The "new" ecological model: user guide

Pavel Sakov

last updated on May 27, 2005
version 0.12

Abstract

The new "modular" ecology code is designed as a plug-in to a host main model (typically, either hydrodynamical or box model), which is supposed to handle i/o and physical side. The main feature of this code is modularity, i.e. the user's ability to build a custom ecological model (EM) from a number of available processes without re-compilation. The modularity is supposed not only to make it easier for a user to set up a particular model but, more importantly, it translates the monolithic flux calculation procedures of the previous version into a library of ecological processes. This should improve maintainability of the ecological code and stimulate further development of the environmental software in the Environmental Modelling Group (EMG).

For a success of the new code, it is critical for a modeller (non-programmer) to be able to set a new model as well as to create new modules (processes) and modify the old ones. This document is supposed to help users with these issues.

Contents

1 Introduction

The “new” ecological model (EM) is the next-generation ecological software used by the Environmental Modelling Group (EMG), CSIRO Marine Research. It has been designed as a plug-in (module) to a host main model (typically, either hydrodynamical or box model), which is supposed to handle i/o and physical side. The main feature of this code is modularity, i.e. the user’s ability to build a custom ecological model from a number of available processes without re-compilation.

(Instead of saying “ecological model” it may be could precise to refer to the new ecological code as “ecological module”, underlying its pluggable nature. However, we will use the former term for historical reasons.)

EM is replacing the existed non-modular ecological code.

The new ecological model design has been strongly influenced by the following two requirements:

- to use C rather than C++ as a coding language, mainly because EMG's transport and hydrodynamical models are written in C, and we wanted to retain this feature and hence the potential ability to vectorise the models if necessary;
- to keep coding of new processes as simple as possible to allow a general user (not a programmer) to write code for new processes.

These requirements have somewhat restricted the possible degree of generality of the developed EM to the specific modelling context EMG exists in. As a result, EM targets marine environments composed of water, epibenthos and sediment. All processes within the model are purely local (cell-based). This limits possible interaction with physical (transport) processes, but still, due to the processing of cells within a column from top to the bottom (Sec. 2.6.3), leaves some capabilities for modelling of column-based processes.

Further on, choosing C as the programming language makes it impossible to fully encapsulate all background work within processes (memory allocation/deallocation, state variable and parameter mapping etc.). Therefore, to write or modify process code, a user still needs to possess a knowledge about the involved structures and follow a certain set of rules.

This programming side of EM is described in Sec. 2.6, while general EM characteristics and set up are described in Sec. 2.

2 Ecological model

2.1 Model structure

The EM structure is outlined on Fig. 1. It basically consists of an engine and a process library. By engine we mean servicing code which does most of the background work: reading parameters, building/destroying necessary objects, communicating with the main model and running the processes. Process library is a collection of processes. A core of a process is a flux- or value-calculating procedure(s) for specified state variables.

2.2 Interfacing

There are several levels of interfacing between the main model and EM:

- EM call level:

  Creation, stepping and destruction of EM are done by calling

  ```
  ecology* ecology_create(char* prm);
  void ecology_destroy(ecology* e);
  void ecology_step(ecology* e);
  ```

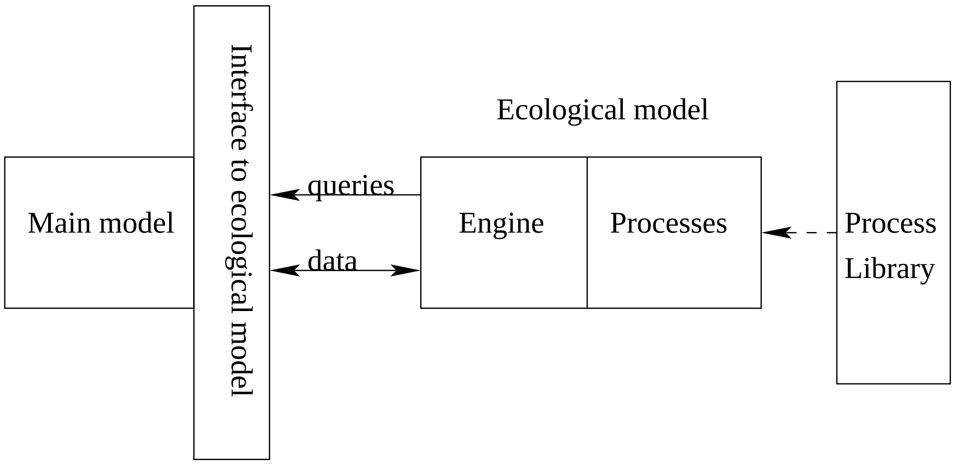

Figure 1: Ecological model structure

Before calling these procedures from the host model, one needs to include the header that contains the necessary definitions:

```
#include <ecology.h>
```

- Procedure level:

  The EM gets the necessary information from the main model via a number of pre-defined calls (queries) declared in "interface.h". Therefore, one needs to define these procedures in the main model to be able to plug in the EM.

  Unlike the previous version of EM, all queries are pretty straightforward and therefore easy to code even without in-depth knowledge of the EM code.

- Run-time level:

  At the initialisation of EM, it expects to find all state variables required by specified processes in the main model. It also expects to find all necessary parameters in the "biology" parameter file. If a necessary variable or parameter is not found, EM exits with error.

2.2.1 Model initialisation

For correct initialisation, the model looks for a number of parameters in the file specified as argument of `ecology_create(char*)`:

`biofname` – file with biology parameters;

`processfname` – file with processes;

`mandatory_water` – flag; optional, default = 1; see Sec. 2.3 for details;

`mandatory_sediment` – flag; optional, default = 1; see Sec. 2.3 for details;

`integrator` – integrator; optional; must be one of: “dopri5” (default), “dopri8”, “adapt1”, “adapt2”, “euler1”;

`integration_precision` – integration precision; optional, default = 1.0e-5;

`check_nans` – flag; optional, default = 0; see Sec. 2.8 for details;

`check_negs` – flag; optional, default = 0; see Sec. 2.9 for details.

2.2.2 Biology parameter file format

The biology parameter file must have the following format:

```
NPARAMETERS <number of parameters = N>

PARAMETER0.name <name, e.g.: "rB">
PARAMETER0.value <value, e.g.: 2.0>
PARAMETER0.units <units, e.g.: "d-1">
PARAMETER0.desc <description, e.g.: "Breakdown rate of labile resin
acids">
PARAMETER0.adjust <value, flag: whether the parameter needs temperature
adjustment>

...

PARAMETER<N-1>.name <name>
PARAMETER<N-1>.value <value>
PARAMETER<N-1>.units <units>
PARAMETER<N-1>.desc <description>
PARAMETER<N-1>.adjust <value>
```

The parameter name is a tag by which it is identified in one or more processes; the parameter value may be either a number or a string, depending on what is expected to be read for a given parameter; the units are somewhat arbitrary except that if “d-1” substring is found there, it is replaced by “s-1” and the parameter value is divided by 86400; the description is an arbitrary one-line string; “adjust” flag should be set to 1 if the parameter value should be adjusted with temperature, and to 0 otherwise. (Currently, all temperature adjustments are handled explicitely from the process code, and therefore this flag has no effect on the model performance).

Process parameter file format The process parameter file must have the following format:

```
water {
<process name>[(<process parameters>)]
...
<process name>[(<process parameters>)]
}

epibenthos {
<process name>[(<process parameters>)]
...
<process name>[(<process parameters>)]
}

sediment {
<process name>[(<process parameters>)]
...
<process name>[(<process parameters>)]
}
```

Here `<process parameters>` is a set of comma-separated strings. The exact meaning of each parameter is defined by the process initialisation code.

2.3 Spatial structure: columns and cells

The EM media consists of a number of independent vertical columns. Some columns are declared as "boundary". Each column consists of a number of cells. There are 3 types of cells: "water" cells, "sediment" cells and "epibenthic" cells:

water cells – all water cells except the bottom one

sediment cells – all sediment cells except the top one

epibenthic cells – bottom water column cell + top sediment cell + epibenthic layer in between

There is no one-to-one static correspondence between the water and sediment cells of the ecological and the host model. A dynamic map is created at each time step. The ecological model maps the host model column cells to the ecological model column cells by the following rules:

1. Only the main model water cells in the index interval [`get_wc_topk()`, `get_wc_botk()`] and sediment cells in the interval [`get_sed_topk()`, `get_sed_botk()`] are mapped. Following are the basic rules governing the creation of model cells:
2. a cell thinner than some minimal specified thickness is assumed to be "empty" and is skipped;
3. if no processes are defined in the corresponding group in the process parameter file, no cells of this types are created;

4. if there are at least one non-empty water cell and at least one non-empty sediment cell, and if there is at least one epibenthic process, then an epibenthic cell is created; otherwhile bottom water cell and top sediment cell become "normal" water/sediment cells;
5. if there are no non-empty water or sediment cells, and if the corresponding parameter, `mandatory_water` or `mandatory_sediment`, is set to 1, then the model exits with error; otherwhile all non-empty cells are created as "normal" water or sediment cells.

2.4 Model stepping

The ecological model is stepped by calling `ecology_step(ecology*)`. Internally, the ecological model step consists of a number of independent column steps. They are potentially suitable to be run in parallel. Only non-boundary columns are stepped. The column step consists of a number of cell steps. The cell stepping is guaranteed to be done from the top of the column to the bottom, starting from the top water cell, to bottom water cell, following by epibenthic cell, then top sediment cell to bottom sediment cell.

Each cell step consists of a "prestep", containing calculations to be done before the integration loop, integration, and "poststep", with calculations to be done after the integration.

2.4.1 Modelling of column-based processes

Processing cells within a column from top to the bottom gives a possibility to model some column-based processes, e.g. vertical light propagation. The intermediate results may be passed between processes via either common column variables (Sec. 2.6.5) or via value diagnostic tracers (Sec. 2.5.1).

Still, due to local (cell-based) character of stepping within EM, one must assume that the influence of upper cells on the lower ones is constant through the EM time step. For example, if the light attenuation coefficient changes substantially during the time step, there is no regular way to update the calculated average light intensity value for the lower cell. One should take this into account when choosing the EM time step.

2.5 Variables

Stepping of EM results in change of state variable values. There are 2 major types of state variables:

3D state variables – "tracers" – represent concentrations of state variables either in water or sediment.

2D state variables – "epibenthic" variables – originally represented area density of state variables attributed to the bottom biological layer. Currently may refer to any 2D state variable.

Apart from state variables, there are a number of EM internal ("common") variables for passing intermediate values between processes – see more on that in Sec. 2.6.5.

2.5.1 Tracers

The EM used to extract the necessary information about its 3D variables from an array of `tracer_info` structures obtained from the main model. However, it makes a little use of most of the parameters and flags contained in this structure apart from a very few ones. Because of that (and to eliminate the unnecessary dependence of the EM code on an external structure), EM has been modified to get the information about tracers via a few queries declared in `interface.h` and defined in the main model:

```
/* @param model Pointer to host model
 * @return Number of tracers
 */
int einterface_getntracers(void* model);

/* @param model Pointer to host model
 * @param i Tracer index
 * @return Diagnostic flag value for the tracer
 *         (0 - non-diagnostic,
 *          1 - flux diagnostic
 *          2 - value diagnostic)
 */
int einterface_gettracerdiagnflag(void* model, int i);

/* @param model Pointer to host model
 * @param i Tracer index
 * @return Tracer name
 */
char* einterface_gettracername(void* model, int i);
```

The tracer values are passed to the EM as an array of pointers to the tracer storage locations arranged for the whole water column or sediment column upside-down:

```
/* @param model Pointer to host model
 * @param b Column index
 * @return Array of pointers to watercolumn tracer values [k * n]
 * [(n=0,k=0), (n=1,k=0), ..., (n=ntr-1,k=0), (n=0,k=1), ..., (n=ntr-1,k=nk-1)]
 *
 * Requires freeing memory with free() after a call.
 */
double** einterface_getwctracers(void* model, int b);
```

```
/* @param model Pointer to host model
 * @param b Column index
 * @return Array of pointers to sediment tracer values [k * n]
 * [(n=0,k=0), (n=1,k=0), ..., (n=ntr-1,k=0), (n=0,k=1), ..., (n=ntr-1,k=nk-1)]
 *
 * Requires freeing memory with free() after a call.
 */
double** einterface_getsedtracers(void* model, int b);
```

The order of tracers in these arrays for a given cell must correspond to the tracer index used to get the tracer's name by using `einterface_gettracername(void* model, int i)` procedure.

The above interface functions are supposed to be used only in the "external" (default) mode of initialisation of variables. On the "internal" mode, see Sec. 2.5.3.

"Diagnostic" tracers. Setting the "diagnostic" flag `diagn` for a tracer to a non-zero value means that the tracer is not an independent state variable, but rather holds some derived or intermediate information about the ecological system. Such 3D variables are ignored in stepping of the main model: they are not advected, diffused, resuspended etc., hence there is no need of setting most of the fields necessary for the "real" state variables. Yet, diagnostic variables are perfectly normal from input/output point of view, and this (having them in the model output) is the main reason for their use.

There are two types of diagnostic tracers, which are defined by the value of `diagn` flag:

1. If the flag is set neither to 0 nor 2, this is a "flux" diagnostic variable. Its value is set by EM to 0 before stepping a cell. After stepping the cell, EM divides the calculated values by the time step value, thus obtaining a mean flux value of the tracer for this time step.

   A typical flux diagnostic tracers describe flux, production, grazing, removal or other dynamic tracer characteristics (e.g. production of Ammonia in the example above).

2. If the flag value is set to 0, this is "value" diagnostic. This means that its value is expected to be set directly outside integration loop. The values of "value" diagnostic tracers are set to 0 before stepping a cell and are not changed after the cell step.

   From the EM point of view, it is usually possible to replace a value diagnostic tracer by an EM internal ("common") variable (see Sec. 2.6.5); yet that such a variable would not be presented in the model output.

2.5.2 Epibenthic variables

Similar to tracers, the EM extracts the necessary information about 2D state variables from the main model by means of the following queries defined in `interface.h`:

```
/* @param model Pointer to host model
 * @return Number of epibenthic variables
 */
int einterface_getnepis(void* model);

/* @param model Pointer to host model
 * @param i Index of an epibenthic variable
 * @return Name of the variable
 */
char* einterface_getepiname(void* model, int i);

/* @param model Pointer to host model
 * @param i Index of an epibenthic variable
 * @return Diagnostic flag value for the variable
 *         (0 - non-diagnostic,
 *          1 - flux diagnostic
 *          2 - value diagnostic)
 */
int einterface_getepidiagnflag(void* model, int i);
```

The values of epibenthic variables are passed between the main model and EM via and array of pointers to the corresponding variables:

```
/* @param model Pointer to host model
 * @param b Column index
 * @return Array of pointers to epibenthic variable indices
 *
 * Requires freeing memory with free() after a call.
 */
double** einterface_getepivars(void* model, int b);
```

2.5.3 Initialisation of tracers and epibenthos

There are two different modes of tracer initialisation. In the first, “external” (default) mode, the host model provides lists of tracers and epibenthic variables via the following procedures:

```
int einterface_getntracers(void* model);
char* einterface_gettracername(void* model, int i);
int einterface_gettracerdiagnflag(void* model, char* name);
int einterface_getnepis(void* model);
char* einterface_getepiname(void* model, int i);
int einterface_getepidiagnflag(void* model, char* name);
```

while the ecology module maps particular tracers in each process to this list. It is assumed that the tracer values given by

```
double** einterface_getwctracers(void* model, int b);
double** einterface_getsedtracers(void* model, int b);
double** einterface_getepivars(void* model, int b);
```

are arranged according to the order of tracer in this list.

In the second, “internal” mode, the host model queries the ecology module about the variables it uses by the following procedures:

```
int ecology_getntracers(ecology* e);
int ecology_getnepis(ecology* e);
char* ecology_gettracername(ecology* e, int i);
char* ecology_getepiname(ecology* e, int i);
int ecology_getdiagnflag(char* name);
```

, and accepts the obtained lists for further communications (i.e when providing the tracer values). This mode is switched on by setting the flag

```
internal_tracers 1
```

in the ecology parameter file.

When there are a lot of tracers not related to the ecology module, the internal mode allows to reduce the number of variables in integration by excluding the unrelated variables. The external mode may be a preferred option when most of the variables in the host model are used by the ecology module.

2.6 Processes

The processes define changes in ecological state variables given the current state of the system. Because the EM engine takes care of integration, these changes are described simply in terms of setting the corresponding flux values.

There are a few issues arising from splitting of an ecological system dynamic into a number of processes.

- **Updating of value diagnostic variables**. A number of state variables may not belong to a particular process, but rather describe the system as a whole. E.g., “total nutrients” concentration `TN` is defined as a sum of all tracers containing Nitrogen. While dynamics of a particular tracer contributing to `TN` may be described by a number of processes, only one of them should be allowed to take care of updating `TN` in regard of this tracer. It is assumed therefore that this process is always included if any of the processes for this tracer are included in the model.
- **Variable pre-calculation**. Certain common variables or value diagnostic tracers may need to be set before a given process is run. It is assumed therefore that the corresponding processes have been run prior to this one.

At the moment, no strict mechanisms have been introduced in EM to detect errors associated with non-inclusion of a certain process or wrong process order in the process parameter file. One may hope to pick a bulk of errors in process setup by detecting `NaN`s in the model output or by model failing to find necessary common variables etc. Still, it is up to user to ensure the overall integrity of the model.

All process available to EM are listed in `processlist[]` array of `process_entry` structures in file "allprocesses.c", where `process_entry` is defined as

```
typedef struct {
    char* name;
    PROCESSTYPE type;
    int nparams;
    int delayed;
    process_initfn init;
    process_initfn postinit;
    process_initfn destroy;
    process_calcfn precalc;
    process_calcfn calc;
    process_calcfn postcalc;
} process_entry;
```

Here again `name` is a tag for the process identification; `type` and `nparams` are fields for control over the type of the process and the number of parameters it must have; `delayed` gives a possibility to demand a late execution of a process; `init`, `postinit`, `destroy`, `precalc`, `calc` and `postcalc` are the procedures associated with the process. Following are more detailed descriptions of these fields.

2.6.1 Types of ecological processes

There are 4 possible types of ecological processes:

`PT_GEN` – processes that can be run either in water or sediment

`PT_WC` – the processes that can be run in water only

`PT_SED` – the processes that can be run in sediment only

`PT_EPI` – the processes that can be run only in epibenthos

If a process is entered in a wrong section of the process parameter file, EM exits with error.

2.6.2 Process parameters

Each process entry in the process parameter file must contain a number of parameters defined by the process entry in "allprocesses.c". As was mentioned in Sec. 2.2.2, process parameters in the process parameter file are comma-separated strings in brackets following the process name. It is up to the process to decide how it interprets and uses the supplied parameters.

If a wrong number of parameters is entered for a process in the process parameter file, EM exits with error.

2.6.3 Process ordering. "Delayed" processes

For a given cell, the processes are executed by EM in the same order as they were entered in the process parameter file. There is no direct mechanism to enforce a certain order; it is entirely up to a user to ensure the right sequence.

Within a group of cells, the cells are stepped "upside down", from the very top cell to the very bottom one. In the same manner, within a "compound" epibenthic cell, first are executed processes for watercolumn "child" cell, then process for epibenthos and, finally, processes for sediment "child" cell.

However, it is possible to have a process that must be executed after all other processes for the cell. E.g., to check mass balance in an epibenthic cell, the model has to know values of relevant variables in *both* water and sediment after integration step.

`delayed` flag in the process entry in "allprocesses.c" gives such a possibility. For each cell, at running `precalc` or `postcalc` process procedures, first the procedures for processes with the `delayed` flag value of 0 are run; after that the remaining procedures are executed. The `delayed` flag has no influence on the order in which `calc` process procedures are run.

E.g., to check mass ballance for an epibenthic cell, one may need to know first the totals in the child water and sediment cells. This may be done by setting the `delayed` flag value for the epibenthic mass balancing process to one.

2.6.4 Process procedures

Each process has to define a number of procedures for its initialisation, execution and destruction:

`init` – first stage initialisation of a process. Executed once for each process in the process parameter file.

`postinit` – second stage initialisation; run after all processes have been initialised. Executed once for each process.

`destroy` – process destructor.

`precalc` – calculations to be done prior to entering integration loop. Executed once for each cell step.

`calc` – flux calculations within integration loop. Normally called several times at each cell step.

`postcalc` – calculations to be done after the integration loop. Called once per cell step.

If there is no need in one or more of these procedures for a process, `NULL` may be used in the process entry in "allprocesses.c".

2.6.5 Interprocess communication: common variables

To pass information between processes, so-called common variables are used. These variables are arrays of doubles, each having a tag stored in a parallel array of strings.

A common variable is initialised by adding a new common variable tag during the process initialisation.

There are 3 levels of common variables: model-wide, column-wide and cell-wide. All common variable values are set to `NaN` at the beginning of the corresponding step.

A typical example of using a common variable is the light calculation. To calculate light in a cell, one needs to know light intensity at the top of the cell. After the calculation is finished, the light intensity at the bottom of the cell becomes known and may be used as the top value for the next cell. To pass this value between cells, a column-wide common variable "lighttop" may be used.

During the light calculation, the value of "lighttop" common variable will be checked. If `NaN`, then the cell is the top one, and should request the light intensity at the water surface from the main model. Otherwise, it should use the value of "lighttop" and update it after finishing calculations.

Note that the calculated light values stored as the tracer "Light" represent mean (not top) light intensity for a water cell and therefore can not be used directly for light calculation in the next cell.

Another typical use of common variables is passing values between calculation procedures (`precalc`, `calc` and `postcalc`) of a process, provided that these values do not change during the step. E.g: storing a temperature-adjusted parameter value (provided that temperature does not change through the cell step). (See also Sec. 3.3.1).

2.7 Integration issues

During the cell step, integration is carried out by an integrator specified in the EM parameter file, e.g.

```
integrator dopri5
integration_precision 1.0e-4
```

At the moment, there are 5 integrators available:

- `dopri8` – 8th-order Dorman-Prince integrator with adaptive step control. Requires 13 function evaluations per step. Needs a very smooth function, in which case represents the best option for higher precisions. Here and in `dopri5` precision is a mix of absolute and relative error.

- `dopri5` – 5th-order Dorman-Prince integrator with adaptive step control. Requires 7 function evaluations for the first step and 6 for each step after. Probably the best all-around choice. Still needs a smooth function to operate properly.

`adapt2` – 2nd-order integrator with adaptive step control.

`adapt1` – 1st-order integrator with adaptive step control. Precision is a maximal relative change allowed for each variable marked with non-zero value in the external array of flags.

```
extern int* essential;
```

`euler1` – Explicit 1st-order scheme with no precision control. Makes one function evaluation per step, which makes it the best debugging option.

2.8 Checking for NaNs

Many variables in EM are initialised to `NaN` to guarantee that a possible omission of initialisation would be detected. If `NaN`s have been detected during the model run or in the model output, to pinpoint the error source, one should rerun the model with the flag `check_nans` being set to 1 in the EM parameter file. In this case, EM model checks all state variables for being `NaN` or `inf` after every process procedure.

2.9 Checking for negative concentrations

To ensure that no negative tracer concentrations occur as a result of running ecological code, one may run the model with the flag `check_negs` set to 1.

2.10 Multi-threading

On machines with symmetric multi-processing (SMP) capabilities, it is possible to perform column stepping within the ecological model in parallel. To compile multi-threading into the ecology code, `-DNCPU=2 -pthread` must be added to the compiler flags in the ecology code directory; `-pthread` must be added to the compiler flags in the directory of the main model. Having multi-threading compiled in, it may be switched on by

```
multithreaded 1
```

in the EM parameter file.

Because running multi-threaded model involves some overhead, there may be small or even no gain in some cases. The multi-threading is particularly useful for stiff models, when the number of integration substeps within a single ecology step is of order of 10 or more, when (on a 2-processor machine) it can accelerate a model by a factor of 1.5 or more.

3 Creating a new process: basic steps

At run time, for each ecological process specified in the process parameter file (Sec. 2.2.2), the following process variable is created:

```
struct process {
    ecology* ecology;
    PROCESSTYPE type;
    char* name;
    char* fullname; /* name + arguments */
    stringtable* prms;
    int delayed; /* flag */

    process_initfn init;
    process_initfn postinit;
    process_initfn destroy;
    process_calcfn precalc;
    process_calcfn calc;
    process_calcfn postcalc;

    void* workspace;
};
```

Here `ecology` is a pointer to the ecological model; `type` – the process type (Sec. 2.6.1); `name` – the process tag; `fullname` – the full entry in the process parameter file; `prms` – array of strings containing process parameters; `delayed` – flag (Sec. 2.6.3); `init`, `destroy`, `precalc` and `postcalc` – process procedures (Sec. 2.6.4); `workspace` - local data.

Most of these fields are filled directly from the corresponding entry in "allprocesses.c" found by the process name. The process-specific data are stored in local structures pointed by `workspace`.

Following are descriptions of main steps necessary for adding of a new process to the EM process library.

3.1 Writing a process header file

The process header file is necessary for definition of `allprocesses` array in "allprocesses.c". This is an easy part, as the header file must simply contains declarations of process procedures according to the following type definitions:

This is the easiest part: the *.h file should only contain declarations for the init, destroy, precalc, calc and postcalc procedures for a process (see Sec. 3) according to the following type definitions:

```
typedef void (*process_initfn) (process* p);
typedef void (*process_calcfn) (process* p, void* pp);
```

Thus, a typical process header file for an ecological process would look like:

```
/* file: dinoflagellate_grow_wc.h */

#if !defined(_DINOFLAGELLATE_GROW_WC_H)
```

```
void dinoflagellate_grow_wc_init(process* p);
void dinoflagellate_grow_wc_postinit(process* p);
void dinoflagellate_grow_wc_destroy(process* p);
void dinoflagellate_grow_wc_precalc(process* p, void* pp);
void dinoflagellate_grow_wc_calc(process* p, void* pp);
void dinoflagellate_grow_wc_postcalc(process* p, void* pp);

#define _DINOFLAGELLATE_GROW_WC_H
#endif
```

3.2 Writing a process .c file: general issues

This is the step where most of the effort goes in. Before describing the sub-steps involved, the problem of accessing model state variables and parameters is considered.

3.2.1 Accessing the media

To make a column or step cell, corresponding column and cell variables are created. To access column or cell-wide information, one needs to get the corresponding handle (pointer) from within `calc`, `precalc` and `postcalc` process procedures. (The other process procedures – `init`, `postinit` and `destroy` – are system-wide, so there must be no need to access cell media from them).

To access cell variable from `precalc` or `postcalc` procedures, one needs simply to cast the second procedure variable:

```
void dinoflagellate_grow_wc_precalc(process* p, void* pp)
{
    cell* c = (cell*) pp;
<...>
}
```

For `calc` procedure, the casting is done in two steps,

```
void dinoflagellate_grow_wc_calc(process* p, void* pp)
{
    intargs* ia = (intargs*) pp;
    cell* c = ((cell*) ia->media);
<...>
}
```

because of the way integration has been design.

The corresponding column variable is accessed through the cell variable,

```
    column* col = c->col;
```

and the model-wide information may be accessed via the model variable stored in process, cell and column variables, e.g.:

```
    ecology* e = c->e;
```

3.2.2 Accessing EM parameters

The EM parameters are normally accessed by using one of the following procedures declared in "utils.h":

```
/** Gets string value for a parameter. Exits with error message if the
 * parameter is not found.
 * @param e Pointer to ecology
 * @param s Parameter name
 * @return Parameter string
 */
char* get_parameter_stringvalue(ecology* e, char* s);

/** Gets parameter value for a parameter. Exits with error message if the
 * parameter is not found.
 * @param e Pointer to ecology
 * @param s Parameter name
 * @return Parameter value
 */
double get_parameter_value(ecology* e, char* s);

/** Gets parameter value for a parameter. Unlike get_parameter_value(),
 * goes on if the parameter is not found.
 * @param e Pointer to ecology
 * @param s Parameter name
 * @return Parameter value if found; NaN otherwise
 */
double try_parameter_value(ecology* e, char* s);
```

3.2.3 Accessing state variables

All processes in EM are run locally for a given cell. Before stepping the cell, a local array `y` filled with state variable values is created. After stepping the cell, these values are copied back into the main model.

For "simple" water or sediment cells, the index of a variable in this array coincides with the index of the corresponding tag (variable name) in a parallel array of tags stored in structure `ecology` (and with index of corresponding entry in `tracer_info` array in the main model). To get this index, two procedures,

```
/** Wrapper to stringtable_findstringindex(). Exits with error message
 * if the string is not found.
 * @param st Pointer to stringtable
 * @param s String
 * @return Index associated with the string
 */
int find_index(stringtable* st, char* s);
```

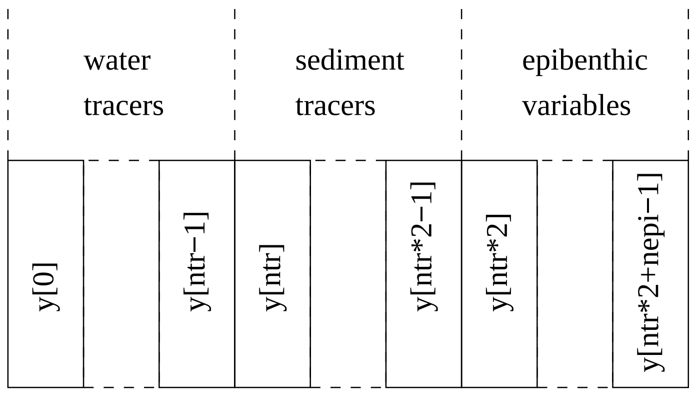

Figure 2: Composition of state variable array in epibenthic cells

```
/** Wrapper to stringtable_findstringindex(). Unlike find_index(), goes on if
 * the string is not found.
 * @param st Pointer to stringtable
 * @param s String
 * @return Index associated with the string
 */
int try_index(stringtable* st, char* s);
```

are used with corresponding model-wide string arrays `tracers` or `epis` as the first argument, e.g.:

```
void dinoflagellate_grow_wc_init(process* p)
{
    ecology* e = p->ecology;
    stringtable* tracers = e->tracers;
<...>
    ws->Phy_N_i = find_index(tracers, "PhyD_N");
<...>
}
```

For “compound” epibenthic cells, this array is constructed of three consecutive concatenated arrays of water tracer concentrations, sediment tracer concentrations and epibenthic variables (Fig. 2).

Therefore, indices returned by `find_index()` must be offset by the number of tracers for sediment tracers and by the doubled number of tracers for epibenthic variables.

In reality, the things are a bit more complicated. The above description refers to the “external” mode of initialisation (see Sec. 2.5.3), when the final lists of variables are known at the process initialisation stage. In the “internal” mode, the variables are added to the lists while the processes initialise. It is therefore necessary to expand the lists rather than exit with error if a variable is not found in a list for this mode. This is done by using `find_index_or_add()` instead of `find_index()` or `try_index()` in this mode.

The ecology code uses the appropriate procedure by setting the functions `find_index` and `try_index` in the ecology structure to the functions required for the used initialisation mode. For a process code to be able to operate in both internal and external initialisation mode one must use the functions from the ecology structure rather than the functions `find_index()` or `try_index()` directly, e.g.:

```
void massbalance_wc_init(eprocess* p)
{
    ecology* e = p->ecology;
    stringtable* tracers = e->tracers;
    workspace* ws = malloc(sizeof(workspace));

    p->workspace = ws;

    ws->TN_i = e->find_index(tracers, "TN", e->verbose);
    ws->TP_i = e->find_index(tracers, "TP", e->verbose);
    ws->TC_i = e->find_index(tracers, "TC", e->verbose);
    ws->Nfix_i = e->try_index(tracers, "Nfix", e->verbose);

    <...>
}
```

3.2.4 Accessing/creating common variables

Depending on the media they relate to (whole model, a column, or a cell), common variables are stored in the corresponding media variables under the name `cv`. Similarly to state variables, to get an index for a needed common variable, one needs to find index of the corresponding entry in the parallel tag (name) array. Like all name arrays, these are stored in the `ecology` structure under names `cv_model`, `cv_column` and `cv_cell`. To find the index, `try_index()` or `find_index()` procedures may be used.

Unlike state variables, common variables are dynamically created within EM by processes. To add a new entry for a common variable if necessary, procedure `find_index_or_add()` should be used:

```
/** Finds string index in a stringtable; adds a new entry if not found.
 * Sorts the stringtable after adding a new entry.
 * @param st Pointer to stringtable
 * @param s String
 * @return Index associated with the string
 */
int find_index_or_add(stringtable* st, char* s);
```

E.g.:

```
void light_wc_init(process* p)
{
    ecology* e = p->ecology;
    workspace* ws = malloc(sizeof(workspace));
<...>
    ws->lighttop_i = find_index_or_add(e->cv_column, "lighttop");
<...>
}
```

3.2.5 Temperature adjustment

There are a number of ways of adjusting a parameter according to the cell temperature (or other state variable). A specific scheme employed in the current process library is described below.

Temperature adjustment is made by multiplying "base" values of relevant parameters by a temperature dependent factor `Tfactor`, which is calculated by a special process "tfactor" and stored as a common cell variable:

```
/* file: dinoflagellate_grow_wc.c */
<...>
typedef struct {
<...>
    int Tfactor_i;
    int umax_i;
} workspace;
<...>
void dinoflagellate_grow_wc_init(process* p)
{
    ecology* e = p->ecology;
<...>
    workspace* ws = malloc(sizeof(workspace));

    p->workspace = ws;
<...>
    ws->Tfactor_i = try_index(e->cv_cell, "Tfactor");
    ws->umax_i = find_index_or_add(e->cv_cell, "DFumax");
}
<...>
void dinoflagellate_grow_wc_precalc(process* p, void* pp)
{
    workspace* ws = p->workspace;
    cell* c = (cell*) pp;
    double* cv = c->cv;
<...>
    double Tfactor = (ws->Tfactor_i >= 0) ? cv[ws->Tfactor_i] : 1.0
<...>
    cv[ws->umax_i] = ws->umax_t0 * Tfactor;
<...>
}
<...>
```

Because the temperature-adjusted value is cell-specific, it should be stored as a common cell variable as well rather than in the process workspace.

If "tfactor" process has not been entered in the process parameter file, the value of `ws->Tfactor_i` in the above example is initialised to -1, and the value of the parameter is set to its base value.

3.3 Writing a process .c file

Following are descriptions of principal stages in writing a process .c file.

3.3.1 Creating the process workspace

To run effectively, process should move maximum of calculations to the initialisation stage. Typically, this means getting indices of the necessary state and common variables as well as storing the necessary parameter values.

These local process data is stored in a local structure `workspace`, e.g:

```
/* file: light_wc.c */
<...>
typedef struct {
    /* parameters */
    double k_swr_par;

    /* tracers */
    int Light_i;
    int Kd_i;

    /* common variables */
    int lighttop_i;
} workspace;
<...>
```

As there may be several column steps run in parallel, the process workspace must contain no media-dependent data. To store such data, common variables may be used (Sec. 2.6.5).

3.3.2 Writing the process "init" procedure

The process `init` procedure creates and initialises the process workspace. It contains operations to be done once and only only once before running the model. E.g.:

```
void light_wc_init(process* p)
{
    ecology* e = p->ecology;
    stringtable* tracers = e->tracers;
    workspace* ws = malloc(sizeof(workspace));

    p->workspace = ws;

    ws->k_swr_par = try_parameter_value(e, "k_swr_par");
    if (isnan(ws->k_swr_par))
ws->k_swr_par = 0.53;

    ws->Light_i = find_index(tracers, "Light");
    ws->Kd_i = find_index(tracers, "Kd");

    ws->lighttop_i = find_index_or_add(e->cv_column, "lighttop");
}
```

When finding indices of state variables in an epibenthic cell, it is important to add necessary offsets for sediment tracers and epibenthic variables (see Fig. 2), e.g.:

```
/* file: example_epi.c */
<...>
typedef struct {
    int tracer_wc_i;
    int tracer_sed_i;
    int epivar_i;
} workspace;

void example_epi_init(process* p)
{
    ecology* e = p->ecology;
    workspace* ws = malloc(sizeof(workspace));
    stringtable* tracers = e->tracers;
    stringtable* epis = e->epis;

    int OFFSET_SED = tracers->n;
    int OFFSET_EPI = tracers->n * 2;

    p->workspace = ws;

    ws->tracer_wc_i = find_index(tracers, "tracer");
    ws->tracer_sed_i = find_index(tracers, "tracer") + OFFSET_SED;
    ws->epivar_i = find_index(tracers, "epivar") + OFFSET_EPI;
}
<...>
```

```
}
```

3.3.3 Writing the process "postinit" procedure

Sometimes, when process needs to make decisions depending on presence of other processes, a need in second-stage process initialisation occurs, e.g.:

```
void dinoflagellate_grow_wc_postinit(process* p)
{
    ecology* e = p->ecology;
    workspace* ws = (workspace*) p->workspace;

    ws->do_mb = (try_index(e->cv_model, "massbalance_wc") >= 0) ? 1 : 0;
}
```

Here `do_mb` is a flag of whether the mass balance calculations are run in the model or not. Putting this code in `postinit` procedure guarantees that at the moment when it runs, all first-stage initialisation procedures have been completed.

3.3.4 Writing the process "precalc" and "postcalc" procedures

The process pre- and post-integration procedures contain calculations to be done outside integration loop. This is usually confined to setting of value diagnostic tracers and common variables.

3.3.5 Writing the process "calc" procedure

The process `calc` procedure modifies or sets fluxes for relevant state variables. State variable fluxes are stored in an array `y1`, which is accessed in exactly the same way as the array `y` of state variable values:

```
void example_calc(process* p, void* pp)
{
    workspace* ws = p->workspace;
    intargs* ia = (intargs*) pp;
    double* y = ia->y;
    double* y1 = ia->y1;
<...>
    y1[ws->tracerA_i] += <...>;
<...>
}
```

The compound flux procedure for a cell consists of a sequence of process flux procedures ran in the order corresponding process entries in the process parameter file. All flux values are initialised to 0.

3.3.6 Writing the process ”destroy” procedure

The process `destroy` procedure must take care of releasing the memory allocated by the process. Typically, this involves just deallocating the process workspace:

```
void light_wc_destroy(process* p)
{
    free(p->workspace);
}
```

3.4 Modifying “allprocesses.c”

The file ”allprocesses.c” is a repository with entries for all processes available in the process library (on definition of the `process_entry` structure – see Sec. 2.6). After writing code for a new process, the user must update the repository: include the corresponding header file and update process_entries array, e.g.:

```
/* file: allprocesses.c */
<...>
#include "process_library/light_wc.h"
<...>
process_entry process_entries[] = {
<...>
    {"light_wc", PT_WC, 0, 0, light_wc_init, NULL, light_wc_destroy,
     light_wc_precalc, NULL, NULL},
<...>
};
<...>
```

The file also contains a list `diagnflags` of names of all variables (tracers and epibenthic variables) with associated values of diagnostic flag used in the processes in the process library:

```
diagnflag_entry diagnflags[] = {
    /*
     * tracers
     */
    {"temp", 0},
    {"salt", 0},
<...>
    /*
     * epibenthic variables
     */
    {"MA_N", 0},
    {"MA_N_pr", 1},
<...>
};
```

The purpose of this list is solely to allow verification that the correct value of the flag is set in the host model. If the new process introduces a new tracer, a corresponding entry must be added to the list.

3.5 Adding/removing a new process to/from the model

Adding or removing a new process to/from an ecological model is as easy as adding or removing the corresponding string entry to the process parameter file (Sec. 2.2.2). The new process may require a number of new tracers, epibenthic variables and parameters, otherwhile the ecology code will exit with error during initialisation.

3.6 Parallelisation issues

Because the individual columns may be stepped in parallel (Sec. 2.10), it is necessary to ensure that no data used in stepping a column can be modified by stepping another column. In practical terms, this means that no process should contain any local (column or cell) data; such data should be stored by using cell or column common variables (Sec. 2.6.5).

Note: a typical indication of violating this rule is a random appearance of mass balance errors during a model run with multi-threading switched on.