Code modernization and modularization of APEX and SWAT watershed simulation models

Robin A. J. Taylor[1*], Jaehak Jeong[1], Michael White[2], Jeffrey G. Arnold[2]

(1. *Blackland Research & Extension Center, Texas A&M AgriLife Research, Temple, Texas 76502, USA*;
2. *Grassland Soil & Water Research Laboratory, USDA-ARS Temple, Texas 76502, USA*)

Abstract: SWAT (Soil and Water Assessment Tool) and APEX (Agricultural Policy/Environmental eXtender) are respectively large and small watershed simulation models derived from EPIC (Environmental Policy Integrated Climate), a field-scale agroecology simulation model. All three models are coded in Fortran and have evolved over several decades. They are widely used to analyze anthropogenic influences on soil and water quality and quantity. Much of the original Fortran code has been retained even though Fortran has been through several cycles of development. Fortran now provides functionality originally restricted to languages like C, designed to communicate directly with the operating system and hardware. One can now use an object-oriented style of programming in Fortran, including inheritance, run-time polymorphism and overloading. In order to enhance their utility in research and policy-making, the models are undergoing a major revision to use some of the new Fortran features. With these new programming paradigms the developers of SWAT, APEX, and EPIC are working to make communication between the two models seamless. This paper describes the ongoing revision of these models that will make them easier to use, maintain, modify and document. It is intended that they will converge as they continue to evolution, while maintaining their distinctive features, capabilities and identities.

Keywords: code modernization, modularization, object-oriented programming, Fortran 2008, landscape-scale models, APEX, EPIC, SWAT

DOI: 10.3965/j.ijabe.20150803.1081 Online first on [2015-03-03]

Citation: Taylor R A J, Jeong J, White M, Arnold J G. Code modernization and modularization of APEX and SWAT watershed simulation models. Int J Agric & Biol Eng, 2015; 8(3): 81－94.

1 Introduction

FORTRAN, an acronym from FORmula TRANslating and now called Fortran, has been in use for 60 years since its public release by IBM in 1954 to translate scientific equations into computer code[1]. There are many millions of lines of Fortran code in daily use throughout the scientific and engineering communities. It is the primary language for some of the most intensive supercomputing tasks, such as astronomy, weather and climate modeling, structural engineering, computational physics, fluid dynamics, chemistry, economics, animal and plant breeding, and hydrological modeling. At the time Dr. Jimmy Williams started working on the field-scale model, EPIC (Erosion Productivity Impact Calculator) in the late 1970s[2-4], Fortran was essentially the only high level mathematically oriented computer language, and of course EPIC was written in Fortran. EPIC evolved over the next decade and was renamed Environmental Policy Integrated Climate[5-7]. Subsequently two spatially-explicit models, APEX

Received date: 2014-02-14 **Accepted date:** 2014-10-18

Biographies: Jaehak Jeong, PhD, Assistant Professor. Research interests: environmental and water resources engineering and modeling. Blackland Research & Extension Center, Texas A&M University, 720 East Blackland Road, Temple, Texas 76502, USA. Email: jjeong@ brc.tamus.edu; +1-254-774-6118. **Michael White,** PhD, Agricultural Engineer. Research interests: biosystems engineering. Grassland Soil & Water Research Laboratory, USDA-ARS, 808 East Blackland Road, Temple, Texas 76502, USA. Email: mike.white@ars.usda.gov; +1-254-770-6523. **Jeffrey G. Arnold**, PhD, Agricultural Engineer. Research interests: hydrologic and water quality modelling. Grassland Soil & Water Research Laboratory, USDA-ARS, 808 East Blackland Road, Temple, Texas 76502, USA. Email: jeff.arnold@ars.usda.gov; +1-254-931-4010.

***Corresponding author: Robin A. J. Taylor**, PhD, Senior Research Scientist. Research interests: systems ecology and modelling. Blackland Research & Extension Center, Texas A&M University, 720 East Blackland Road, Temple, Texas 76502, USA. Email: rtaylor@brc.tamus.edu; Phone: +1-254-774-6122.

(Agricultural Policy/Environmental eXtender) and SWAT (Soil and Water Assessment Tool), were developed using many of Williams' EPIC algorithms.

The APEX is a multi-field landscape model that essentially models multiple georeferenced EPIC fields within a watershed[4,6-8]. The Soil and Water Assessment Tool (SWAT) is a model for simulating large basins[6,9,10]. Where APEX models details of farming and land-use practices in small- to medium-sized watersheds up to about 10 000 km^2, SWAT sacrifices agricultural detail in order to simulate very large river networks of 100 s of thousands of km^2. Thus, EPIC, APEX and SWAT form a continuum of models[6] with EPIC simulating the evolution of a point in space (typically a single field) through time, while APEX and SWAT are explicitly spatially distributed, permitting simulations across an entire landscape. Despite these differences, and differing input and output structures, all three models have very similar internal organization. They are process-based simulation models that can be used either deductively to derive specific results or inductively to obtain general results. Thus, they can function tactically to solve an immediate problem, or strategically to investigate concepts and seek predictions that may be tested by new observation or experiment.

All three models are used around the world for environmental and conservation assessments[6,8,10-13]. In addition, APEX and SWAT are important components of USDA-Natural Resource Conservation Service in national Conservation Effects Assessment Program (CEAP) which assesses the efficacy of NRCS soil and water conservation programs and contributes to USDA conservation policy[14,15].

Both EPIC and APEX operate on a daily time step, although some processes are computed more frequently. SWAT is also primarily applied using a daily time step although the model can be executed with sub-hourly time steps as described by Jeong and colleagues[16,17]. Weather data drive the models, while the agricultural management schedules together with weather modify the simulated environments, including water, soil and plant growth. Although there are some differences, the landscape in APEX and SWAT is divided into georeferenced subareas with homogeneous slope, soil, weather and management practices; SWAT subareas or subbasins are more finely divided into non-georeferenced areas called Hydrologic Response Units (HRUs) that may differ in some other characteristics. The models comprise a set of nested loops that are executed annually, daily, by subarea (APEX and SWAT), HRU and sub-daily (SWAT only). Properties of the HRUs, subareas and soil layers are maintained in a set of vectors and arrays that are updated daily and output at programmed intervals: daily, monthly and/or yearly.

The models are constantly being refined with new algorithms added as required by USDA and other model users. The broad nature of these models cover many aspects of the environment; development of model routines requires not just programming skills but extensive knowledge of meteorology, soil chemistry and physics, limnology, hydrology, plant physiology, climatology and instream dynamics. Few people possess more than a couple of these knowledge sets, limiting the number of interested users able to make substantive contributions. Simplification and modularization of model code and reduced documentation overhead would allow the pace of model development to be accelerated by allowing researchers to concentrate on their specialization without the need for a detailed knowledge of the entire model code. In addition, the newer features of Fortran enable economical recoding of large parts of the models, which translate to greater efficiency and execution speed. The objective of this paper is to present the changes to SWAT, APEX and EPIC, we are making that will make them more useful research and policy tools. To this end, we will demonstrate how the new object-oriented functionality in Fortran will improve model structure and ultimately facilitate modifications and maintenance. The revision of SWAT, APEX and EPIC code is also intended to bring the models' input and output into a common form to facilitate communication between the models and to permit them to use the same databases. This will greatly facilitate their use in conjunction with one another.

In this paper, we present code and logic of features already implemented, currently being developed, and in

the planning stage. Actual code segments are presented in code boxes regardless of their development status in order to illustrate the new programing paradigms being incorporated into the models. The task of revision was started first with SWAT; consequently, more of the features we describe here are already incorporated into SWAT than either EPIC or APEX. We start with an overview of the modern Fortran language and its history, and proceed by describing the features that have been applied to the models and conclude with features still in the planning or early implementation stage.

2 The modern Fortran language

Fortran is a procedural, imperative, compiled language with syntax designed for efficient numeric processing that is highly efficient for processing data in arrays or requiring many iterations. Originally developed by IBM in 1952 and formally released in 1954, Fortran has evolved to include extensions to the language while usually retaining backward compatibility facilitating the maintenance of big simulation models (Table 1). As a result, Fortran continues to be the language of choice for high performance numeric processing. Successive versions have added support for abstract data types and dynamic data structures, enabling object-oriented and parallel programming paradigms.

Table 1 A brief history of Fortran

Year	Name	Features
1952	FORTRAN	IBM developed the Mathematical Formula Translating System
1954	FORTRAN	IBM released FORTRAN to users
1958	FORTRAN II & III	Procedural programming introduced: CALL, SUBROUTINE, FUNCTION, RETURN
1961	FORTRAN IV	Boolean expressions introduced
1966	FORTRAN 66	COMMON memory introduced
1977	FORTRAN 77	Block structures introduced: IF, THEN, ELSE
1991	Fortran 90	Recursion, Pointers, Dynamic memory (ALLOCATE), Operator overloading (INTERFACE), Derived (structured) data types (TYPE), Structured multi way selection (SELECT CASE) introduced from the C language.
1995	Fortran 95	Incremental revision to Fortran 90
2003	Fortran 2003	Inheritance & Procedure pointers introduced; seamless interoperability with C/C++
2008	Fortran 2008	Incremental revision to Fortran 2003; enhanced parallel processing features added

A few features have been deprecated (declared obsolescent in standard Fortran) but are retained for backward compatibility by many compliers (e.g. Intel® Fortran). Examples of obsolescent statements that we have removed include the ENTRY, computed GOTO, and arithmetic IF statements. Other obsolescent statements that retain some values in certain circumstances are statement functions, CHARACTER*(*), DATA, and unconditional GOTO statements. The concept of the MODULE for specifying and limiting the scope of variables while valuable in most applications has limited value in many models where large bodies of data need to be globally available. An important feature of the MODULE statement is it enforces object-oriented programming habits, although MODULEs are not necessary to develop object-oriented programs.

The goals of object-oriented programming are increased understanding, ease of maintenance, and ease of evolution. It is a programming approach for constructing modular reusable software systems and is a formalization of good design practices that go back to the earliest days of computer coding. Rather than structure program code and data separately, an object-oriented system integrates the two using the concept of an "object". An object has state (data) and behavior (code). A common source of errors in programs occurs when one part of the system accidentally interferes with another part. This is avoided by creating modular objects which manage their own data and are responsible for their own behavior. This feature, known as encapsulation avoids accidental interference of one piece of code by another by placing data where they are not directly accessible by the rest of the system. Instead, the data are accessed by calling specially written functions, called "methods". These act as the intermediaries for retrieving or modifying the data they control. The object's methods typically include checks and safeguards specific to the data types the object contains.

Alterations can be made to the methods of an object without requiring that the rest of the program be modified. For example, three different objects might provide methods for printing their data, with each object executing a print method tailored to a different kind of data (INTEGER, REAL, or DOUBLE PRECISION), but the methods are all called by a uniform statement (CALL

Output (Argument)) so that program code does not need to be modified in order accommodate a new data type (e.g., COMPLEX). Object-oriented programming practices become especially useful when more than one programmer is contributing code to a project; a common occurrence with EPIC/APEX/SWAT. Thus, one objective in recoding these models is to facilitate cooperation with other developers interested in using these models' framework to solve other environmental or policy problems.

A number of large models with which we are familiar have undergone or are undergoing transformation to this object-oriented style of programming. For example, the plant growth models comprising Decision Support System for Agrotechnology Transfer (DSSAT)[18,19], the groundwater simulation model MODFLOW[20] and the Object Modeling System (OMS)[21] modeling environment utilize these newer programming features by wrapping Fortran subprograms in object-oriented Java wrappers. The reconstructed DSSAT plant growth models use a modular structure developed by van Kraalingen[22] in which the same routines for computing soil water, soil nitrogen, weather, and sensitivity analysis are used by all plant growth models. Furthermore, the plant growth models are all driven by the same control program and conform to the same data standards and protocols for input and output. One objective of reconstructing EPIC, APEX and SWAT is to make their input and output procedures conform to the same rules or methods so that the models can access data from the same databases and output tables readable by the same post processor or dashboard. Another incentive for modularization is to be able to execute selected portions of a watershed routing structure in both APEX and SWAT. This ability will profoundly improve our ability to accurately represent natural systems with our models.

As some of Fortran's newer features borrowed from C and C++, may be unfamiliar to users and modifiers of these models, it is worth describing some of them before examining their application in EPIC/APEX/SWAT. There are six important features which have been introduced over the past two decades (refer to [1], [23], and [24] for details):

(1) When Fortran was first defined, the sizes of vectors and arrays needed to be declared at the beginning of the program segment to reserve memory at compile time for use in execution. With the introduction of dynamic memory, space for vectors and arrays no longer need to be reserved at compile time but can be created during execution. Thus, the size of the problem determines the size of memory to be used rather than the size of memory determining the size of the problem. Dynamic memory allocation has been implemented in all three models where appropriate. In those (rare) situations where dynamic allocation cannot be used, variables may be declared as pointers (see below) in labelled COMMON.

(2) Inheritance is a way to establish relationships between objects defined by classes, such that a new or derived class can inherit attributes and behavior from a pre-existing class. The relationship of classes through inheritance gives rise to a hierarchy. This has a use in EPIC/APEX/SWAT for output handling when averages over different variables or periods may be required. Inheritance has not been implemented, but is likely to become useful as we develop closer linkages between SWAT and APEX.

(3) Polymorphism is a language feature that allows values of different data types or functions to be handled using a uniform interface. Polymorphic functions may be created to operate on real and integer data, or to extend the precision of a function with 4, 8 or 16 bytes. A special case of polymorphism is operator overloading where an arithmetic operator (+, -, *, / or **) can be given special meaning via an INTERFACE statement or can have different implementations depending on the type of argument(s) in a CALL statement. For example, + (plus) is normally a binary operator (it adds two numbers), but can be redefined as a matrix operator adding two vectors or arrays. Since Fortran 90, simple matrix operations are possible using arithmetic operators; for example, element by element addition, subtraction, multiplication and division. With numeric applications the value of polymorphic overloading is the ability to create new arithmetic operators to evaluate equations or transforms. Polymorphism and overloading are being incorporated

into EPIC on a trial basis.

(4) Structures and abstract data types enable the storage and organization of a variety of different data types in a single efficient record. The elements of a record are usually called fields or members and are functionally equivalent to the fields of a relational database. Records are distinguished from arrays by having fields of different types. The classic example is the Personnel Record that contains fields for name (character variable), rank (integer variable), and salary of employees (real variable). A collection of records can be organized as an array with an index identifying each entry. Abstract data structures have great value in EPIC/APEX/SWAT as they allow variables, such as nutrients loading and sediment in water to be defined as properties of the watershed subarea. The subarea thus becomes the computational unit and, combined with operator overloading, arithmetic can be conducted on the watershed as a unit. All three models now use abstract data types to some degree and their use will expand as revision continues.

(5) Pointers are a data type that point to locations in memory containing data. Thus a pointer's value is not program data but an address pointing to program data. Pointers to data significantly improve performance for repetitive actions. In particular, it is often much cheaper in time and space to copy and use a pointer than it is to copy and access the data to which the pointers point. It should be noted that the ease with which pointers can be misapplied is directly proportional to their power. To avoid instability, it is vital that a strict protocol for creating and destroying pointers is rigidly adhered to, especially in a multi-programmer environment. Fortran subprogram arguments have always been pointers, contributing to the language's speed and efficiency. Pointers are being used in SWAT and EPIC, and planned for APEX.

(6) Recursion is a computational operation in which a process (a function or subroutine) can call itself. A key feature of recursion is self-similarity, in which a function is defined as a similar version of itself. The classic example is the definition of the factorial function:

$$N! = \prod_{n=1}^{N} n$$

where, the function calls itself recursively to multiply N by $(N-1)$, decrementing N by 1 at each step, until $N=1$. The great advantage of recursion is that an infinite set of possible operations on data can be defined or produced by a finite computer program. Another advantage of recursion in an algorithm is its simplicity. However, it has the disadvantage that the algorithm may require large amounts of memory if the depth of the recursion is very large, as the amount of memory grows geometrically with depth. So far recursion has not been incorporated into any model; the planned implementation for APEX is presented.

Combining these features provides for powerful new programming paradigms:

(1) By combining derived types with pointers, linked lists may be created permitting the storage in memory of relational databases and nested data structures permitting rapid access times. In a relational database each individual record is represented as a row, and each attribute as a column. One or more columns identify the entry as coordinates in a space of n-dimensions ($n \geq 1$). The remaining columns hold the data or attributes of the point defined by the coordinates. This is an example of one-to-one relationship between coordinates and data. In a one-to-many relationship, the data are nested or hierarchical and organized into a tree-like structure as described below. The structure allows information to be represented in a parent/child relationship: each parent can have many children, but each child has only one parent. Parents and children are tied together by pointers. Thus, a parent will have a list of pointers to each of its children. Linked lists have been incorporated into EPIC and are being tested in SWAT.

(2) Combining derived types with operator overloading enables arithmetic to be performed on entries in relational databases and nested data structures using purpose-built operators. Thus, operations are programmed such that the records are operated on as a set in a linear algebra-like way. Operator overloading is being planned for APEX.

(3) The combination of dynamic memory and

recursion enables arithmetic to be performed on nested or hierarchical data structures using multiple instances of a single function or subroutine. This is a very powerful feature enabling complicated processes to be addressed by a single, simple executable statement. The proposed implementation for APEX is presented.

At this time the recoding status is as follows: SWAT is most advanced and APEX least, with EPIC in between. None of the recoded models have been released for beta testing.

3 Applying the new Fortran

Until recently, the current construction of the three models was built around fixed dimension arrays as shown in Box 1. In this construction, vectors and arrays, for example those describing soil properties were dimensioned at compile time with explicit DIMENSION statements. By using allocatable arrays, the soil arrays can be dimensioned at execution time according to the number of soils and layers in the primary database. In place of explicitly dimensioning the arrays, they are given the attribute ALLOCATABLE, and the dimensioning is deferred until the required size has been read in or computed as seen in Box 2.

Box 1 Declared data arrays - A vector or array for every soil property dimensioned for a maximum number of soil types and layers using PARAMETER statements

```
INTEGER, PARAMETER :: $NST = 100          ! Maximum
number of soil types
INTEGER, PARAMETER :: $NSL = 15           ! Maximum
number of soil layers
!
COMMON /SOILS/ SALB($NST), HSG($NST), FFC($NST),
WTMN($NST), WTMX($NST), WTBL($NST), GWST($NST),
GWMX($NST), RFTT($NST), RFPK($NST), TSLA($NST),
XIDS($NST), RTN1($NST), XIDK($NST), ZQT($NST),
ZF($NST), ZTK($NST), FBM($NST), FHP($NST), Z($NST,
$NSL), BD($NST, $NSL), UW($NST, $NSL), FC($NST,
$NSL), SAN($NST, $NSL), SIL($NST, $NSL), WON($NST,
$NSL), PH($NST, $NSL), SMB($NST, $NSL), WOC($NST,
$NSL), CAC($NST, $NSL), CEC($NST, $NSL), ROK($NST,
$NSL), CNDS($NST, $NSL), SSF($NST, $NSL), RSD($NST,
$NSL), BDD($NST, $NSL), PSP($NST, $NSL), SATC($NST,
$NSL), HCL($NST, $NSL), WPO($NST, $NSL), EXCK($NST,
$NSL), ECND($NST, $NSL), STFR($NST, $NSL), ST($NST,
$NSL), CPRV($NST, $NSL), CPRH($NST, $NSL),
WLS($NST, $NSL), WLM($NST, $NSL), WLSL($NST, $NSL),
WLSC($NST, $NSL), WLMC($NST, $NSL), WLSLC($NST,
$NSL), WLSLNC ($NST, $NSL), WBMC($NST, $NSL),
WHSC($NST, $NSL), WHPC($NST, $NSL), WLSN($NST,
$NSL), WLMN($NST, $NSL), WBMN($NST, $NSL),
WHSN($NST, $NSL), WHPN($NST, $NSL)
```

Box 2 Dynamic memory allocation - Runtime creation of arrays with the ALLOCATABLE attribute still requires every variable to be individually declared and dimensioned

```
! Soil property vectors made allocatable for later allocation
 REAL*4, ALLOCATABLE :: SALB(:), HSG(:), FFC(:), &      .
     .
 etc.
     .
 Z(:,:),SAN(:,:),SIL(:,:),BD(:,:),UW(:,:),FC(:,:)&
     .
 etc.
     .
 WBMN(:,:),WHSN(:,:),WHPN(:,:)
     .
     .
     .
 NST = 100                   ! Number of soil types defined in
                             ! code or as input
 NSL = 15                    ! Number of soil layers defined in
                             ! code or as input
     .
     .
     .
! Allocate memory for the soil property arrays
!
 ALLOCATE (SALB(NST), HSG(NST), FFC(NST),&
             STAT=Ierr)
 ALLOCATE (Z(NST, NSL), SAN(NST, NSL), &
             SIL(NST, NSL), STAT=Ierr)
!
 RETURN
 END
 COMMON /SOILS/ Soil                  ! labeled COMMON
                                      ! for speed & efficiency
```

A further enhancement to data storage is the concept of the derived type or data structure. In EPIC/APEX/SWAT there are both static and dynamic data that comprise manifold properties. The soil vectors in the above examples are read in initially and then evolve as the simulation progresses. Weather, crops and management practices all impact the soil composition of a subarea. Collecting them together in a data structure as in Box 3 eliminates the need to dimension every property separately (whether by declaration or by allocation). There is just one structure per soil. A vector of soil structures with all their properties is dynamically allocated in each instance of the model. An alternative to allocating a vector of structures is shown in Box 4. Here we create a linked list to simulate a database; each soil has a pointer called Next that points to the next entry in the database. In this construction, we have also made the soil LAYERS component of the SOILS structure a pointer so the vector of LAYERS structures can be specified by reading the number of layers from the external database during execution.

Box 3 Derived data types – Soil properties are collected together in data structures. 43 global properties and 20 layer properties contained in a single variable can be addressed individually or collectively by structure name. In this form, the number of soil layers is dimensioned explicitly using a PARAMETER statement

```
  INTEGER, PARAMETER :: $NSL = 15
                         ! Maximum number of soil layers
!
  TYPE LAYERS      ! Soil layer properties in a derived type
    REAL*4 Z       ! Depth to bottom of layers (m)
    REAL*4 PH      ! Soil pH
    REAL*4 SAN     ! Fraction of sand (%)
    REAL*4 SIL     ! Fraction of silt (%)
      .
    etc.
      .
    REAL*4 CAC     ! Calcium carbonate concentration (%)
  END TYPE LAYERS
!
  TYPE SOILS       ! Soil properties defined in a derived type
    REAL*4 SALB    ! Soil albedo
    REAL*4 HSG     ! Hydrologic soil group
                   ! (1=A; 2=B; 3=C; 4=D)
    REAL*4 FFC     ! Fraction of field capacity for water storage
      .
    etc.
      .
    REAL*4 FHP            ! Fraction of humus in passive pool
    TYPE(LAYERS) Layer($NSL)       ! TYPE LAYERS a
                          ! vector property of TYPE SOILS
  END TYPE SOILS
!
  TYPE(SOILS), POINTER :: Soil(:)   ! TYPE SOILS is
                          ! allocatable and placed in COMMON
```

Box 4 Combining derived data types and pointers to create a soils database in a linked list. The layers sub-structure is also created as a linked list enabling dynamic dimensioning of these variables also

```
 TYPE SOILS
  INTEGER        ID
  CHARACTER*32 Name
  UNION
  MAP
    REAL*4   Global(19) ! Vector equivalenced to 19 properties
  END MAP
  MAP
    REAL*4     SALB     ! Soil albedo
    REAL*4     HSG      ! Hydrologic soil group (1=A; 2=B;
                        !     3=C; 4=D)
      .
    etc.
      .
    REAL*4     FHP      ! Humus passive pool fraction (0.3-0.7)
  END MAP
  END UNION
  INTEGER     NSL                  ! Number of Soil Layers
  TYPE(LAYERS), POINTER :: Layer(:)        ! Pointer to soil
                                           ! layer structure
  TYPE(SOILS),   POINTER :: Next           ! Pointer to next
                                           ! soil type in list
 END TYPE SOILS
!
 TYPE(SOILS), POINTER :: Soil(:), ThisSoil   ! Linked list
                                   ! pointers are placed in
 COMMON /SOILS/ Soil, ThisSoil     ! labeled COMMON
                                   ! for speed & efficiency
```

The process of reading the external soils database is streamlined as the organization of the soil properties is coded into the structure (Box 5) and the internal organization of the soils database is the same as the external source. Additions to or subtractions from the external database require only corresponding changes to the structure definition; recoding of the input process is not required. The example here reads from a flat file defined with an OPEN statement to channel K, but the same logic would apply with reads using calls to a Fortran-compatible Structured Query Language (SQL) or Open Database Connectivity (ODBC) library.

Box 5 Reading data with derived types and pointers simplifies the code. Soil global (ThisSoil%Global) and layer (ThisSoil%Layer) data are each read with a single statement that does not need to be revised if the database is extended by addition of new variables to the structures. The soils linked-list is extended automatically with the ALLOCATE (ThisSoil%Next) statement for each additional soil defined by a unique identifier (ThisSoil%ID) and the number of layers (ThisSoil%NSL). An additional variable reads the soil series name or other text identifier (ThisSoil%Name). Any read error terminates execution with a diagnostic to identify the problem

```
OPEN(UNIT=K, FILE='Soil file.dat', ACTION='READ')
                                  ! Open the database file
ALLOCATE(Soil)                    ! Allocate first soil
ThisSoil => Soil                  ! Point to first soil
NULLIFY(ThisSoil%Next)            ! Nullify pointer to Next soil
DO WHILE(.TRUE.)
  READ(K,*,END=10, ERR=99) ThisSoil%ID, ThisSoil%NSL
                                  ! Read soil series header
  IF (ThisSoil%ID.EQ.0) EXIT      ! Exit if all soils read
  READ(K,'(A32)', END=99, ERR=99) ThisSoil%Name
                                  ! Read soil series name
  READ(K,*, END=99, ERR=99) ThisSoil%Global
                                  ! Read global propeties
  ALLOCATE(ThisSoil%Layer (ThisSoil%NSL))
                                  ! Allocate soil layers
  READ(K,*, END=99, ERR=99) ThisSoil%Layer
                                  ! Read Soil layer properties
  ALLOCATE (ThisSoil%Next)        ! Allocate Next soil
  ThisSoil => ThisSoil%Next       ! Point to Next soil
  NULLIFY(ThisSoil%Next)          ! Nullify Next soil pointer
ENDDO
10 CONTINUE
ThisSoil => Soil                  ! Point to first soil
        .
        .
99 WRITE(*,*)'Error in soil read'
```

The foregoing example assumes the model will read all elements in the external database. The addition of a filter that selects only those to be used is a trivial addition to the code in Box 5. Similar constructions are implemented for other databases used by the models, for example the crop, fertilizer, and pesticide databases used in land management. In the case of soils that evolve

during the course of a simulation, copies are created and are incorporated into the subarea as a property of the subarea. Similarly, the weather and management schedule are attached to each subarea. Databases that do not change during a simulation (crops, fertilizers, pesticides, etc.) remain separate and are accessed by each subarea's management schedule as the schedule is executed, similar to the existing model structure. In most instances, changes to the code executing a process in the model are minor, as it is only the memory management that has changed by the introduction of derived types. The following example illustrates this. Box 6 shows the original code for the subroutine that computes the phosphorus flux between the several phosphorus pools. Five arrays are defined in a module called PARM (the same would be achieved by using the INCLUDE statement for file definitions).

Box 6 Executing code with arrays. With action variables in arrays defined in COMMON or a MODULE (BK, PSP, WPMA, WPML, WPMS), a subroutine to compute phosphorus flux is called with arguments (ISL, ISA) defining the variables in the soil layer and subarea for execution

```
 CALL NPMIN(ISL, ISA)              ! Operate on Layer ISL
                                   ! of soil in Subarea ISA
   .
   .
 SUBROUTINE NPMIN(ISL,ISA)
!
! PROGRAM APEX0806
! SUBROUTINE NPMIN     Computes phosphorus flux
!          between the soluble, active mineral & stable
!            mineral phosphorus pools.
!
 USE PARM                   ! BK, PSP, WPMA, WPML, WPMS
                            ! defined in MODULE PARM
!
! Arguments
!
 INTEGER ISL     ! The current layer of the subarea's soil series
 INTEGER ISA     ! The current subarea
!
 RTO = MIN(0.8, PSP(ISL, ISA)/(1.0 − PSP(ISL, ISA)))
                               ! P sorption coefficient
 RMN = PRMT(84)*(WPML(ISL, ISA) − WPMA(ISL, ISA)*&
        RTO)                   ! Flow rate labile->active
 X1 = 4.0*WPMA(ISL,ISA) − WPMS(ISL,ISA)
 IF( X1.GT.500.) THEN          ! Flow rate active->stable
   ROC = 10.0**(LOG10(BK(ISL, ISA)) + LOG10(X1))
 ELSE
   ROC = BK(ISL, ISA)*X1
 ENDIF
 ROC = PRMT(85)*ROC
 WPMS(ISL, ISA) = WPMS(ISL, ISA)+ ROC    ! New soluble P
 WPMA(ISL, ISA) = WPMA(ISL, ISA) − ROC + RMN
                                          ! New active P
WPML(ISL, ISA) = WPML(ISL, ISA) – RMN    ! New labile P
!
 RETURN
 END
```

The call to SUBROUTINE NPMIN has arguments specifying the subarea and the soil layer to be operated on, ISA and ISL, respectively. These indices are used to access data BK and PSP and update the phosphorus data contained in arrays WPMA, WPML, and WPMS. The corresponding code using structures and pointers as shown in Box 7, is very similar, except that a pointer is used to identify the soil defined in the subarea structure, and here we are using an INCLUDE statement rather than a USE statement to define the soil structure. As before, the particular layer to be operated on is a scalar argument. The only changes to the code involve specifying the phosphorus properties as part of the soil structure which is defined in the Fortran definition file Structures.fd.

Box 7 Executing code with derived types and pointers. The variables for calculating phosphorus flux (bk, psp, wpma, wpml, wpms) are elements of a structure This Soil which is an argument to the subroutine with L defining the layer for to be operated on. Both input and output are contained in the structure This Soil but the logic of the subroutine remains unchanged from the separate arrays approach

```
 CALL NPMIN(ThisSoil,L)        ! ThisSoil points to the current
                               ! subarea's soil
   .                            ! L defines the layer of
                               ! ThisSoil's soil
   .
 SUBROUTINE NPMIN(Soil, L)
!
! PROGRAM APEX0806
! SUBROUTINE NPMIN - Computes phosphorus flux between
! the soluble, active mineral & stable mineral phosphorus pools.
!
 INCLUDE 'Structures.fd'          ! TYPE(SOILS) defined in
                                  ! Structures.fd
!
! Arguments
!
 TYPE(SOILS) Soil        ! Local variable aliased with ThisSoil
 INTEGER L               ! The current soil layer
!
 RTO = MIN(0.8,Soil%psp(L)/(1.0 − Soil%psp(L)))
                               ! P sorption coefficient
 RMN = PRMT(84)*(Soil%wpml(L) − Soil%wpma(L)*RTO)
                               ! Flow rate labile->active
 X1 = 4.0* Soil%wpma(L) − Soil%wpms(L)
 IF (X1.GT.500.) THEN          ! Flow rate active->stable
   ROC = 10.0**(LOG10(Soil%BK(L)) + LOG10(X1))
 ELSE
   ROC = Soil%bk(L)*X1
 ENDIF
 ROC = PRMT(85)*ROC
 Soil%wpms(L) = Soil%wpms(L) + ROC           ! New soluble P
 Soil%wpma(L) = Soil%wpma(L) – ROC + RMN ! New active P
 Soil%wpml(L) = Soil%wpml(L) − RMN            ! New labile P
!
 RETURN
 END
```

The farm or watershed study may involve several fields or subareas. Each subarea is homogenous in climate, topography, soil, and land management schedule. Therefore, the heterogeneity of a watershed/farm is determined by the number of subareas. Each subarea may be linked with each other according to the water routing direction in the watershed, starting from the most distant subarea towards the watershed outlet. Two network characteristics are recognized; headwaters or extreme reaches that have only an outlet, and downstream reaches with both an inlet and outlet. The current algorithm uses three variables to instruct the model in the topology of the network. Extreme (headwaters) areas in a watershed are identified by defining the channel length (CHL) and channel length of routing reach (RCHL) to be the same length (CHL=RCHL). Downstream subareas have unequal channel length and routing reach (CHL>RCHL). Outlet data are added to the next downstream reach until either a negative watershed area (WSA<0) or another headwaters reach (CHL=RCHL) is encountered when the subarea outlet data are stored. A negative WSA indicates that stored information from a prior subarea in the list is to be added to this subarea. On encountering a negative WSA, the routing processor searches back through the list of processed subareas for one that has not been added into the network. Thus, the order of subareas (reaches) in the subarea file is critical for the correct routing topology.

A simple watershed with only four or five subareas is not difficult to parameterize correctly, but a large watershed with scores or hundreds of subareas requires a standalone program (APSUBLDM.FOR) to create the file describing the topology of the watershed's subareas from three input files. One file contains the identification numbers of the entering and receiving subareas. Those that flow out of the watershed have a receiving number of 0. However, some of the subareas are extreme or headwaters areas that do not have an inflow. A number of subarea-specific variables associated with each subarea follow the inflow and outflow numbers. Another file defines subarea properties including channel characteristics, irrigation schedule, and fertilizer and/or manure applications. The third file controls the build process. Changes to large subarea files are fraught with problems. Reliably combining or dividing subareas to reduce or increase the number of reaches is extremely difficult unless APSUBLDM.FOR is used. An alternative method has been developed for constructing the SWAT and APEX subarea files using an interface to ArcInfo (® ESRI) to build the subarea file from a digital elevation model[25,26]. This approach has the advantage of being graphical enabling the user to visualize the basin under study.

As alluded to above, the properties of a subarea include its soil, crop(s), weather, and management schedule. In addition, the subarea's water balance and chemical constituents are also properties defined in structures that are part of the subarea property set (Box 8). In each daily iteration of the model, subareas receive and transmit water, sediment and chemicals requiring careful bookkeeping in order to reliably simulate their flow through the watershed.

Box 8 Using the concept of the derived type, subareas become a collection of data structures containing subarea properties (Weather, Soil, Water, Chems, Crop, Sched, Outflow). Three pointers specify input and output subareas (Inlets, Outlet) and the order of execution (Next)

```
 TYPE SUBAREA
 INTEGER                    IDSA       ! Subarea identifier
 CHARACTER*40               Name       ! Subarea name
 TYPE(WETHR)                Weather    ! Structure with today's
                                       ! weather for subarea
 TYPE(SOILS), TARGET ::       Soil       ! Soil characteristics
 TYPE(WATER), TARGET ::       Water    ! Water balance
(above & below ground)
 TYPE(CHEMS), TARGET ::       Chems        ! Chemicals
                              ! (C, N, P, K, metals & pesticides)
                       !     (Cross-referenced with Soil & Water)
 INTEGER              nCrops      ! Number of crops in the
                                  ! management schedule
 TYPE(CROPS), POINTER ::      Crop(:)      ! Growth status of
                                           ! crop(s)
 TYPE(SCHED                  Sched      ! Management schedule
                             !     (Cross-referenced with Crop)
 TYPE(SUBAREA), POINTER :: Inlets(:) ! Pointer(s) to
                                     ! upstream subareas
                             !     (NULL if headwater subarea)
 TYPE(SUBAREA), POINTER :: Outlet    ! Pointer to
                                     ! receiving subarea
                             !     (NULL if watershed outlet)
 TYPE(SUBAREA), POINTER :: Next          ! Pointer to the
                             ! next subarea for processing
                             !     (NULL if watershed outlet)
 TYPE(FLOW)     Outflow     ! Outflow to downstream subarea
 END TYPE SUBAREA
!
 TYPE(SUBAREA), POINTER :: Subs(:), FirstSub, ThisSub
 COMMON /SUBAREA/ Subs,FirstSub
  .
  .
 ALLOCATE Subs(Site%NSA)        ! Number of Subareas is a
                                ! Site property
```

Two special properties of a subarea are its upstream and downstream neighbors which can be referenced via pointers. Each subarea (except headwaters) receives water, sediment and chemicals from one or more upstream subareas and transmits water, sediment and chemicals to a downstream subarea. Figure 1 represents both the geographic relationship of reaches in a watershed and the internal structure of a hierarchical linked list (1-24). In order that daily flow generated in an upstream subarea is recorded in the correct downstream subarea at the end of each day, a Next pointer specifies the order in which the subareas are to be computed.

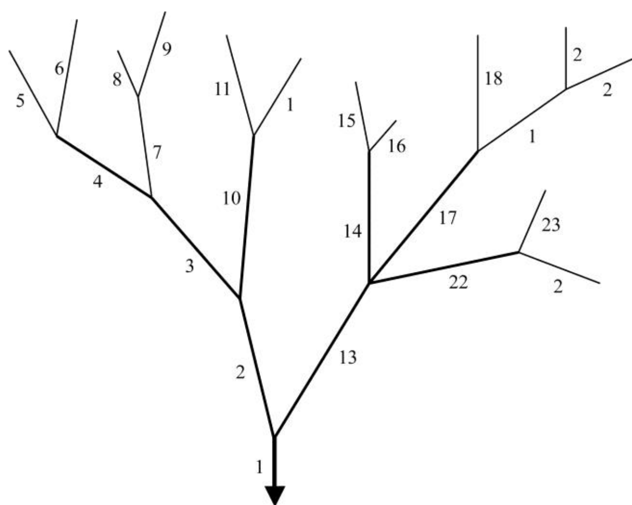

Figure 1 A watershed can be represented as a one-to-many type hierarchical linked list that starts at the outlet and progresses up each branch, ending at multiple distant sources

In the recursive subroutine (RankSA; Box 9) a loop is executed in which the subroutine calls itself but with a pointer to each Inlet in turn. After executing the first Inlet, any ASSOCIATED pointers, are executed in turn. As the recursion progresses, the program follows the order of reaches (i.e. stream orders) given by the numbers in Figure 1; first Reach 1, then Reach 2, followed by 3, 4, etc. At each node, execution either goes upstream to a new reach or drops back one level to a previous node.

As the execution progresses through the watershed, the rank of each reach (stream order) is assigned. In this example ranks of A to E are assigned, A to the outlet and E to the first-order reaches. On completion of ranking, the first-order reaches are numbered from 1 to 24 with the highest ranked (E) reaches numbered 1 to 13, then the next rank (D) numbered 14 to 19 for second-order reaches, and so on ending with number 24 (Rank A) for the fifth-order reach at the outlet. This order is shown in Figure 2. With this order of execution assigned, FirstSub (Box 9) is pointed at subarea 5, Subs(5), Subarea 5's pointer Subs(5)%Next is pointed at Subs(6), and so on. By specifying the order of execution using Subs(:)%Next pointers the first-order reaches are computed first (Rank E), then the next order reaches and so on, ending with the highest-order reach (Rank A).

Box 9 Efficient processing a watershed of a network of reaches as in Figure 1 is facilitated by recursively calling the processor as the program works its way through the stream network, starting at the outlet and ending at a distant headwater source

```
SUBROUTINE MakeNextPointers
TYPE(SUBAREA), POINTER :: Subs(:), FirstSub, ThisSub
COMMON /SUBAREA/ Subs,FirstSub
!
ThisSub => Subs(1)              ! Point to the subarea vector
CALL RankSA(ThisSub)            ! Call the recursive subroutine
!
CALL AssignPointers   ! Assign pointers to the ranked subareas
!
RETURN
END
!
RECURSIVE SUBROUTINE RankSA(Subarea)
TYPE(SUBAREA), POINTER :: Subs(:),FirstSub, ThisSub
COMMON /SUBAREA/ Subs,FirstSub
TYPE(SUBAREA), POINTER :: Upstream, Downstream
!
Upstream     => Subarea%Inlet
Downstream => Subarea%Outlet
!
CALL AssignRank(Subarea)       ! Assign rank to this subarea
!
DO WHILE (ASSOCIATED(Upstream)       ! Quit when no
                                     ! more upstream subareas
  CALL RankSA(Upstream)        ! RankSA calls itself for next
                               ! subarea
  Upstream => Upstream%Inlet   ! Point to next upstream subarea
ENDDO
!
RETURN
END
!
SUBROUTINE ProcessSA
TYPE(SUBAREA), POINTER :: Subs(:), FirstSub, ThisSub
COMMON /SUBAREA/ Subs, FirstSub
!
ThisSub    => FirstSub
DO I=1, Site%NSA                   ! Cycle through the subareas
  CALL DailyRoutine(ThisSA)        ! Daily subarea functions
                                   ! called from here
  ThisSub => ThisSub%Next          ! Point to next subarea
ENDDO
!
RETURN
END
```

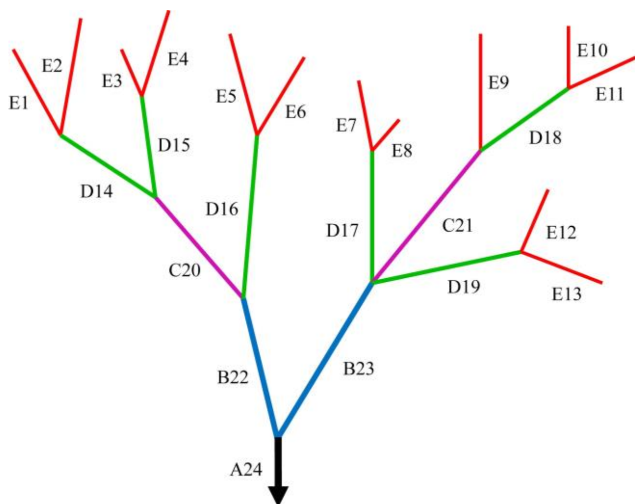

Figure 2 The watershed processes are executed from headwaters (Rank E) down to reaches of Rank D, C, and B, and then on to the outlet (Rank A) following the order defined by the chain of Subarea%Next pointers in the order given by the numbers following the Rank

As each subarea is operated on, structures for reach inflow and outflow are updated. The inflow structure is updated by reach(s) upstream that have access via their downstream Outlet pointers. The outflow is subtracted from the reach's initial state and added to the inflow of the downstream reach. When all reaches of the current rank have been computed, the reaches of the next order streams (downstream reaches) are processed by adding in the accumulated inflows, computing the flux and passing the outflow to the next order streams, and so on down to the outlet. Thus, by the use of pointers specifying the geometry of the watershed and the proper order of execution, the entire watershed may be operated on efficiently with a minimum of statements.

The above description refers to stream network processing, but the same logic applies to processing the subareas through which the streams flow. A single call in the control loop initiates the daily operations for each subarea in the order shown in Figure 2; the weather is calculated or read in, scheduled land management operations are executed, soil properties, water dynamics, sediment detachment and routing, nutrient cycling and transport, pesticide fate and transport, and other processes are computed, and the results stored in the structures Soils, Water, Chems, and Outflow.

Box 9 shows the code required to recursively process the subareas in the watershed to obtain the chain of pointers to execute from headwaters to outlet (RECURSIVE SUBROUTINE RankSA). There is very little overhead cost to analyzing the watershed to create the chain of pointers, which is more than offset by the efficiency of addressing the properties collectively in structures instead of separately in independent arrays.

An important benefit of this approach is that the subareas may be read into the model in any order, provided they each have an unique identifier (IDSA) and the identifier of the subarea into which it drains is specified. This simplifies editing the watershed: if it is necessary to add reaches, only the upstream output identifier needs to be edited to point to the inserted reaches. Adding headwater reaches requires no editing to existing subarea definitions.

Subtracting reaches is just as easy. An added benefit of this system is that a user can select just a subset of subareas (and thus just a portion of the routing structure) for a specific simulation. For example, a watershed calibration process can require hundreds of simulations, but may require examination of results from just a few sub-portions of the watershed. This approach allows users to run only those parts of the model needed for the calibration comparison and provides a much more flexible routing scheme than has been traditionally used in SWAT[27-29] and similarly for APEX[7]. Another feature of this system is that routing may be defined to model an estuary or delta. The example shows a typical upland watershed, with a single outlet per subarea. This method provides the means to define multiple outlets using a chain of outlet pointers.

Finally, there is the combination of structure and operator overloading that provides a powerful link to the object-oriented programming available in C++ and other newer languages. Two important features of object-oriented programming are inheritance and polymorphism. The former uses the concept of class to define a subprogram that can be used as the basis for secondary subprograms that make similar computations. By defining the processing of phosphorus in Box 7 as a class, a sub-class could process nitrates using the phosphorus code as the starting point for nitrate processing. Inheritance is a useful paradigm for manipulating objects

in a graphical user interface (GUI) for example, but its utility in process-oriented models programming is limited.

The final feature, polymorphism, enables the programmer to define new operators. For example, the standard operators (+ - * / **) can be extended or redefined for specific purposes. Its great power for EPIC/APEX/SWAT is the ability to create new operators, which do not replace function calls, but call functions in a different way. For example, computation of the mean of a list of numbers is traditionally achieved by calling a function:

Average = Mean(Input,N)

Using an operator (.MEAN.) to do the same thing looks like this:

Average = Input.MEAN.N

Box 10 Defining a new operator to compute averages combined with a pointer to the numbers to be averaged results in a succinct statement

```
 INTERFACE OPERATOR(.MEAN.)
  REAL*4 FUNCTION Mean(Input, Period)
   REAL*4,   INTENT(IN) :: Input(:)
   INTEGER, INTENT(IN) :: Period
  END FUNCTION MEAN
 END INTERFACE
!
 TYPE(Water), POINTER :: ThisWater      ! Pointer to
                                        ! the target variables
   .
 etc.
   .
 ThisWater => ThisSubarea%Water
 DO Mnth=1,12
  ThisSubarea%Month(Mnth)%Nitrate = &
                    ThisWater%Nitrate.Mean.Mnth
 ENDO
 ThisSubarea%Annual%Nitrate   = &
                    ThisWater%Nitrate.Mean.Year
 ThisSubarea%Annual%Sediment = &
                    ThisWater%Sediment.Mean.Year
   .
 etc.
   .
 END
!
 REAL*4 FUNCTION Mean(Input, Period)
! Computes the average value for variable pointed to
! by Input for interval Period
   .
   .
 RETURN
 END
```

On its own, this would seem to be just a semantic change, but in combination with pointers, it creates a simple and powerful extension to bookkeeping in EPIC/APEX/SWAT. The key feature of operator polymorphism is the procedure interface block that defines the operator (Box 10). It creates a link between the function performing the operation (REAL*4 FUNCTION Mean) and the new operator (.MEAN.) so that a function call can be written like an ordinary assignment statement:

ThisSubarea%Annual%Nitrate =
ThisWater%Nitrate.Mean.Period

The combination of a pointer to the data to be averaged and the new operator creates a uniform yet flexible statement with the second operand (Period) defining the interval over which averaging is to be computed; the function must be able to interpret the value of Period in terms of the desired interval. In the current implementation, the function interprets the interval value of 0 as annual average, values of 1-12 produce monthly averages, and a negative number is interpreted as the number of days to be accumulated for the mean.

4 Conclusions

All the changes described facilitate easier modification and documentation to EPIC, APEX, and SWAT, as well as simplifying maintenance. In addition, these modifications will bring EPIC and APEX into line with the design of a pest population dynamics model that will be the basis for expansion of these models to encompass multitrophic ecological processes. Incorporating more realistic pest-crop interactions will enable cost-benefit analysis of pesticide use as well as provide daily feedback between pest population and growing crop. Several other additions are also planned for EPIC/APEX. Like the pest management module, incorporation of more realistic spatial distributions of plants and plant community dynamics will both require a more modular approach. These features are especially important for analyses of rangeland grazing management, which is the latest component of the NRCS CEAP initiative[31].

These models are neither perfect nor complete; they are continually being updated and improved as our understanding of the natural world grows. The simplification and modularization of model code and reduced documentation overhead allows the pace of

model development to be accelerated by involving more researchers. The broad nature of these models cover many aspects of the environment; development of model routines requires not just programming skills but extensive knowledge of meteorology, soil chemistry and physics, limnology, hydrology, plant physiology, climatology and instream dynamics. No single individual or small group can possess an adequate in-depth scientific understanding in all these areas to keep these models state-of-the-art. By simplifying and restructuring model code a very detailed knowledge of the entire model code is no longer a prerequisite for developers. Collaborating experts with very specific disciplines (i.e. soil carbon or lacustrine nutrient cycling), but little code experience, can be leveraged into more active development. The accumulation of knowledge from this broad base of experts is needed to keep these models current and applicable to todays and tomorrows environmental challenges.

Acknowledgements

We thank Phil Gassman, the editor for the IJABE SWAT Special Issue and two anonymous referees for their critical reading and excellent suggestions. USDA is an equal opportunity employer and provider.

[References]

[1] Chapmann S J. Fortran 90/95 for Scientists and Engineers, 2nd Ed, McGraw-Hill Higher Education. 2004.

[2] Williams J R, Jones C A, Dyke P T. A modeling approach to determining the relationship between erosion and soil productivity. Transactions of the ASAE, 1984; 27: 129–144.

[3] Williams J R. The erosion productivity impact calculator (EPIC) model: A case history. Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences, 1990; 329: 421–428.

[4] Williams J R. The EPIC model. pp. 909-1000 in Computer Models of Watershed Hydrology, (Singh VP, ed.). Water Resources Publications, Highlands Ranch, CO, USA. 1995.

[5] Williams J R, Nearing M, Nicks A, Skidmore E, Valentin C, King K, et al. Using soil erosion models for global change studies. Journal of Soil and Water Conservation, 1996; 51: 381–385.

[6] Williams J R, Arnold J G, Kiniry J R, Gassman P W, Green C H. History of model development at Temple, Texas. Hydrological sciences journal, 2008; 53: 948–960.

[7] Williams J R, Izaurralde R C, Steglich E M. Agricultural Policy/Environmental eXtender Model: Theoretical Documentation, Version 0604. BREC Report 2008-17. Temple, Tex.: Texas AgriLife Blackland Research and Extension Center. 2008. Available at: http://epicapex.tamu.edu/downloads/user-manuals/. Accessed on [2014-08-17].

[8] Izaurralde R C, Williams J R, McGill W B, Rosenberg N J. Simulating soil C dynamics with EPIC: Model description and testing against long-term data. Ecological Modelling, 2006; 192: 362–384.

[9] Gassman P W, Williams J R, Wang X, Saleh A, Osei E, Hauck L M, et al. The Agricultural Policy/Environmental Extender (APEX. model: An emerging tool for landscape and watershed environmental analyses. Transactions of the ASABE, 2010; 53: 711–740.

[10] Arnold J G, Srinivasan R, Muttiah R S, Williams J R. Large area hydrologic modeling and assessment, Part I: Model development. Journal of the American Water Resources Association, 1998; 34: 73–89.

[11] Arnold J G, Moriasi D N, Gassman P W, Abbaspour K C, White M J, Srinivasan R, et al. SWAT: Model use, calibration, and validation. Transactions of the ASABE, 2012; 55: 1491–1508.

[12] Gassman P W, Williams J R, Benson V W, Izaurralde R C, Hauck L M, Jones C A, et al. Historical development and applications of the EPIC and APEX models. ASAE/CSAE Meeting Paper No. 042097. ASAE, St. Joseph, MI. 2004.

[13] Gassman P W, Reyes M R, Green C H, Arnold J G. The Soil and Water Assessment Tool: historical development, applications, and future research directions. Transactions of the ASABE, 2007; 50: 1211−1250.

[14] Wang X, Williams J R, Gassman P W, Baffaut C, Izaurralde R C, Jeong J, et al. EPIC and APEX: Model use, calibration, and validation. Transactions of the ASABE, 2012; 55: 1447–1462.

[15] Duriancik L F, Bucks D, Dobrowolski J P, Drewes T, Eckles S D, Jolley L, et al. The first five years of the Conservation Effects Assessment Project. Journal of Soil and Water Conservation, 2008; 63: 185A–197A.

[16] Wang X, Kannan N, Santhi C, Potter S R, Williams J R, Arnold J G. Integrating APEX output for cultivated cropland with SWAT simulation for regional modeling. Transactions of the ASABE, 2011; 54: 1281–1298.

[17] Jeong J, Kannan N, Arnold J G, Glick R, Gosselink L, Srinivasan R. Development and integration of sub-hourly rainfall–runoff modeling capability within a watershed model. Water Resources Management, 2010; 24: 4505–4527.

[18] Jeong J, Kannan N, Arnold J G, Glick R, Gosselink L, Srinivasan R, et al. Development of sub-daily erosion and sediment transport algorithms for SWAT. Transactions of the ASABE, 2011; 54: 1685–1691.

[19] Jones J W, Keating B A, Porter C H. Approaches to modular model development. Agricultural Systems, 2001; 70: 421–443.

[20] Jones J W, Hoogenboom G, Porter C H, Boote K J, Batchelor W D, Hunt L A, et al. The DSSAT cropping system model. European Journal of Agronomy, 2003; 18: 235–265.

[21] Harbaugh A W. MODFLOW-2005, The U.S. Geological Survey Modular Ground-Water Model—the Ground-Water Flow Process. Techniques and Methods 6–A16. U.S. Geological Survey, Washington, DC, USA. 2005.

[22] Ascough J C, David O, Krause P, Fink M, Kralisch S, Kipka H, et al. Integrated agricultural system modeling using OMS 3: Component driven stream flow and nutrient dynamics simulations. 2010 International Congress on Environmental Modelling and Software Modelling for Environment's Sake. Swayne D A, Yang W, Voinov A A, Rizzoli A, Filatova T, eds. International Environmental Modelling and Software Society, Ottawa, Canada. 2010.

[23] van Kraalingen D W G. The FSE System for Crop Simulation: Version 2.1 (Quantitative Approaches in Systems Analysis Report No. 1). C.T. de Wit Graduate School for Production Ecology, Wageningen University, Wageningen, Netherlands. 1995.

[24] Metcalf M, Reid J, Cohen M. Modern Fortran Explained. Oxford University Press, Oxford, UK. 2011.

[25] Markus A. Modern Fortran in Practice. Cambridege University Press, Cambridge, UK. 2012.

[26] Olivera F, Valenzuela M, Srinivasan R, Choi J, Chou H, Koka S, et al. ArcGIS-SWAT: A Geodata Model and GIS Interface for SWAT. Journal of the American Water Resources Association, 2006; 42: 295–309.

[27] Tuppad P, Winchell M F, Wang X, Srinivasan R, Williams J R. ArcAPEX: ArcGIS interface for Agricultural Policy Environmental eXtender (APEX) hydrology/water quality model. International Agricultural Engineering Journal, 2009; 18: 59–71.

[28] Arnold J G, Srinivasan R, Engel B A. Flexible watershed configurations for simulating models. Hydrological Science and Technology, 1994; 10: 5–14.

[29] Williams J R, Arnold J G. A system of erosion-sediment yield models. Soil Technology, 1997; 11(1): 43–55.

[30] Arnold J G, Kiniry J R, Srinivasan R, Williams J R, Haney E B, Neitsch S L. Soil and Water Assessment Tool Input/Output File Documentation: Version 2009. U.S. Department of Agriculture – Agricultural Research Service, Grassland, Soil and Water Research Laboratory, Temple, TX and Blackland Research and Extension Center, Texas AgriLife Research, Temple, TX. Texas Water Resources Institute Technical Report No. 365, Texas A&M University System, College Station, TX. 2011. Available at http://swat.tamu.edu/documentation. Accessed on [2014-08-17].

[31] Weltz M A, Jolley L, Goodrich D, Boykin K, Nearing M, Stone J, et al. Techniques for assessing the environmental outcomes of conservation practices applied to rangeland watersheds. Journal of Soil and Water Conservation, 2011, 66: 154A-162A.