Author: M. Loose, J. v. Zwieten	Date of issue: 2007-Mar-30 Kind of issue: Public	Scope: CEP/BB Doc.nr.: LOFAR-ASTRON-SDD-053
	Status: Draft Revision nr.: 1.0	File: BBS-SDD.tex

BlackBoard SelfCal (BBS) Software Design Document

G.M. Loose, J.E. van Zwieten

Verified:			
Name	Signature	Date	Rev.nr.
R.J. Nijboer	....	2007-Mar-27	0.2

Accepted:		
Work Package Manager	System Engineering Manager	Program Manager
R.J. Nijboer .... *date:*	A. Gunst .... *date:*	J. Reitsma .... *date:*

Distribution list:

Group:	For Information:
J. Noordam	O. Smirnov
R. Nijboer	M. Mevius
J. Romein	S. Yatawatta
G. van Diepen	V. Pandey
R. Overeem	
G. de Bruyn	
M. Wise	

Document revision:

Revision	Date	Section	Page(s)	Modification
0.1	2006-Nov-21	-	-	Creation
0.2	2007-Mar-23	-	-	Major update for CDR
1.0	2007-Mar-30	-	-	Incorporated review comments

Abstract

DID YOU FINISH THE TECHNICAL EVALUATION?
FIRST DRAFT.
www.dilbert.com scottadams@aol.com
UM. . . THIS SAYS NOTHING BUT LITERALLY "BLAH, BLAH, BLAH."
© 2007 Scott Adams, Inc./Dist. by UFS, Inc.
I LIKE TO NAIL THE SPACING FIRST.

Contents

1 Introduction

1.1 Purpose of This Document

This document provides a detailed description of the architectural and software design of the Blackboard Selfcal System (BBS) that will be used for the calibration of the LOFAR observations. The primary goal of this document is to provide information that is detailed enough to help the reader understand the design considerations, choice of software architecture and global design. We will not delve into the level of detailed software design, since this will likely cause discrepancies between the actual code and this document. For this level of detail, the reader is suggested to consult the on-line code documentation. This document supersedes the previous version of the BBS SDD [1].

1.2 Abbreviations

ACC	Application Configuration and Control
BBS	BlackBoard Selfcal system
OLAP	On-Line Application Processing
SAS	Specification, Administration, and Scheduling

2 Architectural Design

2.1 Design Considerations

The BlackBoard SelfCal (BBS) system is designed to do the calibration of LOFAR in an efficient way. Although BBS is mainly developed for LOFAR, it may also be used to calibrate other instruments as soon as their specific algorithms are plugged in.

2.1.1 Data Volume

The volume of the data coming from the LOFAR correlator is *very* large. During initial operation (mid 2008) the amount of data generated during an average observation will be in the order of several terabytes. Once LOFAR is fully operational, this number will have increased to almost a hundred terabytes. Given the output data rate of the correlator and the storage capacity of harddisks, it is obvious that the data cannot be stored on a single system, not even if an array of harddisks were used. The only feasible way to handle these large data sets is to distribute them to multiple computers. Each computer will have to store and manipulate part of the data.

2.1.2 Distributed Processing

The Selfcal application will be running on the off-line and auxiliary processing clusters of the central processing facility (see [2]). These clusters consist of Linux PCs in a high bandwidth network. The BBS application will run on a large cluster, typically consisting of several hundred nodes. Data stored in the CEP intermediate storage facility will be distributed over multiple disks and will be accessed by multiple nodes concurrently. Reordering tens of terabytes of data takes too much time and should be avoided. Therefore the data should be distributed such that the various applications (e.g., calibration and imaging) can operate well without reordering. The distribution should be such that large chunks of data can be processed locally and only small amounts of data need to be sent to other machines. There are a few axes along which the data may be distributed:

Time is probably not a good candidate, because a time slot contains a lot of data (up to 0.7 Gbytes during initial operation). This may lead to problems in the on-line system when all data of a time slot are sent to a single machine and written there. Another problem is that parallelization of imaging gets hard because the data of all time slots have to be combined.

Baseline seems a better candidate, but will lead to imaging problems. This is because a single image needs data from different machines, so large amounts of gridded or FFT-ed data have to be sent around.

Frequency seems to be the best candidate. Creating an image is usually done per channel or for a few channels, so in principle the whole imaging process can be done locally. It will result in an image cube distributed over many machines, so the image display and analysis software have to be able to handle this. The image cube can be very large (e.g., 256 Gbytes for 1000 channels of 4000 $\times$ 4000 pixels for the 4 Stokes parameters). Distribution in frequency means that, e.g., each subband is stored on a separate machine. If needed, each subband can be distributed further. Of course, each machine should contain about the same amount of data to get good load balancing.
Note that this distribution matches well with the way the correlator and on-line system is designed.

The BBS calibration software is not dependent on a specific distribution, so in the future other distributions can be used when applicable. However, it has not been evaluated yet if that is also true for the imaging software.

2.1.3 Scalable Architecture

One important requirement is scalability. In order to avoid any performance bottlenecks, unnecessary coupling between the different computing nodes should be avoided as much as possible. When distributing data over frequency, we can almost completely decouple the computing nodes, as we saw in the previous section. Another way to reduce coupling is to make communication indirect as well. Computing nodes should communicate through some kind of global shared memory. There are several architectural patterns that describe this approach. One of the oldest and best known is the Blackboard pattern, which we will describe briefly below.

Computing nodes should communicate through some kind of global shared memory. One obvious candidate for such shared memory is a database system. It provides locking and notification (trigger) mechanisms, and sometimes even command queuing. We have to be careful, though, that the database will not become a bottleneck.

2.2 Blackboard Pattern

The idea behind the Blackboard architecture is a collection of independent processes that work cooperatively on a common data structure. Each program is specialized in handling a particular part of the overall task, and all programs work together on the solution. These specialized programs work independent of each other. They do not call each other, nor is there a predetermined sequence for their activation. Instead, the direction taken by the system is mainly determined by the current state of progress. A central control component evaluates the current state of processing and coordinates the specialized programs. This data-directed control regime makes experimentation with different algorithms possible, and allows experimentally derived heuristics to control processing. This architecture is described in [3] and [4].

The Blackboard architecture is ideal for solving problems for which no predetermined algorithm or solve strategy is known. However, for the design of the BBS system, we've come to the conclusion that the operational system will benefit in terms of performance when using a predefined solving strategy. The ”best” algorithm to perform a self-calibration run can be chosen from a relatively short list of calibration strategies in advance (based, e.g., on heuristics, or suggested by research done with the MeqTree system). In fact, the Shared Repository pattern [5], which can be seen as a generalization of the Blackboard pattern, is probably a better match for the BBS system. It realizes indirect communication using a repository as shared memory. Components put their output into memory that is accessible by other components and retrieve their input from this shared memory (also called a repository). Figure 1 shows the specialization hierarchy of patterns based on the Shared Repository pattern.

Controller pattern introduces a control component in the system, which rules the system and schedules activation of other components. This pattern can be applied to deterministic problems where sequences of components activation can be determined off-line and coded in the controller using various techniques.

Repository Manager pattern is applicable in a distributed environment. It introduces a repository manager which sends notification of data creation or modification to the software components.

Blackboard pattern refines the Controller pattern to deal with non-deterministic problems.

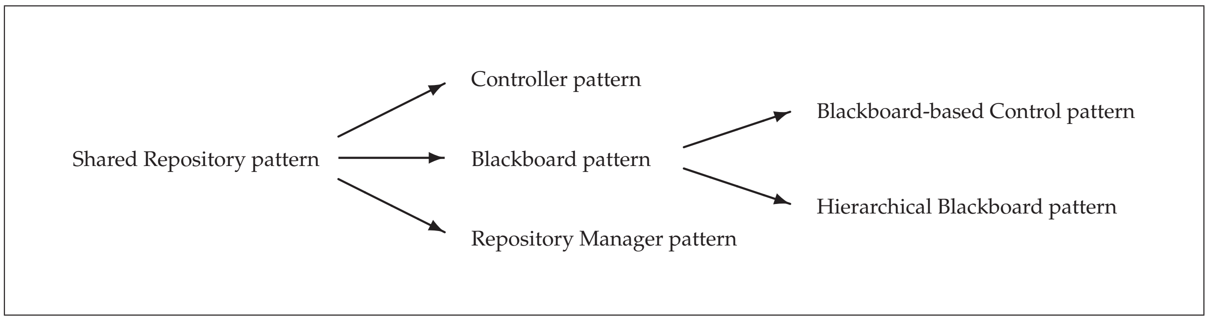
Shared Repository pattern
Controller pattern
Blackboard pattern
Repository Manager pattern
Blackboard-based Control pattern
Hierarchical Blackboard pattern

Figure 1: Patterns based on the Shared Repository pattern, after Lalanda [5].

For BBS, we will need a global controller, which could be implemented using the Controller pattern; and a notification or trigger mechanism to inform the computing nodes of changes to the shared memory, which could be implemented using the Repository Manager pattern. The shared memory is used as the common knowledge base for the self-calibration process, and will be implemented as a database. Using a database system has the advantage that locking, notification (trigger) and sometimes even command queuing mechanisms are provided out-of-the-box.

The database will be separated into two parts. One part, the Command Queue, will contain a list of commands (or work orders) to be sent to each computing node. The other part, the Parameter Database, will contain the values and quality of the (intermediate) solutions computed by each node. The database can be used as a source for assessments of the solutions with external tools.

3 System Overview

3.1 Subsystems

BBS is split into three parts. BBS Control takes care of the distributed processing by means of the Blackboard pattern. BBS Kernel does the actual processing; it executes a series of steps, where each step consists of an operation like SOLVE or CORRECT. BBS Database consists of two databases: the Command Queue and the Parameter Database that together constitute the blackboard.

3.1.1 BBS Control

The BBS Control subsystem is responsible for controlling the execution of a self calibration strategy. A strategy consists of an ordered list of commands, which will be executed by the BBS Kernel subsystem.

The key idea is that a subset of the data (the so-called *work domain*) is kept in memory; as many commands as possible are executed on these data before the next data chunk is accessed. A strategy defines the size of the work domain (in time and frequency) and optionally which stations and correlations are contained in the work domain. It is also possible to define an integration interval in time and frequency to achieve that, say, a longer time interval can be used. The basic concept is that on each machine the data contained in the work domain have to fit in memory. The BBS Kernel iterates over the work domains to process all the data. For each strategy a number of steps can be defined. For instance, when peeling 10 Cat I sources, at least 30 steps can be defined. For each source, step 1 is solving for the gain in the direction of the source, step 2 is subtracting the source, and step 3 shifts to the next source. Note that only after the last subtraction the residual data need to be written. In this way the data are read and/or written only once per strategy.

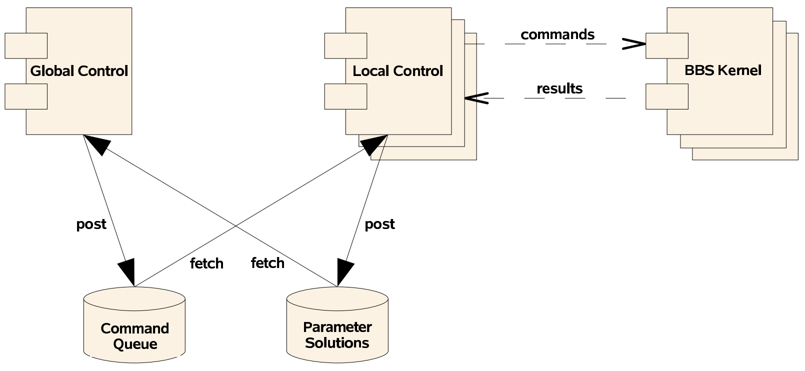

Figure 2: Global design of the BBS Control system. Global Control posts commands to the Command Queue. These commands are asynchronously retrieved by Local Control and forwarded to the BBS Kernel. After execution of the command, BBS Kernel returns the result to Local Control, which in turn posts it to the Parameter Database. Global Control checks the quality of these solutions and takes appropriate action.

The calibration process is controlled by the BBS Control subsystem. Figure 2 depicts the general control structure. The BBS Control subsystem consists of one global controller, which acts as the main process, and multiple local controllers, each controlling one BBS Kernel subsystem. The global controller posts one or more commands (steps) to the Command Queue. Each local controller fetches the next command from the Command Queue and forwards the command to the BBS Kernel subsystem. The kernel returns parameter solutions and their quality metrics to the local controller, which, in turn, posts the results to the Parameter Solutions database. The global controller inspects the results and decides which action should be taken next.

Since all communication takes place via the Blackboard, there is no need for a direct connection between the BBS Control and the BBS Kernel subsystems. The Blackboard contains all the relevant information about the current state of the self calibration process. This information can be used by other (external) processes to monitor the calibration process and to plot results. See [4] for more details on the Blackboard architecture and roles of the controller.

3.1.2 BBS Kernel

The BBS Kernel is responsible for the number crunching inherent in calibration. It implements the *measurement equation* [6, 7], or ME, which models the response of an interferometer given a description of the sky, the environment, and the interferometer. Based on the ME the Kernel can predict visibilities, subtract sources, correct visibilities for a given reference position, and solve for model parameters.

The BBS Kernel subsystem can be split up conceptually into two components: the *kernel* and the *solver*. The kernel was previously known as the Prediffer (from Predict – Differentiate). However, the Prediffer performs much more tasks than predict and differentiate, which is why it has been renamed kernel.

The kernel is controlled by the local controller, and ultimately by the global controller. The solver runs as a separate process and communicates with the local controller/kernel via (unix domain) sockets. The kernel and the solver cooperate whenever a SOLVE operation has to be performed.

3.1.3 BBS Database

The BBS Database (or blackboard) actually consists of two different databases.

Command Queue stores the commands to be executed by the BBS Kernel and the status results returned by each kernel. In principle, commands are executed in the order they were posted by Global Control. However, in the future, we may need a way to send *out-of-band* commands, which could be implemented as a high priority command. This has not been fully decided yet.

Parameter Database stores (intermediate) solutions of the model parameters calculated by the BBS Kernel. Access to the Parameter Database is minimized in order to avoid performance penalties. If partial solutions can be kept in memory of a local (kernel) node, they will not be written to the database, unless requested explicitly.

3.2 Interfaces

3.2.1 Context Diagram

The BlackBoard Selfcal system interfaces with several other components. The context of BBS is shown in figure 3.

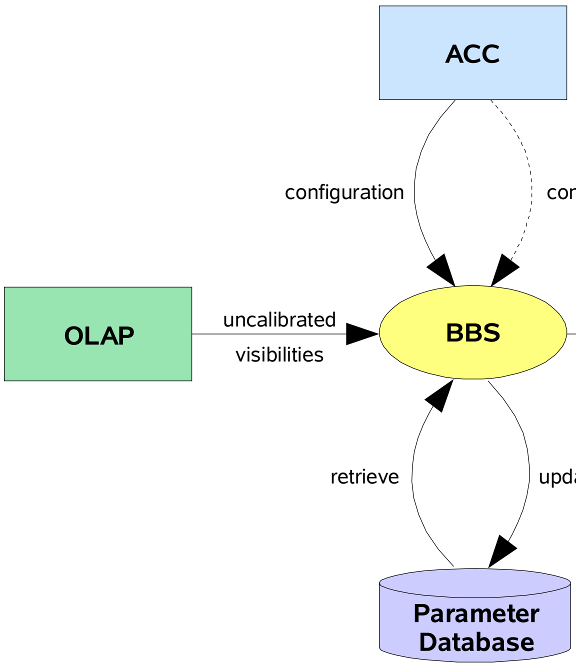

Figure 3: Context diagram for the BlackBoard Selfcal system. OLAP supplies the input data, which are stored as a Measurement Set. ACC configures and controls BBS. The Parameter Database is both read and updated by BBS.

ACC configures and controls BBS. Configuration is done using a so-called *parset* file, which is generated by ACC prior to starting the BBS applications. Each BBS application reads, during initialization, its *parset* file, containing a number of key-value pairs that set several run-time configuration parameters of the applications. All BBS applications implement the control interface that is provided by ACC [8], which enables ACC to control these applications.

OLAP stores the observational data as visibilities into one or more Measurement Sets. In section 2.1.2 we argued that the visibility data could probably best be distributed along the frequency axis; this also matches the way the data are produced by the correlator [9]. So, in the current design, we assume that each BBS Kernel will process one or more subbands of data.

Imager will Fourier transform the residual visibilities produced by BBS into an image of the sky.

Parameter Database will store the parameters of the various (sub)models used in self calibration. Examples of such (sub)models are: local sky model, minimal ionospheric model, and instrument model. The values of the model parameters are estimated by the BBS Kernel.

3.2.2 BBS Control

In this section we will briefly describe the most important interfaces that are provided or implemented by the BBS Control subsystem.

ACC provides the Process Control interface [8] that will be implemented by each executable in the BBS Control subsystem. It provides commands like `define`, `init`, `run`, and `quit`. Furthermore, ACC provides each executable with a so-called *parset* file, which contains important configuration parameters. See appendix A for a complete list of all key-value pairs that are defined for the BBS applications.

BBS Strategy describes the strategy to be used for a self calibration run. Configurable strategy parameters are read from the *parset* file that is supplied by ACC. A strategy consists of one or more steps.

BBS Step describes a (single or multi) step to be executed. Configurable step parameters are read from the *parset* file that is supplied by ACC.

Command Queue contains the queue of commands to be executed by the BBS Kernel. Commands are posted by Global Control and retrieved by each Local Control. Commands are usually executed in the order in which they appear in the queue, unless a particular command is marked *high priority* (TBD). A command can be a single step (a manageable piece of work that is forwarded to the kernel), or a control message like `initialize`.

3.2.3 BBS Kernel

In this section we will briefly describe the most important interfaces that are provided or implemented by the BBS Kernel subsystem.

Operations BBS Kernel supports several operations, which are described in section 4.4. Before an operation is executed, the context of the operation can be set. The context determines which part of the input data is to be processed, and can contain operation specific options. The local controller will request BBS Kernel to perform operations based on the commands it reads from the Command Queue.

Measurement set Measurement Sets are used to exchange visibility data with OLAP on the input side and the imager on the output side.

Parameter Database BBS Kernel reads the current values of the model parameters from the Parameter Database when required, e.g. to compute the response of an interferometer. Also, after estimation of new model parameter values it writes the updated parameters to the Parameter Database.

3.2.4 BBS Database

The current BBS Database is implemented as a relational database. The database management system implements the SQL language and provides a number of useful extensions. The interface to BBS Control and BBS Kernel is implemented as a set of stored procedures, that hide the underlying implementation.

The BBS Database subsystem implements the following interfaces:

Command Queue A set of stored procedures is provided to e.g. place commands in the queue, retrieve commands from the queue, and to report and inspect results.

Parameter Database A set of stored procedures is provided to retrieve and store the values of parameters.

4 Software Design

4.1 Terminology

4.1.1 Domains

A central concept in BBS is that of a *domain*: A 2-D rectangular region in *frequency* and *time*. In table 1 we define seven different domain types that are useful for discussing the design of BBS. Figure 5 illustrates the different domain types and how they relate to each other.

Name	**Description**
`data domain`	The domain covered by a single observation
`local data domain`	Part of the data domain that is local to (i.e. stored at) a given compute node
`work domain`	Part of the data domain which is processed together
`local work domain`	Intersection of the work domain and the local data domain (should fit in the main memory of a node)
`solve domain`	Part of the data domain that is used to solve for a set of unknowns
`local solve domain`	A solve domain that *intersects* the local data domain
`validity domain`	Domain on which a *funklet* is considered valid

Table 1: An overview of the domain types used in BBS.

4.1.2 Models, Parameters, Funklets, and Coefficients

Self calibration revolves around fitting a *model* to the observed data by adjusting the model parameters. On the coarsest scale a model for the calibration of LOFAR describes the instrument, the environment, and the sky. This model can be decomposed into smaller (sub)models, such as a model for the beamshape, the bandpass, or the ionosphere (see also [10, sec.2]).

In general, a *model parameter* can be a constant or a continuous function of one or more variables such as *frequency*, *time*, and *direction*. The value of a parameter is represented by a set of *funklets*. A funklet is an approximation of the value of a parameter on a bounded domain, termed *validity domain*. Figure 4 illustrates the relation between parameters and funklets. A commonly used type of funklet is a polynomial of arbitrary degree in *frequency* and/or *time*. Other possibilities include e.g. Fourier series expansion, shapelets, and splines.

Validity domains and solve domains are related concepts. The validity domain of a funklet equals the solve domain used to fit the coefficients of the funklet. For example, suppose we try to fit a parameter `gain:11:phase:CS10`[1] using 0^{th}-degree polynomials and a solve domain size of a single channel width times 5 integration periods. The solve domain size defines a grid of solve domains on the work domain (see figure 5). For each solve domain a funklet exists with a matching validity domain that represents the value of parameter `gain:11:phase:CS10` on that domain. To fit the coefficients of a funklet we need the values of other parameters as well. The validity domains of the funklets that represent the additional parameters will most likely be different, because they were

[1] The phase of element g_{11} of the 2×2 diagonal G-jones matrix of station CS10.

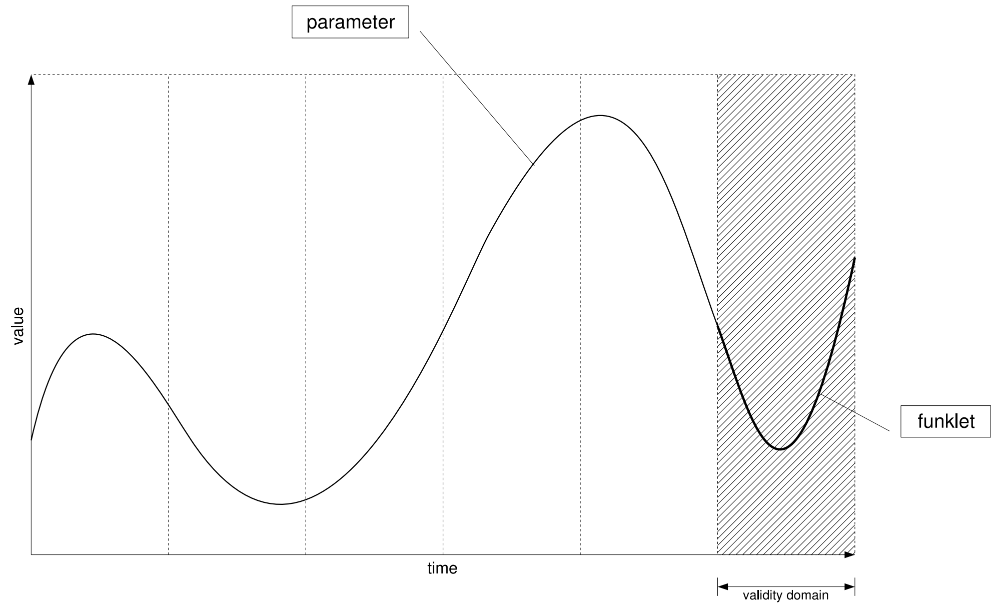

Figure 4: This figure shows a set of funklets with associated validity domains that together represent the value of a parameter on a larger domain.

fitted at some earlier time using a potentially different solve domain size. In summary: The validity domain of a funklet determines in what region the value of the funklet is considered valid; A solve domain determines which part of the observed data is used to fit the coefficients of a funklet.

The problem of fitting a parameter can now be restated as fitting the coefficients of a *set* of funklets defined on a set of non-overlapping solve domains.

An important consequence of using funklets is that one implicitly assumes that the 'true' parameter value can be approximated well by a set of funklets of the selected type. For example, consider polynomial funklets. Given a polynomial degree n and irrespective of the size of the validity domain, one implicitly assumes that the parameter value can be approximated well by a n^{th}-degree polynomial on the validity domain. To make this assumption more explicit, it may be useful to interpret the n^{th}-degree polynomial as part of the model, instead of as an integral part of parameter handling.

4.2 Distribution

The unit of work for most operations (i.e. PREDICT, SUBTRACT, CORRECT, SHIFT) is a single complex visibility. These operations are embarrassingly parallel: communication between nodes is not necessary.

The only operation that is not guaranteed to be embarrassingly parallel is the SOLVE operation. The unit of work for this operation is a single solve domain. As long as every solve domain is completely contained within the local data domain of a node, the SOLVE operation can be performed locally. However, if a solve domain intersects the local data domains of multiple nodes, communication between these nodes becomes necessary.

4.2.1 Data Distribution

Figure 5 shows how the data is typically distributed over the compute nodes. The visibility data computed by the correlator is distributed according to frequency, as is shown in the figure. Visibility data may be unavailable, because the subbands recorded during an observation are not required to be consecutive. All *available* data is distributed equally over the nodes.

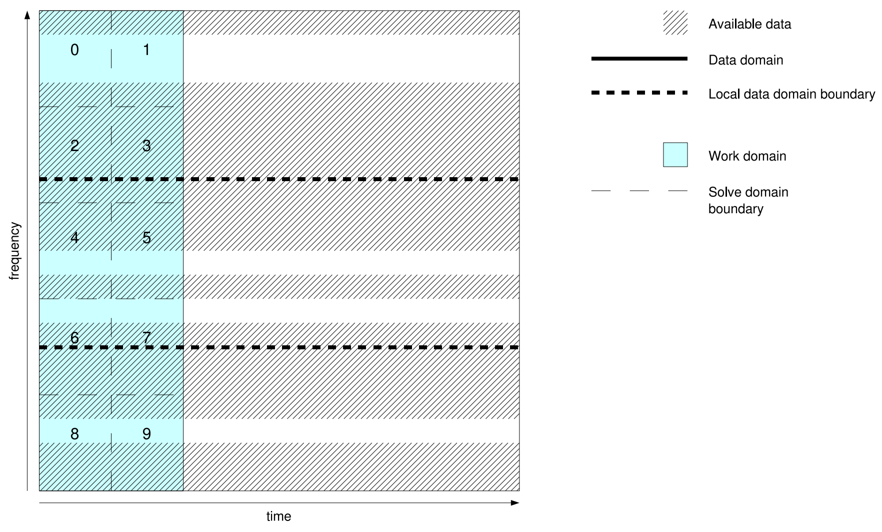

Figure 5: Data distribution and partitioning of the work domain into solve domains during a SOLVE operation. See Table 1 for the definition of the different domain types.

The size of the work domain is specified by the user. It should be chosen such that each node can read its local work domain into main memory. Thus, all steps of the calibration strategy can be executed on the work domain without having to read data more than once. After the work domain has been processed, the global controller instructs the local controllers to continue with the next chunk of data.

During execution of a SOLVE operation, the work domain is partitioned into solve domains based on a user specified solve domain size. Figure 5 shows an example where the work domain is partitioned into ten separate solve domains. Six solve domains are entirely local: solve domains 0, 1, 4, 5, 8, and 9. The other solve domains span multiple nodes.

Solve domain size Because the local work domain is determined by the intersection of the work domain and the local data domain, and the visibility data is distributed in frequency, the work domain has to span the *entire* frequency axis to keep all the nodes busy. This implies that the largest possible size of the local work domain *in time* is more limited than necessary.

The largest possible *area* (channels × integration periods) of the local work domain is (assuming 77 stations, 8GB of local memory):

$$area = \frac{8 \cdot 2^{30}\ bytes}{\frac{1}{2} \times (77 \times 76)\ interf. \times 4\ correlations \times 8\ bytes/correlation} \approx 85441$$

However, due to the way the work domain is defined, the largest possible size in time is limited to the area divided by the number of channels stored per node. Assuming 32000 channels and a cluster of 200 nodes:

$$time = \frac{85441}{32000\ channels \div 500\ nodes} \approx 534\ integration\ periods$$

Assuming an integration period of 1 second, 534 integration periods correspond to approximately 9 minutes. It is currently an open question if this limit is within requirements. This will depend on the type of SOLVE operations the user wants to perform.

Solve domain truncation If the size of the work domain along any dimension is not an integer multiple of the specified solve domain size, the solve domains at the work domain boundary will get truncated.

It is currently an open question if solve domain truncation is acceptable or not. In principle, the work domain size could be adjusted automatically to avoid solve domain truncation. This is not completely trivial, because a strategy can contain multiple solve steps each of which can have a different solve domain size. Because *all* steps of a strategy are executed on the work domain, the size of the work domain must be an integer multiple of the size of each of the specified solve domain sizes to completely avoid truncation. Assuming the size of the work domain is specified in, or converted to, integers (or fractional numbers), e.g. number of channels times number of integration periods, the size of the work domain should be set to a multiple of the least common multiple of the different solve domain sizes.

4.2.2 Communication

Figure 6 shows the communication paths used in BBS. The blackboard is essentially used as *shared memory* that retains the state of the (distributed) calibration process. Shared memory allows decoupling of the global controller and the local controllers, and easy monitoring by external tools.

A potential risk of this design is that the shared memory becomes a performance bottleneck. All local controllers need to access the shared memory and this access will typically be (quasi-) concurrent, because all local controllers perform the same operations on approximately equal amounts of data. A related concern, raised in [1, p.19], is that local controllers may have to poll the shared memory to check for updates. This depends on the implementation of the shared memory component. Most modern databases, for instance, support asynchronous notification of client processes thus avoiding the need for polling. Performance tests should be carried out in the near future to asses if the design meets the requirements.

Figure 6 shows that there is a direct link between the local controller and the solver. One could wonder why the communication is not routed via shared memory. There are two reasons for this. First, exchanging information between local controllers and solvers via shared memory imposes unnecessary synchronization. To fit the funklets defined on a given solve domain, information about that solve domains needs only to be exchanged between the local controller(s) *that share the solve domain* and a solver. Second, the amount of data that needs to be communicated from a local controller to a solver is considerable and encoded in a form that is unsuitable for monitoring.

The updated coefficients communicated from a solver to the relevant local controllers *are* suitable for monitoring.

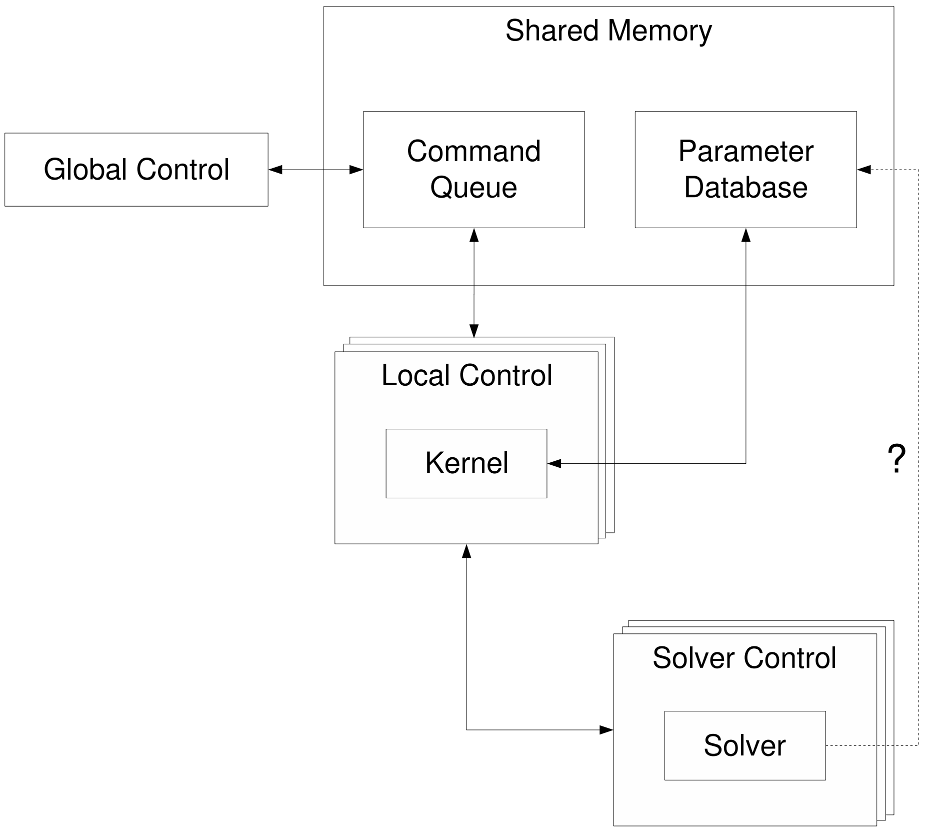

Figure 6: Overview of the communication paths used in BBS.

Of course, using direct communication between kernel and solver does not prohibit writing the updated coefficients to shared memory as well. Assuming monitoring coefficient values during iteration is primarily used for debugging purposes, this is probably acceptable.

At the start of a SOLVE operation a solver needs the initial values of the funklet coefficients for the solve domains it will process. At the end of the SOLVE operations the updated coefficient values have to be written to shared memory. (If monitoring coefficient values during iteration is required, the updated coefficients need to be written after each iteration as well.)

In the current design the local controller sends the initial coefficient values to the solver and writes the updated coefficients to shared memory. Thus, the design of the solver component can be kept simple. Yet it also implies that the updated coefficient values of solve domains that are shared between multiple nodes will be written multiple times. This is not a problem from a data consistency point of view, but may aversely affect performance. Multiple writes could be avoided if the solver is made responsible for updating the shared memory.

4.3 BBS Control

4.3.1 BBS Strategy

One iteration in the so-called *Major Cycle* [10, sec. 4.1] can be described by a BBS Strategy. A strategy defines a relationship between the data set of a given observation, which is stored in a Measurement Set [11], and the Parameter Database holding (intermediate) values of the model parameters that will be estimated as part of the self calibration process. At least two models are used in the current self calibration setup: the Local Sky Model (LSM) and the Instrument Model. The Data Selection associated with a BBS Strategy defines the selection of the observed data that will be used for the complete strategy. It allows one to specify, for example, which frequency bands, time intervals, and baselines should be used during this self calibration run. A strategy is defined in terms of one or more BBS Steps (see section 4.3.2 below).

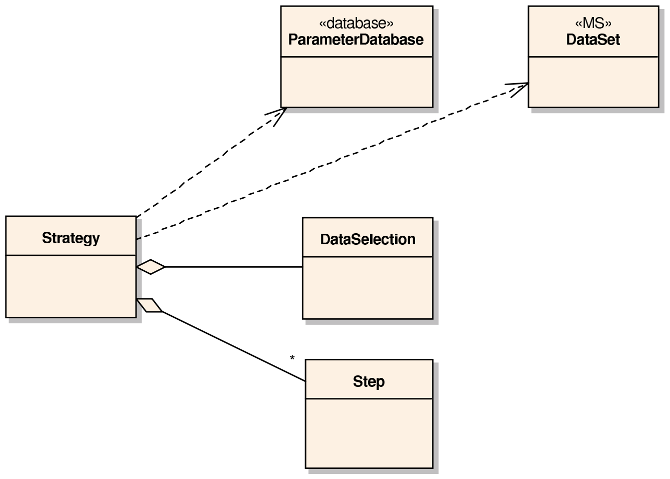

Figure 7: The BBS Strategy class defines the strategy to be used for the current self calibration run.

4.3.2 BBS Step

The BBS Step class is designed as a Composite pattern [12], which means that each BBS Step can itself be made up of one or more BBS Steps. The Composite pattern provides an easy way to define a tree-like structure. Leaf classes, like SolveStep cannot be further subdivided; they describe one single piece of work that can be handed over to the BBS Kernel. Currently, there is a total of seven leaf classes, each defining one single piece of work.

4.3.3 Global Control

BBS Global Control is reponsible for managing the execution of a self calibration run. The program flow is shown in the activity diagram (see figure 9).

Main Flow At start-up, Global Control reads the *parset* file that was supplied by ACC, and initializes itself. Next, it queries the Command Queue database to see if it is starting a new run, or recovering from an ”aborted”

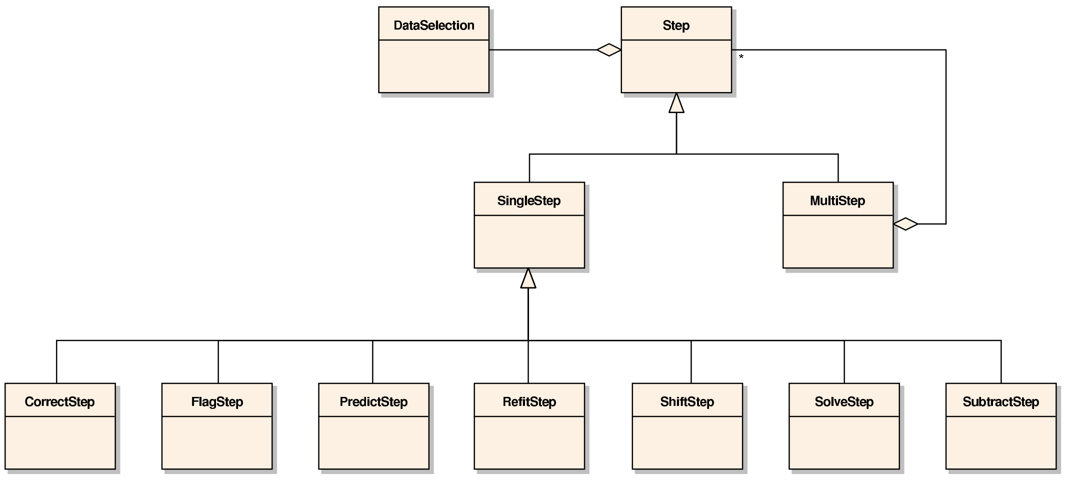

Figure 8: The BBS Step class family defines single pieces of work that can be executed by the BBS Kernel as part of the current self calibration run.

run. If it started a new run, it starts by posting the Strategy to the Command Queue, followed by an *initialize* command, which is needed to inform the Local Controllers that a Strategy is available now. Next, it posts a *next chunk* command, indicating that the whole sequence of steps (as represented by a strategy) should be repeated for the next chunk of data. The size of a data chunk is determined by the work domain size.

Global Control now enters a loop, posting steps to the Command Queue, until either there are no more steps left in the strategy, or the step posted last is a synchronization point. In the former case, it will send a *next chunk* command an re-execute the loop. In the latter case, it will wait until all Local Controllers have finished processing all steps posted up to now, before sending the next step. Iteration over a chunk of data ends when there are no more steps left in the Command Queue.

Alternative Flows If all Local Controllers return an `OUT_OF_DATA` result to the *next chunk* command, then the self calibration run is completed; A *finalize* command is sent to inform all Local Controllers and set the *done* flag for the current strategy.

If Global Control is recovering from an ”aborted” run (possibly due to a crash of itself), it sends a high priority *recover* command and checks if all Local Controllers respond to it. Next, Global Control queries the Command Queue database to get the last step in the Command Queue and resumes operation.

Reminders

- Determine if a step is a synchronization point or not.
- Strategy needs a ”done” flag, which must be set by the Global Controller.
- Strategy needs a unique identifier which must be supplied by SAS.
- Step needs a sequence number
- Check result code (e.g., `OUT_OF_DATA`) of ”next chuck” command

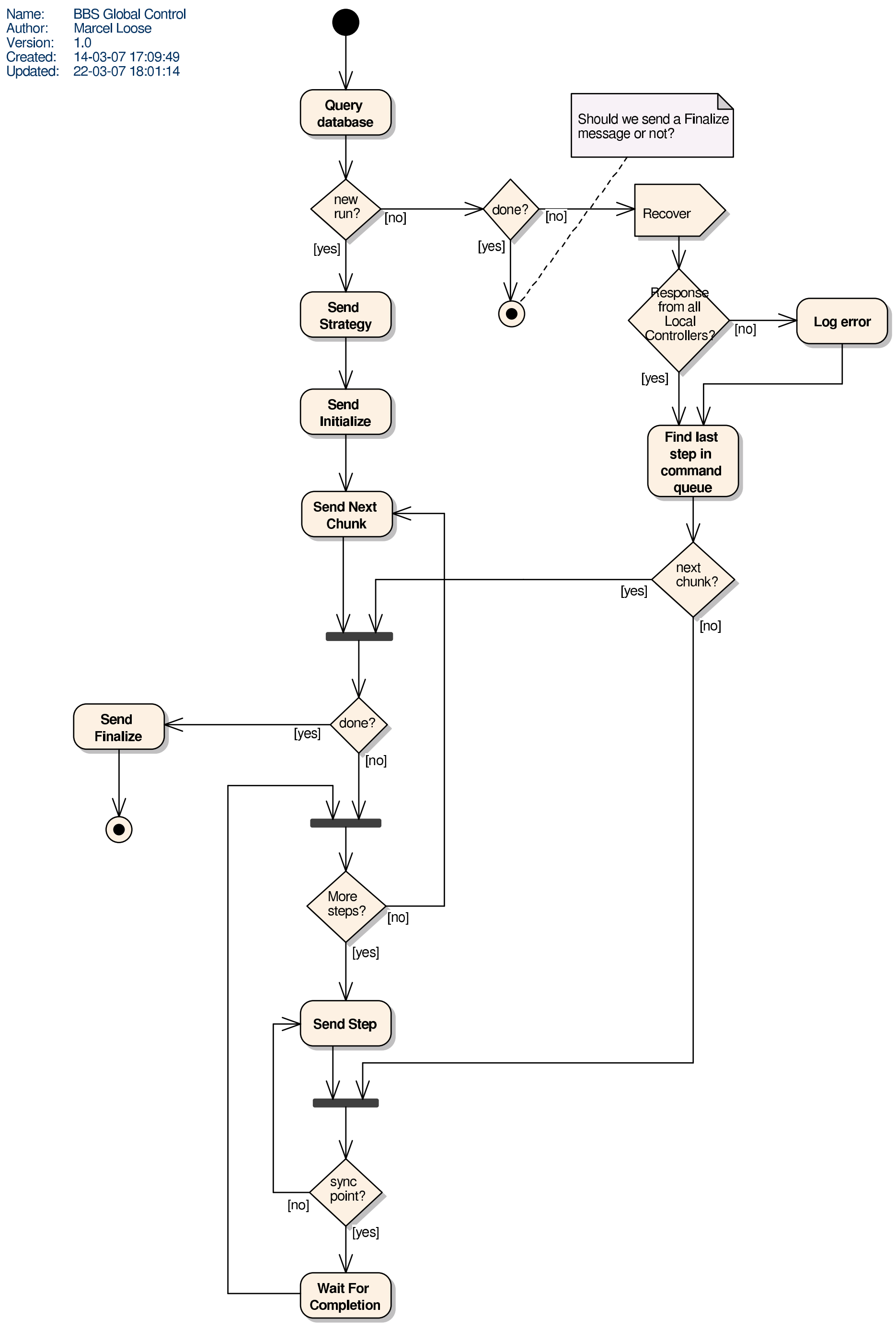

Figure 9: Activity diagram of the BBS Global Control

- Once a cluster node is out of data it will respond with `OUT_OF_DATA` on all subsequent "next step" or "next chunk" commands.
- SAS should provide the number of Local Control nodes.

4.3.4 Local Control

BBS Local Control is responsible for managing the processing of one command (e.g., a BBS SingleStep) by the BBS Kernel. The program flow is shown in the activity diagram (see figure 10).

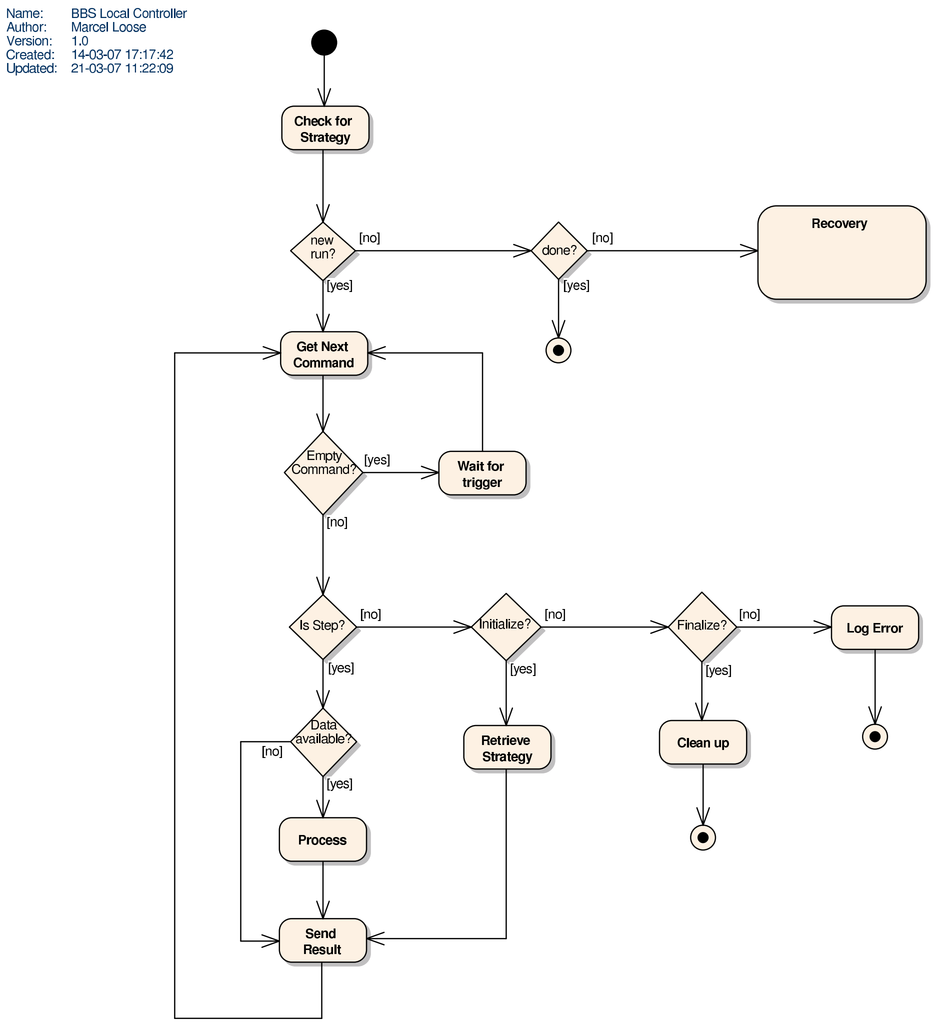

Figure 10: Activity diagram of the BBS Local Control

Main Flow At start-up, Local Control reads the *parset* file that was supplied by ACC, and initializes itself. Next, it queries the Command Queue database in order to find out whether it is starting a new calibration run, or recovering from an "aborted" run. The logic to derive whether a new run is started is somewhat complicated. Here it is for completeness:

> If a strategy with the (by SAS) given ID is not yet present in the database, then this is a new run; else if the strategy is present and its *done* flag is set, then we're done; else if the next command in the Command Queue is *initialize*, or if there are no commands at all, then this is a new run; else we're recovering from, e.g., a crash.

If it is a new run, Local Control enters the main loop. It tries to retrieve a command from the Command Queue database. If there are no commands present in the queue, it will wait for a trigger from the database to retrieve the next command. Currently, three different commands can be handled by the Local Controller: *initialize*, *finalize*, and any of the BBS SingleSteps. If the command is a BBS SingleStep, it will be forwarded to the BBS Kernel, which will process it. The result returned by the kernel will be posted to the result table of the Command Queue.

Alternative Flows If Local Control is not starting a new run, it might be recovering from a crash. Recovery is not yet completely modeled. However, care has been taken to ensure that the information stored in shared memory is sufficient to reconstruct the state of a local controller.

If the received command is *initialize*, a Strategy with the given ID will be retrieved from the Command Queue. It is an error if this strategy is not present in the Command Queue database.

If the received command is *finalize*, Local Control will clean up and exit.

4.4 BBS Kernel

BBS Kernel subsystem is responsible for the number crunching inherent in calibration. It implements the *measurement equation* [6, 7], or ME, which models the response of an interferometer given a description of the sky, the environment, and the interferometer. Based on the ME it can predict visibilities, subtract sources, correct visibilities for a given reference position, and solve for model parameters.

The BBS Kernel subsystem can conceptually be split into two separate components: the *kernel* and the *solver*. The solver is used to estimate new parameter values. It is discussed separately in section 4.4.3.

The kernel (component) supports the following operations:

- PREDICT
  Predict (simulate) visibilities based on a model that describes the sky, the environment (e.g. ionosphere), and the instrument.
- SUBTRACT
  Predict visibilities for one or more source(s) and subtract the result from the observed visibilities.
- CORRECT
  Correct the observed visibilities for a given reference (source) direction.
- GENERATE EQUATIONS
  Generate condition equations in a form that can be fed to the solver. Condition equations relate the model parameters to the difference between the observed visibilities and the predicted visibilities (based on the model).
- SHIFT
  Phase shift the observed visibilities to a different phase center.
- FLAG
  Flag visibilities that have been contaminated by, e.g., radio frequency interference.

- REFIT
  Refit funklet coefficients, e.g. in a multi-resolution approach.

Some of the operations listed above can be subdivided from the kernel's point of view. The SUBTRACT operation consists of predicting visibilities and subtracting them from the observed visibilities. The GENERATE EQUATIONS operation consists of predicting visibilities and partial derivatives, subtracting the predicted visibilities from the observed visibilities, and generating condition equations in the parameters based on the computed differences and partials. However, seen from outside the kernel, the operations listed above behave as 'atomic' operations.

Note that the SOLVE operation does not appear in the list. Remeber that the local controller orchestrates the execution of the commands it retrieves from the Command Queue. All commands except SOLVE can be executed by the kernel alone. However, the SOLVE command involves both the kernel *and the solver*. Each iteration, the local controller instruct the kernel to perform the GENERATE EQUATIONS operation. The result is sent to the solver, which responds with updated parameter values. Basically, the GENERATE EQUATIONS operation constitutes half of the SOLVE command.

Before an operation is executed, the *context* of the operation can be set. The context contains information about which data needs to be processed (data selection), and operation specific options (e.g. the parameters to be fitted in case of a GENERATE EQUATIONS request).

4.4.1 Measurement Equation Evaluation

The Measurement Equation (ME) is a parameterized model of an interferometer in terms of a model of the sky, environmental effects (e.g. ionosphere), and antenna based instrumental effects [6, 7, 13].

The measurement equation is represented in BBS as a directed acyclic graph, although it is commonly referred to as *tree*. The nodes of the tree represent atomic expressions, while the branches represent dependencies between atomic expressions. Compound expressions can be build by combining multiple nodes into a (sub)tree. The leaf nodes represent the parameters of the expression, e.g. source positions and Stokes vectors. As an example, figure 11 shows a subtree for computing the visibilities produced by two linearly polarized point sources at different positions.

Specification by the user is limited to selecting predefined subtrees to include in the model. Currently implemented subtrees are:

- Prediction of a (possibly polarized) point source
- Complex gain per station (G-jones)
- Complex gain in the direction of a source, per station (E-jones)
- Bandpass (B-jones)

BBS will automatically generate a complete expression tree based on the selected predefined subtrees. For example, it will instantiate a subtree for each selected source in the local sky model and will include these at the appropriate place in the ME tree for each selected interferometer (baseline).

In [10, sec. 2][13] an overview of the various models, such as beamshape, bandpass, and ionosphere is provided.

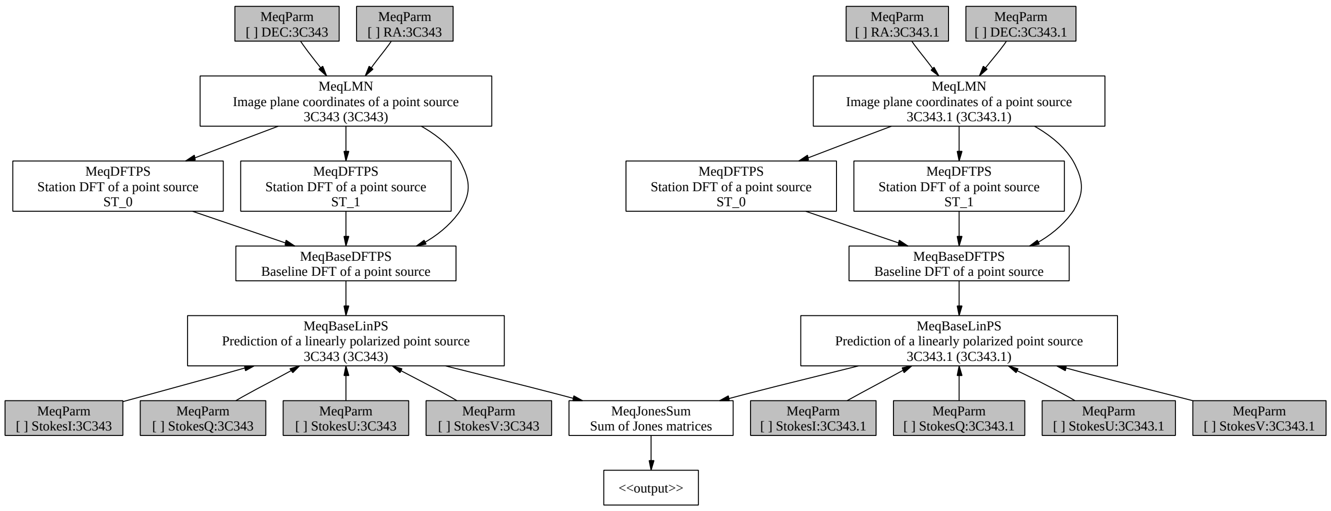
MeqParm [ ] DEC:3C343
MeqParm [ ] RA:3C343
MeqLMN Image plane coordinates of a point source 3C343 (3C343)
MeqDFTPS Station DFT of a point source ST_0
MeqDFTPS Station DFT of a point source ST_1
MeqBaseDFTPS Baseline DFT of a point source
MeqBaseLinPS Prediction of a linearly polarized point source 3C343 (3C343)
MeqParm [ ] StokesI:3C343
MeqParm [ ] StokesQ:3C343
MeqParm [ ] StokesU:3C343
MeqParm [ ] StokesV:3C343
MeqJonesSum Sum of Jones matrices
<<output>>
MeqParm [ ] RA:3C343.1
MeqParm [ ] DEC:3C343.1
MeqLMN Image plane coordinates of a point source 3C343.1 (3C343.1)
MeqDFTPS Station DFT of a point source ST_0
MeqDFTPS Station DFT of a point source ST_1
MeqBaseDFTPS Baseline DFT of a point source
MeqBaseLinPS Prediction of a linearly polarized point source 3C343.1 (3C343.1)
MeqParm [ ] StokesI:3C343.1
MeqParm [ ] StokesQ:3C343.1
MeqParm [ ] StokesU:3C343.1
MeqParm [ ] StokesV:3C343.1

Figure 11: This figure shows a subtree for computing the visibilities produced by two linearly polarized point sources at different positions. The leaf nodes, i.e. the parameters of the expression, have been colored gray.

4.4.2 Model Parameter Administration

To evaluate an expression tree, the kernel needs to know the values of the model parameters (leaf nodes). Typical examples of model parameters are: source position and Stokes vector, complex station gain, ionospheric phase shift. The total number of parameters in the expression trees of the entire LOFAR array will depend on the number of sources that are included in the sky model. Several thousands of parameters seems to be a realistic estimate [13].

Naming scheme Because of the large number of parameters, keeping track of them all becomes a challenge. A naming scheme can be used to simplify this task. The naming scheme achieves two goals:

- Identification of groups of related parameters
- Specification of default values

Each parameter is assigned a unique name, which can be made up of several parts separated by colons. Grouping can be achieved with UNIX-like wildcards (*, {}), see Table 2. If the kernel cannot find an exact match when searching for a certain parameter, it will try to find a default value. Default values are specified according to the same naming scheme as parameters. However, if no default value can be found that matches the name of a parameter exactly, the kernel will strip off the last part of the name and retry. This process continues until either a match is found or the name becomes empty. Thus, one can specify, for example, a default flux for every source by specifying `StokesI`. If the kernel needs the value of `StokesI:3C343` but that parameter does not exist as a regular parameter, it will try to find a default value called `StokesI:3C343`. If that default value also does not exist, it will strip off the last part of the name and search for a default value called `StokesI`, which does exist in our example.

Reusing parameter values It often makes sense to reuse the parameter values determined during calibration of an observation A for calibration of an observation B. For instance, to provide good initial values for the calibration

Identifier	Referenced parameter(s)
`ra:3C343`	Right ascension of source 3C343
`dec:3C343`	Declination of source 3C343
`gain:11:phase:CS10:3C343.1`	Phase of the g_{11} element of the G-jones matrix of station CS10 in the direction of source 3C343.1
`gain:11:phase:*:3C343.1`	Phase of the g_{11} element of the G-jones matrix of every station in the direction of source 3C343.1
`gain:{11,22}:phase:*:3C343.1`	Phase of the g_{11} and g_{22} elements of the G-jones matrix of every station in the direction of source 3C343.1.

Table 2: Examples of typical parameter names and the use of wildcards to identify a group of related parameters.

of B and hopefully save a few iterations.

Also, instead of straightforward copying of parameter values interpolation could be considered. For example, suppose we have already calibrated two observations, A and C, one taken before and one after the time at which the observation to be calibrated, B, was recorded.

Reuse of parameter values can also be applied *during* calibration of an observation, when moving from one work domain to the next. One could even consider something like the following algorithm:

1. Solve a small number of work domains spaced relatively far apart
2. Determine initial values for the intermediate work domains by interpolation
3. Solve for the intermediate work domains.

Refitting Refitting is useful in a multi-resolution approach. A typical scenario would be:

1. Solve for a parameter using funklets of a low order on relatively large solve domains to get the global trend
2. Refine the solution by using funklets of a higher order on relatively small solve domains, using the values found in 1. as a starting point

Funklets are defined on a normalized domain. The normalization depends on the validity domain. Therefore, the coefficients of funklets defined on a coarse scale cannot be copied directly to funklets defined on a fine scale. For polynomials the required transformation is easy to derive. In more complicated cases, the solver could be used to find the new values.

4.4.3 Solver

The solver is responsible for estimating values for the parameters that minimize the difference between the model and observation. It also takes care of merging information of solve domains that are shared between multiple kernels.

The solver minimizes the sum of squared differences using an implementation of the Levenberg-Marquardt algorithm. This algorithm uses partial derivatives with respect to the funklet coefficients, which are computed using forward differences:

$$\frac{\partial M}{\partial p} \approx \frac{M(p+\epsilon) - M(p)}{\epsilon}, \text{ for small } \epsilon$$

Each partial derivative that needs to be computed requires an extra evaluation of the model (to compute $M(p+\epsilon)$). Adding alternative algorithms is possible, but will involve a revision of the interface between the kernel and the solver.

Each solve domain represents an *independent* minimization problem. Therefore, the partitioning of the data domain into solve domains is strongly linked to the problem of interest. For instance, fitting station gain every *minute* is a different problem than fitting it every *hour*.

Any combination of parameters can be fitted simultaneously. However, for all parameters the same solve domain grid will be used. Therefore, it is not possible to fit, for example, station gain *per 10 minutes* and station phase *per minute* simultaneously: This would require the solve domain grid for gain to be different than that for phase. Such fitting problems could be handled by alternating between the different grids every iteration. In the example, a single iteration of the simultaneous fitting problem would consist of one iteration for gain, then one iteration for phase.

Each node will run a local solve process that will handle solve domains that are entirely local. For solve domains that span several nodes, a separate pool of global solver processes could be used or the solver process at one of the involved nodes could be designated to act as global solver.

We intend to use one implementation for both local and global solver. Communication will be performed via sockets. If performance turns out to be poor, we can always (re)integrate the local solver and the kernel. This avoids the memory and communication overhead of serializing and sending equations for entirely local solve domains over a (unix domain) socket.

4.5 BBS Database

The BBS Database actually consists of two databases: a Command Queue database and a Parameter Database. However, they will probably reside on the same node.

4.5.1 Command Queue

The Command Queue is, as the name suggests, a command queue. Commands posted by Global Control are stored inside the Command Queue. Each entry in the strategy table represents one BBS Strategy, which is associated with one or more entries in the step table, representing the BBS SingleSteps in the BBS Strategy. The Local Controllers fetch these steps from the Command Queue, one by one. When one step is completed, the status result is posted to the result table. For each single step, there are as many entries in the result table as there are Local Controllers, processing these steps. For example, suppose a Strategy consists of 10 SingleSteps and there are 20 Local Controllers. Then, when the Strategy is done, we will have $10 \times 20 = 200$ entries in the result table.

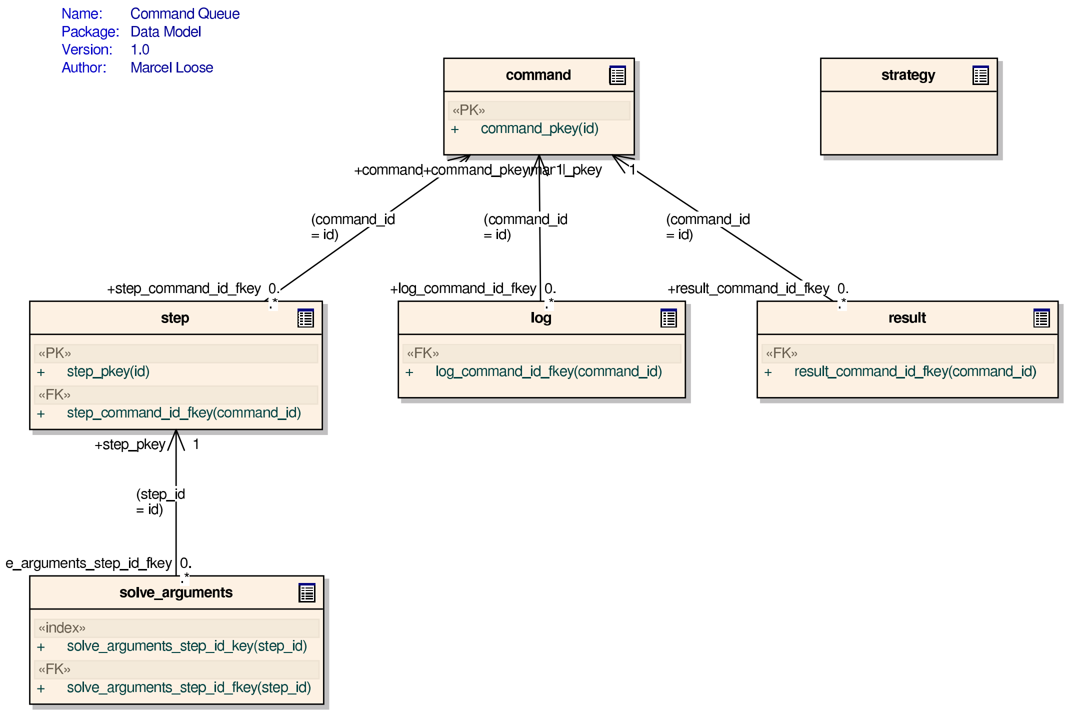
Name: Command Queue
Package: Data Model
Version: 1.0
Author: Marcel Loose
command
«PK»
+ command_pkey(id)
strategy
+command+command_pkey
+nr 1l_pkey
1
(command_id = id)
(command_id = id)
(command_id = id)
+step_command_id_fkey 0..*
+log_command_id_fkey 0..*
+result_command_id_fkey 0..*
step
«PK»
+ step_pkey(id)
«FK»
+ step_command_id_fkey(command_id)
log
«FK»
+ log_command_id_fkey(command_id)
result
«FK»
+ result_command_id_fkey(command_id)
+step_pkey 1
(step_id = id)
e_arguments_step_id_fkey 0..*
solve_arguments
«index»
+ solve_arguments_step_id_key(step_id)
«FK»
+ solve_arguments_step_id_fkey(step_id)

Figure 12: Data model of the Command Queue database

4.5.2 Parameter Database

Currently, the Parameter Database is stored as an AIPS++ table, but in the near future it will be migrated to a ”real” database. A data model has not been finalized yet. Updated parameter values are only written at the end of a solve operation (not for each iteration), in order to reduce the amount of I/O. It remains possible to store the parameter values each iteration for debugging purposses.

4.6 Performance Considerations

4.6.1 Alternative Observation File Formats

The AIPS++ MeasurementSet (MS) format [11] is currently used for input and output of visibility data. Memory mapped I/O is used to access the data in the MS. This approach is fast, because it avoids both the overhead of going via the standard MS access routines and a memcopy by the OS kernel. However, it also imposes certain constraints on the structure of the measurement sets that BBS can handle and the sequence of operations it can execute.

Avoiding the use of standard access routines makes BBS less robust, because (small) deviations from the expected structure may cause it to crash. Also, BBS may fail to report an error if an MS with an unsupported structure is fed in and just read in the wrong data records.

On the short term, we plan to replace memory mapped I/O with the standard MS access routines. This will probably incur a significant amount of overhead. Therefore, alternative (more raw) formats will be considered as well, especially for the interface between OLAP and BBS (the input side).

4.6.2 Spatial Queries

A common query to the Parameter Database involves finding all the funklets of which the validity domain intersects a given domain. The performance of such queries can be improved dramatically by using some form of spatial indices, e.g. kd-trees, quadtrees, or r-trees.

Modern databases often support efficient spatial queries out of the box. If it is decided not to use a database for parameter storage, then it still makes sense to use or implement spatial indices.

4.6.3 Parallel Computation

The compute nodes on which BBS is run will be multi-core and/or multi-CPU machines. The easiest way of using the additional computing power is to start multiple kernel processes on each node. The main advantage of this approach is that the code can remain single threaded and therefore easier to read and understand.

Multi-threading is another way to use achieve parallel computation. It requires (slightly) more complex code. First, multi-threading constructs have to be added to the code, which makes it less clear. Second, additional code is needed to ensure correct parallel execution.

For example, because multiple parents can depend on the value of a single node in the expression tree, straightforward parallel evaluation of the tree is not possible. Instead, dependency analysis is required to find groups of nodes that can be executed in parallel without explicit locking and waiting.

Process level parallelism will be the first thing to try. If it does not yield the expected speed up, multi-threading will be considered. A working multi-threaded implementation is already available, although some revision may be necessary.

Another technique that can be used in combination with either approach discussed above to improve performance is vectorization. A special code generator has been developed to assist in using vector instructions for this purpose.

4.6.4 Caching

More elaborate caching schemes may be implemented to optimally trade memory versus time in the evaluation of an expression tree. For instance, when multiple evaluations of the same tree are required (e.g. for computing partial derivatives, or when executing a SOLVE operation), the execution time of each node in the tree could be measured during the first evaluation. Given the amount of memory needed per node to cache its value, and the total amount of available memory, an optimization problem could be formulated to determine which nodes should cache their data and which should not.

4.6.5 Parameter Estimation

Performance of the SOLVE operation can be improved by using funklets of low order. Low order funklets have few coefficients, which is important for two reasons:

1. The fitting algorithm makes use of partial derivatives with respect to the funklets coefficient. For each partial derivative an extra evaluation of the model is required on the solve domain.
2. A time consuming step in the fitting algorithm is the decomposition of a matrix. The decomposition requires $O(N^3)$ work, where N equals the total number of coefficients that need to be fitted. Doubling the number of coefficients entails eight times more work.

On the other hand, a large solve domain size may be used to increase the signal to noise, or to get enough condition equations when fitting multiple parameters simultaneously. From a practical point of view, a large solve/validity domain also decreases the number of funklets that need to be stored. Storing a set of funklet coefficients per visibility will most likely require too much storage space.

Large solve domains generally require high order funklets to allow for enough variability over the domain. In short, there is a trade-off between processing time on the one hand and signal to noise, number of parameters that can be fitted simultaneously, and storage space on the other.

References

[1] Ger van Diepen and Ellen van Meijeren. Blackboard selfcal (BBS); software design document. Technical report, ASTRON, 2006. LOFAR-ASTRON-SDD-052.

[2] Kjeld van der Schaaf and Auke Latour. LOFAR central processing facility; architecture description, version 2.0. Technical report, ASTRON, 2007. LOFAR-ASTRON-ADD-012.

[3] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, Ltd., 1996.

[4] Kjeld van der Schaaf. Requirements analysis and architectural design of the blackboard selfcal application. Technical report, ASTRON, 2005. LOFAR-ASTRON-SDD-002.

[5] Philippe Lalanda. Shared repository pattern. In *Proceedings of PLoP'98 (Pattern Languages of Programs'98)*, 1998. `http://jerry.cs.uiuc.edu/~plop/plop98/final_submissions/P24.pdf`.

[6] J.P. Hamaker, J.D. Bregman, and R.J. Sault. Understanding radio polarimetry. I: Mathematical foundations. *Astronomy & Astrophysics Supplement Series*, May 1996.

[7] Jan E. Noordam. *The Measurement Equation of a Generic Radio Telescope*, February 1996. `http://aips2.nrao.edu/docs/notes/185/185.html`.

[8] Ruud Overeem. Application control and configuration; global design. Technical report, ASTRON, 2005. LOFAR-ASTRON-SDD-037.

[9] J.W. Romein. On-line application processing (OLAP); software design document, version 2.0. Technical report, ASTRON, 2007. LOFAR-ASTRON-SDD-036.

[10] Kjeld van der Schaaf and Ronald Nijboer. LOFAR calibration implementation, version 2.0. Technical report, ASTRON, 2007. LOFAR-ASTRON-SDD-050.

[11] A.J. Kemball (NRAO) and M.H. Wieringa (ATNF), eds. *MeasurementSet definition version 2.0*, January 2000. `http://aips2.nrao.edu/docs/notes/229/229.html`.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[13] Jan E. Noordam. LOFAR calibration framework. Technical report, ASTRON, 2006. LOFAR-ASTRON-ADD-015.

A Configuration Syntax

This appendix describes the syntax of the BBS configuration file.

Global Settings

DataSet : *string*
Path to the input measurement set.

BBDB : *BBDB* (see page 34)
Information about the black board database.

ParmDB : *ParmDB* (see page 35)
Information about the parameter databases (e.g. instrument model parameters, local sky model parameters).

Example

```
DataSet                 = "test.ms"          # name of Measurement Set

BBDB.Host               = "127.0.0.1"        # hostname/ipaddr of BB DBMS
BBDB.Port               = 12345              # port used by BB DBMS
BBDB.DBName             = "blackboard"       # name of the BB database
BBDB.UserName           = "postgres"         # username for accessing the DBMS
BBDB.PassWord           = ""                 # password for accessing the DBMS

ParmDB.Instrument       = "test.instrument"  # instrument parameters (MS table)
ParmDB.LocalSky         = "test.sky"         # local sky parameters (MS table)
```

Strategy

A strategy consists of one or more (multi-)steps with an associated work domain size and optional data integration.

Steps : *vector<string>*
The names of the steps that compose the strategy. It is an error to leave this field empty.

Stations : *vector<string>*
Names of the participating stations. All stations will be used if this field is left empty.

InputData : *string*
Name of the column in the measurement set that contains the input data.

Correlation : *Correlation* (see page 35)
Specifies which correlations to use.

WorkDomainSize : *DomainSize* (see page 35)
Size of the work domain in frequency and time. A work domain represents an amount of input data that is loaded into memory and processed as a single chunk.

Integration : *DomainSize* (see page 35)
Cell size for integration. Allows the user to perform operations on a lower resolution, which should be faster in most cases.

Example

```
Strategy.Steps                 = ["MultiStep", "SingleStep2"] \
                                                # (multi-)steps that compose this strategy
Strategy.Stations              = [ 0, 1, 2, 3 ] # ID's of stations to use
Strategy.InputData             = "INDATA"       # MS input data column
Strategy.Correlation.Selection = ALL            # one of AUTO, CROSS, ALL
Strategy.Correlation.Type      = ["XX", "YY"]   # which (cross)correlations to use
Strategy.WorkDomainSize.Freq   = 1e+6           # work domain size: f(Hz)
Strategy.WorkDomainSize.Time   = 10             # work domain size: t(s)
Strategy.Integration.Freq      = 1              # integration interval: f(Hz)
Strategy.Integration.Time      = 0.1            # integration interval: t(s)
```

Step

A *single-step* describes a single unit of work of the strategy. A step that is defined in terms of a number of other steps is known as a multi-step. The attributes of a *multi-step* should be interpreted as default values for the steps that compose the multi-step. These default values can always be overridden.

Steps : *vector<string>*
The names of the steps that compose this step (for multi-steps), or absent (for single steps).

Baselines : *Baselines* (see page 36)
Baselines to use.

Sources : *vector<string>*
Sources to use. All sources will be used if this field is left empty.

ExtraSources : *vector<string>*
Additional sources to include when predicting visibilities. If this field is left empty, no extra sources will be included.

Correlation : *Correlation* (see page 35)
Specifies which correlations to use.

Integration : *DomainSize* (see page 35)
Cell size for integration. Allows the user to perform operations on a lower resolution, which should be faster in most cases.

InstrumentModel : *vector<string>*
The parts of the measurement equation that should be included.

Operation : *string*
The operation to be performed in this step. One of SOLVE, SUBTRACT, CORRECT, PREDICT, SHIFT, FLAG, or REFIT. Only relevant for single steps, should be absent for multi-steps.

OutputData : *string*
Column in the measurement set wherein the output values of this step should be written. If left empty, no data will be written.

Single steps should define one of the following fields, depending on the value of ***Operation*** :

Solve : *Solve* (see page 36)
Arguments of the SOLVE operation.

Example

```
Step.MultiStep.Steps                   = ["SingleStep1", "SingleStep2"] \
                                                                # steps that compose this multi-step
Step.MultiStep.Baselines.Station1      = [0, 0, 0, 1, 1, 2]          # baselines to use
Step.MultiStep.Baselines.Station2      = [0, 1, 2, 1, 2, 2]          # (all if empty)
Step.MultiStep.Sources                 = ["3C343"]                   # list of sources
Step.MultiStep.ExtraSources            = ["M81"]                     # list of sources outside patch
Step.MultiStep.InstrumentModel         = ["BANDPASS", "TOTALGAIN"] # instrument model
Step.MultiStep.Integration.Freq        = 2                           # integration interval: f(Hz)
Step.MultiStep.Integration.Time        = 0.5                         # integration interval: t(s)
Step.MultiStep.Correlation.Selection   = CROSS                       # one of AUTO, CROSS, ALL
Step.MultiStep.Correlation.Type        = ["XX", "XY", "YX", "YY"]  # which (cross) correlations to use

Step.SingleStep1.Baselines.Station1    = [0, 1]                      # baselines to use
Step.SingleStep1.Baselines.Station2    = [1, 2]                      # (all if empty)
Step.SingleStep1.Sources               = []                          # list of sources
Step.SingleStep1.InstrumentModel       = ["BANDPASS", "TOTALGAIN"] # instrument model
Step.SingleStep1.Operation             = SOLVE                       # one of PREDICT, SUBTRACT, CORRECT, \
                                                                # SOLVE, SHIFT, FLAG, REFIT
Step.SingleStep1.OutputData            = "OUTDATA1"                  # MS output data column
Step.SingleStep1.Solve.MaxIter         = 10                          # maximum number of iterations
Step.SingleStep1.Solve.Epsilon         = 1e-7                        # convergence threshold
Step.SingleStep1.Solve.MinConverged    = 0.95                        # fraction that must have converged
Step.SingleStep1.Solve.Parms           = ["gain:\{11,22\}:*"]        # names of solvable parameters
Step.SingleStep1.Solve.ExclParms       = []                          # parameters excluded from solve
Step.SingleStep1.Solve.DomainSize.Freq = 1000                        # f(Hz)
Step.SingleStep1.Solve.DomainSize.Time = 1                           # t(s)

Step.SingleStep2.Baselines.Station1    = []                          # baselines to use
Step.SingleStep2.Baselines.Station2    = []                          # (all if empty)
Step.SingleStep2.Sources               = []                          # list of sources
Step.SingleStep2.InstrumentModel       = ["PATCHGAIN"]               # instrument model
Step.SingleStep2.Operation             = CORRECT                     # one of PREDICT, SUBTRACT, CORRECT, \
                                                                # SOLVE, SHIFT, FLAG, REFIT
Step.SingleStep2.OutputData            = "OUTDATA2"                  # MS output data column
```

BBDB

Contains information on how the blackboard database and the parameter databases can be reached.

Host : *string*

Hostname or IP address of the host on which the black board database and the parameter databases reside.

Port : *int*
Port number on which the blackboard database server is listening.

DBName : *string*
Name of the black board database.

UserName : *string*
Username to access the black board database.

PassWord : *string*
Password to access the black board database.

ParmDB

Temporary settings while AIPS++ tables are still being used to store parameter values instead of the blackboard.

Instrument : *string*
Path to the AIPS++ table containing the instrument parameters.

LocalSky : *string*
Path to the AIPS++ table containing the local sky model parameters.

History : *string*
Path to the AIPS++ table containing the solve history.

Correlation

Selection : *string*
Station correlations to use. Should be one of ’AUTO’, ’CROSS’, or ’ALL’.

AUTO: Use only correlations of each station with itself.
CROSS: Use only correlations between stations.
ALL: Use auto and cross correlations both.

Type : *string*
Correlations of which polarizations to use, one or more of ’XX’, ’XY’, ’YX’, ’YY’. As an example, suppose we select ’XX’ here and set Selection to ’AUTO’, then the correlation of the X-polarized signal of each station with itself will be used. However, if we set Selection to ’CROSS’ then the correlation of the X-polarized signal of station A with the X-polarized signal of station B will be used for each base line (A,B)

DomainSize

Freq : *double*
The size of the domain in frequency (Hz).

Time : *double*
The size of the domain in time (s).

Baselines

The selected baselines. A baseline is a pair of stations. The first station of the pair is contained in Station1, the second in Station2. For example, suppose we have six baselines: (A, B), (A, C), (A, D), (B, C), (B, D), (C, D). Then Station1 would contain [A, A, A, B, B, C] and Station2 would contain [B, C, D, C, D, D]. The lengths of Station1 and Station2 should always be equal. If both fields are left empty, all baselines are used.

Station1 : *vector<string>*
One name for each baseline: the first station in the pair that forms the baseline.

Station2 : *vector<string>*
One name for each baseline: the second station in the pair that forms the baseline.

Solve

MaxIter : *int*
Maximum number of iterations.

Epsilon : *double*
Minimal difference between the old and the new parameter values after each iteration. When the difference falls below this threshold, the solver will stop iterating.

MinConverged : *double*
Minimal fraction of solve domains that must have converged to declare overall convergence.

Parms : *vector<string>*
Parameters to solve for. Wildcards are allowed, e.g. gain:{11,22}:*.

ExclParms : *vector<string>*
Subset of the parameters selected by Parms that should not be solved for. For example, if we would like to solve for the gain (amplitude, phase) of each station, but we would also like to fix the phase of the first station (STATION0) this can be specified as follows:

```
Solve.Parms = ["gain:{11,22}:*"]
Solve.ExclParms = ["gain:{11,22}:phase:STATION0"]
```

DomainSize : *DomainSize*
Size of the solve domain. The work domain is divided in solve domains and a solution is computed for each solve domain independently.