Chapter 1: Introduction and Background

Dennis Timlin and Yakov Pachepsky

1.1 The importance of two-dimensional soil modeling

Farm management practices lead to dependencies of soil conditions on the distance from the plant row (Allmaras and Nelson, 1971; Logsdon et al., 1990; Logsdon and Allmaras, 1991). Management practices that affect the two-dimensional spatial distribution of soil and canopy conditions in row crops include the creation of soil ridges, furrow irrigation, placement of plant residues, banded fertilizer application, and mixed canopy geometry. Comprehensive soil models are needed to adequately represent the two-dimensional nature of management practices in row crops and their effects on the below-ground root environment.

Differences in soil conditions between row and interrow zones may have a large impact on the fate of applied agro-chemicals (Timlin et al., 1992). Agricultural chemicals that are not taken up by plants (or degraded into harmless by-products) may find their way into water supplies. A current research priority is to determine which agricultural management practices minimize the movement of chemicals to groundwater. Two-dimensional crop and soil simulation models have several advantages for research. One advantage is the ability to test the leaching potential of various hazardous compounds without actually applying them in the field. Another is to rapidly investigate the effects of different management actions on chemical movement in soils.

Comprehensive soil models, such as RZWQM and LEACHM, are one-dimensional. Existing two-dimensional soil models take into account very few soil processes (Šimůnek et al., 1992; Benjamin et al., 1990). Nevertheless, these models demonstrate the usefulness of a two-dimensional approach for quantifying the multi-dimensional nature of the soil environment under row crops. Plant growth models, which recently have become quite sophisticated, still rely on

simplified representations of soil processes. As a result, the predictive capabilities of these models are less than optimal.

A major drawback to the effective incorporation of sophisticated soil codes into plant models has been the complexity of the soil codes. Usually, the more processes a modeler attempts to incorporate, the more complex the model becomes.

The purpose of this project was to develop a comprehensive simulator for two dimensional soil problems with a modular structure that would simplify model development and maintenance. This goal is to simplify the integration of soil code with a crop model. This document describes a soil simulator, 2DSOIL, that provides for ready replacement of modules, simple addition of new modules, and consequently, an ability to select modules that correspond to the soil properties under consideration. The majority of soil-crop modelers use FORTRAN, and so this language has been chosen for this soil simulator. 2DSOIL is not offered as a finished product but as a vehicle for other scientists to test their own modules and to assemble plant-soil models that correspond to their own problem environments.

1.2 The design of 2DSOIL

1.2.1 Modular Design

The purpose of decomposing a system into modules is to be able to hide code and data from programmers modifying other modules in the system (Einbu, 1994). The goal is to separate the user of the module from the developer (Thomas, 1989). Modules should be designed in a manner that will enhance reuse, maintainability, and reliability of the code. This goes beyond previous philosophy where modularization was influenced by the 'everything is a hierarchy' view of programming (Kirk, 1990). Modules should have loose inter-unit coupling and high internal cohesion (Witt et al., 1994). Loose coupling is characterized by independence, a simple interface between modules, a limited number of interfaces, and a minimum of information passing. High

cohesion is characterized by interdependent elements packaged together and information hiding (Blum, 1992; Witt et al., 1994).

The design requirements for modules as outlined above do not require a special programming language (Blum, 1992), although they are more easily implemented using an object oriented programming (OOP) language. In OOP, each object (module) contains specific information (data) and is coded to perform certain operations (functions). Our design, which is coded in FORTRAN, follows OOP precepts as far as possible, but within the constraints of FORTRAN. Loose coupling is implemented by having each module input and manage its own data. By doing this, the functions and data are kept together in the object (module). High cohesion is achieved by developing distinct, independent modules, each for a specific soil or root process.

Soil and root processes belong to one of the following classes:

- transport processes, e.g., water, heat, nitrate, and oxygen movement;
- root processes, e.g., water uptake, NH^{+4} uptake, root respiration;
- interphase exchange processes, e.g., reversible Ca^{2+} exchange;
- biotransformation processes, e.g., denitrification, CO_2 respiration;
- management processes, e.g., tillage, ammonium phosphate application.

The soil and root process modules in 2DSOIL are based on this grouping (Fig. 1.1). Note that every class is on the same level, and hence that there is no hierarchy among the classes. Each class may include several process modules. The design assumes that only process modules needed in a particular application will be included in the simulator.

1.1.1 Data and the interaction of modules

Modules interact by passing or sharing data. To share data among process modules, the data must be represented consistently in all modules. To provide for loose coupling of modules, the amount of data accessible to all modules must be minimized. The minimum data set available to all modules has to be independent of the model or algorithms used to represent the processes, and must also be sufficient to describe the state of the system at any particular time. The soil and

root environment is partly characterized by the volumetric contents of substances (e.g., water content, bulk density, oxygen concentration, and root length density). Potentials of physical fields and related physical values are also used, e.g., matric potential, and temperature. These values are state variables of the soil-root system. The state variables are subject to changes caused by fluxes of energy or matter into or out of the system. Calculations of changes in one soil state variable may require the values of several other soil state variables. For instance, temperature and soil water content are needed to calculate changes in ammonium concentration caused by nitrification. Similarly, several processes may contribute to changes in the concentration of a particular substance. For example, both root respiration and decomposition of organic materials contribute to carbon dioxide production and uptake. Therefore, soil state variables and fluxes have to be available to all modules.

Values of the soil and root state variables can be recorded and calculated for specific locations within a soil profile; these locations must be the same for all modules. The spatial reference system in the generic simulator is introduced by a spatial grid which is a polygonal geometric structure representing either a vertical profile of the soil for the one-dimensional mapping of soil variables, or a vertical plane cross-section for the two-dimensional case. The grid is discussed in greater detail in Section 3 of this manual.

Modules have to be synchronized to calculate state variables and fluxes for simulated times which are the same for all modules. To do this, we used a sequential iteration approach (Yeh and Tripathi, 1991) in which process modules are execute sequentially as shown in Fig. 1.2. It is assumed that the values of fluxes are available at the beginning of each time step, before the transport codes are called. Each process module may have its own requirements for a time increment at any point in the simulation. For example, the time interval between fertilizer applications can be two months, the atmospheric boundary can be modified hourly, and calculations of infiltration may require time increments on the order of seconds. Combining modules which work with different time steps into one simulator, is referred to as asynchronous

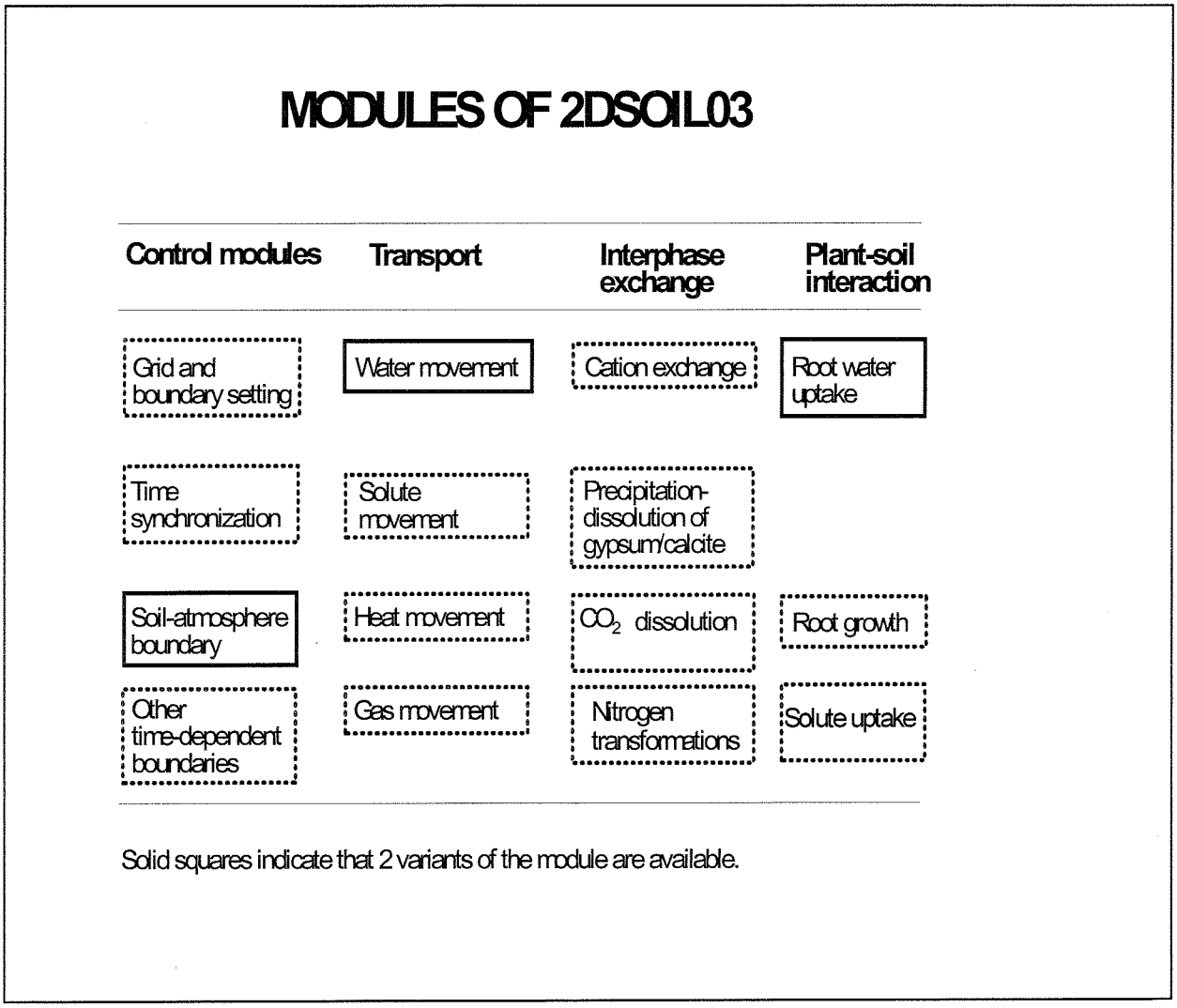

Figure 1.1 Modules of 2DSOIL release 03

coupling (ten Berge et al., 1992).

The general data structure that allows modules to be loosely coupled constitutes the framework of the generic simulator. This framework is supported by control modules. If a process module (such as a plant model) follows this framework, it can be included in the

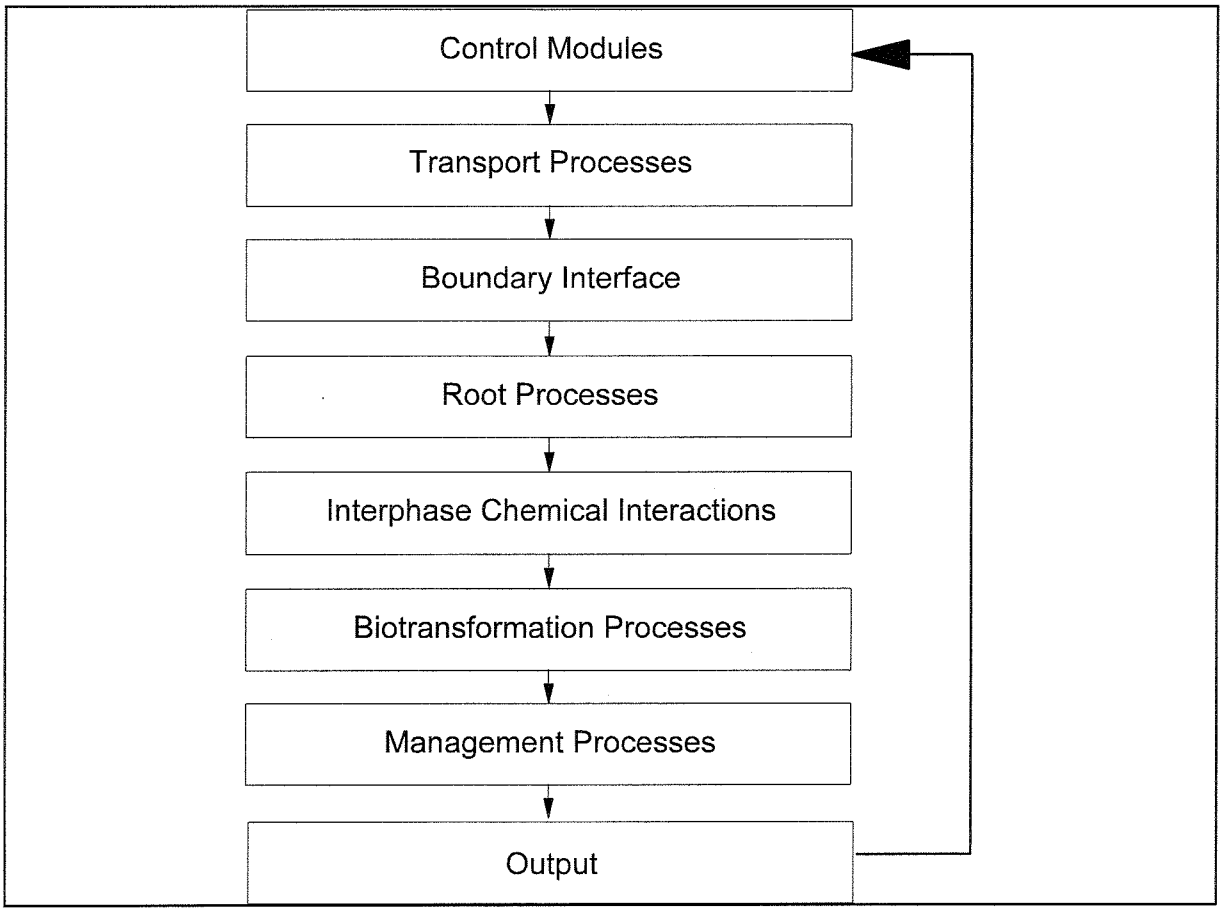

Figure 1.2 Flow chart illustrating the sequence of operations in 2DSOIL

simulator. The control module '*Synchronizer*' chooses the actual time step that the whole simulator will use. It is invoked at the beginning of every time step and uses as input the requirements of all modules for the next time step and the numbers of iterations that have occurred during the previous time step as. *Synchronizer* calculates the optimum time increment that (a) allows for convergence of iterations within modules and, (b) enables the simulator to read or to deliver data at specified times. If there are no modules with time step requirements, the time

step will increase gradually until it reaches a prescribed maximum value. Current time step, simulated time, number of iterations, and time step limitations are also available to all process modules to use.

1.1.2 Data structure

We used the concept of encapsulation of information to facilitate the independence of the modules. Encapsulation, a characteristic of object oriented programming, is the grouping into a

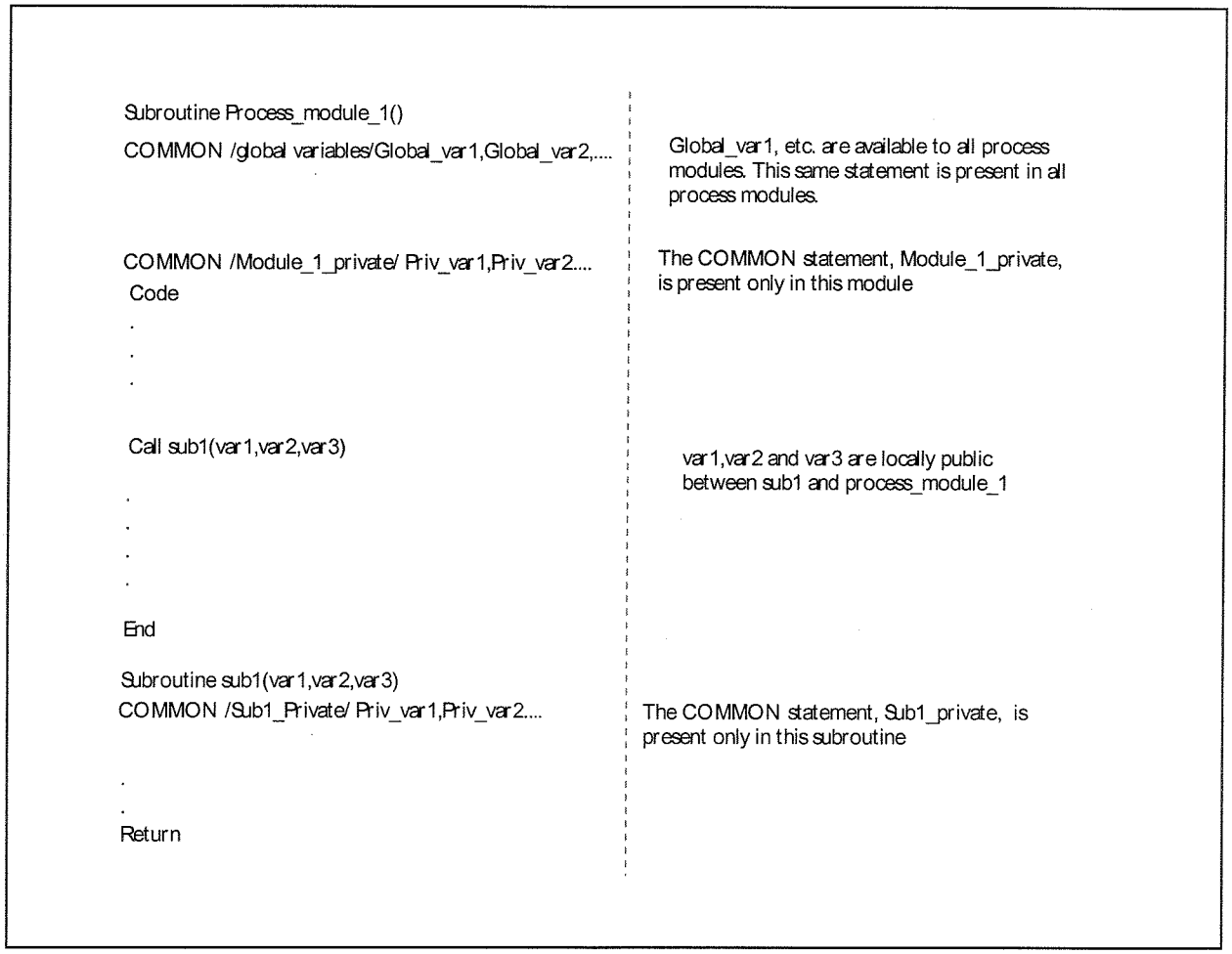

Figure 1.3 Illustration of the implementation of global, locally public, and private variables in 2DSOIL

single module of both data and the operations that modify or use those data (Wirfs-Brooks et al., 1990). The data and operations, encapsulated in a single module, can be hidden from the developers of other program units.

In order to implement data encapsulation in FORTRAN, public variables, i.e. variables available to all modules, were gathered into a separate public data field (Fig. 1.3). This global public field includes soil and root state variables, fluxes, nodal coordinates and temporal variables described in the previous section. Each process module also has its own private data field which is not available to other process modules (Fig. 1.3). Such a field may include control variables to read or print data, parameters of the numeric algorithms coded in the module, state variables and fluxes from the previous time step, and variables for communication between submodules that calculate coefficients in the model equations. If any module requires data from an external file to fill it's private data field, the module opens and reads that file itself. A module can also produce output according its own schedule.

Submodules that calculate coefficients for specific equations are also encapsulated within modules. Neither data nor equations of submodules are needed for the functioning of other process modules. The recommended data structure includes a subdivision of the private information of each process module into information that is locally public within the module, i.e., available to all submodules, and information that is hidden within submodules (Fig. 1.3). The hidden information is read independently and therefore the submodule can be easily replaced or modified. A need to replace a submodule is as common as a need to replace a process module. Consider, for example, the water movement module that requires hydraulic conductivity values and water content as a function of water potential. It may be desirable to be able to replace the equations that calculate these values (Alessi, 1992). The replacement will not affect the process module which needs the value of the hydraulic conductivity. Therefore, the submodule can read its own parameters from its own data file and have them hidden from the process module and the whole simulator.

Global public variables are passed in COMMON blocks to modules and submodules; no arguments are used in CALL statements at the process level. Errors are minimized by the use of INCLUDE statements, so as to insert a file containing a list of named COMMON blocks into

each process module. If it is desirable to transfer certain private variables into a public field, then only the insert file referenced in the INCLUDE statement has to be changed. Locally public variables, i.e. variables shared between process modules and submodules, are passed by means of CALL statements.

COMMON blocks are also used to store private information within a module or submodule. This feature saves the value of any private variables between invocations of a subroutine. COMMON blocks containing private information are present in only one module. Because there is no reference to this block in other program units, the information remains hidden.

1.1.3 Structure of a process module

The general sequence of operation in each process module (Fig. 1.3) consists of reading the time-independent data, calculating the auxiliary variables to determine the optimum time step, checking whether it is time to execute a block of code representing a process, and (if it is time to execute) reading the time-dependent data, changing the public variables, changing the private variables, calculating the requirements for time increments, and providing output data. Several of these steps may be absent in certain process modules.

```
    Subroutine Mngm()
    Include 'public.ins'                                  | Public variables are in "Public.ins'

    Common /Mngmt/ tAppl, cAppl, NumAp, nAppl(NumBPD),    | Private information is in a
                Mod_Num                                   | common block

    If(LInput.eq.1) then                                  | Input time independent information
     Open(40,file='Param_M.dat')                          |     Time of appplication
       Read(40,*,Err=20) tAppl, cAppl, NumAp              |     Total mass of chemical per node
                                                          |     Number of nodes where applied
       Read(40,*,Err=20) (nAppl(i),i=1,NumAp)             |     Node numbers to which chemical
                                                          |     is applied
       Num_Mod=Num_Mod+1                                  | Update the number of modules
       Mod_Num=Num_Mod                                    | and assign a number to this
                                                          | module
       tNext(Mod_Num)=tAppl                               | Assign the execution time for
       Close(40)                                          | this module

    Endif
    If(Abs(Time-tNext(Num_Mod)).lt.0.1*Step) then         | If it is time to carry out any calculations:
       Do i=1,NumAp                                       |
            Conc(nAppl(i))=cAppl/ThNew(nAppl(i))          |     change public variables
       Enddo                                              |
       tNext(ModNum)=1.E+32                               |     generate the next proposed time
    Endif                                                 |     step (only one application here so
    Return                                                |     the next time step is assigned an
20  Stop 'Mngm data error'                                |     unreachable value)
    End
```

Figure 1.4 Basic structure and sequence of operations in a process module.

An example of a typical module is shown in Fig. 1.4. This module simulates a chemical application. At the start of the module, designated by the public variable *lInput*, the module reads time-independent private data that are listed in the private COMMON block. The number, *NumMod*, of the module in the invocation sequence is used as an index for arrays that hold time step requirements for each module. This number is stored as private information. The public variable *tNext* stores the time- to- execute code for this particular activity; therefore, one of the future time steps has to terminate exactly at this time. When the execution time comes, concentrations of the chemical (which are public) in prescribed nodes are changed by dividing mass of chemical applied (which is private) by soil water content (which is public). Finally, the

public variable *tNext* is assigned an unreachable value which means that there are no further calls on this module.

1.2 Modules of 2DSOIL Release 03

The second version of 2DSOIL contained a number of soil processes: water flow, simultaneous convective-dispersive transport of several solutes, heat transport, gas movement, root water uptake, root growth, cation exchange, and carbonate-hydrogen chemistry in the soil solution. In this third version we have added nitrogen transformations along with an additional example (see Chapter 12). 2DSOIL has also been interfaced with a potato model; details are available in a separate documentation. A soil grid manager, a time synchronizer, a soil-atmosphere boundary manager, and a time-dependent boundary definer are the required modules.

The management of time synchronization and ordering of modules has also been changed. The module *Synchronizer* is now called at the beginning of the program. There is no longer an Object_Time module and Obj_Time.Dat file. The latter has been replaced by the file Time.Dat. The modules are now given identifying numbers (ModNum) dynamically as the program executes.

Several modules of 2DSOIL were adopted from other simulators in the public domain. All adopted codes were transformed to fit the modular structure of 2DSOIL. A careful comparison and selection process was followed in order to find the most reliable and computationally efficient codes. References are given in the sections that describe each module.

Chapter 2: Definition of Objects and Time Stepping; the *Synchronizer* Module

Yakov Pachepsky and Dennis Timlin

2.1 Function of the modules

The structure of 2DSOIL allows a modeler to use a subset of the modules that represent the soil-root processes of interest. There is no need to prepare data files for processes that are not needed in the simulations. As each process module is called, a variable, *NumMod*, is incremented. Use of this variable allows an identifying number to be associated with each module; this number is generally used as an array index.

Selection of the time step is an extremely important part of the total calculation procedure. Several rules were adopted from other studies (Shcherbakov et al., 1981; Vogel, 1987; Šimůnek et al., 1991) to regulate the time stepping in 2DSOIL:

(1) Any transport module may have restrictions on the time step that are connected with the numerical stability of calculations. These possible time step values are supplied by the modules themselves and are stored in the public *DtMx* array. *DtMx(i)* corresponds to the *i*th module (identified by the variable *ModNum*) according to Table 2.1.

(2) If a certain module has an iterative numerical procedure and convergence of this procedure depends on the time step, then the calculations will start with a very small time step, *Step*, to ensure convergence.

(3) If the number of iterations during the previous time step was less than a particular threshold, the time step is increased by *dtMul1* times. If the number of iterations was larger than a threshold value during the previous time step, then the time step is reduced by *dtMul2* times. The public variable *Iter* contains the number of iterations.

(4) If a scheduled change in the boundary conditions or in the spatial distribution of a variable is expected, then one of the time steps must be finished exactly at the time of such a change.

(5) If some module has its own time stepping schedule, then the end of its last time step must coincide with the end of one of the time steps generated for the whole program.

(6) If the results are to be printed to a file, then the time step must be chosen so that the time coincides with the time specified for output.

(7) If there is a large difference between the time step values allowed by rules (1)-(3) and by (4)-(6), then the time step allowed by (1)-(3) is halved to prevent sharp changes in the time step value.

(8) If during the calculations the time step becomes smaller than some threshold value, then the calculations are ended.

Applied together, these rules, in general, provide smooth time stepping. The **Synchronizer** module reads the step control constants from the **'Time.dat'** file and calculates time steps for the overall program according to the rules mentioned above. The model itself has only one time step at any particular point in the simulation. Although each process module may have its own requirement with respect to the time step, the purpose of the **Synchronizer** module is to evaluate all requirements and then choose a time step to satisfy the the process module with the most limiting requirement. The file **Time.dat** contains the parameters for controlling the time step control of overall program which are global in scope.

2.2 Input file *Time.dat*

The structure of this file is shown in Table 2.2. A reasonable value for the initial step may be $(tFin\text{-}Time)/10^6$. A reasonable value for the minimum time step may be $(tFin\text{-}Time)/10^{12}$. The value *DtMul1* is within the range (1.1;1.5) and the *DtMul2* value has the range.

Table 2.1. Format of the file **Time.dat.**

Record	Type	Variable	Description
	Time Stepping Control Parameters		
1	-	-	Comment line.
2		*Time*	Initial time.
2		*Step*	Initial time increment Δt [T].
2		*dtMin*	Minimal acceptable time step [T].
2		*dtMul1*	If the number of iterations required at a particular time step is less than or equal to 3, then Δt for the next time step is multiplied by a dimensionless number *dMul1* ≥ 1.0.
2		*dtMul2*	If the number of required iterations is greater than or equal to 7, then Δt for the next time step is multiplied by *dMul2* < 1.
2		*tFin*	Time to stop calculations.

The following example of the file **Time.dat** illustrates the format for the data required by the module:

```
**** Example 2.1: INPUT FILE 'TIME.DAT'
Time  Step    dtMin  dtMul1  dtMul2  tFin
 0    1.E-05  1.E-11   1.33     0.7   10
```

A free-format is used in 2DSOIL. The position and format of numbers in a record are arbitrary, although the sequence of numbers cannot be changed. Note that the units of time can range from days to seconds. Any consistent set of units must be used. Hence the units for other parameters, such as the hydraulic conductivity (LT^{-1}), must be consistent with the adopted time.