can predict the differential refraction of a smooth atmosphere. This APPLY calibration would only be applicable to data and associated sky models (not point sources) which contain lots of sources.

I guess what I am saying is: just make sure you are able to define the spatial phase behavior over each antenna primary beam in some way. How it is used (most of it will hardly ever be used, only the phase at the center of the beam). Also, have descriptors which say how much of the description is used (or actually stored) and at what time intervals.

Although APPLY calibration works for any one time duration, the parameters will change with time. So that each part of APPLY must have many entries, separated by appropriate amount of time. The frequency of the entries must be arbitrary and aips++ needs a good way of dealing with this.

This has been a rambling note. I hope my basic point has come out. Again, I'm sorry I can't spend more time on this.

Ed

From: Andrew Klein
Date: Fri, 3 Apr 92 13:27:24 CST

Brian, et.al. -

Well, my thinking cap has been somewhat overused recently dealing with starting up this project at Motorola. I've seen the aips2 mail off and on, including Ed Fomalont's recent reply. I wish I could tell you the perfect answer and hand you the ideal design (you probably wish I could too!) but that isn't possible.

Let me restate the basic point: the object-oriented design should reflect what it is you want the system to do now and what you think it will do in the future. The class structure is a static picture that reflects how you are managing your code for current ease-of-use and future reuse. The object structure -- which objects are created at run-time, who they talk to and what they say to each other -- is a dynamic picture that reflects the functionality of your system.

Returning to the static picture, when I say "ease-of-use" I am referring to the ease of the aips++ designers/programmers in designing and building aips++ and to the ease of your customers (and yourself) in extending aips++. I am not referring to the end-user/astronomer who will use aips++ to do data reduction and analysis: he or she should find that the system presents a user interface that they understand and can use to make the system do the job that they want it to do.

So, to make your own job as designer and programmer easier, you've turned to OOD/OOP and C++. It promises a cleaner mapping from the entities of the problem domain to the objects of the solution domain. With that cleaner mapping comes a greater localization of knowledge,

so that problem domain details can stay together in a single class, avoiding the entangling interactions that diffuse details all over the code.

The problem is that it isn't easy to find these clean divisions. And in fact, the aips++ group has been fighting over what the right kind of divisions should be. The usual tests, whereby you evaluate a class by the cohesiveness of the data it holds and the services it provides, and by its match to the natural entities of the problem domain, aren't helping very much. From my point of view it appears that there aren't any 'natural' problem domain entities that you can agree on. The entities that have been identified (by the Green Bank group, by Chris Flatters, and by Ed Fomalont) are abstractions of one sort or another, models and model corrections and model parameters, that interact in different ways.

Since I don't have answers to these problems, let me offer the next best thing: questions.

1. What are the processing steps that you want to use on your data? What would make it easier to organize the code to do those steps?

   If it is essential that blocks of observations must be called up, combined together, processed, evaluated -- and then based on the evaluation, split apart for different processing in different combinations -- then this would argue for the need for an object that holds onto these dynamic associations. It also means that you need to keep track of the original data, the mutated (no, I didn't say "nutated") data, and its processing history.

2. Is the data (I'm thinking of the differences in the data you get from different instruments) you process similar, or at least similar enough that it shares a common set of behaviors? At design time (that is, right now) can you say of a kind of data that you know what operations you can apply to it, and what parameters each of these operations require?

   If an operation that can be applied to the data is common over several kinds of data, so that the type returned by the operation and the number and types of parameters taken by the operation are the same, then you have discovered a conventional member function. And in the process you have also discovered that the commonality of "the several kinds of data" can form a C++ class.

   If an operation that can be applied to different kinds of data is only similar conceptually (e.g. thisdata.solve()), where the design of the operation is wildly different in each case, taking entirely different kinds and numbers of parameters, then what do you do?

      Well, you could still use conventional member functions. To get the parameters to the operation you could:

      1. Pass them in a general parameter object. This hides the basic problem one level away. The code for the parameter object would have to be modified every time you added another item to the parameter protocol, as I'm imagining that the parameter object contains all possible parameter items. If the parameter object was initialized incompletely (it is missing a valid value for a parameter item that the current operation needs to have) you won't find out about the problem until run-time.

      2. Get the parameters you need from the data you are processing. This is just like 1., except you've moved the problem.

      3. Get the parameters from some global store or association object... Again, just like 1.

      4. Use a parameter object, but this time make it an association list of keyword/value pairs (e.g., (("offset" 1.337) ("fieldbrt" 234.12)...)). The operation could search the association for what it needs. The advantage is that you could extend (but never retract) the parameter protocol without having to recompile the world. You still have the problem of dealing with run-time errors as in 1. This is not

the kind of processing overhead that belongs in tight loops; the assumption is that the processing time of the operation on the data far exceeds the overhead of getting the parameters.

To the extent that this dynamic kind of processing becomes primary, you will find that you are leaving behind the compile-time type safety of C++ and creating your own world of run-time type identification and method dispatch.

Andrew V. Klein