Eco-Hydrological Simulation Environment (echse)

Documentation of the Generic Components

Author	David Kneis
Affiliation	Institute of Earth and Environmental Sciences
	Hydrology & Climatology Section,
	University of Potsdam, Germany
Contact	david.kneis [at] uni-potsdam.de

Project	PROGRESS
Sub-project	D2.2
Funding	German Ministry of Education and Research (BMBF)

Last update May 7, 2014

Please help to improve this document by sending suggestions, corrections, wishes, and other useful feedback to the author (see above).

Contents

Chapter 1

Introduction

1.1 The echse simulation environment

The idea of a *simulation environment* is to provide a tool which can be used to simulate different systems and/or processes in a single unified software environment. Terms sometimes used more or less synonymously are *modeling framework*, *generic model* or *open structure* model. Examples of existing modeling frameworks in the field of earth and environmental sciences include the *Object Modeling System* (Ahuja et al., 2005) and the *Earth System Modeling Framework* (Hill et al., 2004). Examples from the field of water quality modeling include, for example, *AQUASIM* (Reichert, 1998), the biogeochemical reactions network simulator BRNS (Regnier et al., 2002; Thullner et al., 2005), and the *ECO Lab* software (DHI, 2006).

The benefit of a modeling framework usually emerges in situations, where

- new models have to be developed in short time.
- a preliminary model has to be build and later improvement (possibly by different staff) is planned.
- alternative model structures are to be compared (to find an optimum structure or to learn about structural uncertainty).
- different people are involved in collaborative model development.
- a larger number of individual models must be used and a common (user) interface for all models is required (in operational forecasting, for example).

A basic characteristics of a modeling framework is the flexibility to simulate *objects* of different *classes* [1].

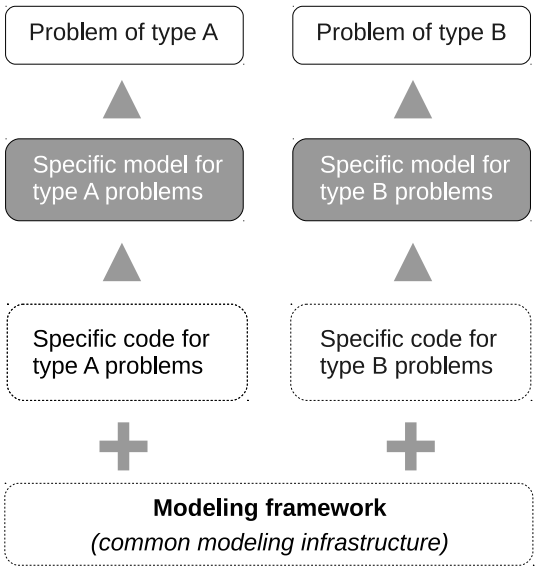

Figure 1.1: *Basic idea of a modeling framework.*

Typically, the features of a class, which include *data* and *methods* [2] are declared/defined by the developer of a specific model for a specific purpose. In contrast to that, the generic core of the modeling framework represents the static part of the software, providing the basic infrastructure for all models (Fig. 1.1).

The **echse** is intended to be a lightweight, simple to use modeling framework, being applicable to many (but not all) simulation problems, arising in the field of (eco)-hydrology. Details on potentials and limits are summarized in Sec. 1.2 and discussed in more detail in Chap. 2.

It is important to understand that the **echse** simulation environment actually consists of two parts:

[1] An approximate synonym is *types*.

[2] An approximate synonym is *functions*.

The generic model core This is a collection of source files. These files provide the common modeling intrastructure shown at the bottom of Fig. 1.1.

The code generator This is a software (currently implemented in R) to generate a large part of the *application-specific* source code from basic information provided by the model developer. The generated source code is guaranteed to be compatible with the source code of the generic model core.

In order to create a specific model (grey boxes in Fig. 1.1), the model developer finally has to complement the generated source code by implementing a set of methods (functions) with a simple, well defined interface. Only at this step, source code has to be written manually.

1.2 Potential uses and limits

Since the potential model applications in the field of eco-hydrology are so diverse, there is (and there cannot be) a modeling framework which is equally suitable for all those applications. Consequently, a 'good' modeling framework is usually one that is specialized on a certain range of applications (as opposed to a 'normal' model, that is specialized on a certain application alone).

The **echse** has been developed in the context of hydrological catchment modeling and water quality modeling. Therefore, this modeling framework is particularly specialized on

- the simulation of a collection of objects representing instances of different classes (e. g. catchments, river sections, lakes, etc.).
- the simulation of object interactions that are mostly of the *feed-forward* type, i. e. the simulated flow of mass, energy, or information is mostly unidirectional. *Feedbacks*, i. e. two-way interations between objects, may also be simulated but there are currently limitations with respect to the accuracy of results.

The current version of the **echse** is *not* recommended for building models that

- are dominated by feedback interactions between the simulated objects. That is, for example, the case in ground water or hydrodynamic modeling, where *partial differential equations* (PDE) have to be solved. The concept of the **echse** currently does not support high-accuracy solutions of PDE.
- consist of a single object only. Simualting a single object is not a practical problem, but the use of other modeling tools may simply be more efficient.

1.3 Required user skills

1.3.1 Use of existing models

The skills required for using an existing model built with the **echse** are the same as for any other dynamic system model. You basically need to

- understand the characteristics of the implemented classes (from a documentation of the specific model).
- know which input files are required (see Chap. 3).
- be able to create all input files. This can be done manually (for small projects only), by writing skripts (for example using R, Matlab, Python, etc.), and/or by using other programs such as spreadsheet software, geographical information systems, or data bases.
- understand the limits of the implemented model with respect to your specific application.

1.3.2 Development of models

As with all modeling frameworks, the **echse** aims at reducing the effort for building new models and for changing/extending existing ones. Thus, you don't need to be a professional code writer. However, to successfully create or modify models, you should

- understand the meaning of the terms 'class' and 'object' (see any introduction on object-oriented programming),
- know the different features of a class supported by **echse** and understand the meaning of the classes' 'simulate' methods (see Chap. 2),
- have basic knowledge of ordinary differential equations and their use in the simulation of dynamic systems,
- be able to program simple algorithms in any language,

- be willing to get familiar with the most basic elements of C++ (basic data types, operators, and flow-control) or find someone who will translate (or wrap) your code if written in another language.

Chapter 2

Basic concepts

2.1 Important terms

To understand the concept behind all models created with the **echse** simulation environment, one must know the meaning of the terms *class*, *object*, and *object group*. These terms are defined in the following sections (see also Fig. 2.1).

2.1.1 Objects

Objects are the basic building blocks of any model created with the **echse** simulation environment. An object in the model typically represents a real-world object (such as a tree, a lake, a soil column, etc.). Usually, the object in the model is a simplified, abstract description of the corresponding real-world object, i. e. it describes only its most important characteristics (for example height, average diameter, and age of a tree). However, an objects does *not necessarily* correspond to an entity existing in the real-world. For example, the function of such a more abstract object may be to simply collect information on some other objects and to supply this information to a third object (like a kind of observer).

Technically speaking, an object always represents an instance of an underlying class (see Sec. 2.1.2). For example, a single tree object is an instance of the tree class. In a typical model, (1) there are multiple instances of the *same* class (such as multiple trees) and (2) multiple objects of *different* classes do co-exist (such as trees and lakes).

The basic features of an object, i. e. the information and functionality linked to that object, are always determined by the corresponding class (a tree has a diameter and may grow, a lake has a depth and its storage may change). The general features of classes are described in Sec. 2.2.

In a typical model, the objects (no matter, of which class) do interact in some way. These interactions typically represent the exchange of matter, energy, or information between the corresponding real-world entities. For example, two lakes could exchange water via a connecting channel or the growth of a tree might depend on a lake's water level.

The collection of all interacting objects is typically called the *model*.

2.1.2 Classes

A class represents an abstract prototype for a certain type of object (*type* is an approximate synonym for *class*). A class describes the features of *all* objects that are instances of that particular class. In the language of object-oriented programming, the features of a class are typically called 'class members'. Such member either represent data (i. e. information) or methods (i. e. algorithms, describing the functionality of an object of that class). For example, a class 'lake' might have the water level, the storage volume, and the geo-coordinates of its center as data members. These data members (not to be confused with the actual values) are then common to all instances of the class, i. e. all lake objects.

Generally, a class is distinguished from other classes by

- its data members (for example, the number and names of state variables) and/or,
- its methods. In the context of the **echse**, each class has only a single visible method called 'simulate'. This method typically describes the dynamics of the class' state variables.

The members of classes are introduced in detail in Sec. 2.2.

2.1.3 Object groups

The term object group is used for all instances (i. e. objects) of a particular class. If a forest of individual trees is modeled, for example, all trees belong to the same object group. Though, in many instances, the terms *class* and *object group* are (and can be) used synonymously, they are not actually interchangeable:

A class is the prototype of all objects with the same data and functionality.

An object group represents the array of all objects (i. e. instances) of a particular class.

2.1.4 Summary

The relation between the terms *class*, *object*, *object group*, and *model* is illustrated with an example in Fig. 2.1. Another view on the relations between these terms is provided in Fig. 2.2. This figure is intended for those who are familiar with basic techniques of object-oriented programming. Shown are 8 objects, which belong to 2 different classes 'A' and 'B'. All these 8 objects inherit from an abstract base class 'abstractObject'. It is therefore possible to keep handles to all these objects (of different classes) in a single array by using base-class pointers (i. e. by treating them as objects of the base class). In the same way, handles to all object groups can be stored in a single array since they all inherit from an abstract base class 'abstractObjectGroup'. In each object group, an arbitrary number of objects may be declared. In contrast to that, only a single instance of each object group can exist.

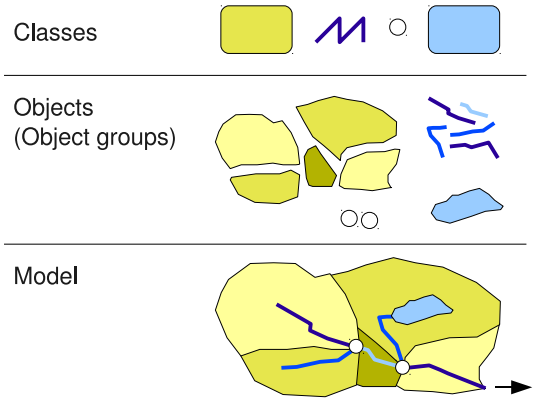

Figure 2.1: *Relation between the terms* class, object, object group, *and* model *with the example of a hydrological catchment model, consisting of sub-catchments (green polygons), river reaches (blue lines), lakes (blue polygons), and river nodes (circles).*

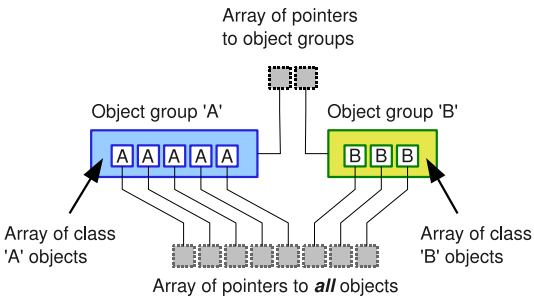

Figure 2.2: Classes, objects, *and* object groups *from a programmers point of view.*

2.2 Features (members) of a class

2.2.1 Overview

An overview of the features (precisely: members) of a class is given in Fig. 2.3. Details on the data members are provided first in Sections 2.2.2 to 2.2.5. The 'simulate' method, which is the most important member function of a class, is addressed in 2.2.6.

2.2.2 State variables

State variables describe the state of an object at a certain point in time. State variables are dynamic data, i. e. their values may change over time. Consequently, at the start of a simulation, their values must be initialized. In **echse**-based models, a class may contain both single-valued (scalar) and vector-valued state variables.

Scalar state variables

Scalar state variables are state variables that take a single value only. Looking at a reservoir, for example, the storage volume is a scalar state variables, since it can be expressed as a single number. In contrast to that, the reservoirs' water depth (as it is spatially variable) is not a scalar variable by nature. The average depth, however, may be treated as a scalar state variable.

Vector state variables

In contrast to a scalar state variables, *vector state variables* are vector-valued, i. e. their value(s) cannot be adequately expressed by a single number. For example, a vector state variable may be required to adequately describe the temperature in a deep reservoir. Due to stratification, there are often significant vertical temperature gradients which often cannot be (convieniently) described by a single value (i. e. a scalar state variable). Another example of a vector state variable is the water level of a river reach, measured at multiple stations along that reach.

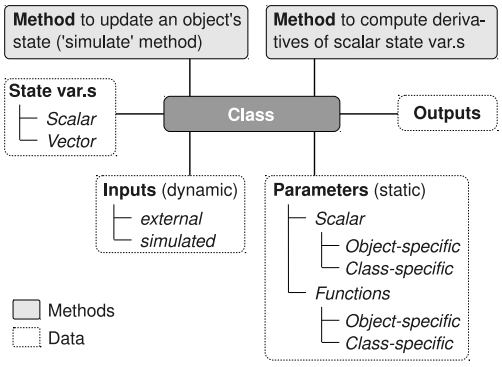

Figure 2.3: *Overview of the features (members) of a class. The dashed line separates data members (below) from class methods (above line).*

2.2.3 Input variables

Input variables (also called forcings) represent time-variable data, representing the dynamic environment of an object. Typically, changes in the values of an object's state variables are triggered by changes in the input variables. In **echse** models, *external* and *simulated* (synonym: internal) inputs are distinguished.

External inputs

External input variables are variables, whose dynamics is *not* simulated by the model. Instead, the dynamics is prescribed, i. e. the values must be known in advance for the entire modeling period. The model reads those data from time series files (see Sec. 3.8). When simulating the temperature of a reservoir, for example, solar radiation and air temperature are typically external input variables (since the atmosphere itself is not part of the model). Values of the external input variables usually represent observations (when simulating the past). In the context of forecasting, the values often originate from forecasts which have been produced by an external model. For example, a hydrological model for medium-term stream flow forecasting uses the forecasts produced by a numerical weather prediction model as input.

Simulated inputs

The values of simulated input variables are computed *within* the model itself. From the perspective of an object, a simulated input variable is a variable, whose values are supplied by another object, i. e. the existence of such variables is bound to interactions between objects. More precisely, a simulated input variable of an object 'A' is always linked to an output variable (see Sec. 2.2.5) of another object 'B'. This is due to the fact that only the output variables of an object are visible to (and accessible by) other objects. A typical example for the use of simulated inputs is the 'reservoir' class

in a hydrological model. From the perspective of a reservoir, the inflow is a simulated input variable, if the values are supplied by an upstream object. The corresponding output variable of the upstream object is usually an outflow rate (of a reach) or a runoff rate (from the reservoir's catchment).

2.2.4 Parameters

Those properties of an object which are static (i. e. which do not change over time), are called *parameters*. As outlined in Fig. 2.3, different kinds of parameters are supported by the **echse**. These are described in detail in the subsequent sections.

Scalar parameters

Scalar parameters are, like scalar state variables, characterized by the fact that they are single-valued. Thus, the value of a scalar parameter is always just a single number. In the **echse**, two types of scalar parameters are distinguished:

Object-specific scalar parameters : The value of these parameters are specific for a particular object (of a particular class). For example, a 'catchment' class could have an object-specific scalar parameter 'area'. Then, values of the area may be assigned to each catchment object individually.

Group-specific scalar parameters : The value of such a parameter cannot be set for individual objects. Instead, a common value is assigned to *all* objects of a particular class. For example, in a 'catchment' class, the long-wave emissivity of the snow cover could be declared as a group-specific scalar parameter, if a common value for all modeled catchments is appropriate.

Note: Hard-coded scalar parameters, i. e. the definition of constants in the 'simulate' method of a class, provide(s) an alternative to group-specific scalar parameters. The use of hard-coded parameters is preferable *only* if it is known that the values are strictly constant. This is typically the case for physical constants with a well known value (such as the specific heat capacity of water). The drawback of using hard-coded parameters is that any modification of the values requires the source code to be re-compiled. Note that, strictly speaking, such hard-coded parameters are not *data members* of the class and, therefore, they do not show up in Fig. 2.3.

Parameter functions

In many situations, some static object properites need to be represented by functions instead of scalar parameters. An example is the relationships between water depth and storage volume in a river reach or lake. The **echse** basically supports two concepts of functions:

1. Tabulated functions (synonym: lookup tables).
2. Analytical expressions.

Tabulated functions: Lookup tables provide a means to describe also those functional relations between two entities which cannot be reasonably captured by an analytical expression. Although, in many cases, a piecewise polynomial representation might be possible, lookup tables offer a more flexible and convenient alternative. The **echse** supports tabulated functions as long they have a single argument only. There is support for both functions with regular (i. e. equally spaced) arguments and functions with non-regular arguments. Like in the case of scalar parameters, two types of such lookup-based parameter functions may be distinguished:

Object-specific parameter functions : This type of function is object-specific, i. e. an individual lookup table is assigned to each object (of a particular class). For example, the rating curve might be declared as an object-specific parameter function in a 'gage' class, since each gage has its own characteristic rating curve.

Group-specific parameter functions : Such a function is not associated with an individual object. Instead, it represents a common function which is accessible to all objects (of a particular class).

Analytical expressions: If a function can be captured by a single (or few) analytical expression(s), then it is typically hard-coded, i. e. the function is defined in the 'simulate' method of a class. It is then, strictly speaking, not a *data member* of the class and, therefore, hard-coded functions do not show up in Fig. 2.3. Hard-coded analytical functions include, for example, polynomials, and linear, exponential, or power functions. The advantage of using them is that the function's return value may usually be computed more quickly as compared to table-lookup.

A typical case of an analytical function that one would hard-code is the Magnus-Formula, which is is an empirical expression relating the air's maximum humidity to air temperature. It is an example of a function which is *not* object-specific since it is practically applicable everywhere on earth.

However, it is quite straightforward to make hard-coded functions object-specific. This is simply achieved by passing the coefficients of analytical expressions via the functions interface and to define those coefficients as object-specific *scalar parameters*. For example, a rating curve may sometimes be expressed by a power expression like $Q = a \cdot H^b$, with a and b being empirical coefficients and Q and H representing discharge and stage, respectively. In such a case, one may declare a and b as object-specific *scalar* parameters in a 'gage' class to let each gages have its individual rating curve.

Note that hard-coded analytical functions provide the only way of implementing functions that take multiple variable arguments (multi-dimensional functions). This is due to the fact that there is currently no support for multi-dimensional table lookup. In some situations, however, it may be possible to split a multi-dimensional function into several single-argument functions which may then be represented by lookup tables.

2.2.5 Output variables

To make information about an object visible to (and usable for) its environment, *output variables* must be declared in the respective class. In particular, output variables have to be declared in a class for all data, which

- should to be passed from an object of that class to another simulated object.
- are of interest to the modeler and should (potentially) be available in the output files.

In a hydrological catchment model, for example, a 'catchment' class might have an output variable 'runoff'. Then, the values of that variable may serve as an input to an object of class 'reach', for example, provided that a corresponding *simulated input* variable (see Sec. 2.2.3) exists in the 'reach' class. Furthermore, the existance of the output variable 'runoff' allows for writing the computed runoff for user-selected catchments to the respective output files.

In each class, at least a single output variable should be defined because objects of that class are otherwise useless. Typically, output variables are used to retrieve information on

- state variables.
- flux rates, such as time-step averages of energy or mass fluxes.

However, there are practically no limitations, i. e. any scalar value which is computed (or which is accessible) in the 'simulate' method of a class (see Sec. 2.2.6) can be assigned to an output variable.

2.2.6 The 'simulate' method

Purpose and interface

The 'simulate' method of a class represents the class' most important member function which needs to be defined by the model developer.

The purpose of the 'simulate' method is to simulate the evolution of an object over a period of length Δt. This is usually equivalent to solving a so-called *initial value problem* which means that

1. the values of the object's n state variables at an initial time t_0 are known.
2. the values at time $t_0 + \Delta t$ are to be computed by integrating n ordinary differential equations (one ODE per state variable).

Thus, the 'simulate' method usually implements a solution of the initial value problem. The ordinary differential equations (ODEs) to be solved are specific for each class and they typically describe either a mass or energy balance (see example in Sec. 2.4). Whether the integration can be performed using simple approaches (such as a first order Euler method) or whether sophisticated ODE solvers (see e. g. Press et al., 2002) are required, depends on the specific problem. In addition to the updating of state variables, the 'simulation' method is responsible for calculating all auxiliary numeric data, which are of interest to the model user, i. e. model outputs (see Sec. 2.2.5).

In C++ notation, the interface of the 'simulate' method looks like

```
.simulate(const unsigned int delta_t)
```

where `delta_t` is the function's (only) argument. It is of type unsigned integer and represents the simulation time step in seconds. Note that no other information are passed via the function's argument list.

Class-specific behavior

The leading dot in the function's name (see interface above) indicates that this function is a class member. Formally speaking, it is a *virtual* member of the *abstract* class 'abstractObject' describing a generic object (Fig. 2.4). The child classes describing a specific type of (usually real-world) objects are derived from that abstract parent class. Through the mechanisms of inheritance, each child class automatically has a 'simulate' method with the interface shown above. Although the function's name is the same, the interior of the method is *class-specific*, since the implementation is only present in the child classes but not in the base class (Fig. 2.4). This makes it possible to use identical calls like

```
reservoir_xy.simulate(3600)
catchment_288.simulate(3600)
```

to trigger the simulation of two objects, which are instances of *different* classes (a 'reservoir' and a 'catchment', in this example). The appropriate code for each object is selected automatically at run-time, based on the type information.

Access to an object's data

As mentioned earlier, the time step `delta_t` is the only information passed to the 'simulate' method via the argument list. All object-related data, such as the values of parameters, inputs, and state variables, are available through class methods. These methods, which may also be called *data access methods*, are summarized in Tables 2.1 and 2.2.

The read-only methods (Table 2.1) are intended for retrieving information. They can appear at the right-hand side of assignments and the methods with a scalar result type may be used in mathematical expressions or comparisons just like normal variables of type `double`. The method for retrieving the values of a vector state variable does not return a scalar result but a constant reference to a numeric vector (`const vector <double> &`). Note that it depends on the *usage* of the return value whether a copy of the retrieved data is generated or not. This is important in terms of computational efficiency if the vectors are large size. To avoid the creation of a copy, you have to use the returned value to initialize a const reference. In C++, this would look like

```
const vector <double> & = stateVect(name);
```

where `name` is the name of the respective vector state variable. If you assign the return value to a 'normal' variable, i. e. a non-constant numeric vector which is not a reference, using

```
vector <double> = stateVect(name);
```

a copy of the data will be created. Note that this distinction is also relevant when passing the vector to a function via the function's argument list. Since you often want to pass a constant reference instead of a copy of the values, the dummy argument should be declared accordingly.

The purpose of the write-only methods (Table 2.2) is to assign new values to an object's state or output variables (see example in Sec. 2.4). The write-only methods all return non-const references to scalars or vectors. These methods typically appear at the left-hand side of assignment statements. They may also be used as actual parameters in function calls, if the corresponding template parameter is a non-const reference of the appropriate type (i. e. the parameter represents an output of the function).

Mandatory actions

According to the purpose of the 'simulate' method (see above), there is a minimum set of statements that should be present in this method for every class (see example in Sec. 2.4). In particular, the method should contain

1. statements to update the values of all state variables using the method(s) from row 1 & 2 of Table 2.2. As discussed earlier, this usually means that ordinary differential equations are solved (see also Sec. 2.2.7).

2. statements to set the values of all output variables (see last row of Table 2.2).

2.2.7 The 'derivsScal' method

As described in Sec. 2.2.6, the purpose of the 'simulate' method is usually to integrate a single (or a set of) ordinary differential equation(s) over time. In some situations, the use of a simple first-order approximation (Euler's method) may be sufficient. Such methods, however, are neither accurate nor stable. If more accurate and stable solutions are needed, an ODE solver must be used which yields higher-order estimates and automatically adjusts the size of time (sub)steps.

```
// Abstract base class (generic object class)
class abstractObject {
  // The virtual 'simulate' method remains unimplemented here
  virtual void simulate(const unsigned int delta_t)= 0;
  // ... more data/function members ...
};

// A 'reservoir' class  (child of 'abstractObject')
class object_reservoir: public abstractObject {
  void simulate(const unsigned int delta_t) {
    // ... Reservoir-specific implementation of 'simulate' ...
  }
};

// A 'catchment' class  (child of 'abstractObject')
class object_catchment: public abstractObject {
  void simulate(const unsigned int delta_t) {
    // ... Catchment-specific implementation of 'simulate' ...
  }
};
```

Figure 2.4: *Specification of the 'simulate' methods in the abstract base class (parent class) and the application-specific child classes. Only relevant parts of the C++ code of the classes are shown.*

The **echse** comes with a built-in ODE solver based on the 5-th order Runge-Kutta method described in Press et al. (2002). This is a quite robust algorithm. However, its applicability is restricted to *non-stiff* systems of simultaneous ODE. This may be relevant if the number of state variables of an object is > 1.

As with all ODE solvers, one must pass a method to the solver which computes the derivatives of the state variables (with respect to time, here). The name of the corresponding class method is 'derivsScal'. Like 'simulate', it is a virtual method. As the method's name indicates, it computes the derivatives of the scalar state variables only (see Sec. 2.2.2). ODE solver support for vector state variables is currently not implemented.

The interface of the 'derivsScal' method is shown in Fig. 2.8. The meaning of the dummy arguments is as follows:

`t` This scalar *input* argument represents the time. It is only relevant when simulating non-autonomous systems, i. e. if the value(s) of the derivative(s) are time-depend. Note that this is only the case, if the forcings are variable within a time step. In many models, the forcings are treated as constant within a time step and the value of `t` is not used in computing the derivatives.

`u` *Input* vector holding the values of the state variables whose derivatives are to be computed.

`dtdt` *Output* vector, containing the derivatives corresponding to the state variables in `u`.

`delta_t` *Input* value, representing the length of the simulation time step. The intention of this argument is to allow for unit conversions. For example, external forcings (precipitation, radiation) may be given as sum values for a time step (mm/time step or J/m^2/time step, for example). When computing the derivatives, such values need to be converted into rates (m/s or W/m^2, for example) using the value of `delta_t`.

Table 2.1: *Data access methods, part I: Read-only methods. The dummy argument* `name`*, has to be substituted by the name of the particular variable, parameter, or function to be accessed. The names are defined by the model developer. Note that names must not be quoted since they do not represent strings but (automatically defined) index constant. For the dummy argument* `arg`*, a numeric expression representing the function's argument has to be supplied.*

Type of feature	Call	Result type
Scalar state variable	`stateScal(name)`	`double`
Vector of scalar state variables	`stateScal_all()`	`const vector <double> &`
Vector state variable	`stateVect(name)`	`const vector <double> &`
External input	`inputExt(name)`	`double`
Simulated input	`inputSim(name)`	`double`
Parameter function (object-specific)	`paramFun(name, arg)`	`double`
Scalar parameter (object-specific)	`paramNum(name)`	`double`
Parameter function (group-specific)	`sharedParamFun(name, arg)`	`double`
Scalar parameter (group-specific)	`sharedParamNum(name)`	`double`

Table 2.2: *Data access methods, part II: Write-only methods. See Table 2.1 for details on the methods'* `name` *argument.*

Type of feature	Call	Assigned type
Scalar state variable	`set_stateScal(name)`	`double &`
Vector of scalar state variables	`set_stateScal_all()`	`vector <double> &`
Vector state variable	`set_stateVect(name)`	`vector <double> &`
Output variable	`set_output(name)`	`double &`

In order to actually use this method, the code for computing the derivatives needs to provided in a separate file (see `#include` directive in the 'derivsScal' method in Fig. 2.8). This file must contain code which assigns a value to all elements of vector `dudt`. At the right hand side of these assignments, one can use the access functions listed in Table 2.1 with one *important exception*: One *cannot* call the function `stateScal` to access the value(s) of scalar state variable(s)! Instead, one must use the respective element of the input vector `u` (appropriate constants for accessing a particular element are provided in the generated class header). This is because of the fact that the ODE solver internally computes derivatives for various estimates of the state variables' values. These estimates are passed in vector `u`. Note that the body of the 'derivsScal' method can remain empty if there is no need for it (because the ODE(s) can be solved analytically, for example).

See Sec. 2.4.4 for an example showing an implementation of the 'derivsScal' method and the use of the built-in ODE solver in the 'simulate' method.

2.3 Automatic code generation

2.3.1 Role of generated code in the `echse` framework

As stated in Chap. 1, the **`echse`** is a generic modeling framework. As such, it consists of a generic, reusable model core complemented by problem-specific extensions. The generic model core provides basic infrastructure for *any* dynamic simulation model. The

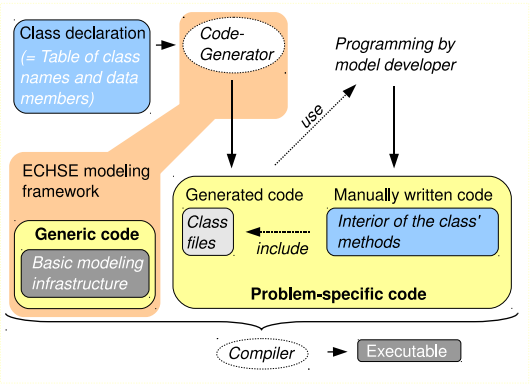

Figure 2.5: *Major components of the* **echse** *modeling framework.*

problem-specific extensions are required to build simulation models for a particular (type of) system.

In the case of the **echse** modeling framework, problem-specific extensions are equivalent to user-defined classes with the features described in Sec. 2.2. The relation between the generic model core and the problem-specific extensions is illustrated in Fig. 2.5.

To sucessfully build a simulation model with the **echse** framework, the problem-specific part of the source code (class definitions) must be perfectly compatible with the generic model core on the one hand. On the other hand, good practice of software development requires that the generic core and the problem specific extensions are well separated. In fact, a developer who implements the problem-specific classes should not need to understand or even know any details of the generic core.

In the **echse** modeling framework, this dilemma is solved by means of automatic source code generation (Fig. 2.5). In this concept, the model developer first *declares* a class by specifying the names and types of all data members (see Fig. 2.3 for possible member types). In a second step, a program automatically generates the class' basic source code from the provided declaration. This generated code is guaranteed to be compatible with the generic core. It provides entry points for additional source code which has to be manually written in a third step. This manually written code comprises the bodies of the 'simulate' and the 'derivsScal' methods (see Sections 2.2.6 and 2.2.7).

The three steps of

1. Declaration of data members
2. Code generation
3. Implementation of methods

are illustrated in Sec. 2.4 with the example of a linear reservoir class.

The advantages of the strategy of automatic code generation can be summarized as follows:

- The model developer does not need to manually write all the abstract code related to the classes and the corresponding object groups (recall Sec. 2.1.4). This reduces development times for new models.
- The model developer does not need to care for the compatibility of the application-specific code with the code forming the generic core of any **echse** model. This makes model development really a simple and save task.

2.3.2 The code generator

Installation

The code generator is currently implemented in the R programming language. It is contained in the R-package **codegen** which provides a single method whose name is `generate`. The package is provided as a tarball with name `codegen_x.y.tar.gz` where x.y is a version number. See Kneis (2012) for details on how to install the R software and add-on packages. The **codegen** package depends on no other packages.

Standard documentation

After the **codegen** package has been loaded, for example using the R command

```
library("codegen")
```

the documentation of the `generate` method can be displayed by typing the question mark followed by the method's name.

```
?generate
```

Examples

To run an illustrative example, one can use the following R command.

```
example("generate")
```

It generates sample input files for the `generate` method and then runs the method on these files. Another practical example, can be found in Sec. 2.4.

2.3.3 Inputs of the code generator

The code generator assumes that each class is declared in a *separate* file. Such a class declaration file must be a plain, TAB-separated text file with two columns 'type' and 'name' (see Fig. 2.7 for an example). Each record in this table declares a single data member of the class. The meaning of the two columns is as follows:

`type` (*string*) The type of the feature to be declared. Valid entries are listed in Table 2.3.

`name` (*string*) The name of the variable, parameter, or function to be declared. The name must be a valid C++ identifier.

In addition to these two mandatory columns, the table may have additional columns which are ignored during processing by the code generator. For the purpose of documentation, it is recommended to append at least one column with a short description of each feature and probably the physical units. Moreover, the file may contain comment lines starting with the # character (see example in Fig. 2.7). It may be convenient to prepare the table in a spreadsheet software first and to save the contents to a text file later (by copy & paste, for example).

Table 2.3: *Description of the keywords expected in the 'type' column of a class declaration table (see example in Fig. 2.7).*

Keyword	Type of feature
`stateScal`	Scalar state variable
`stateVect`	Vector state variable
`inputExt`	External input variable
`inputSim`	Simulated input variable
`paramNum`	Object-specific scalar parameter
`sharedParamNum`	Group-specific scalar parameter
`paramFun`	Object-specific parameter function
`sharedParamFun`	Group-specific parameter function
`output`	Output variable

2.3.4 Outputs of the code generator

The code generator produces several C++ header files (file extension '.h'). The individual files are only briefly described here. The files' contents is not shown.

Instantiation function definition file This file contains a function which, when called, creates a single instance of each object group based on class template. The function returns a handle to the object groups (in the form of a pointer vector).

Header bundle file This file contains C++ include statements, referencing all header files created by the code generator, except for the file itself. This provides a means to include a variable (application-specific) number of header files into the generic source files without the need for any modification there.

Class header file(s) For each class declared by the model developer, a header file is generated. It describes the abstract prototype of an object of the respective class. Note that the implementation of the class' methods 'simulate' or 'derivsScal' are *not* contained here. Instead, references to include files are generated and these include files must be manually filled with code by the model developer.

Index constants file(s) For each class declared by the model developer, a file with index constants is created. These index constants must be used when querying or manipulating an object's data via the methods described in Tables 2.1 and 2.2. The automatically defined constants allow for referencing a particular variable, parameter, or function *by name* (see the 'name' argument in Table 2.1 and the examples in Sec. 2.4.4). This makes data access convenient, efficient, and save at the same time.

2.4 Example: Implementing a new class

2.4.1 Linear reservoir

In this example, we implement a class describing a so-called 'single linear reservoir'. The linear reservoir is a widely used conceptual model in the field of hydrology. Applications range from describing the storage of water in catchments to flow routing in rivers. A single linear reservoir (Fig. 2.6) is fully described by two equations:

the continuity equation represenring the mass balance (Eqn. 2.1) and the linear outflow equation (Eqn. 2.2).

$$\frac{dv}{dt} = q_{in} - q_{ex} \tag{2.1}$$

$$q_{ex} = \frac{1}{k} \cdot v \tag{2.2}$$

The symbols in the above equations are defined below where L and T are generic units of length and time, respectively.

v	Storage volume (L^3)
q_{in}	Rate of inflow (L^3/T)
q_{ex}	Rate of outflow (L^3/T)
k	Retention constant (T)

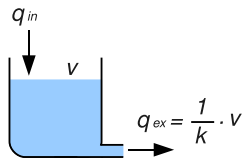

Figure 2.6: *Sketch of a single linear reservoir.*

The ordinary differential equation that results from combining Eqns. 2.1 and 2.2 can be solved analytically. With the simplest assumption of a constant inflow rate q_{in} over a time step of length Δt the integration yields Eqn. 2.3, where $v(t_0)$ is the initial storage at time t_0.

$$v(t_0+\Delta t) = (v(t_0) - q_{in} \cdot k) \cdot e^{(-\Delta t/k)} + q_{in} \cdot k \tag{2.3}$$

Using Eqn. 2.2 one can also transform Eqn. 2.3 into an expression for the outflow rate q_{ex} (Eqn. 2.4).

$$q_{ex}(t_0+\Delta t) = (q_{ex}(t_0) - q_{in}) \cdot e^{(-\Delta t/k)} + q_{in} \tag{2.4}$$

2.4.2 Step 1: Declaration of the class

To declare a new class, the model developer simply needs to specify the class' *data members* (recall Sec. 2.2.1). With respect to the example of the linear reservoir (Sec. 2.4.1), one would have to declare

- a single scalar state variable (storage volume v).
- a single scalar parameter (retention constant k). We assume here that this parameter is object-specific, i. e. each linear reservoir has an individual k.
- a single input variable (inflow rate q_{in}). We assume here that this is a simulated input rather than an external input (recall Sec. 2.2.3).
- a single output variable (outflow rate q_{ex})

As described in Sec. 2.3.3, this information has to be collected in a table-formatted text file for later processing by the code generator (Sec. 2.4.3). An appropriate input file for the code generator is shown in Fig. 2.7.

```
# Declaration of a
# linear reservoir class

type    name

stateScal       v
paramNum        k
inputSim        q_in
output  q_ex
```

Figure 2.7: *Input file for the code generator, containing the declaration of a linear reservoir class. See Table 2.3 for the entries allowed in the 'type' column.*

2.4.3 Step 2: Code generation

Once all data members of all classes have been declared in the required form (see Sec. 2.3.3 and Fig. 2.7), the table is further processed by the code generator. Assuming that the contents of Fig. 2.7 is saved in a file 'linRes.txt', an appropriate call to the code generator could be:

```
library("codegen")
generate(
  files=c(linReserv="linRes.txt"),
  outdir="generated_code"
  overwrite=TRUE
)
```

Note that the vector of class declaration files passed to the `files` argument must have as many elements as there are classes in the model. In our minimum example with only a single class, this vector is of lenght 1. Also note that this must be a *named* vector because the class' names are generated from the elements' names. Thus, the name for the linear reservoir class would be 'linReserv' in the above example.

The complete output from the above call to the `generate` method is not presented here. An an

overview of the created files was already given in Sec. 2.3.4 and a central part of the generated code is shown in Fig. 2.8.

2.4.4 Step 3: Implementing the class' methods

As mentioned in Sections 2.3.1 and 2.3.4, the implementation (i. e. the body code) of the 'simulate' and 'derivsScal' methods has to be provided by the model developer. The code generator only creates appropriate method interfaces and include statements. For the linear reservoir class introduced in Sec. 2.4.1, part of the generated code is shown in Fig. 2.8.

Two complete, alternative bodys of the 'simulate' and 'derivsScal' methods of the linear reservoir class are presented in Fig. 2.9 & 2.10. This is the code which would be imported by the `#include` directives in Fig. 2.8.

2.4.5 Step 4: Compilation

Once the code generator has run successfully and the methods for all classes are implemented, the application specific simulation software (i. e. the 'model engine') has to be build. This is achieved by compiling and linking all parts of the source code, namely

1. the static part of the code, providing the basic infrastructure for every model.
2. the application-specific code created by the code generator (see Sec. 2.4.3).
3. the body code of the 'simulate' and 'derivsScal' methods, manually written by the model developer.

The GNU C++ compiler is used for this purpose and the procedure has been successfully tested on several platforms. To assist the developer in the compilation process, platform-specific makefiles are available.

If invalid code is detected in the manually written parts of the code, the compilation will fail, of course and one has to go through the usual steps of debugging. One should keep in mind, however, that a successful compilation does not necessarily mean that the code is 'correct' in the sense that it produces the desired results. The correctness of the code can only be verified by analyzing the model's output.

```
1
2  // ----- Frames of method implementations follow -----
3
4  // Import auxiliary definitions to be used in 'simulate' and 'derivsScal'
5  #include "userCode_linReserv_aux.cpp"
6
7  // The 'simulate' method
8  namespace object_linReserv_simulate {
9    static const T_index_paramNum k= {0};
10   static const T_index_inputSim q_in= {0};
11   static const T_index_stateScal v= {0};
12   static const T_index_output q_ex= {0};
13 }
14 void object_linReserv::simulate(const unsigned int delta_t) {
15   using namespace object_linReserv_simulate;
16   try {
17     #include "userCode_linReserv_simulate.cpp"
18   } catch (except) {
19     stringstream errmsg;
20     errmsg << "Simulation failed for time step of length " << delta_t << ".";
21     except e(__PRETTY_FUNCTION__,errmsg,__FILE__,__LINE__);
22     throw(e);
23   }
24 }
25
26 // The 'derivsScal' method
27 namespace object_linReserv_derivsScal {
28   static const T_index_paramNum k= {0};
29   static const T_index_inputSim q_in= {0};
30   static const unsigned int INDEX_v= 0; // for use with vectors u & dudt
31   static const T_index_output q_ex= {0};
32 }
33 void object_linReserv::derivsScal(const double t,
34   const vector<double> &u, vector<double> &dudt, const unsigned int delta_t) {
35   using namespace object_linReserv_derivsScal;
36   try {
37     #include "userCode_linReserv_derivsScal.cpp"
38   } catch (except) {
39     stringstream errmsg, data;
40     for (unsigned int i=0; i<u.size(); i++) data << u[i] << " ";
41     errmsg << "Calculation of derivatives failed (Time: " << t << " Number" <<
42       " of deriv.s: " << dudt.size() << " Value(s) of state(s): " << data <<
43       " Length of time step: " << delta_t << ").";
44     except e(__PRETTY_FUNCTION__,errmsg,__FILE__,__LINE__);
45     throw(e);
46   }
47 }
48
49 #endif
```

Figure 2.8: *Part of the generated header file for the linear reservoir class showing the frame of the 'simulate' and 'derivsScal' methods. The manually written code is imported by the* `#include` *directives.*

File 'userCode_linReserv_aux.cpp'

```
// Empty file
```

File 'userCode_linReserv_simulate.cpp'

```
// Compute the new value of the state variable (v) at the end of the time
// step. We save the value in a temporary variable instead of updating v.
// This is because we still need the initial value in the next statement.
double v_new= (stateScal(v) - inputSim(q_in) * paramNum(k)) *
  exp(- delta_t / paramNum(k));

// Update the output variable (q_ex). We use the reservoir's mass balance
// to compute the time-step averaged outflow rate. Alternatively, we could
// simply return the instantaneous value at the end of the time step.
set_output(q_ex)= inputSim(q_in) - (v_new - stateScal(v)) / delta_t;

// We can now update v (since the initial value is no longer needed).
set_stateScal(v)= v_new;
```

File 'userCode_linReserv_derivsScal.cpp'

```
except e(__PRETTY_FUNCTION__,"Method not implemented.",__FILE__,__LINE__);
throw(e);
```

Figure 2.9: *Bodies of the 'simulate' and 'derivsScal' methods for the linear reservoir class if an analytical solution is adopted.*

File 'userCode_linReserv_aux.cpp'

```
// Empty file
```

File 'userCode_linReserv_simulate.cpp'

```
// Save initial volume for use mass balance
double v_ini= stateScal(v);

// Compute new value of v using the built-in ODE solver with arg.s 1--6
odesolve_nonstiff(
  stateScal_all(),              // 1: Initial value(s) of state variable(s)
  delta_t,                      // 2: Lenght of time step
  1.e-08,                       // 3: Accuracy (adjustable)
  1000,                         // 4: Max. number of sub-steps (adjustable)
  this,                         // 5: Pointer to active object
  set_stateScal_all());         // 6: New value(s) of state variable(s)

// Set q_ex using the mass balance
set_output(q_ex)= inputSim(q_in) - (stateScal(v) - v_ini) / delta_t;
```

File 'userCode_linReserv_derivsScal.cpp'

```
// Characteristic ODE of the lineare reservoir
dudt[INDEX_v]= inputSim(q_in) - u[INDEX_v] / paramNum(k);
```

Figure 2.10: *Bodies of the 'simulate' and 'derivsScal' methods for the linear reservoir class if a numerical solution is adopted. Note that the value of the volume state variable (v) in the 'derivsScal'method is accessed via* `u[INDEX_v]` *instead of* `stateScal(v)`*.*

2.5 Outline of computational steps

2.5.1 Overview

The essential computational steps carried out when executing a model are summarized in Fig. 2.11. Note that this outline applies to *any* model built with the **echse** simulation environment.

Main function

- Retrieval of command line arguments (Sec. 3.1)
- Reading of the configuration file (Sec. 3.4)
- Instantiation of objects & object groups (see Fig. 2.2)
- Initialization of the objects' data.

Loop over time steps

- Updating of external inputs

Loop over objects

- Call of the 'simulate' method for current object

- Writing of data to output files

- Closing of output files
- Clean-up
- Output of traceback info in case of exceptions
- Setting of return code and termination

Figure 2.11: *Essential computational steps of a model run. The framed boxes represent loops, thus the tasks inside these boxes are executed repeatedly (see Sec. 2.5.2).*

Most of the computational steps listed in Fig. 2.11 appear in any dynamic systems simulation software. Only those aspects which are specific to **echse** models are discussed in the subsequent sections.

2.5.2 Time and object loop

In Fig. 2.11, the innermost framed box represents the so-called *object loop*. The purpose of this loop to iterate through all objects and trigger the simulation for a single time step by calling the objects' 'simulate' methods (see Sec. 2.2.6). In a spatially distributed model, the term 'spatial loop' is often an appropriate synonym for 'object loop'[1].

Note that the *time loop* is wrapped around the object loop (Fig. 2.11). This design can be found in virtually all spatially distributed models that solve *partial* differential equations (PDE) such as groundwater flow models or hydrodynamic models. Note, however, that a few models exist where the two loops are in reverse order, for example in some hydrological catchment models.

The consequence of having the object loop *inside* the time loop is that, for a particular time step, the 'simulate' method is executed for *all* objects, before the computation proceeds with the subsequent time step. This allows for the exchange of information between objects in every single time step. This is a precondition for properly handling *feedbacks* between objects, i. e. two-way interactions (see Fig. 2.13, Sec. 2.6.2).

The only drawback of this approach is the requirement of keeping instances of *all* objects in memory at the same time. Thanks to the large memory capacity of modern computers, this is hardly an issue. If the number of objects should actually be too large to fit into memory, a cheap solution would be to split the model (into spatial sub-domains, for example) in a way that no feedback between the sub-models does occur.

2.5.3 Exception handling

In a complex and flexible software it is not unlikely that an unrecoverable error occurs during computations. The potential causes are manifold, ranging from missing or erroneous input data to mathematical calculations yielding invalid results (NaN, Inf, ect.).

The models built with the **echse** simulation environment use C++'s exceptions mechanism to handle situations like that. Whenever an exception occurs, the normal execution of the program is suspended and priority is given to exception handling. In the case of **echse** models, this means that the currently active unit (i. e. a class method or function) tries to collect as many information as possible about the circumstances of the exception and then gives control back to its calling unit. The calling unit behaves just like the unit where the exception originally occurred. In this way, the error signal is passed through the hierarchy of routines and fi-

[1] In the current version of the software, the object loop is split into two nested loops to enable parallel processing. This is not essential for the understanding for the general understanding, however.

nally causes an exception at the highest level, the 'main' function. Here (and only here), traceback information is generated and the program is forced to terminate (see final step in Fig. 2.11).

The traceback always contains (for every unit) information about

1. the name of the unit, where the exception occurred.
2. a description of the circumstances and possibly the cause of the exception.
3. the name of the source file containing the unit that failed.
4. the line of this file where the exception was thrown.

Based on the traceback information, the location and cause of the error can usually be identified with little effort.

If the model terminated due to an exception, the program issues a non-zero return code. If no exception occurred, a the code of zero is returned as this is widely used convention. The return code should always be checked if the model is embedded in another software such as a scripts or batch files.

2.6 Interactions between objects

2.6.1 Overview and accessible data

Interactions between objects are a typical in natural and technical systems. In this context, an interaction is defined as an exchange of either matter (mass), energy, or information. Although it is possible to simulate just a single object or a group of non-interacting objects, interactions have to be considered in the vast majority of real-world models.

As already outlined in Sections 2.2.3 and 2.2.5, an interaction between two objects is bound to the declaration of

1. a *simulated input variable* in the class corresponding to the object that *uses* data provided by another object.
2. an *output variable* in the class corresponding to the object that *provides* the data to be used by (an)other object(s).

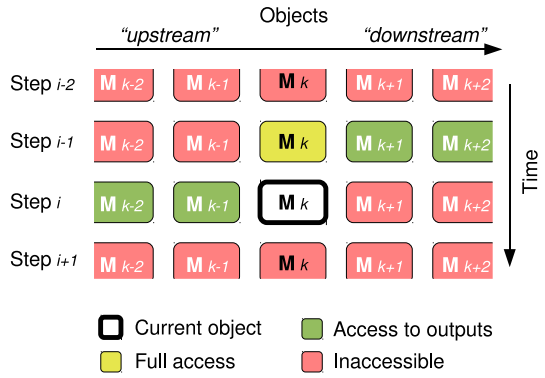

Figure 2.12: *Accessible data from a single object's perspective.*

In a dynamic simulation model with sequential processing of the objects (see Sec. 2.5.2 and Fig. 2.11), further limitations with respect to the acessibility of data do exist. This is illustrated in Fig. 2.12 from the perspective of a single object with respect to a single time step (bold-framed object M_k). Assuming that appropriate simulated input and output variables have been declared in the respective classes (see above), for the simulation of time step i, the object M_k has access to

1. data on the object itself, representing the state at the end of the previous time step with index $i-1$.
2. the output variables of the already processed *upstream* objects. These values are representative for the end of the *current* time step (i).
3. the output variables of the *downstream* objects still waiting for being simulated. These values are representative for the end of the *previous* time step ($i-1$).

2.6.2 Types of interactions

Looking at two interacting objects, one generally has to distinguish between *feed-forward* interactions (also called *one-way* interactions) and *feedbacks*, also known as *two-way* interactions (Fig. 2.13). The difference between the two is illustrated also in Fig. 2.14 on a very simple example. Typical real-world examples of feedbacks in the field of hydrology include

- interactions between river and floodplain. River stage and groundwater level are coupled via infiltration and leakage, respectively.

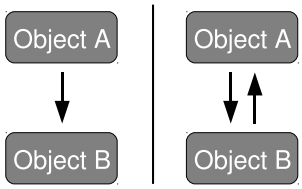

Figure 2.13: *Basic types of object interaction. The arrows indicate exchange of matter, energy, or information. Left: Feed-forward type. Right: Feedback type.*

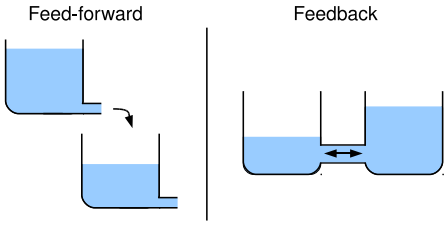

Figure 2.14: *Types of interactions between two buckets filled with a liquid.*

- diffusion problems at the interface of the pelagic and benthic zone. The rate of diffusive transport depends on both, the concentration in the water body and the sediment's pore water.
- the operational control of a reservoir's outflow based on stream flow data observed at a gage downstream of the reservoir.

The feedback (Figs. 2.13 & 2.14, right) represents the more general type of interaction and, actually, the feed-forward interaction (Figs. 2.13 and 2.14, left) may be regarded just as a special type of (missing) feedback. It makes sense, however, to strictly distinguish between the two types of interaction in the context of dynamic simulation, i. e. when modeling the interaction of objects over a sequence of discrete time steps. This is due to the following:

Feed-forward type As long as the exchange of data between two interacting objects 'A' and 'B' is effectively *one-way*, the two objects can be simulated *sequentially*, i. e. one after another. The so-called *source object* ('A' in Fig. 2.13, left) represents the 'data provider' and must be simulated first. The output of 'A' is then used as an input for the *target object* ('B'), which is simulated later.

Feedback type If there is a *two-way* exchange of data between two objects 'A' and 'B', these objects must be simulated *simultaneously*, i. e. at the same time. To put it in other words: The ordinary differential equations, describing the evolution of the state variables in object 'A' form a coupled system with the equations of object 'B'. To get a proper solution, the coupled differential equations must be integrated simultaneously using an ODE solver (see e. g. Press et al., 2002). This, however, conflicts with the facts that (1) objects are geberally treated as well-separated entities and (2) the array of objects is processed sequentially using a fixed order (see innermost loop in Fig. 2.11). Nevertheless, **echse**-based models are capable of handling feedback interactions using the techniques outlined in Sec. 2.6.3.

2.6.3 Handling of feedbacks

Option 1: Compound classes

A straightforward approach to cope with the problem of feedback interactions (Sec. 2.6.2) is to avoid inter-object feedbacks. Taking the objects 'A' and 'B' from Fig. 2.13 (right) as an example, this would mean that the class(es), of which the objects 'A' and 'B' are instances, are joined to form a new (compound) class. The feedback interaction between the former objects 'A' and 'B' is then *internally* present in an object of the compound class. The coupled differential equations related to all state variables (which were originally distributed over object 'A' and 'B') can then be solved simultaneously using a standard ODE solver within the simulate method (Sec. 2.2.6) of the compound class.

The drawback of such an approach is that the compound class may quickly become rather complex. In extreme cases of many feedback interactions, one might end up with a model consisting of only a single object being an instance of a single class which basically integrates 'everything'. In such a case one should think about other strategies (see below) or use another, more appropriate modeling software.

Option 2: Step-wise feedback

The simplest and probably the most common solution for the feedback problem is to treat the differential equations in the two interacting objects 'A' and 'B' as *temporarily independent*. In practice, this works as follows:

Simulation step The objects 'A' and 'B' are simulated independently, as if there was no interaction at all. This means that the state variables of 'A' and 'B' are updated by separately integrating the respective differential equations.

Feedback step After *every* time step, the two objects exchange information about their new states. The information about 'A' is then used in the subsequent simulation step for 'B' and vice versa.

To make the described approach of *step-wise feedback* successfully work in practice, two conditions must be met.

Firstly, the interacting objects 'A' and 'B' must exchange data with a high frequency. This is achieved by running the simulation in small time steps. The longer the time step, the higher the potential numerical error will be.

Secondly, it has to be ensured that the information about object 'B' used by 'A' refers to the *same point in time* as the information about 'A' used by 'B'. In a normal sequential simulation, where either 'A' or 'B' is processed first, this is not the case. However, with the help of a so-called *observer object* it is possible to supply 'A' and 'B' with data of equal up-to-dateness, in spite of sequential processing. This strategy is illustrated by Fig. 2.15. In the shown example, a feedback interactions exists between the objects M_k and M_{k+1}. The role of the auxiliary observer object M_{k-1} is to collect data on M_k and M_{k+1} which is representative for a particular point in time, namely the (end of) time step $i-1$. The collected information is then supplied to M_k and M_{k+1}, respectively, to be used in the simulation over the current time step i.

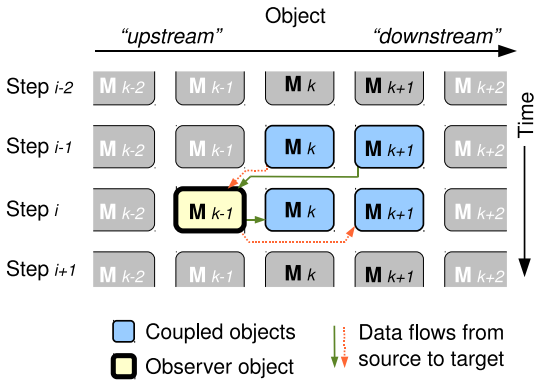

Figure 2.15: *Use of an artificial* observer object *to provide two sequentially processed, feedback-coupled objects with information of equal up-to-dateness.*

Step-wise feedback: Time step issues

As mentioned above, the selection of a sufficiently short time steps is necessary to keep the error associated with the step-wise handling of feedbacks within acceptable limits. A disadvantage of the current version of the **echse** is that the time step is a fixed parameter. Consequently, if a short time step is selected with the intention of increasing the accuracy of feedback solutions, the computation will slow down even for those objects which are not subject to feedback interactions. Thus, a single feedback interaction may impact negatively on the performance of the entire model in terms of computation time. A possible solution to this problem lies in releasing the constraints of the fixed 'global' time step. In particular, it would make sense allow a variable number of sub-steps to be specified for each object. The only restriction would be that the number of sub-steps must be identical for two objects having a feedback relation. If a feedback interaction exists between two objects 'A' and 'B' and another one exists between two objects 'C' and 'D', the number sub-steps applied to the first group ('A', 'B') and the second group ('C', 'D') may still be different, however. It is planned to implement the sub-step approach in an upcoming version of the **echse**.

Step-wise feedback: Accuracy

To really understand the limits of the strategy of a step-wise simulation of feedbacks, a closer look on the solution strategy is required. Let's take the example of Fig. 2.15, where a feedback interaction between the objects M_k and M_{k+1} is simulated under the control of an observer object M_{k-1}. In a first step, the observer object M_{k-1} collects information from both object M_k and M_{k+1} which is representative for time step $i-1$. If the objects M_k and M_{k+1} were the two connected buckets shown in the right column of Fig. 2.14, the observer M_{k-1} would collect information on the water levels in the two buckets and compute the resulting flow rate. Then, the two objects M_k and M_{k+1} would retrieve the computed flow rate from the observer and both objects would use this information to calculate their individual water levels at the end of time step i.

It is important to realize that the accuracy of the resulting solution is limited by the fact that the information on the flow rate between the two buckets is a con-

tant. This rate effectively represents the sitution at the end of time step $i-1$ or (in other words) the situation at the *very beginning* of time step i. This *constant* information is then used in the simulation of the *entire* time step i, neglecting that the water levels change, hereby affecting the flow rate.

Solutions with these characteristics are also called *Euler solutions*. It is well known that the accuracy of such first-order solutions is quite limited. Therefore, with the current version of the **echse** a reasonable simulation of feedbacks can only be expected if

- non-linearities are weak.
- the chosen simulation time step is sufficiently short.

A straighforward method to examine the accuracy of the simulation of feedback interactions is to simply run the model with different time steps and to compare the results for the affected objects. It is possible that support for higher-order solutions (Heun, Runge-Kutta, etc.) and/or methods of automatic time step control will be implemented in a future version of the **echse**.

2.6.4 Conservation of mass or energy

As explained earlier, information is not continuously exchanged between interacting objects but only after discrete time steps. This is true for both feed-forward and feedback interactions. In the usual case of non-linear dynamics, it is quite important to understand the consequences with respect to the loss of accuracy and regarding the conservation of mass or energy in particular.

Recall the example of the two separate buckets shown at the left side of Fig. 2.14. In this feed-forward interaction, the bucket at the bottom receives inflow from the other bucket. If we assume that the upper bucket has the characteristics of a linear reservoir (see Sec. 2.4.1) we known from Eqn. 2.4 (page 23) that the outflow is a *non-linear* (exponential) function of time (Fig. 2.16).

Recalling Sec. 2.2.5, we know that all information about the upstream bucket's outflow needs to be passed to the downstream bucket through output variables. Often, the crux for the model developer is to define the output variables in way that

1. the information on the dynamics is retained.
2. mass (or energy) is conserved.

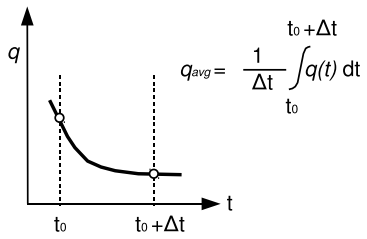

Figure 2.16: *Example outflow q from a linear reservoir within a discrete modeling time step of length Δt.*

Unfortunately, these two goals cannot be achieved at the same time and one needs to set a priority. Based on the example from Fig. 2.16, some possible options are discussed in the following.

Boundary values

A simple solution would be to pass the values at time step boundaries through two output variables:

- The flow rate at the beginning of the time step $q(t_0)$.
- The flow rate at the end of the time step $q(t_0+\Delta t)$.

In this way, at least the information on the change of the flow rate within the time step is retained. However, the information on the actual non-linearity is lost. Consequently, the information on the true cumulated outflow, i. e. the exchanged volume, is lost too.

'Improper' average

Another option would be to pass only an average outflow rate computed as the arithmetic mean of $q(t_0)$ and $q(t_0+\Delta t)$. In most cases, this is not recommended, because neither of the two above-mentioned goals is met. Firsty, the information on the dynamics is completely lost. Secondly, use of the average value as defined above results in a mass balance error if the true dynamics is *non-linear*. This is true in the example (Fig. 2.16) as well as in most real-world situations.

Intermediate values

With this strategy, information on the outflow at intermediate times is passed. For example, one could pass the 3 values $q(t_0)$, $q(t_0+1/2\Delta t)$, $q(t_0+\Delta t)$. Then, the downstream bucket could try to re-construct the non-linear dynamics, for examply by fitting an interpolation function $g(t)$ to the three values.

Interpolation parameters

This is just a special case of the afore-mentioned option. In this case, one does not pass the outflow rates at intermediate times. Instead, the parameters of the interpolation function $g(t)$ are passed. Depending on the specific case and the type of the interpolation function, this may reduce the number of necessary output variables.

'Proper' time-step average

If the priority is on conservation of mass, one needs to pass a *proper* average outflow rate $\overline{q_{out}}$ from the upstream to the downstream bucket. The two above-mentioned approaches allow for the calculation of an approximate average outflow rate as

$$\overline{q_{out}} \approx \frac{1}{\Delta t} \int_{t_0}^{t_0+\Delta t} g(t)$$

with $g(t)$ being the interpolation function. To make this work in practice, the interpolation function $g(t)$ must be integratable and allow for a reasonable fit of the dynamics.

Another strategy which is often more straightforward is based on a discrete mass balance for the upstream bucket. Denoting the volume of the upstream bucket as v and assuming a *constant* inflow rate q_{in}, the time-step averaged outflow rate $\overline{q_{out}}$ is

$$\overline{q_{out}} = q_{in} - \frac{v(t_0 + \Delta t) - v(t_0)}{\Delta t}$$

The approach remains applicable even if more input or loss terms appear in the bucket's mass balance. It is not even necessary that these terms are constant over the time step of length Δt, but then, their cumulated values (integrals over Δt) must be known. In practice, these integral values can be obtained by introducing auxiliary state variables representing the cumulated inputs and/or losses. These auxiliary state variables can be initialized with zero at the begin of each time step.

Combined approaches

In some cases, it may be favourable to combine some of the approaches discussed above. For example, one could pass the values at the time-step boundaries as well as the proper time-step average value. In this way, the loss of information at the interface between the interacting objects is minimized.

Chapter 3

Input of echse-based models

3.1 Mandatory command line arguments

Some basic settings are passed to the model via the command line. Each of these settings is identified by a unique keyword. The keyword must be followed by the equal sign '=', followed by a corresponding value. There must be no spaces before or after the equal sign. The expected keyword-value pairs are summarized in Table 3.1. They may appear at the command line in any order. In addition to these mandatory arguments, further configuration data may be passed via the command line (see Sec. 3.4).

A typical call of the model in a shell script using only the mandatory arguments might look as follows:

```
model file_control=config.txt
file_log=log.txt file_err=err.html
format_err=html silent=false
```

3.2 General notes on file formats

All input files comply with a simple quasi-standard and can easily be created automatically (by scripts or any spreadsheet software). For small projects, the files can even be created manually using just a text editor. The general rules applying to all input files are as follows:

Tabular format All input files actually represent tables. The number of columns varies from file to file and the number of rows (records) depends generally on the particular application. The number of columns must be consistent for all records. The tables are in plain text format.

Column separator The table columns are separated by a reserved character which has to be specified in the configuration file (see Sec. 3.4). Recommended choices are the TAB-character (ASCII code 9), and/or the blank character, or the semicolon (quasi-standards). As an exception to the above, the column separator used in the configuration file is the equal sign ('=') and it cannot be altered by the user.

Table header The first non-blank, non-comment line of a file is interpreted as the table header contaning column names. Tables without header are not supported.

Character set An input file should contain nothing but ASCII characters. Other characters may or may not be interpreted correctly (to be tested).

Comment lines Comment character(s) have to be be specified in the configuration file (see Sec. 3.4). A line starting with one of the selected comment characters is ignored when reading the table.

Blank lines Blank lines are ignored when reading the table, just like comment lines.

Platform independency Line endings may be system specific. On Linux/unix, the standard is `\n`. On Windows, it is `\r\n`. Input files prepared for Linux usually also work on Windows and vice-versa (to be tested). The line endings in the output files depend on the platform on which the model is running (see standards above).

Order of columns With one exception, the columns of a table can be in any order (since they are identified by the columns' names). The only exception are time series data files (see Sec. 3.8.2), where the

Table 3.1: *Mandatory command line arguments of a model.*

Keyword	Data type	Description
`file_control`	string	Name/path of the configuration file. This file contains all configuration data (except for those data specified as additional command line arguments). The configuration data are discussed in detail in Sec. 3.4.
`file_log`	string	Name/path of the log file created during a model run. The log file contains a compact documentation of all major steps of processing. Its contents is usually inspected in the case of abnormal program termination.
`file_err`	string	Name/path of a file where traceback information should be written to. This file will only be generated if the model terminates after occurrence of an exception. In the vast majority of cases, the information found in this file will help to quickly identify what caused the exception.
`format_err`	string	This option controls the format used in the file specified as `file_err`. The supported codes currently include 'xml', 'html', and 'txt'. For visual inspection, the html-format is the preferred choice. The other formats are more useful for automatic extraction of information (if the model is running in a more complex software environment, for example). If an unsupported format code is supplied, the 'txt' format will be used.
`silent`	logical	The model sends basic messages about the current state of processing to standard output (usually the screen) if `silent=false`. This kind of output may be suppressed by setting `silent=true`.

time information must be in the first column. Subsequent columns (containing data values for different locations or variables) may be in any order.

Column types The supported data types of a column are: string, integer, numeric, logical, and datetime. For numerical values the usual f-format (0.1) or the scientific e-format (1.e-01) may be used. Valid logical values are TRUE and FALSE (not case-sensitive). Datetime values must be strings in ISO 8601 format, i.e. in format `YYYY-MM-DD hh:mm:ss`. Date and time must be separated by a single character (recommended is a blank).

File names Some tables contain references to other files. A file name can be specified using either the absolute or relative path.

Empty tables There are no optional input files, which means that all files must exist and must be readable, even if they are not required for a particular application. Even if there is no information to be filled in, you cannot just supply an empty file. Instead you must supply a proper table with the usual header line and (at least) one record of values. The values may (and should be) dummy values that are easily identified as dummies. This procedure may seem overly complicated at first but, in fact, it avoids many other problems (tests in the source code, documentation of optional files, etc.).

3.3 Units of variables and constants

There is no general convention, i. e. arbitrary units may be used for all constants and variables. The only important facts are:

- The units of all variables and constants used in any equations must be consistent. There are no (and cannot be) any built-in checks in the generic part of the source code.

- The length of a modeling time step passed to the classes' simulate methods as argument `delta_t` is given in units of seconds.

3.4 Configuration data

3.4.1 Alternative ways of passing config data

The configuration data comprise all information about a specific model run. This includes, for example, settings like the start and end time of the simulation or the names of the various files which have to be read before or during a model run. The actual data contained in the referenced files (such as time series of external forcings, parameter values, etc.), by definition, do *not* belong to the configuration data.

There are two ways of passing configuration data to the model:

- via a configuration file.

- via the command line, in addition to the mandatory arguments introduced in Sec. 3.1.

3.4.2 Syntax conventions

A single configuration data item generally consists of two parts: A keyword and a corresponding value. The general syntax is shown in the following example:

```
fruit=apple
number=22
apple_data=/home/fred/apples.txt
```

Thus, the keyword must be followed by the equal sign ('='), followed by the value. The value may be a string, a number, a logical value, or a string encoding a datetime value (see Sec. 3.2). To avoid ambiguities, one cannot pass the same configuration data item (identified by its keyword) via the command line *and* via the configuration file. Multiple definitions of the same keyword are generally considered as errors. The configuration data items may appear in any order. This applies to both the configuration file and the command line.

It is important to note that blank(s) right before the '=' character as in `key =value` are *not* allowed (since the blank would be treated as part of the keyword). Some care is necessary if blanks or special characters appear *after* the '=' character . Here are the rules:

If the configuration data item is defined in the configuration file, all blanks after the '=' are treated as part of the value string. You don't need to use quotes here. Typical examples are shown below:

```
item1=2012-01-19 00:00:00
item2=c:\my files\data.txt
```

If a configuration data item containing blanks should be passed via the command line, quotes must be used as in the subsequent example, where `*` stands for the mandatory arguments (see Sec. 3.1). Blank(s) must not appear between the '=' character and the opening quotes.

```
model * date="2012-01-19 00:00:00"
```

If a configuration data item contains special characters, it must not be specified at the command line but needs to be defined in the configuration file. This is due to the fact that those characters may be dropped by the C++ command line interpreter. A prominent example is the TAB character (ASCII code 9).

3.4.3 Indirect file references

The configuration data usually contain both *direct* and *indirect file references*. Since the latter are sometimes confusing to users, the difference between the two types of file references is briefly discussed.

A *direct* reference is present if a configuration data item points to a *data file* containing anything but file names (typically numbers and possibly some alphanumeric IDs). What happens internally is this:

- The model engine reads the configration item and finds the reference to a data file 'A'.
- At the approriate stage of processing, the model engine reads the data from 'A'.

An *indirect* reference is present if a configuration data item points to a file which contains references to further files. What happens internally is this:

- The model engine reads the configration item and finds the reference to a file 'A'.
- The model reads file 'A' and finds references to the files 'B' and 'C'.
- At the approriate stage of processing, the model engine reads the data from 'B' and 'C'.

In theory, it would be possible to use a cascade of such indirect references. The current version of the **echse**, however, uses indirect references of the first level only. See Sections 3.8.3, 3.7.3 & 3.7.4 for examples.

3.4.4 Overview of configuration data items

The various configuration data items expected by an **echse**-based model are described in detail Tables 3.2 – 3.9.

Table 3.2: *Keywords of the configuration file controlling the computational behavior.*

Keyword	Data type	Description
`trap_fpe`	logical	If TRUE (recommended), an exception will be thrown if invalid floating point numbers occur in an object's state or output variables. If FALSE, the computation continues (if possible) and `NaN` or `Inf` values may appear in output files.
`number_of_threads`	integer	The desired number of threads to be run in parallel. Values greater than one will only have an effect on multi-core machines. If the requested number exceeds the maximum possible number of threads on the particular machine, the possible maximum is used. See also keyword `singlethread_if_less_than` and consult Sec. 4.2.1 before setting `number_of_threads` to a value greater than 1.
`singlethread_if_less_than`	integer	This key lets you define a threshold value for parallel processing. If the number of objects of a particular level is $<$ this threshold, these objects will be simulated by a single thread (i. e. in serial mode) even if parallel processing was requested by the keyword `number_of_threads`. If the number of objects of a particular level is geq this threshold, the requested (or possible) number of threads will be used.

Table 3.3: *Keywords of the configuration file dealing with input file formats.*

Keyword	Data type	Description
`input_columnSeparator`	character(s)	Column separator(s) used in input files. One of more character(s) may be specified (typed in). Recommended are TAB and space. Using TAB is especially useful when input files are created from spreadsheet data by copy-and-paste. When typing a TAB, take care that it is not auto-converted to spaces by the editor (depends on the editor's settings). You cannot use characters that are part of legal object or object group names (see Sec. 3.5).
`input_lineComment`	character(s)	Initial character of comment lines in input files. Note that only whole-line comments are supported. A reasonable choice is #, for example. You cannot use characters that are part of legal object or object group names (see Sec. 3.5).
`output_columnSeparator`	character	Column separator used in output files. Must be a single character. Recommended is TAB as it allows the contents of output files to be pasted into spreadsheets.
`output_lineComment`	character	Initial character of comment lines in output files. A reasonable choice is # for compatibility with R.

Table 3.4: *Keywords of the configuration file specifying basic input files.*

Keyword	Data type	Description
`table_objectDeclaration`	string	Name/path of the file declaring the simulated objects (see Sec. 3.5).
`table_inputOutputRelations`	string	Name/path of the file containing information on the objects' input-output relation (see Sec. 3.6).

Table 3.5: *Keywords of the configuration file related to the simulation time & resolution.*

Keyword	Data type	Description
`simStart`	datetime	Start of the simulation time window (= start of the first time interval). Example: `2005-01-01 00:00:00`. Note that a time zone without daylight saving time (DST) is assumed, such as UTC.
`simEnd`	datetime	End of the simulation time window (= end of the last time interval). See `simStart` for restrictions.
`delta_t`	integer	Length of a simulation time step in seconds. Must be ≥ 1. The time step determines the frequency of data exchange between linked objects. It also controls the resolution of model outputs. Note that the temporal resolution of external forcings (time series of boundary conditions) must be equal or greater than the value of `delta_t` (see Sec. 3.8.2 for details). Numerical methods (such as ODE solvers) used in the simulate() method(s) are not affected by the choice of `delta_t` and may internally use smaller time steps.

Table 3.6: *Keywords of the configuration file controling the model's output files.*

Keyword	Data type	Description
`table_selectedOutput`	string	Name/path of the table listing the objects and variables for which time series output is to be generated (see Sec. 3.10.1 for details).
`table_debugOutput`	string	Name/path of the table listing the objects for which time debug output is requested (see Sec. 3.10.2 for details).
`table_stateOutput`	string	Name/path of the table with times, at which the entire model's state should be saved. (see Sec. 3.10.3 for details).
`outputDirectory`	string	Name of the directory where all model outputs requested through `table_selectedOutput`, `table_debugOutput`, `table_stateOutput` should be written to. Must be an existing directory with appropriate permission. The names of the output files are generated automatically. Note that log and error messages are not necessarily saved in this directory. The location of these two files in controlled by the keywords `file_log` and `file_err` (see Table 3.1).
`outputFormat`	string	Selection of the desired format used to print time series of selected variables for selected objects (as controlled through the input table specified after keyword `table_selectedOutput`). Currently, the two valid choices are 'tab' (for TAB-separated table format; file extension '.txt') and 'json' for output in Java Script Object Notation (file extension '.json'). The latter is a slim, self-documenting data interchange format (see, e. g. `http://www.json.org`) supported by many programming languages and softwares (Example: R-package 'rjson'). Note that, using appropriate settings for the column separator, the '.json'-files can still be imported in spreadsheet software.
`saveFinalState`	logical	Should the final model state be saved even though the time corresponding to the end of the simulation period is not listed in the file specified as `table_stateOutput`? This may be particularly convenient in the context of an automatized forecasting environment.

Table 3.7: *Keywords of the configuration file related to initial value files.*

Keyword	Data type	Description
`table_initialValues_scal`	string	Name/path of the table with initial values for the scalar state variables of all objects (see Sec. 3.9.1).
`table_initialValues_vect`	string	Name/path of the table with initial values for the vector state variables of all objects (see Sec. 3.9.2).

Table 3.8: *Keywords of the configuration file related to external forcings.*

Keyword	Data type	Description
`table_externalInput_datafiles`	string	Name/path of the table listing properties and source files for the external input variables (see Sec. 3.8.3).
`table_externalInput_locations`	string	Name/path of the table listing assigning external input locations and weights to the objects' input variables (see Sec. 3.8.4).
`externalInput_bufferSize`	integer	Number of time series records to be kept in memory. Must be ≥ 1. If `externalInput_bufferSize=1`, only a single time series record is read at a time. If the value is choosen too large, memory allocation might fail for large models (many objects and many object variables). Choosing a larger value of `externalInput_bufferSize` may optimze the reading of data from disk. Whether there is an actual gain in performance depends on many factors (including input files and hard ware). Thus, it is recommended that some tests are carried out with an increased buffer size starting from `externalInput_bufferSize=1`.

Table 3.9: *Keywords of the configuration file related to the object groups' parameter tables.*

Keyword	Data type	Description
*name*_`numParamsIndividual`	string	Name/path of the table holding object-specific *scalar* parameters for all objects of an object group, i. e. user-defined class (see Sec. 3.7.1). The name of the object group has to be supplied in the prefix *name*. For example, for a class 'apple', the keyword would be `apple_numParamsIndividual`. The configuration file must contain as many instances of this keyword as there are object groups (i. e. user defined classes).
*name*_`funParamsIndividual`	string	Like 1^{st} row of the table but this key is related to the object-specific parameter *functions* rather than scalar parameters (see Sec. 3.7.3).
*name*_`numParamsShared`	string	Like 1^{st} row of the table but this key is related to the group-specific (shared) *scalar* parameters rather than to object-specific parameters (see Sec. 3.7.2).
*name*_`funParamsShared`	string	Like 2^{nd} row of the table but this key is related to the group-specific (shared) parameter *functions* rather than to scalar parameters (see Sec. 3.7.4).

3.5 Object declaration table

The object declaration table (example given in Fig. 3.1) consists of two columns of type string:

`object` (*string*) Contains the names (ID strings) of all objects to be simulated. Object names must be unique. Valid names consist of the characters `a-z` and `A-Z`, digits `0-9`, the minus (`-`), the underscore (_), as well as opening and closing parenthesis.

`objectGroup` (*string*) Contains for each object the name (ID string) of the corresponding object group (i. e. the name of the object's class). Valid names must also be valid C++ identifiers, hence the character set is restricted to `a-z`, `A-Z`, `0-9`, and the underscore (_). The first character cannot be a digit or underscore but must be a letter.

3.6 Object linkage table

The object linkage table describes the input-output relations of the simulated models. For each object, the table must contain n records, where n is the number simulated input variables of the corresponding object group (i. e. object class). The table consists of four columns of type string and one logical column (see Fig. 3.2 for an example):

`targetObject` (*string*) Names of objects that receive input from other simulated objects.

`targetVariable` (*string*) Name of the target object's input variable defined in the current row.

`sourceObject` (*string*) Name of the object that supplies the input to the target object and variable defined in the current row.

`sourceVariable` (*string*) Name of the source object's output variable that supplied the input to the target model's input variable.

`forwardType` (*logical*) Defines the type of relation. If TRUE, the relation is of the forward type, which means that the source object is simulated before the target object (in every time step). Thus, the input information used by the target model in the simulation of a time interval t_0...t_1 represents the output information of the source model queried at time t_1. In other words: The state of the source model is updated before the state of the target model. In contrast to that, a backward relation is assumed, if the entry in this column is FALSE. Then, the target model is simulated before the source model and, consequently uses 'outdated' information. In general, if the flow of information between two objects 'A' and 'B' of the feed-forward type (see Sec. 2.6.2), the relation is always of the forward type (entry TRUE required). Backward relations (entry FALSE) make sense only in the context of feedback interactions (see Sec. 2.6.3). In the example shown in Fig. 2.15, the data flows from object M_k to M_{k-1} and from object M_{k+1} to M_{k-1} represent backward relations. The reverse data flows ($M_{k-1} \rightarrow M_k$ and $M_{k-1} \rightarrow M_{k+1}$) represent forward relations. Note that, if two models 'A' and 'B' exchange data for more than one variable, the type of the relation must be the same for all those variables. For example, object 'B' cannot use output 'x' of object 'A' in a forward relation and, at the same time, use another output 'y' of object 'A' in a backward relation (because non of the two objects could be simulated before the other one).

```
# Simple model with four objects and two classes (object groups)

object      objectGroup

Rhine       basin
Constance   gage
Danube      basin
Vienna      gage
```

Figure 3.1: *Example of a simple object declaration table.*

```
# Simple river system:
#
# spring1 --> reach1
#               |
#             junction --> reach3
#               |
# spring2 --> reach2

targetObject  targetVariable  sourceObject  soureVariable  forwardType

reach1        inflow          spring1       outflow        true
reach2        inflow          spring2       outflow        true
junction      inflow_1        reach1        outflow        true
junction      inflow_2        reach2        outflow        true
reach3        inflow          junction      outflow        true
```

Figure 3.2: *Example of a simple object linkage table.*

3.7 Object parameters

3.7.1 Object-specific scalar parameters

The numerical (scalar) parameters of the simulated objects are held in different tables if there are multiple object groups (i. e. classes). For each object group, a separate table must be supplied. These tables are in matrix format. There must be one column with name `object` holding the object names (ID strings). The remaining column(s) hold the parameters for the corresponding objects. The number of columns depends on the number of parameters that objects of the particular group (class) have. An example is given in Fig. 3.3.

3.7.2 Group-specific (shared) scalar parameters

In addition to object-specific scalar parameters (see Sec. 3.7.1), group-specific scalar parameters do exist. In contrast to the former, the values are shared by all objects of a particular group. The use of group-specific scalar parameters is often preferred over hard-coded parameters since the latter cannot be altered without recompilation of the model. To be consistent with the object-specific scalar parameters (see Sec. 3.7.1), the information for the different object groups is held in separate tables (see example in Fig. 3.4), each having the following two columns:

`parameter` (*string*) Name of the parameter.

`value` (*numeric*) Value of the parameter.

3.7.3 Object-specific parameter functions

Like the numerical (scalar) parameters, the parameter functions of the simulated objects are held in different tables if there are multiple object groups (i. e. classes). For each object group, a separate table with the following five columns has to be supplied:

`object` (*string*) Names (ID strings) of the objects.

`function` (*string*) Names of the functions to be assigned to the objects.

`file` (*string*) Names of the files containing the function data (see Sec. 3.7.5 for details on the format and restrictions).

`col_arg` (*string*) Names of the column where the argument values (x) reside in the corresponding data file.

`col_val` (*string*) Names of the column where the function values ($f(x)$) reside in the corresponding data file.

For each object, the table must contain n records, where n is the number parameter functions of the corresponding object group (i. e. object class). An example is given in Fig. 3.5.

3.7.4 Group-specific (shared) parameter functions

In addition to object-specific parameter functions (see Sec. 3.7.3), one may define group-specific parameters functions. In contrast to the former, these functions are shared by all objects of a particular group. Like all other parameters, the information for the different object groups is held in separate tables (one table per object group). The layout of the table is similar to the format described in Sec. 3.7.3, except for the fact that the `object` column is omitted (since the functions are *not* object-specific). Thus, the four expected columns are:

`function` (*string*) Names of the functions to be assigned to *all* objects of the object group.

`file` (*string*) Names of the files containing the function data (see Sec. 3.7.5 for details on the format and restrictions).

`col_arg` (*string*) Names of the column where the argument values (x) reside in the corresponding data file.

`col_val` (*string*) Names of the column where the function values ($f(x)$) reside in the corresponding data file.

3.7.5 Function data files

Function data files must have (at least) two columns of numerical values: a column of argument values and column of corresponding function values (see Fig. 3.6). If more columns are present, the additional columns are simply ignored. The columns must have unique names. The values in the arguments column must be in *strictly*

```
# Scalar parameters for the individual objects of a particular class

object  length  slope   roughness

reach1  1000.   1.e-03  20.
reach2  5400.   1.e-03  25.
reach3  3300.   1.e-04  30.
```

Figure 3.3: *Example of a table of object-specific scalar parameters.*

```
# Scalar parameters shared by all objects of a particular class

parameter      value

freezingPoint  0.0
density        1.0
```

Figure 3.4: *Example of a table of group-specific (shared) scalar parameters.*

```
# Assignment of parameter functions to the objects of a particular class

object  function  col_arg  col_val  file

lake_A  storage   volume   stage    data/functions/lake_A.txt
lake_A  outflow   stage    flow     data/functions/lake_A.txt
lake_B  storage   volume   stage    data/functions/lake_B.txt
lake_B  outflow   stage    flow     data/functions/lake_B.txt
```

Figure 3.5: *Example of a table of object-specific parameter functions. The file shows an example of two lakes, each being described by its storage curve ($stage = f(volume)$) and a rating curve at the outlet ($outflow = f(stage)$).*

increasing order (i. e. without duplicate values). The argument values *may or may not* be equi-spaced. Whether the use of equi-spaced arguments is advantageous depends on the specific application. Only some general recommendations can be given:

- If the function is tabulated with high resolution (many records) and/or the assessed argument values are highly variable from one time step to the next, equi-spaced arguments may be preferable. This is due to the fact that the value corresponding to an argument can be determined by index computation.
- If the appropriate resolution changes with the argument value (e. g. use of a logarithmic scale) and/or there is a high chance that the assessed argument values change only slightly (or not at all) from one time step to the next, one better uses irregularly spaced arguments. In such a case, the search always starts at the argument value that has been accessed most recently. This strategy usually allows for smaller input files, because one can use a high argument resolution where function values change rapidly and a low resolution elsewhere.

```
# Storage volume (cbm) of a river reach as a function of the flow rate (cbm/s)

flow    volume

0.0   0.000
0.2   244.778
0.5   460.687
1.0   759.379
2.0   1402.216
5.0   3290.339
```

Figure 3.6: *Example of tabulated function with irregularly spaced argument values.*

3.8 External forcings

3.8.1 Overview

The assignment of external forcings to the objects of a particular class is best illustrated using an example (Fig. 3.7). Let's assume the growth of urban trees is to be modeled. Five trees were selected for the study, located around the world (3 in Berlin, 1 in Tokyo, and 1 in Melbourne). Rain and sunshine are assumed to be the most important time-variable forcings of tree growth and, consequently, the 'tree' class has two external input variables, named 'rain' and 'sun'. Sunshine and precipitation data are available for different sets of climate stations. Fortunately, the stations recording sunshine data are located in the same city as the selected trees (Berlin, Tokyo, Melbourne). However, rainfall data for Berlin and Melbourne are unavailable. As a workaround, we simply use the rainfall data from Tokyo also for Melbourne. For Berlin, we interpolate available data from Moscow and Vienna instead, since Tokyo is really quite far away.

To make this strategy work, two things have to be done:

1. The two external input variables of the 'tree' class have to be linked with two time series files, containing the actual data for a single variable at all available stations. This is illustrated by the solid connecting lines in Fig. 3.7. The model's input file that is used to establish those links between variables and time series files is described in Sec. 3.8.3.

2. Links must also be established between the individual objects and the locations (i. e. climate stations). This is illustrated by the dashed connecting

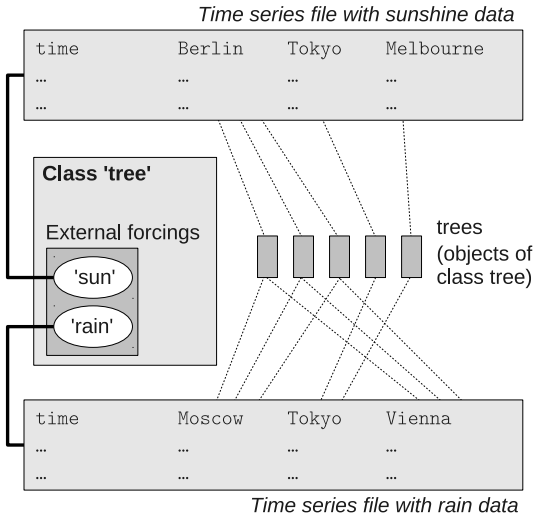

Figure 3.7: *Example illustrating the assignment of external forcings to objects of a particular class.*

lines in Fig. 3.7. Such links exist separately for each external forcing. Since a single object may be linked to more than one location, the links must also have a 'weight' attribute. This allows to account for the fact that Berlin is nearer to Vienna than to Moscow, when the rainfall for Berlin is estimated. The model's input file that is used to establish the links between objects and locations is described in Sec. 3.8.4.

3.8.2 Time series data files

A time series data file is a table with the usual header and *two or more* columns. As an exception to the usual convention (see Sec. 3.2), the time information must always be present in the *first* column of the table. The remaining n column(s) contain the time-dependend values of the respective variable at n locations. The order of these remaining columns is arbitrary since they are identified by their column names (which usually represent location names/IDs). The name of the first column containg the time information must be present but it is ignored. A reasonable name would be the abbreviation of the respective time zone, such as 'UTC'. An example of a time series data file is given in Fig. 3.8.

The entries in the time column (first column), must comply with the subsequent rules:

- Times must be encoded as stings in ISO 8601 format (`YYYY-MM-DD hh:mm:ss`) as already described in Sec. 3.2. Due to this format, the highest possible resolution is 1 second.
- Any character can be used to separate date and time information (blank is a usual convention). It may even be identical with a character used to separate the table columns.
- The times must be in *strictly* increasing order, i. e. the latest data are expected in the file's first record and there must be no duplicate times.
- Regular as well as irregular time series are supported, i. e. the time differences between neighbored records may be variable within a file. This offers the chance to use a higher resolution in periods of increased data variability and to use a low resolution when the values change slowly (or not at all). Such a strategy can save disk space and reduce the effort for reading data.
- The resolution, i. e. the smallest time differences between *any* neighbored records must be $\geq$ the simulation time step (see keyword `delta_t` in Table 3.5). To give an example: If the simulation time step is 1 hour (`delta_t=3600`), one can use time series with a minimum resolution of 1 hour or more (i. e. 1 hour, 2 hours, 1 day, 3 days, etc.). A time series file containing (some/only) 5 minute data, for example, will not be accepted.
- If the resolution of the time series data file Δt differs (for some or all interval(s)) from the simulation time step `delta_t`, the values are automatically transformed (i. e. reduced) if they represent sums (see discussion of column `sums` in Sec. 3.8.3).

With respect to the data values, the following restrictions apply:

- The values always represent averages or sums over a certain time interval (i. e. they do not represent instantaneous values). The respective time interval is determined by the difference in times between two neighbored records (see discussion of column `past` in Sec. 3.8.3 for details).
- Missing values or special values used to identify invalid data (such as `NA`) are *not* supported. Thus, data gaps must be handled by external software (or manual work) prior to model application. It is possible, however, to use special numerical values (often -9999) to mark missing/invalid data and to treat them properly in the classes' simulate methods.

3.8.3 Assignment of time series files and attributes to variables

Each external input variable which has been declared for a particular object class must be linked to a time series file. The time series file contains the actual data and its format is described in Sec. 3.8.2. In addition to that, a time series has further attributes which describe how the times and values are to be interpreted.

All information on the linkage of variables and time series files as well as time series attributes hast to be supplied in a single table with the following four columns:

`variable` (*string*) Names of the external input variables.

```
# Time series data file example

# Note that it depends on the used time series attribute 'past' how the
# data (temperatures) are interpreted. If past=true, the average temperature
# in Berlin between 14:00:00 and 15:00:00 was -1.8 degress Celsius. However, if
# past=false, a temperature of -2.0 is assigned to the mentioned time interval.

UTC                    Berlin  Tokyo  Sidney

2011-11-15 14:00:00:00    -2.0    10.     24.
2011-11-15 15:00:00:00    -1.8    11.     24.
2011-11-15 16:00:00:00    -1.3    11.     24.
2011-11-15 17:00:00:00    -1.0    10.     23.
```

Figure 3.8: *Example of time series data file containing values of a variable at three locations.*

`file` (*string*) Names of the files containing the time series data for an external input variable (see Sec. 3.8.2 for details on the format and restrictions).

`sums` (*logical*) If TRUE, the data value related to a particular time interval is interpreted as a sum. This is appropriate for data generally measured as sums. Examples include precipitation (given in units of mm/interval) or radiation (if expressed in units of Joule/interval). If FALSE, the data are interpreted as averages over the time interval. This is appropriate, for example in case of velocities (m/s) and the like, temperatures, or radiation intensities (expressed in units of Watts).

`past` (*logical*) It is a common practice that time series data files contain a single time column only, even if the stored values represent averages or sums over time intervals instead of instantaneous values (see Sec. 3.8.2). Consequently, there must be a convention whether a given time marks the *begin* or the *end* of an interval. This is accomplished through the setting of `past`. If `past=true`, it is assumed that the times given in the respective column of a time series data file mark *end-of-intervals*. This is a intuitive convention used by many (but not all) data providers. It is important to note that the begin of the time interval related to the very first record in the file is *unknown* if `past=true`. Consequently, the data values of the very first record are ignored. In contrast to that, times given in the time series file are assumed to mark the begin of time intervals, if `past=false`. Then, the end of the time interval related to the last record in the file is unknown and the values are, consequently, ignored. See also Fig. 3.8 for an example.

Note that the table does *not* contain a column like `objectGroup`. Thus, if an external input variable with the same name is declared in multiple object classes, the data are always taken from the same time series file. Thus, the table should contain as many records as there are unique names of external input variables in all object classes. An example of a simple table is provided in Fig. 3.9.

3.8.4 Assignment of external input locations to objects

The table used to establish the links between objects and certain columns of a time series data file (that usually represent different locations) consists of the four colums described below:

`object` (*string*) Names (ID strings) of objects getting external input.

`variable` (*string*) Names of the external input variable(s).

`location` (*string*) Location(s) to be assigned to a particular variable for a particular object. The used location name must be an existing column in the time series data linked to the respective variable (see Sec. 3.8.3).

`weight` (*numeric*) Weights to be assigned to a particular station for a particular variable and object. If,

```
# External forcings: Assignment of time series files & attributes

variable  sums   past  file

rain      true   true  precipitation.txt
flow      false  true  flowrates.txt
wind      false  true  windspeed.txt
```

Figure 3.9: *Example of table holding information on time series data files and attributes for a set of external input variables.*

for a specific variable, an object uses data from only a single location, the weight is generally 1.0. If, for a specific variable, the object is linked to multiple stations, the *sum of weights* over all locations (for that variable) is 1.0. This is just the usual case of spatial interpolation or, more generally, weighted averaging (see Eqn. 3.1).

The value of an external forcing applied to a particular object at a particular time, X is computed according to Eqn. 3.1. In this equation, w_i is the weight of a location with index i as assigned in the `weight` column of the described table. The symbol v_i denotes the value of the external variable for the same location (index i) read from the respective time series data file. The number of involved locations is n.

$$X = \sum_{i=1}^{n} w_i \cdot v_i \tag{3.1}$$

The described weighting approach and its typical use in the context of spatial interpolation is illustrated by Fig. 3.10. In the shown cases (a) and (b), the number of involved locations n with respect to a particular object and variable is 1 and the assigned weight w_1 equals 1.0. In the cases (c) and (d) we have $n > 1$ and multiple weights whose values satisfy Eqn. 3.1.

A minimum example of a table assigning external input locations to objects is given in Fig. 3.11. This table corresponds to the example introduced in Sec. 3.8.1. In realistic, more complex models, such a table may become quite large, especially if many objects are simulated that use information of multiple external input variables and the input (for a specific object and variable) is taken from multiple locations. This is due to the fact that the information for all objects (of all classes) is collected in a single table.

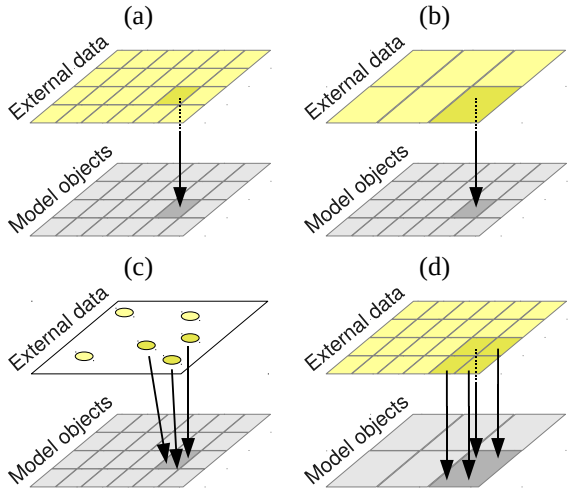

Figure 3.10: *Assignment of values of an external variable measured at multiple locations to the simulated objects (here represented by grid cells). Shown are typical situations arising in spatially distributed modeling: (a) Spatial resolution of the external variable matches with the model's resolution, (b) Use of low-resolution input in a high-resolution model, (c) Estimation of an object's input by interpolation of point data, (d) Use of high-resolution data in a low-resolution model.*

```
# External forcings: Assignment of locations for the 'tree example'

object        variable   location   weight

tree_berlin1  sun        Berlin     1.0
tree_berlin2  sun        Berlin     1.0
tree_berlin3  sun        Berlin     1.0
tree_tokyo    sun        Tokyo      1.0
tree_melbrn   sun        Melbourne  1.0

tree_berlin1  rain       Moscow     0.3
tree_berlin1  rain       Vienna     0.7
tree_berlin2  rain       Moscow     0.3
tree_berlin2  rain       Vienna     0.7
tree_berlin3  rain       Moscow     0.3
tree_berlin4  rain       Vienna     0.7
tree_tokyo    rain       Tokyo      1.0
tree_melbrn   rain       Tokyo      1.0
```

Figure 3.11: *Example of table holding information on the links between the simulated objects and the locations where data of external input variables are available. The table corresponds to the example used in Sec. 3.8.1 (Fig. 3.7).*

3.9 Initialization of states

3.9.1 Initialization table for scalar states

This table contains the initial values of all the scalar state variables of all simulated objects (see Fig. 3.12 for an example). The three required columns are defined as follows:

`object` (*string*) Names (ID strings) of all objects with one or more scalar state variable(s).

`variable` (*string*) Names of the scalar state variable(s).

`value` (*numeric*) Initial values assigned to the corresponding variables of the respective models.

3.9.2 Initialization table for vector states

As opposed to the initialization table for scalar state variables (see Sec. 3.9.1), the initialization table for vector state variables has an additional column named `index`. This column is of type *integer* and contains the element indices corresponding to the vector state variable specified in the `variable` column. The following rules apply to the `index` column:

- The C/C++ convention is used for the vectors' indices, i. e. the index of a vector's first element is 0 (not 1, as in many other programming languages).
- For each model and variable, at least one record must be present with a value of 0 in the `index` column. Thus, an initial value must be present at least for one (the first) element of each vector.
- More records may follow for a particular model and variable, with values of 1 through n in the index column. The index values must increase by 1 from one record to the next, i. e. there must be no gaps in the sequence of indices.
- The highest index value, n, determines the *initial size* of a vector. Since indexing starts at 0, the total vector size (number of elements) is $n - 1$. Note that a vector's size may change during simulation, depending on the code of the simulate method of the corresponding model class.

A simple example of an initialization table for vector state variables is given in Fig. 3.13.

```
# Initial values of all objects' scalar state variables

object        variable    value

reservoir_A   storage     3.e09
reservoir_A   waterlevel  255.8
catchment_X   snowheight  0.
catchment_X   soilmoist   .23
```

Figure 3.12: *Example of table with initial values of scalar state variables.*

```
# Initial values of all objects' vector state variables

object     variable    index   value

tree1      fruitSize   0       92.
tree1      fruitSize   1       88.
tree1      fruitSize   2       90.
# ... more data ...
tree782    fruitSize   0       101.
tree782    fruitSize   1       96.
# ... more data ...
```

Figure 3.13: *Layout of a table with initial values of vector state variables. The example corresponds to a model of apple trees having a variable number of fruits. Consequently, all properties of the individual apples must be held in vector state variables. Initially, the apple tree 'tree1' has three fruits and 'tree2' has only two. Depending on the implementation of the apple tree class, these numbers may change during simulation.*

3.10 Model output control

3.10.1 Selecting output variables for specific objects

For each object, the output of simulated values in the form of time series may be requested. Note that this is restricted to those variables which have been declared as 'outputs' in the corresponding class. Consequently, if time series output for a state variable is required, for example, one must declare an (additional) output variable in the respective object class and the values of the state variable must be assigned to the output variable in each time step. The same procedure is necessary in order to output time series of an object's external inputs, functions, etc. The table used to request output of certain variables for certain objects has the following two columns:

`object` (*string*) Names (ID strings) of objects for which output is requested.

`variable` (*string*) Names of output variables declared in the classes corresponding to the objects. A separate record is expected for each variable.

`digits` (*integer*) Controls the number of digits after the decimal place. Numbers are always printed in a fixed format, i. e. as 0.33 instread of 3.3e-01, for example.

An example of such a table is given in Fig. 3.14. It is important to note that it is *not* checked whether the entries in the `object` column actually represent the names of existing objects. Thus, to turn off `any` output, one could just supply a single record with a non-existing object's name.

Also note that the time series of all variables requested for a particular object are written to a single file. The name of this file is generated automatically by appending an extension (determined by the requested format) to the object's name. The directory where the output file will appear is controlled by the value assigned to key `outputDirectory` in the configuration file (see Table 3.6).

3.10.2 Enabling debug output for specific objects

By requesting debug output for certain objects, it is possible to create time series outputs for basically *all* computed values, namely

- Scalar state variables
- Vector state variables
- External input variables
- Simulated input variables
- Output variables.

This kind of output is particularly useful when debugging a model. Depending on the complexity of an object's data and the number of simulated time steps, the produced output files may become quite large. Therefore, the approach described in Sec. 3.10.1 is usually more appropriate if only some of the computed quantities are actually of interest.

The table used to request debug outputs consists of just a single column with name `object` (no example given). It holds the names (ID strings) of the objects for which output is requested. It is *not* checked whether the entries in that column represent the names of existing objects. Thus, if no debug output is required at all, the table should contain just a single record with a non-existing object's name. Note that the writing of large debug output files that are not actually required may lead to a significant waste of computing time and disk space.

The name(s) of the output file(s) are generated automatically by appending an extension (currently `.dbg`) to the objects' name(s). The directory where the output files will appear is controlled by the value assigned to key `outputDirectory` in the configuration file (see Table 3.6).

3.10.3 Output of the model's state at selected times

In some situation is is desirable to write the values of all state variables of all simulated *at a certain point in time* to disk. Potential uses of the produced file(s) containing the *entire model's state* include

- restarting of the model, using the produced files to initialize the state variables in a subsequent call,
- visualization of spatial patterns.

The table used to request outputs of the model's state consists of just a single column with name `time` (no example given). It holds the times for which the output is requested as strings in the usual ISO 8601 format (see Sec. 3.2). One should note that output is only created if

```
# Selection of for certain variables and objects for time series output

object     variable     digits

lake_1     waterlevel   2
lake_1     outflow      3
lake_1     temperature  0
lake_2     temperature  0
```

Figure 3.14: *Example of a table used to request the writing of time series files for selected output variables of certain models.*

a specified time also represents the end of a simulation time step (exactly to the second). Thus, it is not possible to request the model's state for intermediate times (such as 07:30, if the simulation time step is 1 hour and the model was started at a full hour). The only technique of suppressing outputs of the model's state at all is to specify (valid) times that do not meet the above criteria. Preferably, one specifies just a single time which is far outside the simulation time window and easily identified as a dummy such as `1900-01-01 00:00:00`, for example. Saving state information for many time steps despite of the fact that it is not actually required wastes both computing time and disk space.

For each point in time, two output files are created. One of the files contains the current values of scalar state variables and the other file holds the values of the vector state variables. The used formats are identical to the initialization tables described in Sec. 3.9.1 (Fig. 3.12) and Sec. 3.9.2 (Fig. 3.13), respectively.

The name(s) of the output file(s) are generated automatically using the respective time stamp. The values of the scalar state variables are saved in file `statesScal_YYYYMMDDhhmmss` and the vector state variables are written to file `statesVect_YYYYMMDDhhmmss`.The 14 digits at the end of the file names encode the time (4-digit year, followed by 2-digit month, day, hour, minute, and second). The directory where the output files will appear is controlled by the value assigned to key `outputDirectory` in the configuration file (see Table 3.6).

3.10.4 Precision of printed outputs

The current version of the **echse** writes all output data in a scientific format with three digits after the period. Thus, numbers are formatted like ±`X.YYYe`±`ZZ`, where `ZZ` is the exponent (base 10) corresponding to the number `X.YYY`. Currently, the output format *cannot* be changed by the model user.

A potential issue with the ±`X.YYYe`±`ZZ` format is that the precision of output data is limited to a total number of 4 digits. This is sufficient for many applications but may sometimes be problematic. In such cases, it is recommended to transform the data by adding or subtracting an appropriate constant (at latest before calling the `set_output()` method; see Table 2.2).

Let's consider the example of a reservoir in a mountainous region. Suppose that the reservoir's water level ranges from 2150 to 2180 m (a.s.l.) due to operation and fluctuations of the inflow. If the model writes the simulated water level to output files in units of m a.s.l., the precision is limited to 1 m only! To notice that, one must consider that the minimum and maximum values would be printed as `2.150e+03` and `2.180e+03`, respectively. By subtracting an appropriate constant, say 2100, the output precision can be increased significantly because the new data range is 50–80 (printed as `5.000e+01` and `8.000e+01`, respectively). After the transformation, the precision of the printed data is about 1 cm, i. e. 100 times higher.

Chapter 4

User guidelines

4.1 Model discretization

Often, alternative options of discretizing a real-world system do exist. Each alternative usually comes with specific pros and cons. Depending in the intended application of the model, a particular option may be more or less appropriate (or convienient to use). Basic aspects of proper model discretization are discussed in the subsequent sections Sec. 4.1.1–4.1.2.

4.1.1 Basic rule

The most basic rule is this: If two objects are charavcterized by different sets of state variables, i. e. the names of the state variables are not the same, the two objects belong to different classes.

4.1.2 Sub-discretization

Real-world objects can (or must be) often further discretized into sub-units. This is usually the case, when the object's state variables are subject to spatially variability. For example, a larger catchment can be broken into sub-catchments, that differ with respect to certain properties (proportion of forest, soil type, etc.). In such situations one has to decide whether the sub-discretization should implemented

- within the object, or
- by splitting an object into multiple separate objects.

Such a decision should be made based on the guidelines presented below.

4.1.2.1 Dynamic sub-discretization

If the sub-discretization (i. e. the number of spatial sub-units, for example) is time-variable, one must use vector-valued state variables (see Sec. 2.2.2). This is due to the fact that the size (i. e. the number of elements) of the vectors is allowed to change during the simulation period. A dynamic sub-discretization may be necessary when modeling the travelling of a flood wave in a homogeneous river reach represented by a cascade of linear reservoirs. In such a model, the appropriate number of linear reservoirs typically depends on the flow rate.

The approach of using vector-valued state variables only introduces additional state variables (but not parameters, etc.). Note: Pragmatically, vector state variables may also be 'misused' as scalar parameters, if parameter values are variable among the sub-units. This is, however, not efficient, because the info on parameters then (unnecessarily) appears in output files.

4.1.2.2 Static Sub-Discretization

If, in contrast to the situation discussed above, the number of sub-units is constant over the simulation period, several alternatives do exist:

State variables only If the sub-discretization is limited to state variables (i. e. all sub-units have common parameters and external inputs), one may use vector-valued state variables. The size of the vectors is simply held constant (by not changing it). Note: Pragmatically, vector state variables may also be 'misused' as scalar parameters, if parameter values are variable among the sub-units. This is, however, not efficient, because the info on parameters then (unnecessarily) appears in output files.

Fixed number of sub-units If the number of sub-units is the same for all objects of a class, one should use multiple scalar state variables (and/or parameters). Example: A catchment class, with a fixed number of land-use classes (such as 'forest', 'water', 'urban', 'other').

Remaining cases If (a) the sub-discretization is not limited to state variables (i. e. parameters are variable as well) and (b) the number of required sub-units is specific to individual objects, one should treat the sub-units as objects of a separate (additional) class and define appropriate interactions between the objects of the original and new class. Example: If the number of (relevant) land-uses in a catchment varies from one catchment to the next (and using a fixed number seems inefficient), one may introduce a class new 'landUseUnit'. The sub-units are then implemented as objects of that new class, each being linked to the corresponding catchment object.

4.2 Optimizing for speed

4.2.1 Parallel processing

The **echse** software comes with built-in support for *shared-memory* parallel computing. This is implemented using OpenMP (`http://openmp.org/`) which is supported by many modern compilers. Parallel processing can be enabled/disabled by the user via a model's configuration data (see Table 3.2).

Please note that *shared-memory* parallel computing only works on systems which are capable of running a *single process* as *multiple threads*. Please note, however, that the attempt to use multi-threading does not necessarily increase the performance, i. e. save computation time. In fact, depending on computer architecture (hardware) and the specific model, the model may *slow down* unexpectedly although it should become faster from theory. A possible reasons for this undesired behaviour might be that the overhead for the creation of multiple threads is higher than the actual gain from the parallel simulation. Another possible cause might be that, although the machine supports multiple threads, these threads share a limited ressource (like a floating-point arithmetic module, for example). Then, the multiple threads run sequentially in fact.

Present experience has shown that parallel processing is effective on a true multi-processor machine, i. e. a machine with more than one physical CPU[1]. It was found to be counterproductive, however, on multi-kernel architectures where all kernels are part of one single CPU[2].

Thus, it is recommended to always empirically determine the gain in computation speed (or slow down). This is easily done by inspecting the computation time of a particular simulation with multi-threading being turned on and off, respectively (see Table 3.2).

4.2.2 Miscellaneous

Declaring local variables of the simulate-Method as static does not have an effect (at least when compiling with gcc and optimization). It seems that the optimization performed by the compiler prevents the repeated allocation of memory on every call of the simulate-method.

[1] Dell machine with 4 CPUs of type Intel Core i7-2620M, each of which with 2 kernels, running 32 bit Ubuntu 12.04 LTS

[2] Several dual-core and quad-core machines running Windows 7

Chapter 5

Source code (PRELIMINARY)

5.1 Programming language

Execution time is critical for many operatinal and scientific applications. Therefore, the use of a compiled language (like FORTRAN 95+ or C++) is preferred over the use of a interpreted language (like Java, Matlab, R, . . .). C++ was selected as the language for implementing **echse** for the following reasons:

- Execution speed of the compiled code.
- Full support of object-oriented (OO) programming features.
- A standard way of exception handling exists.
- Availability of libraries.
- Availability of free compilers for any platform (gcc).
- Widespread use.

List of Figures

List of Tables

Bibliography

Ahuja, L.R., Ascough, J.C., David, O., 2005. Developing natural resource models using the object modeling system: feasibility and challenges. Advances in Geosciences 4, 29–36.

DHI, 2006. ECO Lab – A numerical laboratory for ecological modeling. Danish hydraulic institute. URL: http://www.dhisoftware.com.

Hill, C., DeLuca, C., Balaji, Suarez, M., Da Silva, A., 2004. The architecture of the earth system modeling framework. Computing in Science & Engineering 6, 18–28.

Kneis, D., 2012. Eco-Hydrological Simulation Environment (ECHSE) - Installation and Administration Guide. University of Potsdam, Institute of Earth- and Environmental Sciences. URL: http://echse.bitbucket.org/downloads/documentation/echse_install_doc.pdf.

Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P., 2002. Numerical recipes in Fortran 90 - The art of parallel scientific computing. 2 ed., Cambridge university press.

Regnier, P., Vanderborght, J.P., Steefel, C.I., O'Kane, J.P., 2002. Modeling complex multi-component reactive-transport systems: Towards a simulation environment based on the concept of a knowledge base. Applied Mathematical Modelling 26, 913–927.

Reichert, P., 1998. AQUASIM 2.0 - Computer program for the identification and simulation of aquatic systems, User manual. EAWAG. URL: www.aquasim.eawag.ch.

Thullner, M., Van Cappelen, P., Regnier, P., 2005. Modeling the impact of microbial activity on redox dynamics in porous media. Geochimica et Cosmochimica Acta 69, 5005–5019.

Index