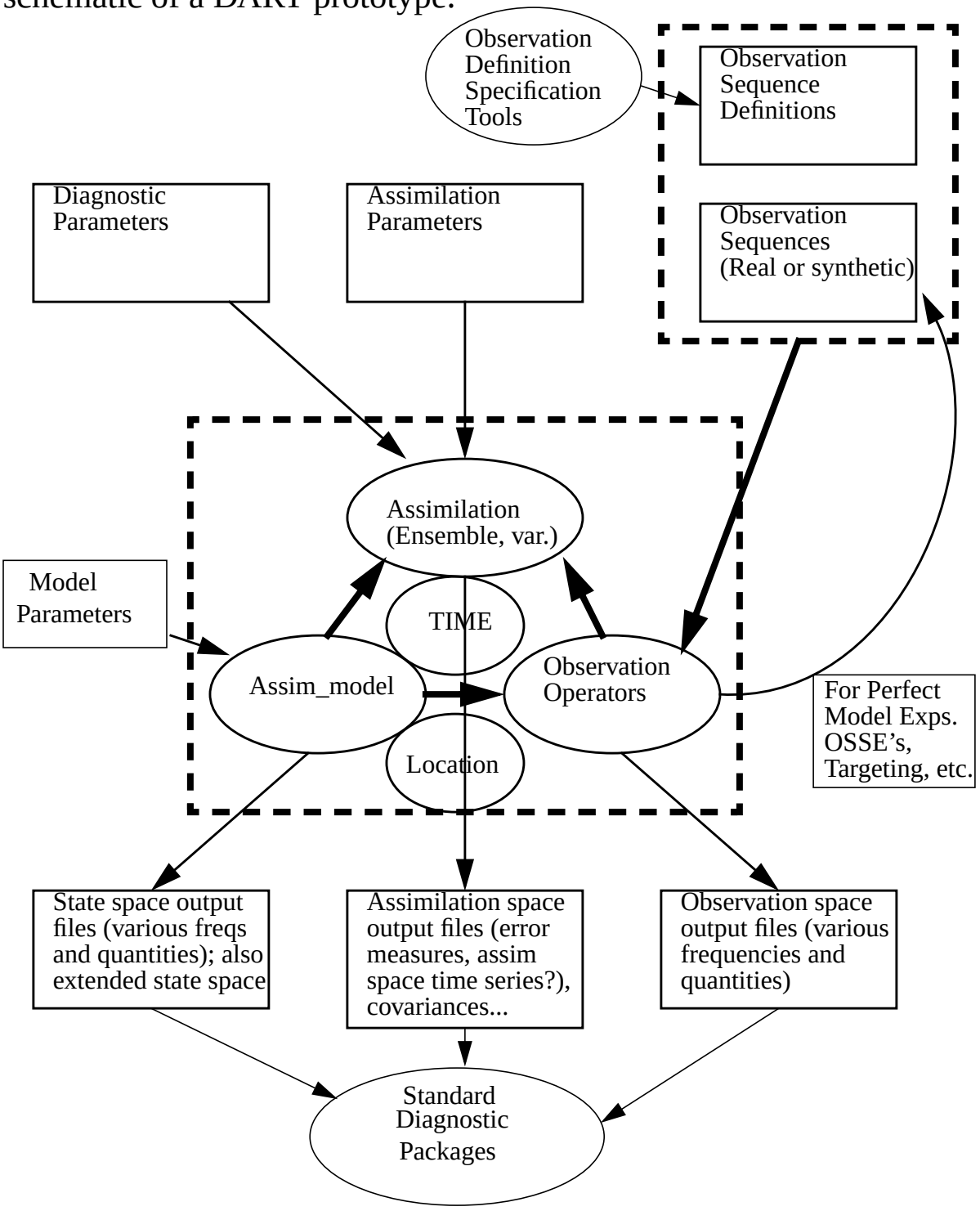
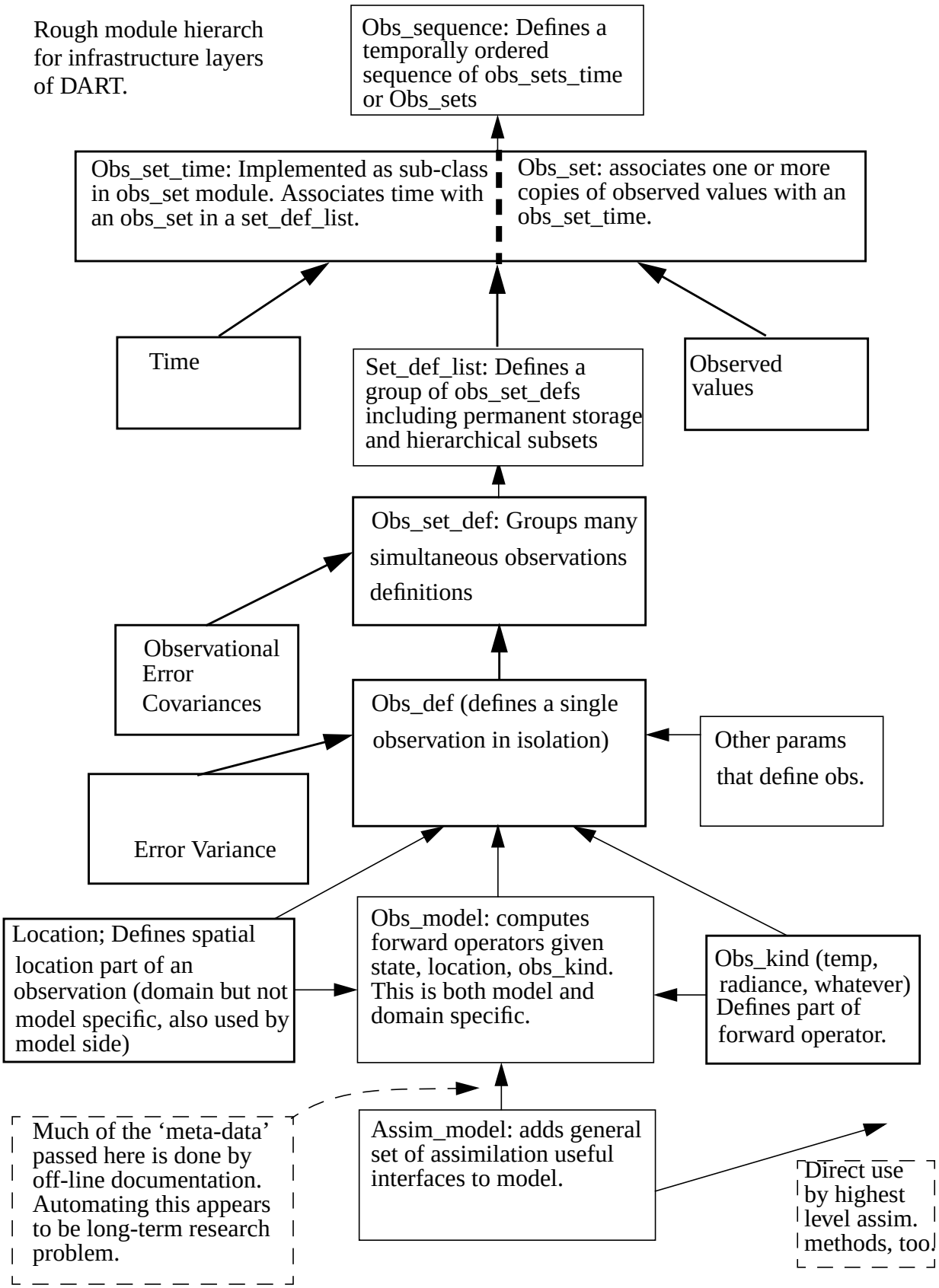
A schematic of a DART prototype.




Rough module hierarch for infrastructure layers of DART.
Obs_sequence: Defines a temporally ordered sequence of obs_sets_time or Obs_sets
Obs_set_time: Implemented as sub-class in obs_set module. Associates time with an obs_set in a set_def_list.
Obs_set: associates one or more copies of observed values with an obs_set_time.
Time
Set_def_list: Defines a group of obs_set_defs including permanent storage and hierarchical subsets
Observed values
Obs_set_def: Groups many simultaneous observations definitions
Observational Error Covariances
Obs_def (defines a single observation in isolation)
Other params that define obs.
Error Variance
Location; Defines spatial location part of an observation (domain but not model specific, also used by model side)
Obs_model: computes forward operators given state, location, obs_kind. This is both model and domain specific.
Obs_kind (temp, radiance, whatever) Defines part of forward operator.
Much of the 'meta-data' passed here is done by off-line documentation. Automating this appears to be long-term research problem.
Assim_model: adds general set of assimilation useful interfaces to model.
Direct use by highest level assim. methods, too.

1. Rough class specifications for DART below assimilation level

Each class begins with a crude type definition and is followed by a series of Fortran class interface descriptions. The interfaces are roughly ordered by class with initialization, calls to get data from the class, calls to set data in the class, logical queries, input, output, and finally termination.

I. obs_sequence (Called obs stream by operational folks)

This is the highest level class on the observations side.

Defines type obs_sequence_type
which contains num_copies of data, meta_data associated with each copy, the maximum number of obs_sets in the sequence and the current number, a set_def_list that contains defintions of all observation sets, and a time ordered list of observations sets.

1. function init_obs_sequence(max_obs_sets, num_copies_in, copy_meta_data) initializes an obs_sequence with room for max_obs_sets. Optional argument num_copies_in with default value 1 (note that 0 is a legal value) specifies the number of copies of data associated with each observation in the sequence.

1b. subroutine obs_sequence_copy(seq_out, seq_in) does a comprehensive copy of seq_in to seq_out allocating all required storage.

1c. subroutine obs_sequence_def_copy(seq_out, seq_in) copies only the definition parts of the sequence seq_in into seq_out which will have num_copies set to 0 by definition.

1d. subroutine inc_num_obs_copies(seq, inc, copy_meta_data) increments the number of copies of the data in the sequence by inc and adds the additional meta data for these new copies. This requires an ugly reallocation in the current implementation.

1e. function get_num_obs_copies(sequence) returns the number of copies of the data contained in this sequence.

2. function get_num_obs_sets(sequence) returns the number of observation sets in this sequence.

3. subroutine get_obs_values(sequence, index, obs, copy_in) returns the observed values associated with the index-th observation set in the sequence. Optional argument copy with default value of 1 indicates which copy of the associated data is to be returned.

4. subroutine set_obs_values(sequence, index, obs, copy_in) set the observed values with arguments as in get_obs_values.

5. subroutine set_single_obs_value(sequence, index, num_obs, obs, copy_in) sets the value of a single observation in the set, indexed by num_obs, with all other arguments as in set_obs_values.

7. function get_copy_meta_data(obs_sequence_type, index) returns the meta-data associated with the indexth copy of the observations. Index is an optional argument with default value 1.

8. function get_obs_set(sequence, index) returns the index-th obs_set in the sequence.

9. subroutine get_diag_obs_err_cov(sequence, index, covariance) returns the diagonal part of the covariance of the observations in the index-th observation set in the sequence.

10. subroutine get_num_close_states(sequence, index, radius, num) returns the number of model state variables that are closer than distance radius to each of the observations in the index-th observation set in the sequence.

11. subroutine get_close_states(sequence, index, radius, num, indices, distance) as in get_num_close_states but returns a list of the indices of each state variable that is close to each observation as well as the distance between the close state variables and the observations.

12. subroutine get_expected_obs(sequence, index, state, obs, num) returns the expected value of the index-th observation set in the sequence given the state vector in state. The array obs returns the expected values. If the optional argument num is present, only the expected value of the num-th observation in the specified observation set is returned.

13. function get_obs_sequence_time(sequence, index) returns the time associated with the index-th observation in the sequence.

14. function get_num_obs_in_set(sequence, index) returns the number of observations in the index-th observation_set in this sequence.

15. subroutine add_obs_set(sequence, obs_set) adds the obs_set to this sequence. The set must have an ending time (time for now) that is later than the last observation set currently in the sequence.

16. subroutine associate_def_list(sequence, def_list) associates the def_list with this sequence. For now, a sequence must be associated with exactly one def_list.

17. subroutine write_obs_sequence(file_id, sequence) outputs the sequence to the file.

18. function read_obs_sequence(file_id) reads an obs_sequence written by write_obs_sequence.

19. function read_obs_sequence_def(file_id) reads an obs_sequence file written by write_obs_sequence but only uses the definition information (i.e. num_copies is 0 in the sequence created).

II. Obs_set

Defines a group (hierarchical set_def_list) of observation set definitions that are all associated with the same time interval and the associated observation values. This could actually be split into two separate modules, one defining an obs_set_time (a time ordered sequence of observation groups with the associated times) and another defining an obs_set which in addition includes one or more copies of values associated with the observations.

This pushes time interpolation to this rather high level. This should be re-evaluated at design review. The higher level assimilation algorithms would need to keep states at different times if model won't return stuff at required time. In mature implementation, time must be general and the lower level assim_model class must return info on the interval over which an observation is taken (weighted time interval with delta function as simplest case, followed by average, weighted average, etc.). The higher level assimilation will have to decide what to do with observation sets that are defined over more generalized time intervals.

Defines type obs_set_type
which contains a two dimensional array of obs, the first dimension is the number of observations associated with the corresponding set_def observation group and the second is the number of copies of data available for each observation.
A two dimensional logical array missing which is if the corresponding obs are not available.
The number of copies and the number of observations, a time_type giving the time of this observation, and an integer index into the set_def_list associated with this set. At present, an observation sequence (next level up) can only be associated with a single set_def_list. The number of copies is set to 0 and no data is associated when this structure is used to represet an obs_set_time type.

1. function init_obs_set(set_def_list, index, num_copies): Initializes the storage for an obs_set associated with the index-th set in the set_def_list and with num_copies of the associated data. Num_copies is optional with a default value of 1.

1b. subroutine obs_set_copy(set_out, set_in) does a comprehensive copy of an obs_set allocating all required storage and copying set_in to set_out.

1c. subroutine obs_set_time_copy(set_out, set_in) copies just the obs_set_time definition portion of set_in to set_out allocating all storage as required. The result has num_copies set to 0 and no data associated with the defintions.

1d. subroutine inc_num_obs_copies(set, inc) increments the number of copies of the observations associated with an obs_set. This is required by the current static allocation of storage and requires a reallocation and copy. inc specifies the number of additional copies to be added.

1e. function get_obs_def_index(obs_set) returns the index in the set_def_list of the parent of the set_def hierarchy for this observation set.

1f. function get_obs_set_time(obs_set_type :: set): Returns a generalized_time_type that describes the characteristics of this set of observations.

1g. function get_num_obs(obs_set) returns the number of observations in this observation set.

2. subroutine get_obs_values(obs_set_type, obs, index) : Returns a vector of 'real' values that came from the instrument(s). Some value might be returned if the data is missing for certain elements, but this is not defined. If integer or other types of obs were happening might have to extend or overwrite this, but can't think of a relevant case. The optional argument index indicates which copy of the observation data should be returned with a default of 1.

4. subroutine set_obs_values(obs_set, obs, copy) : Sets the observation values for an obs_set. The optional argument copy selects which copy of the observations should be set with a default of 1.

4a. subroutine set_single_obs_value(obs_set, num_obs, obs, copy_in) set the value of the single num_obs-th observation in the obs_set to the value in obs. The optional copy_in argument indicates which copy of the observation data should be set with a default of 1.

5. subroutine set_obs_set_missing(obs_set, missing, index_in): The observations corresponding to any true elements are set to the values in the logical array missing. The optional argument index_in indicates which copy of the missing flags is to be set with a default value of 1..

6. subroutine set_obs_set_time(obs_set_type, generalized_time_type) sets the time interval associated with a particular observation.

7. function contains_data(obs_set_type) returns logical, true if this set has observations and false if it only has a definition.

8. subroutine obs_value_missing(obs_set_type, missing, index_in) returns a logical array missing with false for all entries where an observation exists and true if observation is missing. Optional arguement index_in selects which copy of the missing flags is returned with a default value of 1.

9. function read_obs_set(file pointer), reads an obs set from file with this pointer (can just be an index for now)

10. function read_obs_set_time(file pointer), reads the obs_set_time part of an obs_set leaving out information about the data and missing flags.

11. subroutine write_obs_set(file, pointer, obs_set), writes an obs_set to file with this pointer.

12. subroutine write_obs_set_time(file_pointer, obs_set), writes the obs_set_time part of an obs_set leaving out information about the data and missing flags.

ALSO INHERITS the calls from obs_set_def which should act directly on the obs_set_def in the obs_def structure.

IIB. set_def_list

Provides permanent storage with indexing and retrieval for obs_set_def structures. This is between the obs_set_def and the obs_sequence. Only obs_set_def structures that have been registered in the index can be used as subsets or can be referenced in an obs_set. This level also allows for the definition of hierarchically nested obs_set_def subsets. The subset mechanism is not currently implemented.

Defines type set_def_list_type
which contains an ordered list of list_element_types, along with a maximum number of list_element_types that can be in this list and the current number in the list. The maximum should be removed in later implementations by addition of dynamic storage.

Also defines private type list_element_type
which contains an obs_set_def, an integer index, the total number of observations in the set (including subsets when these are implemented), the maximum number of subsets that could be included in this set and the number of subsets currently included as well as a list of indices pointing to the subsets.

At some point, want to make sure that set_def_list structures are frozen once the list is linked into an obs_sequence structure so that things can't be redefined underneath. May need back pointers and other things when subsets are implemented.

1. function init_set_def_list(max_sets) returns a set_def_list with all required storage allocated to hold up to max_sets.

2. subroutine set_def_list_copy(list_out, list_in) does a complete copy of list_in to list_out including allocating all required storage.

3. subroutine get_expected_obs(list, list_index, state_obs or extended state, num) returns the expected values of all observations included in the (hierarchical) list_element with index list_index on the set_def_list list. The optional argument num requests return of only the expected value of the num-th observation in the referenced list_element.

4. subroutine list_element_copy(list_out, list_in) is a private interface doing comprehensive copy of list_element_type list_in to list_out.

5. function add_to_list(list, set, max_subsets_in) returns a set_def_list that that is the same as list but with the addition of set which is specified as having no more than max_subsets_in subsets. Final argument is optional with default number of subsets being 0.

6. function get_def_from_list(list, index) returns the obs_set_def in the index_th list_element in the set_def_list.

7. function get_num_sets_in_list(list) returns the total number of list_elements in this list.

8. subroutine get_diag_obs_err_cov(list, index, cov) returns the diagonal observational error covariance for the subset hierarchy with parent that is the index-th element on the set_def_list list.

9. subroutine get_num_close_states(list, index, radius, num) returns the number of state variables within radius of each observation that is defined in the obs_set_defs that have parent at location index on the set_def_list, list. The numbers are returned in num.

10. subroutine get_close_states(list, index, radius, num, indices, distance) returns the indices of all the state variables that are closer than radius to each of the observatoins included in the hierarchical set_def_list with parent at position index on the list. The distances between the observations and the corresponding state is returned in distance.

11. function get_number_obs_subsetrs(list, index) returns the number of subsets contained in the set_def_list elmenet index.

12. subroutine write_set_def_list(file_id, list) outputs the list and all its contents.

13 function read_set_def(file_id) reads the output of write_set_def_list and returns the corresponding set_def_list.

14. function write_list_element(file_id, element) outputs the contents of a single list element.

15 function read_List_element(file_id) reads the output of write_list_element.

III. obs_set_def

Defines a set of associated observation definitions.

Defines type obs_set_def_type

contains
error_covariance
list of obs_defs
the number of obs_defs currently in the set
the maximum number of obs_defs that could be in the set (this is a storage simplification parameter that should eventually be eliminated.

Note that current implementation only supports diagonal error_covariance.

Earlier requirements for a unique key for assimilation algorithm caching have been dropped.

1. function init_obs_set_def(max_num_obs) returns an obs_set_def with space for up to max_num_obs observation defintions. Might want to make this optional at a later time.

1b. subroutine obs_set_def_copy(set_out, set_in) produces a complete copy of set_in in set_out; handles all allocaton of storage, etc.

2. subroutine get_diag_obs_err_cov(obs_set_def, cov) : Returns the diagonal of the observation error covariance matrix for this set in a one dimensional array.

3. function get_obs_err_cov(obs_set_def :: set) : Returns a two dimensional array containing the observational error covariance for this set. Not needed in initial implementation.

4. subroutine get_expected_obs(obs_set_def :: set, model_state_vector or extended model state vector:: state, obs, num) : Returns a one dimensional array containing the expected values for this observation set given the model state. Argument num is optional and, if specified, indicates that only the expected value of the num observation in the set is to be returned. Need to give more thought to how the extended state is passed around if it is required and whether it is of fixed or variable composition (probably variable for big models). For initial implementation need only implement raw state part.

5. function get_obs_def(obs_set_def, index) returns the index obs_set in the obs_set_def.

6. function get_num_obs(obs_set_def :: set): Returns the number of observations in a set.

Following 3 interfaces push up obs_def functions to this level (inherited)

13. subroutine get_obs_locations(obs_set_def, locations) returns ordered set of locations for the observations in the set.

14. subroutine get_close_states(obs_set_def, radius, number, indices, dist) returns array of numbers plus 2D array of indices for state variables close to each ob in the set and the corresponding distances in the 2D array dist.

15. subroutine get_num_close_states(obs_set_def, radius, number) returns array of number of close states for each of the observations in the set.

16. subroutine add_obs(obs_set_def, obs_def) adds the obs_def to this obs_set. Need error return if the maximum number of obs would be exceeded by this insertion.

17. subroutine set_err_cov(obs_set_def, obs_subset_index, obs_subset_index, obs_index1, obs_inces2, cov) sets error covariance between two obs subsets/obs. Only two of the subset / obs indices can be set in the call. The covariance is a two dimensional matrix with the covariance val-

ues. Error is required if the indices are outside the range for this set or if the covariance matrix is not the correct size for the pair. Not needed in initial implementation. Not currently implemented.

19. function diag_obs_err_cov(obs_set_def :: set) : Returns true if error covariance for this obs set is diagonal, false otherwise. Initial implementation can support only diagonal covariances with great simplification.

20. function read_obs_set_def(file_pointer), returns an obs_set_type with contents read in from a standard file format (this may be relatively sophisticated as it will have to allocate storage, etc. Should make use of calls defined above.

21. subroutine write_obs_set_def(file_pointer, obs_set_def) outputs an obs_set_def in standard file format.

May need to push up additional obs_def functions to this level

IV. obs_def

Defines a single scalar observation’s spatial characteristics and kind.

Defines type obs_def_type

contains a location, an obs_kind, an a real error variance. Other parameters may need to be associated with the type at some point.

Earlier requirements for a unique key for caching have been eliminated.

1. function init_obs_def(obs_location, obs_kind, error_variance) Constructor for obs_def returns an obs_def with this kind, location and error variance.

2. function get_expected_obs(obs_def, model_state_vector / extended state vector), returnsexpected value given state or extended state vector. For initial implementation only need to deal with raw state.

3. function get_error_variance(obs_def), returns observational error variance of observation definition.

4. function get_obs_location(obs_def), returns a location_type for this observation. For some more complex types of observations the definition of location may involve additional parameters, but let’s avoid that complexity for this stage. This would require changing this to a subroutine with multiple returned parameters.

5. function get_obs_def_kind(obs_def), returns the kind of this observation.

7. function get_num_close_states(obs_def, radius) retuns the number of state variables within radius of this observation. This assumes that the model class data is global (only one type of model in use). At some point it would be nice to generalize this, but not now.

8. subroutine get_close_states(obs_def, radius, number, close_state_list, distance) Returns a list of state points within distance radius of the obs_def. The number is redundant to the returned value from get_num_close_states. The list has the indices of the close states (currently no order is implied) and distance returns the distance between each state in the list and the observation location.

10. function set_obs_location(obs_def, location) returns an obs_def with the location set to the input and all other aspects as for the input obs_def.

11. function set_error_variance(obs_def, err_var) returns an obs_def with error variance set to err_var.

12. function set_obs_def_kind(obs_def, obs_kind) returns an obs_def with kind set to obs_kind.

13. function read_obs_def(file_pointer) reads an obs_def from standard format off the file. Will need error returns if read fails.

14. function write_obs_def(file_pointer, obs_def) writes an obs_def in standard format to the file.

15. function interactive_obs_def() does text driven interactive generation of an obs_def and returns this.

V. location

This is the level at which the details of the spatial representation come into play. There will be different location data representations for different spaces (sort of like the one and twod loc and dist mods but more general). Currently, a one dimensional space implementation has been completed.

Defines type location_type

oned implementation contains a single real number in range 0-1

1. function get_dist(loc1, loc2), returns some metric of distance between two locations.

2. function get_location(location_type) returns values of the location in the arguments; inverse of set_loc. This has been implemented as a function in the oned version, but probably needs to be modified to be a subroutine, given that a single value will not define locations in higher order spaces.

3. function set_location(space model specific arguments, for instance, lat, lon, height) returns a location type with these values.

4. subroutine write_location(file_pointer, location_type) writes the location to the file in a standard format.

5. function read_location(file_pointer) reads the location from the file in a standard format. Need error return if read fails; this is related to a general need to re-evaluate error returns from IO.

6. subroutine interactive_location(location) allows for interactive text driven entry of a location. Current oned implementation allows option of selecting a uniformly distributed random location.

VI. generalized_time

For early implementations, let's just assume all times are discrete and use the standard FMS time manager package to do this. However, need to keep possibility of a weighted observation time interval in mind. Need to be able to output and input a time. The FMS time manager is used with the addition of two interfaces for consistent IO in the DART context. The FMS defined type is retained.

Defines type time_type.

Additional interfaces added to FMS:

1. subroutine write_time(file, time) accepts a file identification (currently an integer unit number) and a time and outputs the time in a standard format.

2. function read_time(file) reads the output of write_time and returns a time type loaded with the corresponding time.

VII. obs_kind

This module is designed to provide general information about the kind of observation being taken. It is not clear that a class like this that tries to define observation type independently of the model definition will be useful in the long run. This might be a good place to store other parameters associated with more complicated observation types. For now, the obs_kind is simply implemented as an integer index that corresponds to a particular type of observation. All computation depending on this index is performed in obs_model for now.

Defines type obs_kind_type.

Current implementation has single integer representing observation kind.

1. function get_obs_kind(obs_kind) returns string (integer for now) specifiying the obs_kind.

2. function set_obs_kind(string) returns and obs_kind with this string (integer for now) as its definition.

3. subroutine write_kind(file, kind) outputs a kind to file.

4. function read_kind(file) reads a kind from a file.

5. subroutine interactive_kind(kind) allows interactive input of kind for user driven set-up of observation system.

Can this whole level just be dropped for initial implementation?

VIII. obs_model class

This is the level at which classes become model dependent on the observation taking side of things. This class needs to know considerable detail about how the model data is laid out, how to interpolate, how to do forward operators for different types of observations, etc. Probably the biggest chunks of code someone will have to write in order to get things going? Maybe there should be one more layer that just does interpolation and knows about the model? In the ultimate implementation, the location and the class should be separated as cleanly as possible. In the best case, the class would request model field quantities at specific locations and then operate on these to complete the forward observation operator. Clean separation like this may not be practical.

1. function take_obs(model state vector or extended model state vector, location, obs_kind_id) there may be a variety of different algorithms. Making use of interpolation if needed from assim_model, this computes the observation given the state and returns the value. Current implementation is limited to accepting a simple model state vector.

2. subroutine interactive_def(location, obs_kind) returns a location and obs_kind defined interactively by a user. This can be used for simple text based interfaces to set up observation definitions.

IX. assim_model

This is the interface needed to a model by the assimilation algorithms and observation networks. This is model specific and tries to abstract away the model details. Because of F90 limitations, seems appropriate to think of only being able to work with one model type (class) at a time, i.e. one couldn't work on a separate atmosphere and ocean model at once with these interfaces. Assim_model is a class for a particular kind of model and the instances are the state (including some indexed time). Initialization and end calls are for the class data (for instance setting up transform stuff for a spectral model). Smart choices about timestepping can be pushed into calls at this level. Extended state should not be viewed as part of the class data, but is instead the return from some functions operating on the state. May want a concept of static extended state (things that can be obtained by operating on the state variables without time integration) and extended state that requires integration. For initial implementation, assume that model timestep is fixed at a particular delta_t and that all observations will be specified as falling exactly on one of these timesteps. The general timestepping computations can probably be pushed into an auxiliary module since they will be used for all models.

Defines type assim_model_type
time type with associated global base time that defines model time
state vector

Also defines global meta_data associated with the class that describes the state vector. Also need associated location definitions in global storage.

1. function static_init_assim_model() initializes the class data for the model, for instance transform data for spherical harmonics. Sets up global definition of model state variable locations. Initializes information about model size and things like time stepping capabilities.

1a. subroutine init_assim_model(state) allocates storage for and assim_model instance in state.

2. function init_diag_output(file_name, global_meta_data, copies_of_fields_per_time, meta_data_per_copy) returns file_id for an output diagnostic file that will output a given number of copies (say posterior, prior, truth, ensembles, etc) of fields at each time. For now the global meta-data is a text string describing the file contents and the meta_data_per_copy is an array of text strings describing the contents of each copy of data.

2a. function init_diag_input(file_name, global_meta_data, model_size, copies_of_field_per_time) reads in output of init_diag_output in preparation for reading diagnostic files. Gets global_meta_data, model_size, and copies_of_field_per_time from file. Need to be careful with coordination of files and class since there is all this class static data floating around that might be inconsistent with the diagnostic file.

2b. subroutine get_diag_input_copy_meta_data(file_id, model_size_out, num_copies, location, meta_data_per_copy) reads in the location and copy meta_data given the model size and num_copies previously read by init_diag_input.

3. function get_model_size() returns integer size of model state vector; need to be very careful to understand multiple time-level state interactions. Note that this is class data.

4. function get_max_dt()

5. function get_min_dt()

6. function get_other_dt_stuff()

7. function get_last_state_time_before(time)

8. function get_first_state_time_after(time)

9. function get_closest_state_time_to(assim_model, time) given a model state in assim_model and a time, finds the closest time to which the model state can be advanced to this time.

10. function get_next_state_time(time)

11. function get_prev_state_time(time)

12. subroutine get_initial_condition(x) returns an assim_model_type to start from in x. This might come from a file or from something else, controlled by model runtime paramerters??? This needs to be given some help through some runtime interface I think. One option is that a 'restart' file is available in some standard place, this is read in and the corresponding state returned (watch out for multi-level timestepping stuff here). Another option is just some basic spin-up state. The assim_model_type returned has a time associated with it. For now, this time needs to just come in with whatever is read from file or generated and is probably controlled by a runtime input. Coordinating the time between the model and observation set areas should be automated eventually but can be pushed to runtime for now.

13. subroutine get_state_meta_data(index, location, optional kind) returns metadata for the indexed state variable as a location plus a kind (if the model has more than one kind of variable).

14. subroutine get_extended_state_meta_data(index, location, optional kind) returns metadata for the indexed extended state variable. May need to further refine this for large models with a large number of possible extended state types.

15. subroutine get_close_states(location_type, radius, number, indices, distance) returns list of state indices within radius of the given location and the number of such state variables. May want to make kind an optional input. Need error handling if there is a storage issue. Should probably implement with a clean extensible list. Note that these are class functions that don't depend on a particular state instantiation. Also returns the distance between the close state variables and the location. At some point, may need additional distance information from kind differences?

16. function get_num_close_states(location_type, radius) returns the number of state variables within radius distance of this location.

17. function get_model_time(assim_model_type) returns time type that is the time for this state.

18. function get_static_extended_state(assim_model_type, optional field specifier) returns static extended state variables computed from model state. Not needed in initial implementation.

19. function get_model_state_vector(assim_model_type, plus additional arguments to specify a portion?)

20. subroutine copy_assim_model(assim_model_type_out, assim_model_type_in) returns assim_model_type overloaded to =.

21. function advance_state(assim_model_type, target_time, extended state requests optional) returns state and optionally extended state vector advanced in time to the target_time. If the model has flexibility to do so, it will be advanced to exactly target_time. If it has a fixed time_step, target_time should be within some small tolerance (for floating point stuff) of a time to which the model can go or an error should be returned. The philosophy here is that the requisite time computations should be done first and then the model should be advanced. For initial implementation, assume that model has single dt and that all observations will fall exactly on a dt interval.

22. function interpolate(state vector / extended state vector, location, field_id) returns value for this field interpolated to the location. Need to be very clear here. What is a field_id and how is it obtained. What about 2D fields in 3d models, etc. For now, only implement for state vector and add extended state at later date. Field_id needs to be associated with some sort of string i.d. that is part of the model metadata.

23. subroutine set_model_time(assim_model_type, time_type)

24. subroutine set_model_state_vector(assim_model_type, state_vector, plus additional arguments to specify a portion?

5. subroutine write_state_restart(assim_model_type, file_name) writes out the state vector and the time in a form with machine precision so that time integration can be resumed without evidence of interruption.

26. function read_state_restart(file_name) reads a machine precision restart from file_name including the time and state vector.

27. subroutine output_diagnostics(file_id, state vector, time, copy_index optional) outputs diagnostic state information for this time for the copy_index copy. If copy_index is not present it is assumed to be 1. For now, the copies at a given time must be called sequentially and no other calls for output for this file_id may be made until all copies are output at this time.

28. subroutine input_diagnostics(file_id, state, copy_index) reads in an assim_model_type diagnostics from file written by output_diagnostics.

28. subroutine end_assim_model() shuts down and cleans up class data for model.

2. Contents of diagnostic output and control input files:

I. State space output files: Contain metadata and data for output of state space quantities. These are arranged into a (set of) file(s) with different associated time axes (in the NetCDF sense). Things that may be here include the ensemble members before and after each observation (prior and posterior), and the ensemble mean. Might also want extended state to be available from model which would have to output the appropriate extended state and its associated metadata. Might also want variance of prior and posterior state (more consistent with non-ensemble methods). If available, the truth for state variables should be in a file of this format (but would be generated ahead of time) if output is for a synthetic run. In all cases, the assim_model level is required to provide calls that output metadata for the appropriate output variables to a file.

II. Observation space output files: Contain prior and posterior (ensemble) values for the observations, truth for the observation would also be in a file like this. Could also include variance, etc., if desired. These variables are apt to be irregularly spaced in space and time with potentially very complex metadata. While it is desirable in the long run to attempt to use some standard metadata format to express this, doing this correctly appears to be a difficult research problem. For the initial implementation, it is acceptable to use a customized ascii based format which can be plotted for some subset of metadata types by standard matlab based interfaces. For efficiency, it will be necessary to compact this ascii metadata at the head of the output file as per NetCDF. However, for ease of initial implementation may want to generate two output files, one with metadata and one with data and combine them after execution is completed.

There is significant overlap between the observation definition/observation input files and the observation space diagnostic output files. For the initial implementation, these should use identical basic metadata although the diagnostic output may contain additional instances of the data.

III. Assimilation space output files: Measures of global error, detailed time series of individual assimilation variables (do this for efficiency?), covariance output between state/obs variables. This output is generated directly by the assimilation level which is responsible for generating the appropriate metadata. However, in many simple cases, the majority of this output may in fact be in model space and the model interfaces could be used to generate metadata and output the data. In more general cases, this metadata may be even more messy than that for the observation space files since it will combine observation and model space output along with statistical products between these spaces. Again, it would be desirable to use a standard metadata format, but this is probably impossible at present and a custom metadata format with associated extraction and plotting will be required.

IV. Observation definition/observation input files: These contain efficient metadata describing observations and sets of observations as well as data (time plus observation value(s)) describing the associated observations. Eventually, want to be able to stream real observation sets from some community standard data servers through this interface. Also need a capability to generate observation sets on the fly (i.e. need random access files with read and write pointers) so that targeted

observation types of OSSEs can be performed without modifying the configuration of an assimilation.

V. Parameter control input files: A number of different files are needed to control run-time behavior of different components of the assimilation system. Traditionally, this type of input has tended to be done through F90 namelists but this has proved to be somewhat limited. For now, DART will allow the use of either namelists or more general control input files to be read at run-time. Standard naming convention for input control files will be the module name followed by an appropriate extension.

3. Different views of DART for different users:

I. Diagnostics users (general diagnostics interfaces)

Want to be able to select the output files associated with a particular experiment. The experiment was generated by a particular model, set of observations, assimilation method, and parameterizations associated with the model and the assimilation method. These different aspects will control to some extent how the diagnostics appear, although diagnostics in most cases should be relatively independent of this.

Some example diagnostics that one might like to have readily accessible by a simple GUI. Initially, many of these will be front-end wrappers for Matlab diagnostic routines. Some examples of diagnostics that would be nice to have standardized:

a. Plots of overall RMS, spread, etc. as function of assimilation time
b. Plots of ensemble behavior for individual variables
c. Order statistic histograms for selected variables, for both truth or observations, if observations need to correct for error
d. A catalog of which variables are available and a way to relate them to what they represented in the model
e. Plots or animations of model state (or slabs / hyperslabs of model state) as function of lead time
f. Plots of error as above
g. Plots of spread as above
h. Plots of mean bias as above
i. Plots of individual ensemble members as above
j. Plots of error for individual members as above
k. Diagnostic plot of where the observations are
l. For non-perfect cases can’t do all of the quantities above, may often be limited to working at observation locations which may not be regularly gridded
m. Mean value of innovations, time series of innovations, etc.
n. Plots of prior and/or posterior correlations of variables

In order to do this in the most general form, the output file must contain the following items:

1. Values of observations, along with description of location and error, plus possibly h's?a. Plots of overall RMS, spread, etc. as function of assimilation time
b. Plots of ensemble behavior for individual variables
c. Order statistic histograms for selected variables, for both truth or observations, if observations need to correct for error
d. A catalog of which variables are available and a way to relate them to what they represented in the model
e. Plots or animations of model state (or slabs / hyperslabs of model state) as function of lead time
f. Plots of error as above
g. Plots of spread as above
h. Plots of mean bias as above
i. Plots of individual ensemble members as above
j. Plots of error for individual members as above
k. Diagnostic plot of where the observations are
l. For non-perfect cases can't do all of the quantities above, may often be limited to working at observation locations which may not be regularly gridded
m. Mean value of innovations, time series of innovations, etc.
n. Plots of prior and/or posterior correlations of variables
a. Plots of overall RMS, spread, etc. as function of assimilation time
b. Plots of ensemble behavior for individual variables
c. Order statistic histograms for selected variables, for both truth or observations, if observations need to correct for error
d. A catalog of which variables are available and a way to relate them to what they represented in the model
e. Plots or animations of model state (or slabs / hyperslabs of model state) as function of lead time
f. Plots of error as above
g. Plots of spread as above
h. Plots of mean bias as above
i. Plots of individual ensemble members as above
j. Plots of error for individual members as above
k. Diagnostic plot of where the observations are
l. For non-perfect cases can't do all of the quantities above, may often be limited to working at observation locations which may not be regularly gridded
m. Mean value of innovations, time series of innovations, etc.
n. Plots of prior and/or posterior correlations of variables

In order to do this, output files must contain the following:
1. Values of observations, along with description of location and error, plus possibly representation of forward operators
2. Prior ensembles for observations
3. Model state variables, ensemble members and mean
4. Model state variables before and after each assimilation time (can get innovations from this)
5. Truth state variables along with the ensemble if this is a simulated observations case.

II. University type educational user (student):

This type of user wants to do new runs with existing models (or modified models) and design their own observation set or use previously defined observation sets. Divide this into two categories: Making a run with existing observational set and model but changing some parameters of the run like model or assim parameters and generating output

OR, designing a particular observational set and running either synthetic obs or real obs with this configuration

Will need tools to build observation sets, in particular to build sets of relatively regular observations or small number of irregularly spaced.

III. Sophisticated model developer: Put in a new model and some associated metadata and plotting routines etc. Will need to write the assim_model class, the obs_model class, and the location type if the domain is something new. While this should be much more straightforward than in a naive modeling setup, this is not going to be trivial and will require lots of expertise on DART.

---

4. Sophisticated assim person: Put in a new assimilation scheme.

---

5. Sophisticated data set person, put in a new data set for use with one or more models. This may be relatively straightforward if the forward operator is simple, but could be extremely complicated in the case of the most general forward operators.

Some notes on DART usage in prototype:

Given a model, etc.

1. Observation definitions, observations set definitions, result is a file containing an observations definition set (set_def_list); can end up with an array of set_def_lists associated with a particular spatial dicretization (and model because of obs_kind?)

2. Observation sequence definitions: define a sequence of observations through time, result is a file containing an observation sequence (obs_sequence): have an array of obs_sequence files available for a particular spatial discretization (and model?). These may or may not have some sort of observation values associated with them, but for this purpose those are not relevant.

3. Perfect model output files: given an observation sequence definition (with no obs values required) one can run the model (given some initial state). Have output to a state file which receives the true values of the (extended) state. Also have output to an obs_sequence file which includes not only the same definition, but also adds in the true value of the observation variables and the synthetically generated observations that have samples of the observation error added onto the true observation variables.

4. Assimilation experiments: take as input an obs_sequence with at least one copy of data associated to act as observations. Outputs are assim, state and obs output files. The state output has the values of the state ensemble members both prior and posterior. The obs outputs are another obs_sequence file, this time with the prior and posterior values for the observations associated. The assim output files can be postponed for development for now.

5. Doing analysis: Analysis is done by taking input from a set of files. For observation space analysis, input files might include one that has the perfect obs (if available), the actual obs (always available) and for a filter a file with the prior and posterior obs. In the long run want to be able to sub-sample so that files don’t get huge, but that’s an add-on. For state, might have truth (if available), plus a second file with prior and posterior ensembles. Analysis programs need to be told what file name (and what copy or metadata tag within the file) are associated with the truth, the obs (for obs files), and the ensembles. They can then provide an array of plotting options.