MeqTree Kernel Design Overview (PSS4)

O.M. Smirnov

CVS path: `LOFAR/doc/MEQ/MeqDesignOverview.tex`
Revision: 1.13
Date: 2005/01/21 13:15:08

Contents

Chapter 1

The Fundamentals

The purpose of this document is to provide an in-depth description of the MeqTree kernel and related interfaces. The following subjects will be covered:

- Basic terms and concepts.
- System components.
- Data structures employed in the MeqTree kernel.
- Standard node state & functionality.
- Interaction between nodes, and how it is affected by state.
- Interaction with Glish (and in the future, other scripting languages).
- Examples of some standard and application-specific nodes.

The intended audience for this document is:

The Tree Designer, since a thorough understanding of how nodes interact is critical in construction of complex and efficient trees.

The Node Developer needing to implement specialized node classes.

1.1 Trees

The MeqTree kernel provides a C++ implementation of the MeqTree concept. A **MeqTree** (or simply "tree") is a collection of interconnected **MeqNodes** (or simply "nodes"). Nodes have a directed parent–child relationship; a parent may have any number of children, and a child can have multiple parents. Cycles are forbidden. Technically, this makes the MeqTree a *directed acyclic graph*, but the term "tree" is retained for historical and aesthetical reasons.

We will often use directional terms when discussing trees, **up** is from parent to child, and **down** is from child to parent.

In broad terms, the life of a node generally consists of receiving **requests** from its parent(s), passing them up to its children, receiving **results** in response, performing some calculation on the child results, and returning the result down to its parent(s). Thus, requests generally originate somewhere down the tree and propagate up, while results germinate at the top and percolate down.

1.2 The math

Trees are mostly concerned with evaluating functions (e.g., predicted visibilities), optionally comparing these functions to other functions (e.g., measured visibilities), and solving for adjustable parameters to obtain the best fit. Before we discuss this in detail, it helps to define some basic mathematical terms.

The result of a node usually represents some function defined over $\mathcal{R}^2$ space – $f : \mathcal{R}^2 \to \mathcal{R}^N$ or $f : \mathcal{R}^2 \to \mathcal{C}^N$. The $\mathcal{R}^2$ domain is interpreted as frequency–time space in our application context, but in fact there's very few places in the kernel where this has any specific meaning. The codomain, $\mathcal{R}^N$ or $\mathcal{C}^N$, could represent any number of things, e.g., a single correlation $XX \in \mathcal{C}$, the Stokes parameters $(IQUV) \in \mathcal{R}^4$, four complex correlations in $\mathcal{C}^4$, etc.

The actual function **domain** $[X_1, X_2] \times [Y_1, Y_2]$ is a rectangular subset of $\mathcal{R}^2$. This domain is fully or partially covered by a set of $N \times M$ **cells**. Each cell $\mathbf{c}_{ij}$ is a $\Delta x_i \times \Delta y_j$ rectangle, centered on point (x_i, y_j). The function itself is represented by an object called the **vells**, which is essentially a set of $N \times M$ *samplings* $\bar{f} = \{f_{ij}\}$:

$$f_{ij} = f(x_i, y_j),$$

or $N \times M$ *integrations* $\bar{f} = \{\tilde{f}_{ij}\}$:

$$\tilde{f}_{ij} = \iint\limits_{\mathbf{c}_{ij}} f(x, y)\, dx\, dy.$$

A vells $\bar{f} = \{f_{ij}\}$ can also represent a set of *measured data*, such as observed complex visibilities.

A function may also depend on K real parameters. If we designate the *parameter space* $\mathcal{P} := \mathcal{R}^K$, then our function essentially becomes $f : \mathcal{R}^2 \times \mathcal{P} \to \mathcal{R}^N$ or $f : \mathcal{R}^2 \times \mathcal{P} \to \mathcal{C}^N$. This can also be represented as $f(x, y; p_1 ... p_K)$, or $f(x, y; \vec{p})$. The *model fitting problem* is essentially a minimization problem: finding the value $\vec{p}_0$ that minimizes the function $\chi^2(x, y; \vec{p})$ over a fixed set of cells in (x, y) space. The χ^2 function ties together *predict function* $f(x, y; \vec{p})$, measured data $\{\hat{f}_{ij}\}$, and an optional set of weights $\{w_{ij}\}$:

$$\chi^2(\vec{p}) = \sum_{ij} (f(x_i, y_j; \vec{p}) - \hat{f}_{ij})^2 w_{ij},$$

Solving the minimization problem hinges on estimating the gradients of f in $\mathcal{P}$ space.[1] Given a tree that computes $f_{ij} = f(x_i, y_j; \vec{p})$, we can numerically estimate each partial derivative $\partial f / \partial p_k(x, y; \vec{p})$ by taking a small **perturbation** δ_k, and using the same tree to compute a **perturbed value** for parameter k: $f_{ij}^{(k)} = f(x_i, y_j; p_1, ..., p_{k-1}, p_k + \delta_k, p_{k+1}, ..., p_K)$. In fact, the kernel is designed to automatically compute perturbed values when needed. More precise estimates may be obtained by using two sets of perturbed values ("double-differencing"), with different perturbations $\delta_k^{(1)}$ and $\delta_k^{(2)}$ (generally, $\delta_k^{(2)} = -\delta_k^{(1)}$). The different sets of perturbed values are designated as $f_{ij}^{(sk)} (s = 1, 2)$. These are passed around as a **vellset**, which is composed of the *value vells* $\{f_{ij}\}$, $K \times S (S = 1, 2)$ *perturbed value vells* $\{f_{ij}^{(sk)}\}$, and S vectors of the perturbation themselves $\delta_k^{(s)}$.

1.3 Nodes

Nodes are implemented as C++ objects, subclassed from the abstract `Meq::Node` class[2]. All interaction with nodes is done via the `Node` class interface. Consequently, a node has no direct knowledge of the *type* of its children. Nodes may be connected together in a practically arbitrary manner. Given a rich toolbox of elementary node classes, trees representing arbitrary mathematical expressions may thus be constructed.

All nodes have a **state record** that determines their behaviour. The state record is fully accessible from the outside. The initial state record is also called the **defrec** (from definition record), and is usually supplied from the scripting layer when the node is created.

Nodes also have a **result cache** that may retain the most recently computed result. The cache is intended to save processing time for repeated (or similar enough) requests.

1.4 Design Philosophy

These principles are key to understanding the design philosophy behind the kernel:

Locality: all functionality is defined in terms of local parent–child request–result interactions. There is no centralized "control" as such.

[1] The exact mathematical expressions are slightly different when dealing with integrations (as we do in the case of visibilities), but most operations remain essentially the same.

[2] All the MeqTree kernel classes reside in the `Meq` namespace; we will omit `Meq::` from names from now on.

KISS and rely on emergent behaviour: complex behaviour of the tree as a whole emerges from primitive request–result interactions at the local level. Most node classes are designed to be simple (K.I.S.S.!), with a single well-defined purpose. If some sort of specific functionality is required, it is almost always preferrable to implement it by building the right tree, rather than developing specialized nodes.

Policy-free: kernel node classes are largely policy-free. By policy we mean any sort of application-specific behaviour or concepts. Policy may only emerge at the tree level (by connecting the nodes in a specific way), and/or at the scripting level.

Prohibition is for policy-makers: this is a corollary to the previous principle. Apart from minimal and obvious sanity checks, the kernel imposes very few restrictions on its interface calls. This, of course, is a double-edged sword – it provides great power, but also great opportunities to do something wrong. It is left to the higher-level (i.e. application-level) scripting code to shackle the user and protect him from mistakes.

There is more than one way to do it: (with a nod to Larry Wall and Perl) in a lot of cases, the kernel provides several ways of accomplishing the same result. There is not necessarily a single "right" way, it all depends on the particular application context. This redundancy is by design, as it increases the overall adapatability of the system.

1.5 Software components

An interface to the kernel is provided via a `MeqServer` object. The `MeqServer` maintains a `Forest` (a collection of nodes and trees), and provides operations such as:

- Create, connect and delete node objects.
- Inspect and modify node state records.
- Issue requests and return results.
- Connect trees to data sources (e.g. Measurement Sets).

`MeqServer` plugs into the OCTOPUSSY publish/subscribe framework, and through it can can transparently support any number of local or remote clients, such as Glish sessions. The `MeqServer` object is instantiated inside a `meqserver` process.

All application-dependent logic ("policy") is meant to reside in the scripting layer (Glish, and in the future Python).

Chapter 2

Data Structures

2.1 Basic concepts

The main priciple driving data structure design in MeqTrees is **congruity**: all C++ objects used to *pass information* within a tree must be mappable without any loss of information to and from data structures on the scripting side. Fully *private* structures (e.g., private nested classes) can exist on the C++ side only.

Congruity facilitates **transparency**: most of the inner workings of a tree are readily accessible from the scripting side. This allows for very elaborate monitoring schemes, and is a great debugging aid when something goes wrong.

2.1.1 Data types

The choice of atomic data types is limited by the requirement of congruity. Currently, the only supported scripting language is Glish, but Python support is expected in the near future. In any case, the kernel restricts itself to common primitives that are supported by all mature scripting languages. Thus, kernel data structures are defined in terms of a restricted set of "legal data objects", specifically:

- scalars – bool, integer, float, double, float or double complex;
- strings;
- multidimensional arrays of scalars;
- lists of legal objects (in Glish this is represented either via a vector of scalars or strings, or via a record with fields indexed by number);
- records of legal objects (a.k.a. dictionaries/maps/hashes with a string key and a legal object value).

On the C++ side, data objects are based on the DMI `DataRecord`, `DataField` and `DataArray` classes. Most data classes are in fact derived from `DataRecord`, and are at core a record with some (sometimes loosely) predefined structure.

2.1.2 HIIDs

The `HIID` (hierarchical indentifier) class of the DMI package is used for all data-related indentifiers in the kernel, such as record fields, node groups, request IDs, etc.

A HIID is essentially a vector of integers called *atomic IDs*. Atomic IDs have a string representation: for IDs >=0 this is simply the integer itself in string form, while for IDs <0, a global *dictionary* (i.e. map from strings to numbers) is maintained in the development tree. Another way to look at this is that negative IDs represent *atomic concepts*, or words. Thus, any HIID can be viewed as a mix of words (from a fixed though rather large vocabulary!) and numbers.

The string form of a HIID consists of atomic IDs, separated by periods. For example, `"Request.ID.1"` is the string form of $(-1210, -1087, 1)$. Note that the string form is **not** case-sensitive, so `"request.id.1"` corresponds to the same HIID. An alternative string representation, employing underscore (`"_"`) as the separator, is used for Glish record fields, e.g., `rec.request_id_1`.

The reason we use HIIDs instead of plain strings is efficiency on the C++ side – many data storage classes engage in parsing or building up HIIDs, and vectors of integers are much easier to manipulate than strings. Note that the symbol-to-ID mapping is also available as C++ header files containing `const` declarations for atomic IDs. These may be used as, e.g.:

```
const HIID MyFieldName = AidRequest | AidId | 1;
```

which is a convenient and visually obvious way to define a constant HIID corresponding to `"Request.ID.1"`. The compiler turns this declaration into a `const HIID` object – essentially, a constant vector of integers.

HIIDs are covered in more detail in the documentation for DMI. Here we'll only dwell on their Glish form. There's two forms in which HIIDs appear in Glish:

- As record field names, e.g., `rec.request_id`. The implication of this is that in order to be recognized by the kernel, all record field names must be built up from a fixed vocabulary (which may be extended by the C++ developer as new classes are added).

- As values. In Glish, a HIID value is just a string containing the HIID's symbolic form, tagged by the `::is_dmi_hiid` attribute. The `hiid()` function (in `dmitypes.g`) is a convenient way to create HIID values. For example:

  ```
  my_id := hiid('Request.ID.1');
  my_id := hiid('request','id',1);
  ```

  will both create the same HIID.

  Note the crucial difference between strings and HIID values. Compare the two Glish records:

  ```
  rec1 := [ a = 'a.b.c.1' ];
  rec2 := [ a = hiid('a.b.c',1) ];
  ```

  While they may appear to be practically identical on the Glish side of things – both records contain the string field `a`, except `rec2.a` has an extra attribute tag – when passed to the C++ kernel, `rec1.a` is converted to an `std::string` object, while `rec2.a` is converted to a `HIID` object.

In this document, we will use both forms interchangably depending on context, with the understanding that, e.g., `request_id_1` and `"Request.ID.1"` both refer to the thing as far as the kernel is concerend.

2.1.3 Naming conventions

- In C++, standard data structures & nodes reside in `namespace Meq`. In Glish, corresponding object constructor functions are placed into the `meq` "namespace" (actually just a record), and names are all-lowercase. The DMI dynamic type system uses a `Meq` prefix for the namespace. Thus, the `Meq::Request` class in C++ is registered as a "`MeqRequest`" in the DMI type system, and has a `meq.request()` counterpart in Glish.

- Even though the languages we use are case-sensitive, HIIDs aren't. A good reason to avoid relying on character case to distinguish identifiers is that different languages and contexts have different capitalization conventions – compare, e.g., `Meq::Request` in C++, as opposed to `meq.request` in Glish. Thus we should always avoid assigning case-sensitive names to different entities.

2.1.4 Glish `meq` objects: meqtypes.g

With a couple of exceptions, Meq objects are represented by Glish records of a [mostly] predefined layout. The Glish/C++ conversion layer uses a few "magic" attributes to distinguish these objects from ordinary records, so it is able to map them to specialized Meq C++ classes rather than generic `DataRecord`s.

Specifically, the `::dmi_actual_type` attribute is set to a string which gives the DMI object type. Thus, it is possible to construct a record in Glish, tag it with `::dmi_actual_type`, pass it thorugh the Glish/C++ layer, and have it auto-magically converted to an C++ object of the appropriate class. The only requirement is that the record contain the correct set of fields, which are mapped to class attributes (data members) in C++. In practice, it's a lot more convenient to use the "constructor" functions defined in `meq/meqtypes.g`, which create properly formed and tagged records[1]:

`meq.domain()` creates a Domain object (record).

`meq.cells()` creates a Cells object (record).

`meq.requestid()` creates a request ID from individual components (e.g., domain ID, config ID, iteration ID). A request ID is really just a `HIID`.

`meq.request()` creates a Request object (record).

`meq.polc()` creates a Polc object (record).

In general, all data objects on the C++ side have counterparts on the Glish side. Within this document, we will usually describe data objects in terms of their Glish equivalents – records and record fields – with the implied understanding that there is a 1:1 mapping from that to C++ classes and data members.

2.1.5 Glish lists

Glish does not have a native "list" (a.k.a. "sequence") type. Instead, lists are emulated in one of two ways:

- If all list elements are all of the same *scalar* type, then the list is emulated by a vector.
- If the list elements are of a non-scalar type, or the type is not homogenous, then the list is emulated by a record.[2]

Since Glish records support numeric subscripts, both types of lists can be accessed via the same syntax – `len(list)` returns the number of elements, `list[`n`]` accesses element n. Note though that if a list contains only one element, it should still be accessed as `list[1]` rather than `list` (the latter syntax will actually do the right thing given a vector, but not a record, hence it should be avoided for consistency's sake.)

We will use the term *list* from now on to refer to both types of Glish structures, as it should usually be clear from context which type is actually employed.

2.1.6 1-based and 0-based indexing

Glish array indices are 1-based, while C++ indices are 0-based. This, unfortunately, has always led to all sorts of confusion, since indices pop up on both sides of the Glish/C++ barrier, and sometimes even need to be passed back and forth. As our experience with AIPS++ has shown, keeping track of index conversion on an individual basis is completely impractical.

The Glish/C++ conversion layer attempts to address this issue by providing *automatic conversion* of indices. If a record field's name ends in `_index` (on the C++ side this corresponds to a field HIID ending in the atomic ID `"Index"`), and the field contains a single integer or a list of integers, then the field is assumed to contain indices, and conversion between 0- and 1-base is automatically performed.

Thus, the Glish record `[foo=1,foo_index=1,bar_index=[2,3]]` will be converted to `[Foo=1,Foo.Index=0,Bar.Index=[1,2]]` on the C++ side (and vice versa), while `[foo_index=1.0]` or

[1] Note that some Meq classes may have Glish counterparts but no Glish constructors. At time of writing, these include `Vells`, `VellSets`, and `Results`. The reason for this is simply lack of necessity, since objects of these classes always originate on the C++ side rather than Glish. In the future, constructors for these classes may be added to Glish as required.

[2] When mapping lists created in the kernel to Glish records, the conversion layer assigns field names of the form `"#1"`, `"#2"`, etc.

`[foo_index=[a=1,b=2]]` will not undergo any conversion (since the `_index` field contains a `double` value in one case, and a subrecord in the other case).

Automatic conversion, of course, introduces its own potential for confusion if forgotten about. This is why you simply shouldn't forget about it.

2.2 MeqRequest and related data objects

A `Request` is a *job description* containing a number of commands that a node executes in order to produce a `Result`. A `Request` is implemented as a record with a semi-fixed structure; commands correspond to specific field names, while the value of each field usually carries the command arguments.

Operationally, the critically important command is `cells`: this tells the node to evaluate itself over a given grid in the frequency–time domain. In fact, most other commands are mere housekeeping, while `cells` represents the brunt of the workload. For this reason, the `cells` command gets special treatment: it is placed at the top level of the request record (along with a few related flags), and all nodes are obliged to process it. All other commands are kept inside a **rider** sub-record, and are subject to a *node selection* mechanism that allows commands to be directed to all nodes, individual nodes, or groups of nodes (this is described in further detail in section 5.3).

Each `Request` is assigned a unique request ID (**rqid**, pronounced "arr-cue-d"). This is a `HIID` describing various properties of the request. Request generators are expected to follow a certain *contract*, and assign rqids in a consistent way. This is covered in detail in section 4.1.1.

On the C++ side, a `Request` is derived from `DataRecord`. It can contain the following fields (of which only `request_id` is obligatory).

field name	*type*	*description*
`request_id`	`HIID`	the request ID
`cells`	`Cells`	*[optional]* a `Cells` object (see below)
`calc_deriv`	`int`	*[optional]* compute perturbed values (0, 1 or 2). Default is 0.
`next_request`	—	*[optional]* a hint of what the next request is going to be. This influences caching decisions and speculative execution (section 4.1.2). *Placeholder only, not currently implemented.*
`rider`	`record`	*[optional]* rider subrecord containing additional commands.

2.2.1 Constructing Requests

In Glish, a `Request` record can be created by calling the following function:

```
meq.request := function (cells=F,request_id=F,calc_deriv=0)
```

You can subsequently add commands to the record using `meq.add_command()` and `meq.add_state()`. This is also described in section 5.3.

2.2.2 Domains and Cells

The `Domain` class represents a rectangular domain in frequency–time space. The `Cells` class represents a gridding of that domain. This is illustrated by Figure 2.1.

On the C++ side, both classes are derived from `DataRecord`. A `Domain` record corresponding to $[f_{st}, f_{end}] \times [t_{st}, t_{end}]$ has the following structure:

`[ freq = [` f_{st}, f_{end} `], time = [` t_{st}, t_{end} `] ]`

where $f_{st}, f_{end}, t_{st}, t_{end}$ are `double` values giving the domain boundaries.

Note that the specific concepts of *frequency* and *time* are meaningful to only a handful of nodes. The majority of nodes simply deal in functions defined over abstract two-dimensional domains (in $\mathcal{R}^2$), without associating any semantics with the dimensions. For this reason, the bulk of kernel code is careful to abstract itself from the `freq` and `time` names whereever possible, treating domain components only as "first axis" and "second axis".

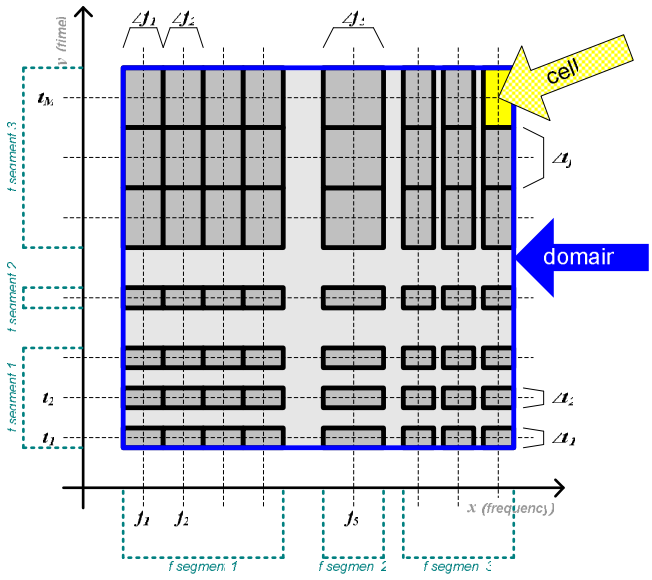

Figure 2.1: Layout of a Cells object and the envelope Domain

The Glish syntax of selecting record fields by number is handy here: if `dom` is a domain record, then `dom[1]` and `dom[2]` refer to the axis subrecords in a name-independent way.

The `Cells` record is structured in the same spirit:

```
[     domain   = [ envelope domain record ],
        grid   = [ freq = [f_1,...,f_N],          time = [t_1,...,t_M] ],
   cell_size   = [ freq = [Δf_1,...,Δf_N],        time = [Δt_1,...,Δt_M] ],
    segments   = [ freq = [start_index=[i'_1,...,i'_n],end_index=[i''_1,...,i''_n]],
                   time = [start_index=[j'_1,...,j'_m],end_index=[j''_1,...,j''_m]] ] ]
```

This represents an $N \times M$ gridding of the given domain. The $\{f_i\}$ and $\{t_j\}$ vectors give the cell centers, while $\{\Delta f_i\}$ and $\{\Delta t_j\}$ give the cell sizes.

The `segments` sub-record contains information on the **regular segments** of the grid.[3] A regular segment is a part of the grid over which the stepping between cell centers, as well as the cell size, remains constant. The `start_index` and `end_index` vectors contain the starting (i', j') and ending (i'', j'') indices[4] of each regular segment. By definition, given n segments and N grid points, $i'_k = i''_{k-1} + 1$, $i'_1 = 1$, $i''_n = N$.

In the simplest case, the entire $N \times M$ grid is regular, in which case the `segments` subrecord looks like this:

```
[  freq=[start_index=[1],end_index=[N]],
   time=[start_index=[1],end_index=[M]]    ]
```

The other extreme, of course, is the completely irregular grid. This is represented by `segments` of the form:

```
[  freq=[start_index=[1,2,...N],end_index=[1,2,...N]],
   time=[start_index=[1,2,...M],end_index=[1,2,...M]]    ]
```

Figure 2.1 shows an 8×7 grid with three regular segments along each axis. This would correspond to the following `segments` sub-record:

```
[  freq=[start_index=[1,5,6],end_index=[4,6,8]],
   time=[start_index=[1,4,5],end_index=[3,5,7]] ]
```

[3] Some calculations, such as the DFT, can be significantly optimized over regular grids.

[4] 1-based in Glish, 0-based in C++, see section 2.1.6.

Note that `segments` information is merely an optimization facility, and can be safely ignored most of the time. The user need not know anything about it, since `Cells` constructors compute `segments` automatically; most application code doesn't care either, instead working directly with the `grid` vectors.

On a related note, the `Cells` record is full of redundant information. For example, `domain` and `segments` can be completely derived from `grid` and `cell_size`. This has two important implications:

- To construct a `Cells`, you need to provide only the minimum sufficient information, while the constructor figures out everything else automatically.
- `Cells` records should be treated as read-only. Directly manipulating any of the values inside can break consistency between the grid/domain/segments fields, and lead to all sorts of confusion down the line.

Empty `Cells`

An empty record corresponds to an uninitialized `Cells` object in C++. Empty `Cells` may appear in situations where the cell information is not defined, so Glish code should be prepared to deal with it. See section 2.4.4 for an example.

Constructing `Domains` and `Cells`

The Glish `meq.domain` function constructs a record corresponding to a `Domain` object. Its usage is pretty much self-evident:

```
meq.domain := function (startfreq,endfreq,starttime,endtime)
```

The `meq.cells()` function is somewhat more elaborate:

```
const meq.cells := function (domain=F,num_freq=F,num_time=F,
                             freq_grid=[],time_grid=[],
                             freq_cell_size=[],time_cell_size=[])
```

All of the arguments are optional, allowing different ways of specifying a `Cells`. For example,

```
cells := meq.cells(meq.domain(0,1,0,1),2,2);
```

creates a regular 2×2 grid for the domain $[0,1] \times [0,1]$: cell centers at (0.25,0.75), cell sizes of (0.5,0.5). The same `Cells` can be alternatively specified as:

```
cells := meq.cells(freq_grid=[0.25,0.75],time_grid=[0.25,0.75]);
```

letting the constructor derive the domain & cell size automatically. The two forms can even be mixed:

```
cells := meq.cells(meq.domain(0,0,0,1),freq_grid=[0.25,0.75],num_time=2);
```

produces the same `Cells` yet again – note that the `freq_grid` values override the `freq` component of the domain.

If not supplied in the function call, the default cell sizes are computed to perfectly tile the specified domain. The `freq_` and `time_cell_size` arguments allow you to supply explicit cell sizes. These may be scalars – implying the same size for all cells along that axis – or vectors. In the latter case, the size of the vector must match the corresponding x`_grid` or `num_`x argument.

2.3 Fail-records and fail-states

Run-time errors during execution are reported via special structures called fail-records. Because fail-records can appear within different data structures (see below), they deserve to be documented separately. A fail-record has the following layout:

field name	*type*	*description*
`message`	`string`	a description of the error.
`node_name`	`string`	*[optional]* name of originating node, if any.
`node_class`	`string`	*[optional]* classname of originating node, if any.
`origin`	`string`	origin: usually just the source file name.
`origin_line`	`int`	origin location: usually just the source line number.

Because most errors tend to cascade from lower-level subsystems up to the application level, accumulating more specific descriptions along the way, fail-records usually come in a list. Lower-level errors then appear at the head of the list, and higher-level errors appear at the tail.

Certain data classes described below (e.g., `Result` and `VellSet`) support a fail-state – i.e., a form of the data object describing a failure. A fail-state is represented by a record with a single field named `fail`, containing a list of fail-records. The kernel uses this layout consistently for indicating fail state, so all Glish code can follow a simple policy and process fails in the same way everywhere:

- Any record with a `fail` field represents an object in fail-state.
- In fail-state, no other meaningful fields exist.
- The `fail` fields always contains a *list* of fail records (even if the list contains only one element).

2.4 MeqResult and related data objects

A `Result` contains the result of a `Request`'s execution. A `Result` is also a record (derived from `DataRecord` on the C++ side). Theoretically, this record is completely free-form, with its contents dependant on the commands in the original `Request` (and in some cases even on node type – e.g., a `Solver`'s result will contain solution metrics). If, however, the original `Request` contains a `cells` command – asking the node to evaluate itself over the given `Cells` – and the command is executed successfully, then the returned `Result` has a well-defined structure:

field name	*type*	*description*
`cells`	`Cells`	the `Cells` of the result (not necessarily matching the request cells – see 5.4)
`values`	`VellSet[]`	list of result values
`integrated`	`bool`	flag indicating if the values are integrations or samplings (default is false, implying samplings)

Two other common types of `Result` are the empty result (empty record), returned when a `Request` does not contain any commands with return values, and the fail-result (see section 2.3), returned when a run-time error arises during execution.

2.4.1 Sampling vs. integration

A `Result` can represent both a sampling of some function at the cell centers, or an integration over each cell. The `integrated` flag is used to indicate this, if missing, `false` (i.e. a sampling) is assumed. Leaf nodes set this flag according to the type of value they return (for example, a `Spigot` reading visibilities from a data set returns integrations; a `Parm` representing gain returns samplings). Non-leaf nodes should take care to pass this flag from child to parent properly. This flag is also taken into account when performing resampling of results (section 5.4).

2.4.2 Multiple values

Note that the `values` field is defined as a *list* of `VellSet`s. In Glish, a list is implemented as a record, using the `rec[1]`, `rec[2]`, etc. syntax to access fields by number. Even if there is only one `VellSet` in the list, you still have to access it as `result.values[1]`.

Each `VellSet` represents a function $f : \mathcal{R}^2 \rightarrow \mathcal{R}$ or $\mathcal{C}$. A set of N `VellSet`s then represents $f : \mathcal{R}^2 \rightarrow \mathcal{R}^N$ or $\mathcal{C}^N$.[5] Another way to look at it is that a set of `VellSet`s allows multiple "planes" for a single `Cells`. For example, a `Spigot` may return four planes at a time for the four correlations. Function nodes expect all child `Result`s to either have the same number of planes, and will apply the function to each set of planes (cross-slice) independently; however, some children may have only one plane, in which case it is re-used in each cross-slice.

The `Selector` and `Composer` nodes can be used to decompose and assemble `Result`s on a plane-by-plane basis. The `Selector` node has one child; it returns a `Result` composed of specific plane(s) from its child. The `Composer` assembles a `Result` from all the planes returned by its children.

[5] In fact, the set can even contain a mix of $\mathcal{R}$ and $\mathcal{C}$ codomains.

2.4.3 VellSet

A `VellSet` is essentially a sampling or integration (see 2.4.1) of some function $f : \mathcal{R}^2 \to \mathcal{R}$ or $f : \mathcal{R}^2 \to \mathcal{C}$ over a certain `Cells`. A `Cells` defines a domain & grid in $\mathcal{R}^2$ space – usually interpreted as frequency-time – specified by the grid vectors $(x_1, ..., x_N)$ and $(y_1, ..., y_M)$. Thus, a `VellSet` contains an $N \times M$ matrix of *function values* $f_{ij} = f(x_i, y_j)$.

If the function is dependent on K real parameters $(p_1, ..., p_K)$, then the `VellSet` may also contain a set of *perturbed values* $\{f_{ij}^{(k)}\}$, which can be used to estimate the partial derivatives $\partial f/\partial p_k$. The math behind this is explained in section 1.2. Within a tree, the parameters p_k are uniquely indentified by their integer **spids** (from *solvable parameter IDs*).

On the C++ side, `VellSet` is derived from `DataRecord`. The `VellSet` record has three forms: regular, empty and fail.

Regular VellSets

The regular form of a VellSet contains the following fields:

field name	*type*	*description*
`value`	`Vells`	the `Vells` containing the function value $\{f_{ij}\}$
optional, only appear if `calc_deriv>0` *was specified in the original* `Request`*:*		
`spids`	`int[]`	a list of K integer spids identifying the parameters
`perturbations`	`double[]`	a list of K perturbations $\{\delta_k\}$ (must be same length as `spids`)
`perturbed_value`	`Vells[]`	a list of K `Vells` containing the perturbed values $\{f_{ij}^{(k)}\}$
optional, only appear if `calc_deriv>1` *was specified in the original* `Request`*:*		
`perturbations_1`	`double[]`	second set of K perturbations $\{\delta_k^{(2)}\}$
`perturbed_value_1`	`Vells[]`	second set of K perturbed values $\{f_{ij}^{(2k)}\}$

Spids, perturbations and perturbed values will only appear if `calc_deriv` is specified in the original `Request`, and the tree above the node contains solvable parameters. A setting of `calc_deriv=2` causes two sets of perturbations and perturbed values to be computed (usually with $\delta_k^{(2)} = -\delta_k^{(1)}$), used to estimate the second-order derivatives of f, which can potentially achieve a better fit, at the expense of almost doubling the computing time.

Empty VellSets

An empty `VellSet` is just an empty record, corresponding to a default-constructed (empty) object in C++. While empty `VellSet`s shouldn't be present in well-formed results, they can still appear in node state records and other structures, thus Glish code should be prepared to deal with them.

Fail-VellSets

A fail-VellSet is used to indicate a run-time error or other failure. This is represented by a standard fail-state record (see section 2.3). The difference between this and a fail-Result is discussed below.

Vells

On the Glish side, a `Vells` object is simply a `double` or `complex` scalar or a 2D array. On the C++ side, `Vells` is a wrapper class around the scalar/array, providing run-time type and size information, and built-in mathematical operations, and copy-on-write semantics. This is discussed in detail in the context of `Function` nodes (section 5.5.4).

Scalar `Vells` represent values with no time-frequency variation. Array `Vells` represent dependence *over a specific Cells*. This implies that all *array* `Vells` within a `Result` must be consistent in shape with the `Result`'s `Cells`; scalar `Vells`, on the other hand, are by definition consistent with any and all `Cells`.

2.4.4 Empty Cells in Results

A `Result` record containing an empty `Cells` field is used to represent *constant values* – or values with no time-frequency dependence. In other words, a `Result` with an empty `Cells` record will be the same for any possible set of real cells. Such a `Result` may only contain scalar `Vells`. This property of a `Result` is important during resampling (see 5.4), since constant values do not need to be resampled.

2.4.5 Fail propagation

Note that a `Result` that is not in a fail-state itself may nonetheless contain one or more `VellSet`s in a fail-state. One way to look at it is that a fail-Result represents a complete fail, while a fail-VellSet represents a partial fail localized to one plane. For example, if a `Spigot` node is configured to return four correlations, and the data source only contains XX and YY, then the XY and YX planes will be represented by fail-VellSets. On the other hand, if an error occurs while reading from the data set, this is represented by a complete fail-Result.

Depending on the tree, partial fails may be recoverable. Fails propagate down the tree in an orderly fashion. For most nodes, a partial fail from one of its children will result in a fail-VellSet at the same position in the output `Result` (the contents of the fail – origin & description – are preserved.) In our example, partial fails could propagate down the XY/YX trees, to a `Sink` node, which could then handle them benignly (by not writing XY/YX data, for example). Partial fails can even "disappear" on their way down a tree – consider a `Selector` node that is configured to select plane 1 of the `Result`. Partial fails in the other planes will simply be discarded.

Chapter 3

Node Initialization & State

The fundamental behaviour of & interface to a node is provided by the C++ class `Meq::Node`. This is an abstract class; at least one pure virtual methods is declared (`Node::getResult()`) that must be defined by subclasses to implement specific functionality.

All nodes share the following basic traits:

- A node may have a number of child nodes. Generally, a node has no knowledge of the types of its children. Subclasses may assign formal child labels (akin to argument names) to specify semantics, or may leave their children unlabeled. Child labels are assigned via the constructor of the subclass.

  A node also has no direct knowledge of its parents, and is only allowed to infer things from the requests that it receives.

- Each node has a unique **node index** (integer>0) and an optional unique **node name**. A `Forest` object acts as a repository of nodes, and maintains a map between names, indices and node objects (see section 6).

- A `Node` maintains a **node state record**, which should completely determine the behaviour of the node.

- A `Node` has an `execute()` method, taking a `Request` parameter, and returning a `Result`. Normally, a node is expected to call `execute()` with the same `Request` on its children, and form its result based on the results of its children. Thus, requests propagate up the tree, and results percolate down the tree. This is discussed in Chapter 5.

The subject of handling requests will be dealt with later. This chapter deals with everything related to node initialization and state.

3.1 Node state in C++ and Glish

Each node's state is mapped to a state record (`DataRecord` in C++, standard record in Glish). The base `Node` class defines this record and provides a number of tools for maintaining it. Note that the internal state of a C++ object, as determined by its data members, is physically different form the state record, and it is up to the object itself to maintain **coherency** between the two. Coherency is critically important, since the Glish layer only has access to the state record, and not to an object's internal data members. As will be seen below, the `Node` class implements a number of facilities that simplify the task of maintaining coherency.

Node state is almost always inherited from superclass to subclass. Subclasses define state in terms of *additions* to state defined by their superclass. State defined by the base `Node` class is common to all nodes.

3.1.1 Access to state

On the C++ side, state may be read via the `Node::state()` method, and changed via the `Node::setState()` method. On the Glish side, these are mapped to the `getnodestate()` and `setnodestate()` methods of the `meqserver` proxy (see Chapter 6). The argument to `setState()` (or `setnodestate()`) does not have to be a complete new state record; instead, it should only contain those fields that actually need to be changed.

Another mechanism of state changes is the *request rider*. A `Request` can contain a command that changes the state of a node or a group of nodes (section 5.3).

When a node is constructed, it is passed[1] an **init-record** (also called the **defrec**, for *definition record*), which is nothing more and nothing less than the complete initial state record of the node. This is the mechanism via which all run-time arguments to a node are specified. Later in a node's lifetime, it may be reconfigured (via `setState()` or request riders). A node is not obliged to be reconfigurable in every single aspect, although it's good design to make it so as much as possible. If some of the node state may only be set once via `init()` and not changed later on via `setState()` – we'll call this **static state** – it should be clearly documented as such. The assumed default is **dynamic state**, i.e., state that may be reconfigured at any time via `setState()`.

The individual fields of the state record are known as **state fields**.

3.1.2 Categories of state fields

All state fields belong to one of the following categories:

Static state can only be set up at construction time, via the init-record. Static state is protected: any attempts to modify it should return an error. By design, static state is kept to a minimum.

Dynamic state can be specified at construction time, and freely changed later on via `setState()` or request riders. Node classes are designed so that most of state is dynamic. This is the assumed default, unless clearly documented otherwise.

Informational state does not affect the behaviour of a node. It is a one-way street: nodes maintain these fields to provide additional info to outsiders (thus improving transparency, i.e., monitoring and debugging), but any changes to these fields from the outside are simply ignored.

Script code can monitor informational state, but should have any operational dependencies on it. Due to performance concerns, the setting of informational state may be compiled out of optimized builds of the kernel.

Other state fields that a node class does not recognize are simply ignored. Outsiders may read and change these fields at will; this can be useful for tagging a node with additional informational attributes.

3.1.3 Clients

It is useful to introduce the term **client**, referring a software component (or even the user himself) that initiates the creation of nodes, specifies state changes, issues initial requests, etc.

From a node's point of view, the client is any external entity that accesses the node interface. From the `MeqServer`'s point of view, the client is the scripting layer, or perhaps another C++ component that interfaces with it via OCTOPUSSY. From the scripting layer's point of view, the client is the user himself, or perhaps a batch script run by the user that uses standard functions in the scripting layer.

3.1.4 The Node Contract

Even the base `Node` class exhibits some non-trivial behaviour with regards to maintaining state and processing requests. This behaviour is not defined or constrained by the node interface as such. The node interface simply defines a collection of methods (`init()`, `setState()`, `execute()`) and data formats. Meanwhile, it's the implementations of these methods that provide additional semantics, such as tying node behaviour to node state.

These additional semantics are known as the **node contract**. For example, maintaining a state record that is coherent with internal C++ object state is part of the basic node contract. Responding to changes in dynamic state is another part of the contract. Other examples will be discussed below. In general, the contract is a set of obligations that a node can be trusted to follow.

A conventional contract brings together at least two parties. In the case of the MeqTree kernel, the other party to the contract is the client (scripting layer, tree builder, tree user, etc.). The obligations of the client are also specified in the contract. In particular, the `setState()` interface to a node is an extremely powerful tool; node classes provide only minimal sanity checking, so it's always possible to configure a node into some sort of senseless state. Correct interaction of nodes within a tree requires nodes to be consistently configured. Thus, the contractual obligations of a node to behave correctly are only valid as long as the client meets its obligations of configuring the node(s) consistently. We will see specific examples of this later on.

[1] via the `init()` method in C++, or via the `createnode()` call in Glish.

It is helpful to view the kernel in the context of a multi-layered software system. The lowest level is the MeqTree kernel itself; on top of that is the `MeqServer` interface, and on top of that a thin scripting layer. On top of that – now completely in the scripting domain – we have application-specific scripts to build trees, and still higher up, user-level tools to operate, manage and visualize trees. The interfaces grow more application-specific towards the upper layers, while contracts grow less specific. The overall design philosophy here relies on the fact that it is far easier to implement complex semantics in a scripting language; thus the higher layers in the scripting domain can ensure more and more elaborate aspects of the node contracts, until the user is completely protected from these complexities.

Note that an end user will hardly ever need to change (or even see) node state directly; a tree developer, however, will be working with this stuff constantly. As will become apparent in this chapter, a node's internals are completely exposed to tweaking and experimentation. The C++ kernel imposes no policy and does very few sanity checks, so it is perfectly possible to thoroughly confuse a node (or an entire tree) through misguided manipulation of its state record. Again, we assume that it is up to the scripting side tools to shield end-users from the more "dangerous" capabilities. Remember, prohibition is for policy-makers!

When developing new node classes, it is very important to define and document their contracts in detail.

3.2 Standard Node state fields

Table 3.1 lists all the state fields defined and maintained by the `Node` class. Here we will only discuss the basic fields; the more advanced ones will be documented in further chapters.

field name	*type*	*default*	*description*
static state:			
`class`	`string`		the node class
`nodeindex`	`int`	`0`	the node index
`children`	—	null	children specification (see 3.2.2)
dynamic state:			
`name`	`string`	`""`	node name
`node_groups`	`HIID[]`	`[]`	list of groups that this node belongs to
`auto_resample`	`int`	`0`	auto-resampling mode (see 5.4)
cache-related dynamic state (see 4.1):			
`cache_policy`	—	null	caching policy *(placeholder; not currently implemented)*
`request_id`	`HIID`	null	rqid corresponding to cached result
`cache_result`	`Result`	null	cached result
`cache_result_code`	`int`	`0`	cached result code
depmask-related dynamic state (see Chapter 4.3):			
`depend_mask`	`int`	`0`	current depmask
`known_symdeps`	`HIID[]`	`[]`	list of known symdeps
`active_symdeps`	`HIID[]`	`[]`	list of active symdeps
`symdep_masks`	`record`	`[]`	current symdep masks
`gen_symdeps`	`HIID[]`	`[]`	list of generated symdeps
`gen_symdeps_group`	`HIID[]`	`[]`	list of groups for generated symdeps
informational state:			
`children_names`	`string[]`		list of child names. Can't be used to specify child nodes: use the `children` field instead.
`request`	`Request`		last handled request

Table 3.1: Base state defined by the `Node` class

3.2.1 Constructing nodes: classes, names, indices

To construct a node, a client must provide the init record – i.e., an initial state record. Note the "default" column in Table 3.1 – only a few state fields need to be specified, since the rest have reasonable defaults that will be filled in by the node at construction time (a default of "null" indicates that the field, if missing, will remain unfilled). Also, the cache-related and depmask-related state fields are not normally set at construction

time, but rather are filled by the node itself during operation. The node state interface provides full access to them mostly for debugging and monitoring purposes.

The class of a node is obviously an external, static property – once a node object is instantiated, its class is defined "for life". The `class` field is usually set by the client in the scripting layer, and used by `MeqServer` as a key into a *node constructor registry*, when determining which node class to actually instantiate. The `class` string is usually formed by concatenating the C++ namespace identifier with the C++ class name. Thus, a `Meq::Parm` is specified as `"MeqParm"`.

The node index (field `nodeindex`) is not normally specified by the client, but rather automatically assigned to the object at construction time, and placed into the `nodeindex` field. The index is also a static property.

The node name (field `name`) is just a free-form identifier supplied by the client. The name is used to identify the object for subsequent operations. The `MeqServer` object maintains a `Forest`, which is essentially just a repository of node objects, with name→node and index→node maps. A node may be created with an empty name (which is the default), such *anonymous* nodes are then only identifiable by their indices. Note that while the name of the node may be changed via a `setState()` call, this is probably a bad idea, since the current `Forest` implementation does not support node renaming (i.e. cannot update the name→object map).

3.2.2 Specifying children

The `children` field of the state record is used to specify a node's children. The children set is a static property[2], and must be specified at construction time. The `children` field takes the form of a list or record of **child specifiers**. A child specifier can be:

- An integer node index referring to an already existing node.
- A string node name, which does not have to match an existing node. The child can be created later on; `MeqServer` includes a mechanism for recusively *resolving* named children (see Chapter 6).
- A record, which will be used as an init-record to create a child node on the fly. This option allows whole subtrees to be specified via one big nested record.

Child labels

Certain node classes can predefine *labels* (`HIID`s) to identify their children. Think of child labels as being the equivalent of named arguments in a programming language. Without labels, children can only be specified by their ordinal number (a-la positional arguments). Thus, labels help distinguish children with specific roles. For example, the `UVW` node (used to compute UVW coordinates) may label its children, e.g., "Ra", "Dec", "X", "Y", "Z". If the child roles are all the same (e.g., the `Sum` node, which sums the results of its children), or are obvious from position (e.g., binary function nodes, including non-commutative ones such as `Sub` and `Div`), then labels are not used.

Labels have the following implications for the `children` field:

- Given a node class that predefines child labels, `children` may be specified as a record. Child labels are matched to field names (on a mismatch, node creation fails, and an error is reported.)
- If `children` is a list rather than a record, then child nodes are simply matched by position, and any labels are ignored.
- If `children` is a record but the node class does define any labels, then child nodes are matched by their position in the record. This way of specifying children is discouraged, since the abstract record type does not provide for a fixed order of fields.

Child information

After a node has been created and all children have been resolved, the `children` field of the state record is replaced with either a list of child node indices if no labels are predefined, or a record of labels to indices otherwise. This field is treated as static state, so any attempts to modify it via the `setState()` mechanism will fail. The (purely informational) `children_names` field is filled with a similar list or record of child names.

[2]The ability to connect children dynamically may be implemented in future versions if necessary. In any case, it would be provided via a separate function rather than the generic `setState()` mechanism, so the `children` field would remain protected.

3.2.3 Node groups

A node may be assigned to one or more node groups. Node groups are used to restrict the commands of a `Request`to a specific set of nodes. This mechanism is discussed in detail in section 5.3. Examples of its use may be found in the documentation for the `Parm` and `Solver` nodes.

3.3 Creating init-records in Glish

The `meq.node()` function (in `meq/meqtypes.g`) can be used to put together a basic init-record. This record can then be extended with additional fields as required:

```
const meq.node := function (class,name,children=F,groups="")
```

The mandatory `class` and `name` arguments are strings specifying the node class and node name. The optional `children` argument is a list or record of child specifiers (see above). The `groups` argument specifies the node groups, this can be a list of `HIID`s or strings; in the latter case (as is usual for most `meq` functions), `meq.node()` will convert the strings to `HIID`s automatically.

Specialized node classes may define functions of their own to put together the corresponding init-records. An example of this is `meq.parm()` (see Glish file for details).

3.3.1 The defrec map

The kernel build system includes a mechanism for automatically generating Glish scripts that define class-specialized init-records. This code – known as the *defrec map* – is generated based on comments found in each node's class header file, and may be used as an alternative to the `meq.node()` function.

The defrec map is made available by including `meq/defrec.g`. This defines the following function:

```
const meqdefrec := function (class,name='',children=F,groups="")
```

The function can be used exactly like `meq.node()`, but with one important difference: it returns a complete init-record for the specified class, including any specialized fields defined by that class. These fields are initially populated with default values.

In addition to this, the init-record returned by `meqdefrec()` is also self-documenting. The record itself is tagged with a `::description` attribute, containing a textual description of the node class. Record fields are also tagged with `::description` attributes of their own.

3.4 The C++ side

The rest of this chapter deals with implementation of node state on the C++ side. You probably don't need to read this unless you're developing your own node classes.

The following is a list of `Node` methods responsible for initializing and changing state:

```
  // public: Initializes node with init record
  //         Note that Ref::Xfer implies that ref to record will be taken over
virtual void init (DataRecord::Ref::Xfer &initrec);

  // public: Reinitializes node with init record (called after de-serializing)
virtual void reinit (DataRecord::Ref::Xfer &initrec);

  // public: Changes dynamic node state (note: non-virtual)
  //         Node can attach to/take over record contents as needed.
void setState (DataRecord &rec);

  // protected: Checks init record for missing fields, fills in defaults where needed
  //            (called from Node::init())
```

```
virtual void checkInitState (DataRecord &rec);

  // protected: Implementation for setting or changing internal dynamic state
  //            (called from Node::setState())
  //            Node can attach to record contents as needed. If initializing,
  //            then record is the state record and should not be changed. If
  //            not initializing, node can take over contents as well.
virtual void setStateImpl (DataRecord &rec,bool initializing);
```

3.4.1 Managing data objects via CountedRefs

Many of the methods described here (and in Chapter 5) take arguments of type *Class*`::Ref::Xfer` or `Ref::Copy`. These arguments are known as **counted refs**. Counted refs are properly documented in the DMI Programmer's Guide; this section provides a brief primer.

Counted refs provide an efficient object management mechanism. Most DMI data objects can be accessed via a counted ref; this allows the same object to be shared by many "owners", and to be passed around efficiently. Refs may be copied (in which case a second ref to the same object is created) or transferred (xferred), in which case the new ref is attached to the object while the old one is detached. When the last ref to an object is detached, it is automatically destroyed. DMI containers (`DataRecord`, `DataFields`) hold their contents via counted ref.

Counted refs may be read-only or read-write. Holders of a read-only ref cannot legally write to an object (without engaging in C++ `const` violations, which are caugfht by the compiler). The holder of a ref may *privatize* it: this operation ensures that a "private" copy of an object is made. The privatization operation is essential for avoiding unnecessary (and presumably expensive) copying of large data objects: refs are smart enough to figure out when actual copying is not required. For example, a singly-referenced object is already private to begin with. An object with no writable refs can also be considered "private" to each read-only ref holder, since there's no legal way to modify the object.

For example, a `Request` object is passed up the tree via read-only refs. This means that all nodes deal with the same object; if a node needs to modify a request, it must privatize its ref first. This ensures that `Request`s are copied only when really necessary. A similar mechanism is employed for `Result`s and the result cache: children returns refs to result objects, and retain refs in their cache. Because most nodes do not modify child results, but rather process them as read-only before discarding, only a single instance of that `Result` object needs to exist.

All counted ref types are instantiations of the `CountedRef<T>` template defined in DMI. Most classes define the nested type *Class*`::Ref` as a shortcut for `CountedRef<`*class*`>`. The `::Ref::Xfer` and `::Ref::Copy` types are aliases for `::Ref` itself, these are used in function declarations to document the function's behaviour, i.e., whether it can be expected to take a copy of the ref, or to transfer the ref.

3.4.2 init()

The base `Node::init()` method is called with an init-record, directly after a node has been constructed. The method is virtual, and thus can be redefined in subclasses if needed. It does the following:

1. Takes over the init record, sets it as the state record, ensures a private & writable copy.

2. Adds the node's classname to the state record if not already present. If present, checks that the name actually matches the node class.

3. Calls the virtual `checkInitState()` method with the state record, to ensure that it's complete, and that any missing defaults are filled in.

4. Calls `setStateImpl(staterec,true)` to set up internal state from the state record (setting the second argument – `initializing` – to `true` indicates that the node is being initialized with a full state record.)

5. Processes the `children` field to set up connections to child nodes (see 3.2.2).

6. Any errors will result in an exception being thrown at the caller. A node object that fails `init()` is under no obligation to be usable; the only method that's not allowed to fail is the destructor.

Derived classes need to implement their own `init()` only if they have some special initialization needs that can't be taken care of via `setStateImpl()`. The `init()` method in a derived class should do the following:

1. Call the parent class's `init()` with the init-record. This should ultimately call `Node::init()`, thus setting up the state record, and calling `setStateImpl()` to set up dynamic state.
2. Set up static state defined at the child class level.
3. Perform any specific initialization.
4. Throw exceptions on any error.

Note that if only static state needs to be implmeneted, then it is easier to handled it in `setStateImpl()` (when `initializing` is `true`); there's little point in redefining `init()` specifically for that purpose only. (Most node classes do find it sufficient to only redefine `setStateImpl()`, and not bother with any of the other state-related methods.)

3.4.3 checkInitState()

Note the virtual `checkInitState()` method called from `Node::init()`. This is meant to check the init-record for mandatory fields, and insert missing defaults. Alternatively, this can be done in `setStateImpl()` when the `initializing` is `true`. The second way is usually easier, since `setStateImpl()` is almost always redefined by subclasses.

A derived `checkInitState()` should call the parent version, then check for defaults and mandatory fields defined by the child class, and throw an exception if any fields are missing.

A couple of macros defined in `Node.h` are meant to help implement this method. The `requiresInitField(rec,field)` macro checks if the specified field is present, and throws an exception otherwise. The `defaultInitField(rec,field,deflt)` macro inserts a default value for a field if it is missing.

3.4.4 setStateImpl()

The virtual `setStateImpl()` method is responsible for setting up and/or modifying node state. Note that this method is protected – the rest of the world must call it via through `init()` or `setState()`.

```
void Node::setStateImpl (DataRecord &rec,bool initializing);
```

The first argument is a record, the second argument tells the method whether the node is being initialized (and `rec` is a complete init-record), or simply updated (and `rec` contains only a subset of state, i.e. only those fields that are actually being changed).

Most node classes can get away with implementing their own `setStateImpl()`, and not worrying about any of the other methods described in this section. A node's `setStateImpl()` should follow this checklist:

1. If the `initializing` is `false`, check the record for protected fields (i.e. for attempts to modify static state). Throw a `FailWithoutCleanup` exception (see below) if any are present. The `protectStateField(record,field)` macro defined in `Node.h` is a convenient way to do this.
2. If `initializing` is `true`, check for mandatory state fields – throw `FailWithoutCleanup` is any are missing – and/or fill in defaults (unless `checkInitState()` has been already been redefined to do the same). Setup static state (unless handled by `init()`).
3. Call the parent class's `setStateImpl()`, which presumably sets up inherited state.
4. Parse the record and modify dynamic state relevant to the child class. Note that the standard DMI hook methods `get()` and `get_vector()` are very handy for doing this operation:

```
if( rec[StateField].get(var,initializing) )
  // field is present, react if needed
else
  // field is missing, react if needed
```

The `get()` method employed here does the following: if `StateField` is present in `rec`, assigns its value to `var` (throwing an exception is the types are incompatible) and returns `true`. If the field is missing, optionally (only if `initializing` is `true`) inits it from the value in `var`, and returns `false`. The standard `setStateImpl()` methods make extended use of this mechanism.

5. Throw exceptions on error. A `Node::FailWithoutCleanup` should be thrown if and only if no internal state was modified. All other exceptions will invoke the "rollback" mechanism described below. You can rely on `DataRecord` (and other DMI classes) to throw an exception when datatypes mismatch or something else goes wrong; throw a `FailWithCleanup` exception if you want to indicate some other kind of failure.

The base `Node::setStateImpl()` method handles all of the dynamic state listed in Table 3.1.

3.4.5 setState()

The non-virtual `setState()` method defined in `Node` provides the public interface for setting state. Basically, it defers parsing the record to `setStateImpl()`, while providing a transaction mechanism of sorts:

1. Calls `setStateImpl(rec,false)` to process the record. The `false` value indicates that state is being modified rather than [re]initialized. (Note that if the supplied record happens to be the node state record itself, `true` will be passed in instead.)

2. Catches `FailWithoutCleanup` exceptions and rethrows them at the caller with no additional action.

3. Catches all other execeptions, and does a cleanup before rethrowing them. The cleanup consists of calling `setStateImpl(staterec,true)`, so as to reset internal state from the current state record. This is meant to roll back from situations where an error midway through `setStateImpl()` could cause internal object state to decohere from the state record.

4. On success, merges the supplied record into the current state record.

This design ensures that if a `setStateImpl()` call fails (i.e., with an exception), both the state record and the internal state of the object are rolled back to their values prior to the call. (Assuming they were mutually consistent to begin with.) In other words, the node object is guaranteed to remain usable.

Note that implementations `setStateImpl()` should ignore any unrecognized fields in the init and state record. This allows the outside world to assign arbitrary informational attributes to node state.

3.4.6 Serialization & persistency issues

`MeqServer` already supports persistent nodes (i.e. being able to save/load nodes and trees to a file). Further down the road, we plan for the the capability to move a node across a network. Both capabilities hinge on being able to serialize a node – i.e., to turn it into a stream of bytes, and to unpack it from a stream of bytes.

Serialization is implemented through standard DMI mechanisms. A `DataRecord` is inherently serializable. To serialize a node, `MeqServer` simply serializes its state record. To unserialize a node, it recreates (un-serializes) the state record, creates a node object (as specified by the record's `class` field), and calls `init()` on it with the record.

Thus, subclasses of `Node` should take care to maintain their state record appropriately. Each node class should ensure that it is completely re-creatable (via `init()`) from a snapshot of its state record at any point in time. Basically, this means that a 1-1 mapping should be maintained between the state record and internal object state. One possible exception to this are internal caches; if these are not maintained in the state record, then the worst than can happen from re-creating a node is a cleared cache.

Chapter 4

The Node Cache, Symdeps & Depmasks

4.1 Caching issues

To avoid unnecessary recalculations, a node's result can be retained in a cache. Obviously, this trades off performance against memory footprint. Three broad caching policies have been identified so far:

Never: no caching at all, values are always recalculated anew.

Always: result is always cached, until a different request comes in. (This is the policy implemented currently). While expensive in terms of memory, is very useful for debugging, since it allows one to pause the system and examine the most recent result of every node.

Smart: result is cached according to memory availability, expected request sequence, etc. Some varieties of this policy are discussed here.

Caching policy can be set on a per-node basis, via the `cache_policy` field of the state record. The cache itself is also part of the node's state record (and thus can even be changed manually if needed.)

From a design point of view, the "Smart" policy is really the only interesting one. As will be discussed in this chapter, the kernel provides a number of mechanisms that allow a node to get pretty smart about its cache.

4.1.1 Result/Request dependencies

The simplest scenario of cache use is when identical requests (i.e. with identical rqids) are sent to the same node. If a node has a cached result, it can return that immediately rather than doing a wasteful recalculation.

Real-life trees involve some more sophisticated scenarios. For example, in a solve-tree, a `Solver` node iterates over a tree by issuing a series of requests, with each subsequent request containing updates to parameter values. Some branches of the tree have no dependence on solvable parameters, it is obviously wasteful to recalculate those, so the cache must somehow be used.

Thus, the decision to use cache hinges on knowing the **dependencies** of a particular `Result`. When a node caches a result, it also caches the rqid. In the trivial case, if the next request has the same rqid, the node can immediately return the cached result. In fact the cache can be more discriminating. Each cached result also has a *dependency mask* (or **depmask** for short) that describes *what properties of a request the result depends on*. The depmask is a bitmask, with each bit indicating a particular dependency. Typical dependencies include:

- The request's `Cells` (envelope domain and grid), obviously enough. Example nodes with this dependency: `Parm` (with a non-zero degree polynomial), `Time`, `Freq`.
- Only the envelope domain of the `Cells`. Example: the `Spigot`, since it always returns data at the native resolution of the dataset, ignoring the resolution specified in the `Cells`.
- Updated `Parm` values sent up by a solver.
- The configuration of a `WSum` node.

- Any combination of the above.

If a node has children, then its result's dependencies are almost always the union of the children dependencies, plus (in some cases) additional dependencies introduced by the node itself (e.g. the `UVW` node always adds a dependency on `Cells`). In other words, the depmask of the result is a bitwise-OR of the depmasks of the children's results, OR the node's own local depmask. Obviously, the set of dependencies grows as results propagate down the tree.

Given a cached result and its depmask, a node can be somewhat more discriminating in choosing when to return a cached result. For example, if the depmask indicates that the result depends on `Cells` only, then all further requests with the same `Cells` can be served from the cache. The same applies to other dependencies. In global optimization terms, this means that when a tree is re-evaluated for a slightly different request, it recalculates only those sub-trees that have been updated. The problem is how to determine if a different request has the same `Cells`, without doing a brute-force comparison (which can be quite expensive if done at every node). This is where the *hierarchical* part of request IDs come in.

An rqid is a `HIID` – essentially, a string of integer indices. Each index corresponds to one bit in the depmask. For example, if the depmask is structured as follows:

bit 0	`Parm` values from solver
bit 1	`WSum` configuration
bit 2	resolution of `Cells`
bit 3	envelope domain of `Cells`

then the rqid is composed of four indices:
<domain index>.<resolution index>.<config_index>.<value_index>

The decision whether a new request can be served from the cache becomes quite simple: just compare all indices of the rqid for which the corresponding depmask bit is set, and use the cache only if none of them differ.

In other words, the components of the rqid describe how a request is different from previous requests. The domain index must change whenever a new domain is requested, the config index must change whenever a `WSum` is reconfigured, the value index must change at each solve iteration, etc.

Of course, this scheme only works if the depmasks returned by the nodes (generally, somewhere up the tree), and the request IDs generated by request originators (generally, down the tree) have the same semantics. The depmask/rqid correspondence represents a *contract* between request generators and dependency generators to apply these semantics consistently. The `Node` class provides a number of mechanisms for automatically setting up consistent semantics throughout the tree, see the discussion on *symdeps* below.

The general scheme implemented at the `Node` level does not assume any application-specific semantics at all. The depmask is simply treated as set of N bits, and the rqid as a corresponding set of N indices. All semantics are defined at the application level!

4.1.2 Smart caching behaviour

The previous discussion focused on how depmasks determine whether a cached result can be reused. In a similar vein, depmasks can also help us define a smart caching policy *(Note: this is not yet implemented.)*

Let's assume that the request generator has knowledge of how the next request is going to be different from the current one. This seems a reasonable assumption: a `Sink` always knows that its next request is going to have a different domain/cells, a `Solver` knows that its next request is going to contain updated parameter values, etc. This information can be expressed as a **diffmask**: a bitmask describing which components of the rqid are going to change. The diffmask can be passed along in the request record, and treated as a *hint* by the caching code.

Now, each node returning a result has the following information: its result depmask, the depmasks of its children's results, and a diffmask hint describing the next expected request. The following caching behaviour appears reasonable:

1. If a dependency will change in the next request (`diffmask & depmask != 0`), then the node can clear cache (since the next request will invalidate it anyway).

2. If the parent dependencies are exactly equal to those of a child, then the parent does not need the child's cache (because any request requiring a recalculation of the parent result will also require a recalculation of the child result.) It can then tell the child to release cache. Note that this decision does not require a diffmask hint.

3. If the parent dependencies are larger than those of a child, then the child should retain cache (unless it has already decided to clear it based on the diffmask hint.)

4. A child can clear cache if **all** of its parents have told it to release cache.

   The child/parent interaction implies that regardless of diffmasks, cache only needs to be retained *at those points of the tree where the dependency set grows.*

Note that the diffmask hint is not required for correct operation of the caching system. If the hint is wrong, then the worst that can happen is that either cache is retained or cleared when it shouldn't be. If the hint is missing, the nodes can fall back to a strategy of aggressive caching (i.e. retaining cache at points where the dependency set grows), or no caching at all. The two strategies can be implemented as variations of the "Smart" caching policy.

We have now defined all cache-related behaviour locally: all decisions are made on the basis of depmasks and diffmasks, and no knowledge of the structure of a tree (or the application domain) is required. The next section will describe how depmasks are actually generated.

4.1.3 Cache and the state record

The cache is actually stored in the node state record. The `cache_result` field contains a read-only counted ref to the result. The `cache_result_code` field contains the result code, which includes the depmask (see 5.2). The `request_id` field is the rqid of the original request.

The node cache may be cleared by assigning a boolean `false` to the `cache_result` field. It is also possible to modify the cache on the fly (for example, substituting in another result). This may be a useful feature for debugging and experimentation, though if abused, it can thoroughly confuse the caching and dependency tracking mechanism.

4.2 The local depmask

The `depend_mask` field of the state record defines the local depmask of the node. The local depmask indicates which dependencies a node introduces into its result. For most `Function`-derived nodes, this mask is null, indicating that the result dependencies are fully determined by child dependencies. Leaf nodes, on the other hand, will generally have non-null masks.

The depmask is just a set of N bits with no specific semantics associated with them. The association between individual bits and specific result properties is set up via the mechanism of *symbolic dependencies*, or *symdeps* for short.

4.3 Symdeps in a nutshell

A *symdep* is a `HIID` (thus, symbolic – since `HIID`s have a symbolic representation) that identifies some application-specific dependency of the result. Node classes will typically define some standard symdeps, such as this set used in the standard nodes:

"Domain": result depends on the requested domain (i.e., the envelope domain of the `Cells`). Most non-trivial leaf nodes have this symdep.

"Resolution": result depends on the resolution of the `Cells`. Most nodes with a time and/or frequency dependence have this symdep.

"Parm.Value": result depends on parameter values passed up from the solver. Solvable `Parm`s have this symdep.

A node's set of symdeps is generally known to the node class at construction time. Then, when a tree is initialized, different symdeps are dynamically associated with different bits of the depmask, as described below. Essentially, this maps abstract concepts (the symdeps) onto specific bits of the depmask. In other words, this mechanism defines the bitmask semantics.

4.3.1 Symdep masks

Note that certain nodes can be viewed as symdep *generators*. These are nodes that generate new requests. For example, the `Sink` node generates requests with different domains and resolutions, thus we say that `Sink` generates the `"Domain"` and `"Resolution"` symdep. The `ModRes` node changes the resolution of requests, thus it generates the `"Resolution"` symdep. The `Solver` node generates the `"Parm.Value"` symdep.

These nodes are responsible for associating a particular bit of the depmask with each symdep that they generate. Typically, the association is fixed when a tree is initialized. These associations (known as *symdep masks*) are then recursively sent up the tree, thus becoming known to all child nodes. Nodes up the tree can then compute their local depmasks by combining the symdep masks of their specific symdeps.

4.4 Symdeps: the hairy details

This section describes the details of how symdeps and depmasks are set up and maintained. Note that all of this is maintained in the node state record.

4.4.1 Known and active symdeps

The *known symdeps* of a node are just that, all the symdeps that a node (typically, its class) knows about. This is usually specified once and for all in the node's constructor, by calling the `setKnownSymDeps()` method.

A subset of the known symdeps – the *active symdeps* set – determines what symdeps currently apply to the node's result. For some node classes, this is always the entire known set. Other classes, however, may change their active set depending on state. For example, if a `Constant` node is configured to provide a constant value as a sampling, then it has no active symdeps at all, as the value will be the same for any domain or resolution. However, if it is (re)configured to provide the constant as an integration, then it begins to depend on resolution – since the integrated value is the product of the value by cell size.

The active symdeps set may be changed by calling the `setActiveSymDeps()` method, or by changing the `active_symdeps` field (a list of `HIID`s) of the state record. Whenever this is done, the local depmask is automatically recalculated using the known symdep masks, by calling the virtual `resetDependMasks()` method.

The known symdeps may also be changed at any time (though I can hardly see why anyone would want to do this), by calling the `setKnownSymDeps()` method, or by changing the `known_symdeps` field of the state record.

4.4.2 Propagating symdep masks

A node will automatically keep track of the symdep masks associated with its known symdep set. These are sent up as commands in a request (see 5.3), and processed at the `Node` class level.

- The `Add.Dep.Mask` command contains a map of symdeps to symdep masks. (This command usually originates at the symdep generator nodes, see 4.4.3 below). In response to this command, `Node` adds all the masks it finds for its known symdeps to its internal map of symdep masks. After this, it calls `resetDependMasks()` to recalculate its local depmask.
- The `Clear.Dep.Mask` command clears all known symdep masks.

One consequence of this design is that each node maintains its own local mapping of symdeps to depmasks. While at first glance this may seem redundant and even wasteful – since the mapping would appear to be the same throughout a tree – consider the following points:

- When a tree is distributed throughout a cluster, maintaining a single "global" map becomes difficult (and actually violates the principle of locality!) Keeping a copy of the map at each node avoids this problem.
- The map is not really global anyway. For example, consider a rippled tree with multiple solvers. The solvable parm set of solver 1 and the solvable parm set of solver 2 need to be represented by different bits in the depmask. Thus, the `"Parm.Value"` symdep of different groups of parms will actually be mapped to different depmasks!

  Note that node group facility (see 5.3) provides an elegant mechanism for distributing different symdep masks to different node groups.

- The map is small, and changes very infrequently (if at all – usually, it will be set up only once when a tree is initialized). Thus there is really no performance cost associated with keeping local copies.

The map of known symdep masks is maintained in the `symdep_masks` field of the node state record. It is possible to change this field on the fly, an automatic call to `resetDependMasks()` always results.

4.4.3 Generating symdep masks

Nodes that generate new requests also need to generate `Add.Dep.Mask` commands containing their symdep masks assignments. `Node` provides a simple facility for doing this automatically.

A node class can specify its mapping of *generated symdeps* to masks by calling the `setGenSymDeps()` method at construction time. The standard node classes specify some pre-assigned masks by default: bit 0 for `"Parm.Value"`, bit 1 for `"Resolution"`, bit 4 for `"Domain"`. This allows for simple trees to be put together using just the defaults.

For more elaborate trees – e.g., rippled trees with multiple solvers – different `Solver` nodes need to be assigned different masks for the same symdep. This can be done via the `gen_symdeps` field of the state record. This field has to contain a map (e.g. record) of symdeps to depmasks; if specified, it overrides any previously set mappings.

Additionally, a node may be configured to generate symdeps for a particular node group only. This useful for, e.g., multiple solvers. By default, the `All` group is used, but this can be overridden via the `gen_symdeps_group` field of the state record.

Once the generated symdeps are configured, a node needs to be told to send them up to its children. This is done by passing it the `Init.Dep.Masks` command via a request rider. In responce to this command, `Node::processCommands` inserts the appropriate `Add.Dep.Mask` commands into the request rider based on the current setting of `gen_symdeps` and `gen_symdeps_group`.

Operationally, all this is typically done once at init time:

- Request generator nodes are created with an initial state record containing their generated symdep assignments (if the default assignments need to be overridden).
- Once all the nodes and trees have been created, a request containing the following two commands is given to all the root nodes: `Resolve.Children` and `Init.Dep.Masks`. (The first command is required to resolve named children.) This is done by `MeqServer` automatically, in response to the `Resolve.Children` command.
- This initial request is propagated up the tree, accumulating symdep masks from other generator nodes along the way.
- All trees are now ready for use.

4.4.4 Order of state updates

Because it is possible to change everything about a node's symdeps and masks by modifying the state record, and a state update may contain multiple fields, the order in which it is updated becomes important. `Node::setStateImpl()` employs the following order:

1. Updates the known symdep set from `known_symdeps`, if specified.
2. If specified, reloads the symdep masks from `symdep_masks`, and calls `resetDependMasks()`.
3. If specified, sets the active symdeps from `active_symdeps`, and calls `resetDependMasks()`.
4. If specified, sets the local depmask from `depend_mask`.

Basically, this order implies that if the local depmask is explicitly changed via `depend_mask`, the specified value overrides any implicit value calculated from, e.g., an `active_symdeps`.

4.4.5 Specialized node behaviour

Note that all these facilities are provided at the basic `Node` level, but special node classes are free to ignore them, or make use of only some subset. For example, the `Parm` class[1] needs to maintain separate predict symdeps and solve symdep sets. It deals with this in the following way:

- It ignores the `Node`-level active symdeps, initially specifying an empty set of active symdeps. This implies that its `Node`-level local depmask stays at zero.
- It maintains two of its own symdep sets: predict symdeps and solve symdeps, and two corresponding depmasks: the predict depmask and the solve depmask.
- It overrides the `resetDependMasks()` method, to recompute the two depmasks whenever known symdep masks change.
- It returns either one or the other mask from `getResult()`, depending on what sort of request is being serviced.

[1] And currently, this is the only example.

Chapter 5

Executing Requests

The virtual `Node::execute()` method is responsible for processing a `Request`:

```
virtual int execute (Result::Ref &resref,const Request &req);
```

The node is supplied a `Request` object, and it is expected to return a `Result` by attaching it to the counted ref passed in as the first argument. The return value is called the **result code**: this incorporates the depmask of the result, plus several additional flags such as `RES_WAIT` and `RES_FAIL` (see below).

A parent node will generally call the `execute()` methods of its children. In the current single-threaded implementation, the entire tree is evaluated via nested `execute()` calls. In a multi-threaded or distributed implementation, the parent will probably call stub methods in the communication layer, which will in turn call `execute()` on the child nodes.

The `Node::execute()` method is the fulcrum of the entire MeqTree kernel. A solid understanding of how it works is vital for both tree design and node development, and also beneficial to advanced users that need to deal directly with trees.

5.1 Node::execute() steps

The `Node::execute()` method looks at the request, and calls a number of virtual *handler* methods for various aspects of processing. Most node classes will override one or more of these handler method(s) to implement their specific node behaviour. `Node::execute()` also provides fundamental node functionality, such as cache management and exception handling.[1]

The base `Node::execute()` performs a number of processing steps. These will now be described in detail, grouped by function.

5.1.1 Checking the cache

Init return code. Set the current return code to 0. It will be accumulated in further steps via a bitwise-OR operation.

Step 1. Compare the request id to the previous rqid, if any. Set a local "new request" flag for the benefit of further logic below.

Step 2. If there's a cached `Result`, and the request id matches the cached rqid/depmask (see 4.1 for a discussion), immediately return the cached `Result` and cached result code. On mismatch, clear the cache and proceed.

Note that the caching policy also determines how fail-results are dealt with. If the cache contains a fail-result, the node may choose to ignore it and attempt to recalculate the result to see if the fail conditions have gone away.

[1] You may have noted that `execute()` is declared virtual. Most node classes will only redefine specific handler methods, not `execute()` itself. The possibility to reimplement `execute()` is reserved for the exotic cases. This is not to be undertaken lightly, however, as the base `Node::execute()` provides so much useful behaviour.

Step 3. For new requests only: call the virtual `readyForRequest()` handler, and if the return value of that is `false`, immediately return the code `RES_WAIT` (result will be empty).

```
virtual bool readyForRequest (const Request &req);
```

The handler is passed the current `Request`, i.e., the `req` argument to `execute()` itself. The purpose of this handler is to support nodes that block on external events. None as such have been implemented, so this is currently just a placeholder. The default handler always returns `true`.

Step 4. For new requests only: if a rider subrecord is present, parse it and call the virtual `processCommands()` handler to process commands targeted at the node (see details in 5.3).

5.1.2 Polling children

Step 5. If node has children, call the virtual `pollChildren()` handler to pass the request on to the children and collect their results. Bitwise-OR the return value of `pollChildren()` into the current return code.

```
virtual int pollChildren (std::vector<Result::Ref> &child_results,
                          Result::Ref &resref,
                          const Request &req);
```

The handler is called with the same `resref` and `req` arguments that were given to `execute()` itself. Child results should be returned via the `child_results`, which is pre-sized to the number of children prior to calling `pollChildren()`.

The default implementation of `pollChildren()` is appropriate for most node classes that pass their requests on to the children unmodified. "Control" nodes (e.g. `Sink`, `Solver`, `ModRes`, `ReqSeq`) will define their own version. This is the default `Node::pollChildren()` behaviour:

- Calls `execute()` (with `req`) on all the child nodes, collects their `Results` (by ref) into the `child_results` vector, and accumulates a return code as a bitwise-OR of the children's `execute()` return values. Note that the refs to child `Results` are expected to be read-only.
- If the accumulated return code has the `RES_FAIL` bit set, its is assumed that at least one of the children has returned a fail-result (see 2.3). In this case, `pollChildren()` creates a new fail-`Result` object, attaches it to `resref`, and fills it with all the fail-records found in child results.
- The accumulated return code is the return value of the handler.

Note that `resref` is passed to `pollChildren()` only as a means of reporting possible fails. If the handler returns a code with `RES_FAIL` in it, it should attach a fail-result to `resref`. If no `RES_FAIL` is reported, the handler should leave `resref` alone (in fact, anything it does to it will be simply ignored when no `RES_FAIL` is returned.)

If the `pollChildren()` return value contains `RES_WAIT` or `RES_FAIL`, `execute()` returns (see section 5.1.5). Otherwise, it proceeds to the next step:

Step 6. If auto-resampling is enabled (see 5.4), compare the resolutions of the child results, figure out a common resolution (`Cells`) to resample them to, and perform the resampling. Throw an exception if this is not possible.

Note that if the child results do not contain any `VellSets` (which will be the case when the `Request` does not have a `cells` command), this step is skipped.

A result `Cells` may be initialized based on how the resampling went (see 5.4).

5.1.3 Evaluating `cells`

Step 7. If the request contains a `cells` command (with a `Cells` object), call the virtual `getResult()` handler to process the command, passing in the vector of child `Results` returned by `pollChildren()`. The return value of `getResult()`, along with the node's current depmask (see 4.2), is bitwise-ORed into the current return code.

```
virtual int getResult (Result::Ref &resref,
                       const Cells::Ref &cells,
                       const std::vector<Result::Ref> &child_results,
                       const Request &req,bool newreq);
```

The `resref` and `req` arguments are the same as those passed to `execute()`. The `cells` argument is a ref to the result cells, if any were initialized during the resampling stage above, or otherwise to the `Cells` in the request (see 5.4 for details). The `child_results` vector is built up in `pollChildren()`, it will be empty if the node has no children. Finally, the `newreq` flag indicates if it is a new request, this flag is set in Step 1 above.

The `getResult()` handler is responsible for attaching a `Result` object to `resref`. In most cases, it will create a new object. Note, however, that certain nodes may pass on child results transparently (e.g., `ModRes`, `ReqSeq`), they do this by simply copying a ref from the `child_results` vector.

If `getResult()` returns a `RES_WAIT` code, it is allowed (and expected) to leave `resref` unattached. Otherwise, a valid `Result` must be provided! Any errors occurring inside `getResult()` can be reported by throwing an exception.

5.1.4 Handling exceptions

Error handling. If an exception is thrown at any stage of the process, `execute()` will catch it, create an output `Result` with a fail-result describing the exception, and add return the current return code, with a `RES_FAIL` flag added in.

Thus, throwing an exception is the normal way for `processCommands()`, `getResult()`, or any other handler to indicate a failure. Note that a node should remain in a usable state (i.e. should be able to process further `Requests`) after most exceptions; methods that are liable to leave the node in a non-usable state should provide their own exception handling code that performs the necessary cleanups and re-throws the exceptions. See the `setState()` rollback mechanism described in section 3.4.5 for one such example.

5.1.5 Caching and returning a `Result`

Returning. Whenever any kind of `Result` is returned, it is stored in the cache according to the current policy, and returned via the `resref` argument. The accumulated return code is returned along with the result. If the result is new (i.e. not returned from cache at step 2), the `RES_UPDATED` flag is added to the return value.

Note that if the accumulated return code contains `RES_WAIT`, then no `Result` is expected (`resref` remains unattached). In all other cases, a valid `Result` object should be attached to `resref` (in case of exceptions, this will be a fail-result).

5.1.6 All Results are read-only!

Prior to returning a `Result`, `execute()` recursively changes `resref` and all other refs found inside the result object to read-only. This implies that the caller of `execute()` (i.e. the parent node, or `MeqServer`) cannot [legally] change the `Result` contents. This is deliberate – the node can now hold a ref to the `Result` in the cache, and be assured that no-one can [legally] change the contents.

In most cases, parent nodes will process the `Results` of their children as read-only, and discard them afterwards (actually, only the parents' refs are discarded – the `Result` objects themselves persist if still referenced somewhere, e.g., in a child's cache). Some nodes may want to modify child `Results` "in place". To do this, they will need to privatize their refs for writing first (see 3.4.1), thus ensuring that a private copy is made if the same object is still referenced somewhere. The same applies to individual components of the `Results`. Essentially, this is a robust copy-on-write mechanism that assures that data is duplicated only when needed, with very little effort required from the node developer.

Note also that when a `Result` is moved across to the scripting layer, some sort of copying is implicitly performed. Glish does not deal with `Result` objects directly, but rather with their Glish representations.

5.2 Result codes

As described above, the return value of `execute()` is simply the accumulated result code. The result code describes certain properties of the returned `Result`. Part of it is simply the result depmask (see 4.1.1, the other part contains a number of bitflags listed below. Note that the property semantics are defined in such a way that, in most practical cases, a flagged property in any child result is inherited by the parent's result. This allows `execute()` to accumulate the correct result code via a simple bitwise-OR. The following additional flags are defined:

`RES_UPDATED`: result has changed from that of previous `Request`. This bit is usually cleared when the node returns a result from the cache, and set otherwise.

`RES_VOLATILE`: result may change in response to external events, even without new requests. *(This is not implemented for now, and only meant as a placeholder for future developments, such as dynamically growing domains, partial integration, etc.)*

`RES_FAIL`: result is a fail. Note that this is not the same thing as a result containing some mix of valid and failed VellSets; rather, this indicates a failure for the whole result overall. `RES_FAIL`s are usually generated in an exception handler. Note that the default implementation of `pollChildren()` and `execute()` causes fails to cascade down the tree.

When this flag is returned, a `Result` object is still expected; it should contain one or more fails describing the error. Note that depmasks can be meaningfully combined with `RES_FAIL`, to indicate that the fail depends on something (i.e. may go away if a particular dependency changes). The "smart" caching policy may make use of this.

`RES_WAIT`: no result available, wait for notification or try later. If this flag is raised, then no `Result` should be returned. Note that dependency flags can be meaningfully combined with `RES_WAIT`, since it usually possible to indicate the dependencies of a node in advance. No current code uses this, however.

The return value of a node's `getResult()` method should describe any **additional** properties introduced by the `getResult()` calculation (additional with respect to the node's current depmask – see 4.2). Most function nodes will return zero, indicating no additional dependencies.

5.3 Commands in request riders

The optional `rider` sub-record of a `Request` is used to send commands to nodes. The `Solver` node, for example, relies on this feature to send up parameter updates during iterative solutions. Commands are specified as sets. Each **command set** is a record (`DataRecord` in C++), with the field names (i.e. HIIDs) being the commands per se, and the field value being the command value a.k.a. arguments. Commands with no arguments are indicated with a boolean `true`, a boolean `false` value implies that the command is **not** issued.[2] If the set contains multiple commands, they are processed in a specific order determined by the node class. Some example commands are:

`state`: changes the state of a node (available for all nodes);

`add_dep_mask`: adds symdep masks (available for all nodes, discussed in 4.3).

`update_values`: incremental updates of solvable parameters (see documentation for the `Solver` and `Parm` nodes).

Since commands are seen as record fields in Glish, and plain HIIDs in C++, we will use both forms (`foo_bar` and `"Foo.Bar"`) interchangably.

[2] This may seem redundant – why not simply omit a command if it is not meant to be issued? Consider though that Glish has no built-in facility for *removing* record fields. The "false=no command" convention allows one to effectively remove a command from an existing set by assigning the `F` value to it.

5.3.1 The command handler

Commands are processed by a virtual handler method, called from `execute()`:

```
virtual void processCommands (const DataRecord &rec,Request::Ref &reqref);
```

The `rec` argument is the command set to be processed (see below). The `reqref` argument is a ref to the current `Request`. Because the `processCommands()` handler is called before `pollChildren()`, it can modify the request (i.e., add commands to it) before it is passed on to the children. `Node` itself, for example, uses this capability when accumulating symdep masks (see 4.3). The `reqref` should be privatized for writing before a `Request` is modified, because requests are normally passed around as read-only.

Node classes implementing their own commands will need to redefine this handler. It is important, however, that the new handler calls the parent class's handler first, so that commands implemented in superclasses (especially `Node` itself) are handled properly. Unknown commands should be ignored – in fact, handlers are usually implemented to simply look up known commands, and ignore the rest.

Any errors arising during command processing may be indicated by throwing exceptions (these will be caught by `execute()`). The handler should take care to perform any cleanup, and leave the node in a usable state.

The base handler, `Node::processCommands()`, processes all standard `Node`-level commands. These are listed in section 5.3.4.

5.3.2 Rider subrecord layout

The rider record is structured so that it is possible to associate a command set with a specific node or a set of nodes. Each node thus checks if the request contains any commands for itself. Note that in a very large tree, such repeated checks may grow quite expensive. For this reason, nodes that need to receive a lot of commands should be assigned to a *node group*.

A node may be associated with one or more groups. Each group is identified by a `HIID`, a node's group assignments are a part of its dynamic state (see 3.2). All nodes implicitly belong to the `all` group. A node's groups are used as a first-level index into the `rider` record. If a node belongs to groups `foo` and `bar`, then `Node::execute()` will check for the *command subrecords* (**CSR**s) `rider.foo`, `rider.bar`, and `rider.all`, in that order, and process any CSRs that it finds.

Judicious use of node groups practically eliminates the overhead of command lookup. The rider itself contains very few fields (if any), so looking up the CSR for a group is very fast. If your tree frequently uses commands to control a small subset of nodes (e.g., a solve-tree uses commands to update solvable parameters), these nodes should be placed in a group of their own, and commands should be kept in that group's CSR. All other nodes will only check for the `all` CSR – which is usually not present. Thus, only the required subset of nodes will engage in the possibly expensive business of CSR parsing and command processing.

The `all` CSR allows for commands to be sent to any node; but because of the associated overhead, this should only be used for infrequent operations (e.g., reconfiguring a tree – as opposed to iterative solving).

CSR layout

Each CSR contains a number of command sets. These command sets may be associated with specific nodes. There are three ways to specify these associations (we will use `csr` here to refer to the CSR itself):

All nodes in group: The command set contained in the field `csr.command_all` applies to all nodes in the group. For example:

```
- req.rider.foo.command_all
[ save_polc=T,state=[solvable=F] ]
```

...will call `processCommands()` on all nodes in group `foo`, with the command set `[save_polc=T,state=[solvable=F]` (meaning save polcs, and set to non-solvable).

Via node index: Command sets may be associated with a node index. This is done via `csr.command_by_nodeindex`, which is essentially a map from node index to command set. For example:

```
- req.rider.foo.command_by_nodeindex
[ #19 = [ value=1,save_polc=T ], #41 = [ value=2,save_polc=T ] ]
```

...will cause a `processCommands()` call on nodes 19 and 41, but only if these nodes actually belong to group `foo`. (Note that since Glish only supports strings for record field names, the *'#ddd'* form is used to specify a "numeric" node index.)

Via lists: The third way is to associate command sets with lists of nodes matched by name or node index. This is done via a list of records in `csr.command_by_list`. For example:

```
- req.rider.foo.command_by_list
[ #1 = [ name="RA DEC",nodeindex=[17,32],
         command=[ save_polc=T,state=[solvable=F] ]
       ],
  #2 = [ command=[ state=[solvable=F] ] ]
]
```

...will call `processCommands()` with the first command set on nodes 'RA', 'DEC', 17 and 32 (if they belong to group `foo`), and with the second command set on all other nodes in group `foo`. To be more specific, the `command_by_list` field is treated as a list of records. Each record in the list must contain a `command` field (the command set itself), plus an optional `name` field (string or list of strings) and/or an optional `nodeindex` field (integer or list of integers). `Node::execute()` will iterate through the list of records one by one; if the node's name is found in `name`[3], or the node index is found in `nodeindex`, then `processCommands()` is called with the contents of `command`. Once a match is found, list processing stops. As a special case, if neither `name` nor `index` is specified, then the entry is a "wildcard" matching any node. Wilcards are only useful at the end of the list, to catch nodes not matched by previous entries.

5.3.3 Command evaluation order

As you can see, the rider allowss for more than one command set per node. In this case, `processCommands()` will be called several times, once for each set. It is important then to define the order in which the rider is parsed:

- The outer loop is over node groups, in the order in which they are specified in the node's state record. The `all` group is checked last.
- Within a group's CSR, the processing order is from least specific to most specific: `command_all`, `command_by_list`, `command_by_nodeindex`.
- Once a command set is passed to a node's `processCommands()` method, the order of processing is determined by the node class implementation. Generally, a subclass should call its parent's `processCommands()` first, so general commands (such as `state`) will be processed before more class-specific commands (such as `save_polc`).

5.3.4 Standard node commands

The following is a list of standard commands implemented at the `Node` level. Commands are listed in the form of Glish record field names. The actual order of processing is the same one as given here:

resolve_children: resolves all children specified by name into actual nodes. An exception is thrown if a name does not match any known node. This command is normally issued by `MeqServer` at initialization time, in responce to a `"Resolve.Children"` request. This command has no value.

state: calls `setState()` (see 3.4.5) with the command value. This command thus allows for any field of the dynamic state to be modified.

clear_dep_mask: clears accumulated symdep masks (see 4.3). This command has no value.

add_dep_mask: accumulates symdep masks (see 4.3). The command value is a record of symdep:mask pairs.

init_dep_mask: if the node is a dependency generator (see 4.3), this command causes it to add its generated symdep masks to the request, as an `add_dep_mask` command placed into the `command_all` field of the CSR for the configured gendep group.

[3] In the future, pattern matching will be supported as well.

5.3.5 Building up command riders in Glish

In Glish, the command rider is simply a part of the `request` record, and may be created and manipulated just like any other [sub]record. In addition, `meq/meqtypes.g` defines some handy shortcuts for adding commands to a request:

```
const meq.add_command := function (ref req,group,node,command,value=T);
```

This function modifies the request object passed in via the `req` argument. A CSR for `group` will be initialized if required, and a `command` with the specified `value` added to the appropriate field of the CSR. The `node` argument specifies the target node(s) as follows:

- an empty list (i.e. `[]`) targets command at all nodes. The command is added to the `command_all` set.
- a single integer is treated as a node index. The command is added to `command_by_nodeindex` (initializing a command set if necessary).
- a list of integers is treated as node indices. The command is added to `command_by_list`, with a `nodeindex` key.
- a string or a list of strings is treated as node name(s).The command is added to `command_by_list`, with a `name` key.
- an empty string array (`""`) adds a wildcard entry to `command_by_list`.

A second shortcut is handy for inserting a state update command:

```
const meq.add_state := function (ref req,group,node,state);
```

This will add a `state` command, with the supplied `state` record as its value. All other arguments have the same meaning as for `add_command()`.

5.4 Resolution & resampling

Resolution & gridding is a complicated business. Most nodes expect their children's results to have the same resolution (i.e. same `Cells`), yet performance constraints require our trees to operate at different resolutions. `Node` implements a flexible mechanism for automatically controlling the resolution of child results. Before going into detail, it helps to take a step back and review some basic requirements:

- In the first instance, the grid/resolution is determined by incoming data. We'll call this the **full resolution** grid.
- It may be prohibitively expensive to evaluate some subtrees (e.g. predict) at full resolution. Thus we should support going from full resolution to **reduced resolution** (**integration**) and back (**upsampling**).
- Parent nodes will not always have sufficient information to determine what the best resolution for a child is (i.e. the optimum predict resolution may perhaps be only determinable within the predict subtree). Thus, a child should be able to return data at any resolution is deems fit, and let the parent deal with it.

5.4.1 Treatment of resolution

To satisfy these requirements, the following behaviour w.r.t. resolution is implemented:

1. The **requested resolution** (or grid) is defined by the `Cells` object supplied with the `cells` command of a `Request`.
2. The requested resolution is merely a hint! A node is not obligated to honor the grid of the `Cells`. It must, however, honor the envelope domain. The gridding of the result is indicated by the `Cells` returned in the `Result` object.

3. Some leaf nodes (e.g. `Const` and `Parm` with no time-frequency dependence, when configured to return samplings and not integrations) return resolution-free `Result`s. This is indicated by a missing `Cells` in the `Result`.

4. Other leaf nodes (e.g. `Parm`, `Freq`, `Time` – generally, nodes meant to evaluate some analytic function over the domain) are easily able to evaluate themselves over any given grid. These nodes will always return a `Result` at the requested resolution; the tree designer may rely on this behaviour as it is part of the contract for those nodes.

5. Data-driven leaf nodes (e.g. `Spigot`) completely ignore the requested resolution. The gridding of their `Result` is fully determined by the layout of incoming data.

6. Most non-leaf nodes with multiple children (e.g., most of the `Function`-derived nodes) can only operate on child results of the same resolution. In the event that different resolutions are returned, any node can be configured to **auto-resample** child results to a common resolution. This is done by `Node::execute()` prior to calling `getResult()`.

7. Some special nodes (such as `ModRes`) can modify the resolution in the request before passing it up.

Note each node's treatment of the requested resolution (i.e. honor/ignore/modify) is part of the node class contract, and should be clearly documented in the Node Reference. Thus, while parent nodes have no control over (or knowledge of) what resolution their children are going to use for their results, the tree designer can always see the global picture, and determine which resolution is used where in the tree. By enabling auto-resampling at key points in the tree, and possibly adding some `ModRes` nodes, the designer can always ensure predictable results.

The brute-force approach (as opposed to careful tree design) is to enable auto-resampling at all nodes. This, however, is a waste of CPU, and is never actually necessary.

5.4.2 Selecting auto-resampling modes

Auto-resampling is enabled via the node state record, on a node-by-node basis. The actual resampling is performed in `Node::execute()`, and it can follow one of several strategies:

Upsample: Find the highest resolution among child results, upsample all other results to it.

Integrate: Find the lowest resolution among child results, integrate all other results to it.

Use Resolution Driver: Resample everything to the resolution of a specific child (this child is called the *resolution driver*).

Use Request: Resample everything to the resolution of the `Request`. If this strategy is selected, the node is guaranteed to honor the requested resolution. If all child results have the same resolution, but it is different from the requested one, resampling is performed.

Fail: Fail outright (return a fail-result, that is) if resolutions differ. This option is mainly useful for debugging; in a properly designed tree, any node that can potentially encounter this situation should be configured to use one of the other strategies.

Note that when selecting a high or low resolution for integration or upsampling, resolution-free results are ignored, since they never require resampling.

The auto-resampling strategy is determined by two fields in the node state record (note that this is dynamic state). The `resample_child_index` field enables the "Use Resolution Driver" strategy, if set to an ordinal child number (≥ 1 in Glish, ≥ 0 in C++ – note the `index` suffix and remember automatic 0-1-base conversion discussed in section 2.1.6) or a `HIID` child label (see 3.2.2). The specified child becomes the resolution driver. If the field is set to boolean `false`, or an out-of-bounds number (< 1 in Glish, < 0 in C++), then this resampling strategy is not enabled.

If the resolution driver returns a resolution-free result (see 2.4.4), then a fail is reported. This situtation indicates an error in tree design, since resolution-free results are returned by only a few nodes, under well-defined circumstances.

If a resolution driver is not enabled, then the `auto_resample` field can define another resampling strategy. It can be set to one of the following values:

NONE (= 0)**:** do nothing, do not even check the child resolutions. This is the default setting initialized by the `Node` class constructor.

FAIL (= −2)**:** check resolutions and return a fail-result if they do not match. This is the default setting initialized by the `Function` class constructor. Since this is somewhat slower that the `NONE` setting, the default for optimized builds may possibly be changed to `NONE` in the future.

INTEGRATE (= −1)**:** select the "Integrate" strategy.

UPSAMPLE (= 1)**:** select the "Upsample" strategy.

REQUEST (= 2)**:** select the "Use Request" strategy.

The `auto_resample` and `resample_child_index` fields are mutually exclusive. If a resolution driver is selected, `auto_resample` will be automatically reset to "none".

5.4.3 A working example

Figure 5.1 shows an example of a tree that operates at different resolutions – full data resolution for "fast" operations such as phase shift and subtract, and reduced resolution for predict. The reduction of resolution is controlled by `ModRes` nodes (see below). Nodes for which resampling is enabled are indicated by a red label in the node box: "UPS" for upsample, "INT" for integrate, "RES/DR" for using a resolution driver. The predict branch is evaluated at a (presumably reduced) resolution determined by its root `ModRes` node, this is visually indicated by a coarser grid background. The `CondEq` node integrates its child results to the lower of the two resolutions: this is appropriate for solving. The `Subtract` node uses original data resolution as the driver.

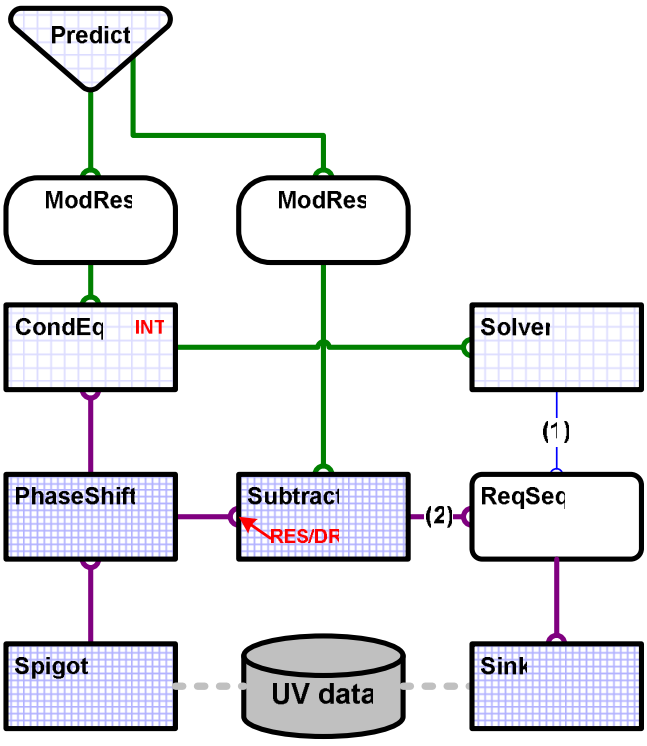

Figure 5.1: A multiple-resoulution tree employing auto-resampling

Implementation issues

For the time being, `Node` only supports integral resamplings. In other words, the higher-resolution cells must be perfect tilings of the lower-resolution cells. A fail-result is generated otherwise. This keeps things simple, and avoids interpolation errors.[4]

[4]In the future, we may add support support arbitrary regridding if needed.

This limitation is less serious than may appear. Resolutions are never chosen randomly – from a tree designer's point of view, they are completely deterministic. Current facilities for resolution control (the `ModRes` node mentioned below) only allow for integral resolutions. As for the original data resolution, it always comes in from the data access nodes (`Sink` and `Spigot`), which are tightly coupled: for each snippet of data, the `Sink` issues a `Request` with a `Cells` corresponding to the data layout, while the corresponding `Spigot` is then ready to return a `Result` with the same cells. Thus, the tree designer can always define working resolutions for subtrees in terms of integral factors of the original data resolution.

Upsampling currently uses linear interpolation. In the future, we may add support for more sophisticated resampling.

5.4.4 Controlling the resolution

The `ModRes` utility node may be used to change resolution mid-tree. A `ModRes` always has a single child; upon receiving a `Request` from its parent, it modifies its resolution up or down by a fixed factor (independent for each axis), as determined by its node state. The modified `Request` is passed on to its child, and the child's `Result` is returned directly to the parent.

In the future, we envision an adaptive resolution reducer (ARR) node, which adaptively selects a minimum required resolution based on some application-specific considerations. These may include:

- For solving, the predict tree need only supply enough cells to constrain the solution. The minimum resolution can be determined by the `Solver` node, and passed along in the `Request` as a hint to any ARR nodes up the tree.
- For subtracting predicted sources, the cells need only be big enough to ensure linearity over a cell.

Note also that the `Identity` node, when configured with a "Use Request" resampling strategy, becomes a *resolution coupler*, ensuring resampling all results to the requested resolution.

5.4.5 How this applies to getResult()

Note that the `getResult()` handler discussed above receives two crucial pieces of information: a vector of child results, and a `Cells` (passed by ref). The `Cells` parameter defines the expected result cells, and it is set up as follows:

- If auto-resampling is not enabled, or the node has no children, then this is simply the cells from the `Request`. The resolutions of child results, if any, are not constrained.

  Note that most nodes' contracts specify that "this node only deals with child results at the same resolution" (this is true, e.g., for all `Function` nodes), and leave it up to the tree designer to ensure it (i.e. by enabling auto-resampling where necessary). The node's behaviour when a contract is violated does not need to be defined (although one can reasonably expect an exception to be thrown, leading to a fail-result).
- If auto-resampling is enabled, then the resolutions of child results are constrained: they will all have the indicated `Cells` (with the exception of resolution-free results, which have no cells). Most nodes are expected to return a result at the same resolution.
- If auto-resampling is enabled and all child results are resolution-free, then the `Cells` ref is empty. In this case, most nodes' result will should be resolution-free as well. In the rare case where the node introduces its own time-frequency dependence (e.g. the `UVW` node), it should get a `Cells` from the request.

5.5 Function nodes

`Function` is an important subclass of `Node`, representing some function of its children's results. The main feature of `Function` is an abstract `evaluate()` method, which takes a vector of `Vells` objects, computes some function over them, and returns a `Vells` result.

Remember that a `Vells` is a an $N \times M$ matrix ($\bar{f} = \{f_{ij}\}$) of $\mathcal{R}$ or $\mathcal{C}$ values, representing a set of sampling or integrations of a function $f : \mathcal{R}^2 \rightarrow \mathcal{R}$ or $f : \mathcal{R}^2 \rightarrow \mathcal{C}$ over some cells. `Vells` are organized into `VellSets`, containing the main value $\{f_{ij}\}$, plus an optional set of $K \times S$ ($S = 1, 2$) perturbed values $\bar{f}_{sk} = \{f_{ij}^{(sk)}\}$, which

correspond to S sets of perturbations w.r.t. K solvable parameters. A `VellSet` will also contain S vectors of the perturbations themselves, $\{\delta_k^{(s)}\}$.

The `Result` of child l will contain Q_l `VellSet`s. Most of the time, $Q_l = 1$; when $Q_l > 1$, the result represents a multidimensional function $f : \mathcal{R}^2 \rightarrow \{\mathcal{R} \text{ or } \mathcal{C}\}^{Q_l}$. Note that each plane of such a function may have its own set of solvable parameters and perturbed values. We will use the notation $\bar{f}_{lq}$ to designate the main value for child l, plane q, and $\bar{f}_{lq}^{(sk)}$ to designate the corresponding perturbed value for parameter k from set s.

The `evaluate()` method of a `Function`-derived node implements some compound function of L child `Vells` defined over the same `Cells`. We will designate this function as $F = F(\bar{f}_1, ..., \bar{f}_L)$. The value of this function is a `Vells` itself, $\bar{F} = \{F_{ij}\}$, also defined over the same `Cells`.

The `Function` class provides a `getResult()` method that iterates over all `Vells` in its child results, calls `evaluate()` to compute F, and collect the resulting `Vells` into an output `Result`. The same operation is done for all perturbed values. Thus, the `Function` class implements a basis for all nodes that calculate a compound function of their children. PSS4 includes an automatic code generator (documented elsewhere) that will generate a full C++ implementation of a `Function`-derived node class, given a symbolic expression for the F function. The kernel also provides a toolkit of `Function`s that implement most primitive mathematical operations.

5.5.1 Dealing with multiple planes

Note that an implementation of `evaluate()` defines F for a single set of `Vells`. When child results contain multiple planes, `Function` deals with it as follows:

- If all child results contain the same number of planes Q ($Q = Q_1 = ... = Q_L$), the function F is applied separately to each plane $q = 1...Q$.
- Child results with a different number of planes are supported in only one scenario: some results may have 1 plane, and some may have $Q > 1$ planes, but Q must be the same for all multi-planar results (a fail is generated if this condition is not met). Results with 1 plane are then artificially expanded to Q planes by reusing the same `VellSet` for all planes from 1 to Q. (This is roughly similar to scalar-vector operations in Glish.)

5.5.2 Dealing with perturbed values

Each plane (aka `VellSet`) q of result l may contain a number of perturbed values. These are also evaluated. Note that the set of parameters (identified by spids) with respect to which the values are perturbed is not necessarily the same for each child result or plane, in fact, some planes or results may contain no perturbed values at all.

To simplify the description here, let's remember that `Function` deals with multiple result planes independently, on a plane-by-plane basis (see above). Hence, we'll assume we're dealing with only one plane, and l child results for that plane. We'll also forget about multiple perturbation sets, since those are all dealt with in the same way.

`Function` starts by figuring out all the spids present in the `VellSet`s of the children. Thus, it ends up with a list of spids, $(\kappa_1, ..., \kappa_n)$, which is simply the union of the spid sets from each child's `VellSet`. For each spid κ_i, it then needs to compute the perturbed value of F with respect to parameter κ_i ($\bar{F}^{(\kappa_i)}$), based on the perturbed `Vells` ($\bar{f}^{(k)}$) found in the child `VellSet`s.

This is done by calling `evaluate()` n times to compute $\bar{F}^{(\kappa_i)}$ for each κ_i, while selecting the input `Vells` from each child's result as follows. If spid κ_i is present in the list of spids for the `VellSet` of child l, the perturbed value $\bar{f}^{(lk)}$ is used (where k corresponds to spid κ_i). Otherwise, the main value $\bar{f}_l$ is used.

If this seems complicated at first glance, remember that the end result is quite simple. A `Function` node computes some function over its children's results. If the subtrees above the node contain solvable parameters, then perturbed values of that same function **w.r.t. all the solvable parameters** must be computed as well, based on perturbed values returned by the children. Everything else follows from this simple requirement.

5.5.3 Restrictions on child results

A `Function` node imposes certain restrictions on the structure of its child results, which the tree designer should be aware of. If the results do not have the right layout, a fail is always generated.

- The first restriction has already been covered above – all results must have the same number of planes, or must be a mix of Q planes and single planes. `Composer` and `Selector` nodes should be placed appropriately to ensure this.
- The second restriction has also been mentioned – all results must have the same `Cells`. Auto-resampling may need to be enabled to ensure this. This implies that all `Vells` in the results will have either the same $M \times N$ shape, or will be scalars (i.e. no time-frequency dependence).
- The child results must be either all samplings or all integrations. The two types cannot be mixed in the same expression.
- The next restriction involves the values of the perturbations (δ_κ) themselves. When F is evaluated over perturbed values for parameter κ, and these perturbed values appear in more than one `VellSet`, the result is sensible only when the perturbation δ_κ is the same thoroughout.
- Finally, if more than one perturbed value set is used (think double-differencing), `Function` requires that all child results have the same number of sets, though any child may always return zero sets (no perturbed values at all).

Note that the last two restrictions are almost always met automatically, since the perturbations δ_κ are usually determined by the parameters (i.e. `Parm` nodes) themselves. Single- or double-differencing is determined by the value of `calc_deriv` in the `Request`, which is also the same for all children. You'd be hard-pressed to construct a tree that didn't meet these two restrictions. Nevertheless, it's something to keep in mind in case you see `Function` nodes failing unexpectedly.

5.5.4 Implementing evaluate(): Vells arithmetic

The `evaluate()` method is defined as follows:

```
virtual Vells evaluate (const Request &req,
                        const LoShape &shape,
                        const std::vector<const Vells*> &args);
```

The `req` argument is simply the original `Request`. `shape` indicates the shape of the output `Vells` (i.e. the shape of the result `Cells`, see 5.4.5). The last parameter is a vector of pointers to argument `Vells`, which `Function` sets up for each call so that the correct combination of main values and perturbed values is used. Note that the resulting `Vells` is returned by value; this is a very fast operation, since `Vells` utilizes copy-on-write for its contents.

Implementations can take advantage of `Vells` *math*. `Vells` is an intelligent wrapper for real or complex 2D arrays or scalars, with overloaded operators and functions for basic math and implicit conversion from scalar values to a `Vells`. It provides automatic type promotion, and can also implictly "expand" a scalar to an array. For example, the doubled product of two `Vells` `a` and `b` may be obtained simply by stating `c=2*a*b`. This notation conveniently hides a lot of messy processing: real `Vells` are automatically promoted to complex if the other argument is complex, scalars are expanded to arrays if the other argument is an array, and the operation is performed on an element-by-element basis.[5] As another example, here's an actual implementation of `evaluate()` for the `Cos` node (computes cosine of child result):

```
Vells Cos::evaluate (const Request&,const LoShape &,
                     const vector<const Vells*>& args)
{
  return cos(*(args[0]));
}
```

Table 5.1 lists all the primitive operators and functions available with the `Vells` class. Most of these operate on `const` arguments and return a new `Vells` object by value. The exception are in-place operators (`"+="` and friends), which modify a `Vells` in place and return it as `"Vells&"` (these obviously require a non-`const` argument). The C++ syntax allows one to freely combine primitive operations into complicated expressions, while the compiler takes care of creating and destroying intermediate `Vells` objects as appropriate.

[5] An exception is thrown if two array `Vells` have a different shape. Note that having the same `Cells` in all child results ensures the same shape for all `Vells`.

basic arithmetic:	
`-`	unary negation
`+ - * /`	standard binary arithmetic operators
in-place arithmetic:	
`+= -= *= /=`	arithmetic, modifies the left-hand `Vells` in-place
unary functions, $\mathcal{R} \to \mathcal{R}$ *and* $\mathcal{C} \to \mathcal{C}$*:*	
`exp() log() log10()`	e^x, $\ln x$, $\log x$
`sqr() sqrt()`	x^2, $\sqrt{x}$
`pow2()...pow8()`	$x^2 ... x^8$
`sin() cos() tan()`	sine, cosine, tangent
`sinh() cosh() tanh()`	hyperbolic sine, cosine, tangent
`conj()`	complex conjugate
unary functions, $\mathcal{R} \to \mathcal{R}$ *only:*	
`ceil()`	nearest integer $\geq x$
`floor()`	nearest integer $\leq x$
`acos() asin() atan()`	arccosine, arcsine, arctangent
unary functions, $\mathcal{R} \to \mathcal{R}$ *or* $\mathcal{C} \to \mathcal{R}$*:*	
`abs() fabs()`	absolute value $\|x\|$ (both names are equivalent)
`norm()`	norm: $\|x\|^2$
`arg()`	argument of a complex number (the ϕ in $x = \|x\|e^{i\phi}$)
`real() imag()`	real and imaginary parts
array reductions:	
`min() max() mean()`	these reduce the argument (presumbaly, an array `Vells`) to a single scalar.
`sum() product()`	Note that `min()` and `max()` are only defined for real arguments.
binary functions:	
`tocomplex()`	$x + iy$
`pow()`	x^y
`atan2()`	$\arctan(y/x)$, also defined for $x = 0$
`posdiff()`	angular difference: for two angles x, y in the range $[-\pi, \pi]$, returns $x - y$ renormalized into $[-\pi, \pi]$ by adding $\pm 2\pi$ as necessary.

Note: some `Vells` *functions are defined for real arguments only. Applying them to a complex* `Vells` *will result in a fail.*

Table 5.1: Available `Vells` operations

Copy-on-write specifics

Since `Vells` employ copy-on-write, using `Vells` math for complicated expressions incurs very little overhead in comparison to an "optimized" implementation with explicit looping over array elements. The amount of messy coding saved, on the other hand, is huge.

Copy-on-write means that when one `Vells` is assigned to or initialized from another `Vells`, the two start off sharing the same data. This makes copy/assignment operations very economical, since the underlying data array is not copied. Any subsequent attempts to modify either `Vells` will cause a true private copy of the data to be made. Thus, copying of data arrays is deferred until it is actually needed (which may be never).

Consider, for example, an implementation of `Add:evaluate()`. The `Add` node adds the results of any number of children:

```
Vells Add::evaluate (const Request&, const LoShape&,
                     const vector<const Vells*>& args)
{
  if( args.empty() )
    return Vells(0.);
  Vells result(*args[0],DMI::READONLY);
  for( uint i=1; i<args.size(); i++ )
    result += *(args[i]);
  return result;
```

```
}
```

The `result Vells` is initialized from the first argument `Vells`, and shares data with it. As soon as the second argument is added in, a private copy of `result`'s data is implicitly created. If, however, only one argument is passed in to begin with, the `result` will still share data with it when returned.

Chapter 6

The MeqServer Interface

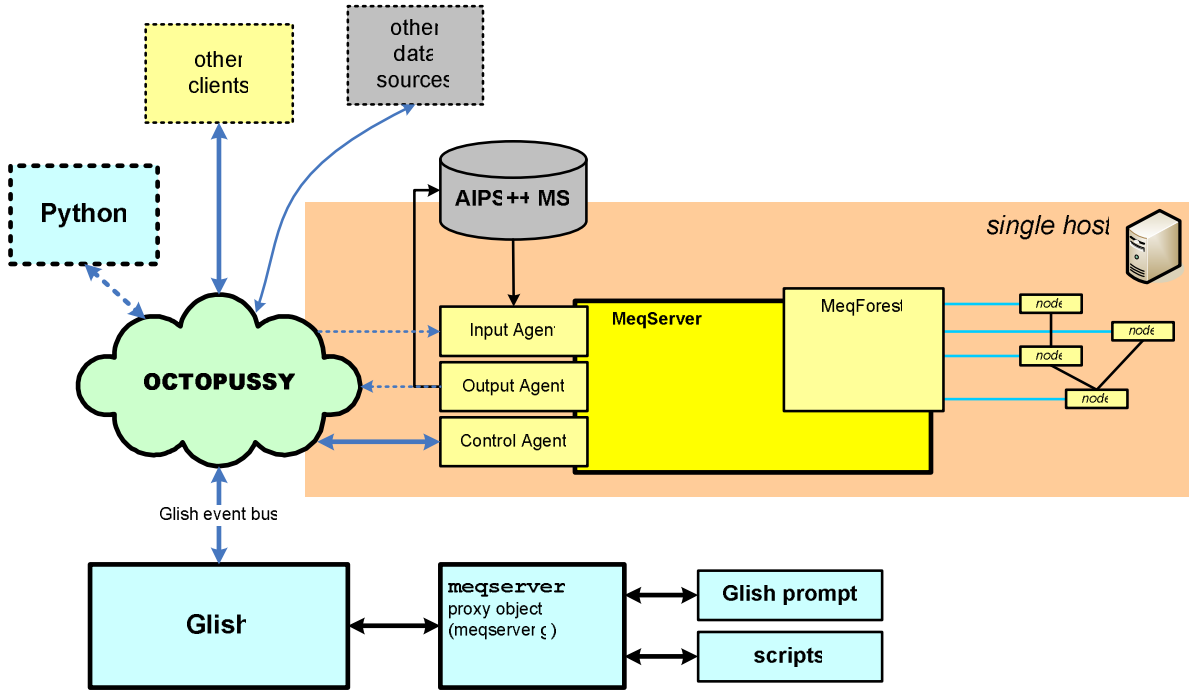

Figure 6.1: MeqServer and MeqTree control