Contents lists available at ScienceDirect

Computers and Geosciences

journal homepage: www.elsevier.com/locate/cageo

Research paper

The introspective may achieve more: Enhancing existing Geoscientific models with native-language emulated structural reflection

Xinye Ji, Chaopeng Shen [*]

Department of Civil and Environmental Engineering, The Pennsylvania State University, University Park, PA 16802, USA

ARTICLE INFO

Keywords:
Reflection
Geoscientific modeling
Efficient programming
FORTRAN
Generic data structure
Data I/O

ABSTRACT

Geoscientific models manage myriad and increasingly complex data structures as *trans*-disciplinary models are integrated. They often incur significant redundancy with cross-cutting tasks. Reflection, the ability of a program to inspect and modify its structure and behavior at runtime, is known as a powerful tool to improve code reusability, abstraction, and separation of concerns. Reflection is rarely adopted in high-performance Geoscientific models, especially with Fortran, where it was previously deemed implausible. Practical constraints of language and legacy often limit us to feather-weight, native-language solutions. We demonstrate the usefulness of a structural-reflection-emulating, dynamically-linked metaObjects, *gd*. We show real-world examples including data structure self-assembly, effortless input/output (IO) and upgrade to parallel I/O, recursive actions and batch operations. We share *gd* and a derived module that reproduces MATLAB-like structure in Fortran and C++. We suggest that both a *gd* representation and a Fortran-native representation are maintained to access the data, each for separate purposes. Embracing emulated reflection allows generically-written codes that are highly re-usable across projects.

1. Current issues

Geoscientific models (GMs), especially process-rich system-dynamics models (Kelly et al., 2013; Laniak et al., 2013), e.g., Community Land Model (CLM) (Lawrence et al., 2011), Noah-MP (Niu et al., 2011), iESM (Collins et al., 2015) and APSIM (Holzworth et al., 2014), manage a large variety of data. A common practice is to organize data into a hierarchical data structure. For example, the 4.0 version of the Community Land Model (CLM) holds data in 5 spatial scales, each possessing multiple sub-structures (e.g., carbon states, water states), and thousands of independent variables, which result from the inclusion of many component modules, e.g., hydrology, energy, photosynthesis, vegetation phenology, carbon and nitrogen cycles, which interact and depend on each other. Providing management support, e.g., input/output (I/O), model save and restart, memory allocation, module interfacing, and data access for such complex and a large count of variables can pose a practical challenge.

While environmental modeling frameworks (EMFs) have been promoted for some time, e.g., see (Argent et al., 2009; Holzworth et al., 2010; Janssen et al., 2011; Rahman et al., 2004; Rizzoli et al., 1998), we realize their adoption is nonetheless slow and systemic coding issues are still prevalent with many GMs: cross-cutting concerns, lack of metadata use, and strong mutual dependency. These issues, which are easily observed from a spectrum of models, are described in the following.

i There are pandemic, behaviorally similar tasks across multiple components of the model, sometimes referred to as "cross-cutting concerns" (Elrad et al., 2001; Kiczales et al., 1997), resulting in different, case-specific implementations of the same action scattered throughout the model. For example, the allocation and initialization of data, checking bounds violation and occurrence of NaN, input/output, data copy between similar types, save/restart are uniform operations that cross-cut all modules. However, in many present models, they need to be hard-coded for each component, resulting in low code reusability and heavy redundancy.

ii The program does not use metadata, or information that describes the data objects, such as the object's position in the data hierarchy, methods, subfields, and their data types, etc. Metadata allows the program to answer the question "*who am I?*": where it is; what it is; what it has; and what it can do. Without using metadata, a program cannot enumerate its subfields to traverse the data hierarchy; nor can it refer to alternative methods.

iii There are strong inter-module dependencies. Module and libraries updates necessitate program-wide refactoring. This dependence is

* Corresponding author.
E-mail address: cshen@engr.psu.edu (C. Shen).

https://doi.org/10.1016/j.cageo.2017.09.014
Received 23 February 2016; Received in revised form 18 November 2016; Accepted 26 September 2017
Available online 28 September 2017

especially troublesome for a highly collaborative project like Community Earth System Model (CESM).

The purpose of this paper is to promote reflection to Geoscientific domain scientists. We show that small investment in a simple, Fortran-native reflection solution can generate a considerable return on code reusability, efficiency, and clarity. Here our examples include four well-published models: CLM (Lawrence et al., 2011; Oleson et al., 2013), PAWS (Shen and Phanikumar, 2010), CHOMBO (Adams et al., 2015), and CONCEPTS (Langendoen and Alonso, 2008; Langendoen and Simon, 2008).

2. Native-language-emulated reflection as an efficient solution

Just as the ability to reflect on oneself immensely reshapes our cognitive abilities, reflection allows the program to possess a level of intelligence (Smith, 1982). Across various disciplines, reflection has provided elegant solutions (Forman and Forman, 2004) to above-mentioned issues, drastically reduced code size, and improved code readability. The cost reduction in maintenance is also not to be under-estimated. While long-adopted in computer science, reflection is not widely adopted in GMs, evidenced by highly redundant hard-coding, some examples for which are provided in later sections.

Philosophically, reflection is the ability of the program to examine its own properties and modifying its structure and behavior at runtime (Forman and Forman, 2004). Reflection can be implemented by interrogating and modifying a first-class metaObject, any object that creates, describes, or manipulates, objects including itself. For example, in Java, the base class, *Class*, is able to be queried for its class, methods, and metadata of its members. Similar approach powers reflection in Python and MATLAB. This type of reflection is commonly termed *structural* reflection. In contrast, *behavioral* reflection involves the inspection of stack or assembly information and modification of a program's own code (Jacques and Demers, 1996). As our objective is to assist Geoscientific models with coding efficiency with minimum effort, we limit our scope to *structural* reflection. A signature of reflection-oriented code is that string input arguments replace hard-coded field access or method invocation. A reflective program receives the request, resolves it during runtime, and chooses the appropriate behavior. A key ingredient to reflection is metadata, with which the program can automatically list all the content and properties of subfields and methods, traverse hierarchical tree and reason if the request can be met. Table 1 compares the behavior of reflective and non-reflective codes regarding handling several different situations for an object A.

Why is the adoption of reflection in GMs so slow, given that reflection-embracing, Java-, C#-based EMFs (references provided previously), which have been available for 2 decades, are likely to improve coding efficiency? The main reasons can be cost and habits. Any reflective solution for GMs must face practical constraints: GMs are predominantly developed in compiled languages focusing on performance, e.g., Fortran and C/C++, which do not have the reflection-supporting metaObjects at the language level. As data management only provides auxiliary services, many scientists desire simple tools to provide the functionality, with as little learning curve as possible, and language-interoperability is often shunned due to various practical reasons. While EMFs have powerful and flexible features, specialized EMFs may not apply to many legacy GMs (David et al., 2013), especially high-performance ones. However, to further promote reflection, there should also be alternative, native-language solutions that deliver solid return-on-investment in terms of code length, clarity, and more importantly reusability. A feather-weight package fully encompassed in a module in Fortran or a class in C++ may much better promote reflection among domain scientists. A successful solution also needs to support message passing interface (MPI) and parallel I/O. Dependencies on external libraries should also be minimized.

Table 1
Comparison of reflective and non-reflective field access (pseudo code). [a]Here we demonstrate retrieval by fieldname. Index-based retrieval is faster. Just an example e, all of the illustrated subroutines can also be recursive to work on all subfields because there is a function to check the type of the subfield, which can be sub-structure.

Task described in normal language	Non-reflective	Reflective (getPtr means "get pointer")
a Access a field with the name 'f1'	A%f1 ! Hard-coded	call getPtr(A, 'f1', localPointer)
b Invoke a method *mtd*	Call mtd(…)	call getPtr(A, 'mtd', localPointer)

c Set all values of all subfields and their children to 0	Hard-code all subfields, which can be hundreds of lines	DO I = 1,nSubField
		Field = A%FieldNames(I) ![a]
		call getPtr (A, Field, localPointer)
		localPointer = 0
		ENDDO

d Copy between same named fields between different object types	A%f1 = B%f1
	! Hard-coded for many fields
	! for C++, this can become complex as there needs to be access methods for each member.
		DO I = 1,nSubField
		Field = A%FieldNames(I) ![a]
		IF ((isReal(A,Field).AND. isReal(B, Field).AND. (dimsOf(A,Field).EQ. dimsOf(B, Field)) THEN
		call getPtr(A, Field, localPointer1) ![b]
		call getPtr(B, Field, localPointer2)
		localPointer1 = localPointer2
		ENDIF
		ENDDO

[a] In the code, we achieve field retrieval by index, which is faster and easier.
[b] In the actual code, there are overloaded operators to allow simpler syntax.

3. The generic, dynamically-linked metaObject (*gd*)

3.1. Implementation of gd

The primary task of data management of GMs is to store, manipulate and access numerical variables of large sizes and various dimensions. With compiled languages, reflection must be emulated using a base metaObjects to store these elemental data types and to provide metadata. The basic structure of the base metaObject (Fortran version), *gd*, is provided in Fig. 1 with explanations (In the attached code: Library). This module was developed independently as a by-product of hydrologic modeling in PAWS + CLM (Shen et al., 2016, 2014, 2013), with applications in water storage and fluxes (Fang et al., 2016; Niu et al., 2014), channel-land interactions, biogeochemistry, carbon/nitrogen states, transport (Niu and Phanikumar, 2015) and scaling (Ji et al., 2015; Pau et al., 2016; Riley and Shen, 2014), and therefore manages a large amount of data. *gd* allows dynamic access, polymorphic (meaning uniform interface for different data types) addition and deletion of fields, procedures (subroutine or function) and sub-structures, permitting recursive (depth-first) access with a dictionary. It can provide counting of the total number of fields and number of fields with repetitive names, check if a field exists (*isempty*), list fieldnames, and return the type, dimension, and sizes of the data. While accessing fields, it will check for the validity (type and dimension) of the output argument and reports error with helpful diagnostic messages.

3.2. Metadata generation

Because *gd* is a generic tool, itself does not create metadata, which needs to be originated through some means. First, realizing it or not, for most Geoscientific models, the input files already contain metadata which is generated during pre-processing by more flexible tools. For example, for the Community Land Model, the input files are the .nc (NetCDF) files. For PAWS + CLM, the input file is a .mat or .nc file (Shen et al., 2014). For CHOMBO, the inputs are text files written in a certain protocol. For off-the-shelf models like MODFLOW (Markstrom et al., 2008), or SWAT (Neitsch et al., 2009), input files also describe all

```
MODULE gd_Mod
 PRIVATE

integer, parameter :: maxDim = 6
integer, parameter :: maxStrLen = 100

type p_t
  real*8, dimension(:), pointer :: p1  => null()
  real*8, dimension(:,:), pointer :: p2 => null()
  real*8, dimension(:,:,:), pointer :: p3 => null()
  real*8, dimension(:,:,:,:), pointer :: p4 => null()
  real*8, dimension(:,:,:,:,:), pointer :: p5 => null()
  real*8, dimension(:,:,:,:,:,:), pointer :: p6 => null()
  integer,dimension(:), pointer :: j1 => null()
  integer,dimension(:,:), pointer :: j2 => null()
  integer,dimension(:,:,:), pointer :: j3 => null()
  integer,dimension(:,:,:,:), pointer :: j4 => null()
  integer,dimension(:,:,:,:,:), pointer :: j5 => null()
  integer,dimension(:,:,:,:,:,:), pointer :: j6 => null()
  logical*1,dimension(:), pointer :: l1 => null()
  logical*1,dimension(:,:), pointer :: l2 => null()
  logical*1,dimension(:,:,:), pointer :: l3 => null()
  logical*1,dimension(:,:,:,:), pointer :: l4 => null()
  logical*1,dimension(:,:,:,:,:), pointer :: l5 => null()
  logical*1,dimension(:,:,:,:,:,:), pointer :: l6 => null()
  procedure(), nopass, pointer :: proc => null()
end type p_t

type gd ! general data type
  ! it comes with a strs array, an integer array, a pointers array, and an integer nf, the number of valid fields
  integer np                                    ! number of associated fields
  type(p_t),dimension(:),pointer::p  => null() ! for all pointers
  type(gd),dimension(:),pointer::g  => null() ! for sub-structures
  character(len=maxStrLen),dimension(:),pointer :: name => null()
                                              ! a name that traces from root to current object
  character(len=maxStrLen),dimension(:),pointer::f  => null() ! fieldnames
  type(gd),pointer:: parent=>null()           ! uplink, this link must be implemented when the hierarchy is created
  integer parent_fnum                         ! field number of this object in parent
  integer*8 :: fileLink=0                         ! an optional link to file handle, etc
end type gd
```

Fig. 1. A portion of the gd module to show members of the type *gd*. A member *p* (type *p_t*) supports REAL*8, INTEGER and LOGICAL variables up to 6 dimensions. Storing and retrieving data to and from p_t is supported by a uniform interface and polymorphism. *gd* can link to sub-structures (member *g*) and procedure pointers (member *proc*). The fieldnames are stored as a character array in *f*. Access to a field in *gd* is done by finding the index, *k*, of the corresponding field in *f* (multiple same-named fields is supported, requiring an additional index) for the requested field.

information needed by the model to carry out a simulation. The input files likely are already in self-descriptive format, e.g., HDF5,.mat, NetCDF, XML or raster files, which are almost free-of-charge sources of metadata. In the supplementary materials, we present subroutines interfacing MATLAB and NetCDF files to assemble the *gd* representation from input files (Code: buildMatTable). In the absence of self-descriptive input files, a text file containing the metadata can be provided by pre-processing utilities and then interpreted by *gd*. We also have templates for interfacing ASCII data. Because of the uniform way *gd* is represented, writing an interface for *gd* is much easier than conventional data format. Moreover, for models without strict enforcement of protocols in their input files, the best approach is to utilize the data types defined in their Fortran source code. Metadata can be generated from parsing the Fortran source code. We have attached a Matlab script to achieve this goal (Code: GeneralTypeScanner, which generates the linkage source code in ioUpdate).

3.3. Supporting legacy code with efficiency: a dual-management system (DMS)

Since most legacy codes use Fortran-native (or C-native) structures, enforcing a whole-model conversion into using *gd* is impractical. However, this is neither necessary nor recommended. We can adopt a DMS approach by creating a *gd* hierarchy for the data, and then establish links (pointer assignment) between *gd* and user-defined Fortran-native data structure (Fig. 2). Developers can call *gd* or native data representation for different purposes. Here, DMS does not refer to a specific library or subroutine, but instead a style of data management in which the *gd* representation and the user-defined Fortran native representation coexist and both provide access to the same data in memory. The linking codes (Code: autoLink) are automatically generated by a script (Code: GeneralTypeScanner), which scans the Fortran type declarations for the hierarchical information from user-defined derived types. During execution, the data is managed by both the *gd* representation and Fortran-native representation. We can use *gd* for cross-cutting, pandemic tasks (data I/O, data validity check, inter-module variable passing, batch operations, etc.) while using Fortran/C natives in computational tasks. Requiring few changes to the legacy code, this strategy helps to solve most imminent software issues while maintaining model design. The mapping step also has an additional significant advantage: it insulates the model codes from changes to external I/O libraries. We note that *gd* can also be used to manage memory efficiently. In the example (e) in Table 1, *gd* can recursively deallocate memory with just one single call.

We also make a distinction between *gd* and the method that uses macros to automatically write certain parts of the code according to a fixed format (here referred to as the templated-macros method). Although we use this technique in autoLink (Fig. 2), autoLink provides a generic, cross-project handling of creating DMS, which serves as a handle to accomplish both computational and cross-cutting tasks. The templated-macros must be modified from case to case depending on the specific tasks, and in various parts of the code where code-writing is needed, the procedure needs to be separately invoked. It interrupts the normal code-writing process. In our approach, the linking between Fortran-native structure and *gd* representation is a one-time, uniform operation that does not need to be repeated for different invocations. Also, the use of the autoLink feature is almost identical from application

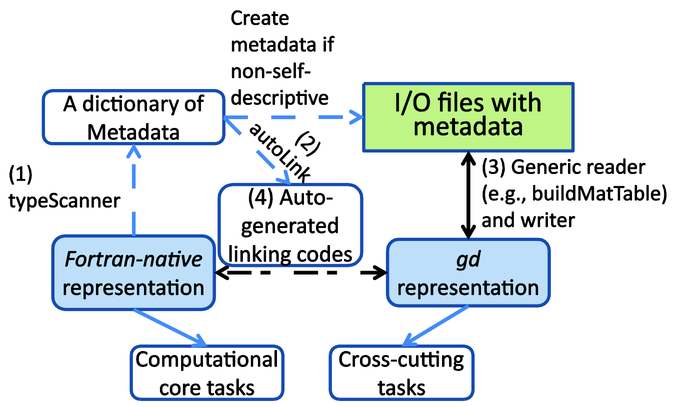

Create metadata if non-self-descriptive
A dictionary of Metadata
I/O files with metadata
(1) typeScanner
(2) autoLink
(3) Generic reader (e.g., buildMatTable) and writer
(4) Auto-generated linking codes
Fortran-native representation
gd representation
Computational core tasks
Cross-cutting tasks

Fig. 2. Workflow of setting up Dual data Management System (DMS) in Geoscientific models. DMS is a style of data management rather than any specific code. Dashed lines indicate optional steps. There are four steps: (1) (optional) use one Matlab command, read Fortran type declarations into a dictionary of metadata; (2) (optional) use several Matlab commands, produce Fortran linking codes that bridges *gd* and Fortran-native representation; (3) use one Fortran statement, read input files and build *gd* representation of data; and (4) use one Fortran call to the auto-generated linking codes to build the linkage. Optionally, if metadata is not available in the input files used by a certain GM, it can be generated using the dictionary of metadata previously created. The memory storage is shared, and the *gd* representation merely stores pointers to data. The *gd* representation can then used for input/output and cross-cutting tasks, while the computational core of the code using Fortran-native language can remain unchanged. Updating I/O library requires only slight modification to the generic reader/writer. Altogether, steps (1–4) requires 4 function/subroutine calls. A real example is provided in the attached code, Example: FortranBridge. *Autolink* relies on same variable names in input files and Fortran variables. In practice, some programmers habitually use different variable names in the files from those used in the model. We have developed an interface to accommodate patterned variable name transformations.

to application, reducing modelers' effort in adapting to different applications. The experiences are entirely different.

3.4. *Using reflection: applications of gd*

Below we show examples of using the feather-weight *gd* library to improve coding efficiency greatly. Table 2 summarizes the examples in plain text. All applications below are real-world examples taken from well-known models.

In a typical application, certain programs read data from a file into memory. The hierarchy in *gd* is simultaneously established, and the pointers in *gd* are assigned to the memory that holds the data (example code: buildMatTable). In this case, *gd* only provides access to the data but many compilers will not allow *gd* to deallocate the pointer in use. In some other uses, e.g., use *gd* to store globally accessible temporary storage, memory can be allocated. In this case, *gd* can also deallocate the memory space that itself created.

3.4.1. *MATLAB-like data self-assembly and storage*

MATLAB (or Octave) is widely used in engineering and science due to its flexibility, code development efficiency and moreover, programmatic access to various data types (e.g., Geographic Information System, GIS, files). It also contains API interfaces to Fortran/C, which are, however, rather challenging to use correctly due to many subtleties. We developed a library, *matTable*, based on *gd* and the MATLAB external interfaces API to load MATLAB/Octave data into Fortran and C++. In upgrading the PAWS model, the reflective solution is astonishingly easy compared to the original API (Fig. 3, Example: fortranBridge). Using *gd*, *matTable* dynamically creates a tree structure corresponding to the data in MATLAB and fills in the metadata, with just one command. As data hierarchy evolves, the reflection-based solution needs no change in the code, while the hard-coded version will cost much more debugging time.

Table 2
Summary of examples in this paper illustrating the use of *gd*. In the examples below, using fortranBridge, batchOper, checkNan codes in another project requires no change of the code. To set up ioUpdate for another project, simply replace one subroutine with a newer one.

Examples	Task in plain language
1. fortranBridge	"*automatically read the data hierarchy stored in a self-descriptive file, without knowledge of its content a priori, establish a corresponding data hierarchy in memory, and be ready to modify the same hierarchy at runtime and write the hierarchy to a different file as needed.*"
2. ioUpdate	"*following the buildMatTable example, finish a migration of I/O to (a). NetCDF file format; and (b). Parallel NetCDF format by only making changes to one subroutine (~150 lines of code in the subroutine; < 20 lines of changes)*"
3. batchOper (gdCopy/ gdSetVal)	a "*(optional: recursively) copy between two derived data types, regardless of their declared data types, while the copy is limited to the same-named, same-dimensioned REAL fields.*" b "*(optional: recursively) set all REAL (or INTEGER/LOGICAL) fields to a certain value, without knowledge of data structure a priori.*"
4. checkNaN	"*without using hardcoding, for every member of a derived data type, the members of members and so forth, if it is a REAL field, check whether NaN exists, and, if so, print out the full name of variables where NaN occurs and their locations in the hierarchy.*"
5. massBalance (Envisioned)	"*A set of generic mass balance checker based on a dictionary of mass balance terms.*"
6.Fortran_C generic coupler (Envisioned)	"*A uniform, generic data (pointer) passing interface between FORTRAN and C to remove the need for case-specific interoperability codes.*"

3.4.2. *Batch operations*

Without reflection, batch operations such as uniformly setting values to 0 and copying between objects, need to be hard-coded, leading to significant redundancy and waste of modeler's effort. Two examples with two different models (Fig. 5) show that this is a prevalent and hazardous issue. Again, similar examples can be found with many other models written in Fortran or C. Reflection and DMS permit recursive traversal of data hierarchy and batch-apply of operations. (Code Example: gdCopy, gdSetVal.

3.4.3. *Separation of concerns*

Separation of concerns (Elrad et al., 2001) using reflection can encapsulate scientists from software infrastructure. In current CLM (and many other codes), I/O are "entangled" with computational routines, meaning they are implemented everywhere in code, and an update of the I/O library resulted in system-wide refactoring (Fig. 4). Using a reflection-based solution shortened the code dramatically. Moreover, by just upgrading the generic I/O subroutine, we can effortlessly transition from one file type to another (.mat to NetCDF), or from serial I/O to a parallel I/O program (Parallel NetCDF) without model-wide refactoring, as the model code is insulated from I/O library updates (Code Examples: ioUpdate, which contains buildMatTable, buildNcTable and buildPncTable).

Because *gd* is generically written, parallel programs can utilize *gd* in the same way as serial programs. Regarding the process of upgrading serial programs to parallel programs, we have only demonstrated upgrading serial I/O to parallel I/O using *gd*. However, there are many other ways *gd* can be used to support the upgrading of serial programs to parallel programs. For example, as a serial program is upgraded to parallel, it may require the exchange of data between different cores. With *gd*, the user can prescribe a list of fields that need to exchanged; then a uniform "*exchange*" command can be issued for the *gd* representation. The user will then avoid the need to write separate exchange commands for different fields, just like other examples describe above. The main advantage of this approach is that, again, the "*exchange*" command is highly reused and written for *gd* representation. We only need to maintain the performance and integrity of a single program during upgrades.

(a)

```
subroutine bridgeGlobal(file_name)
use matRW
use Vdata
implicit none

character(len=length_of_character) ::  file_name
integer*4 status

mwPointer :: file_in,pGlobal,pm
mwIndex :: index = 1
mwIndex :: i
mwSize :: ndim
mwPointer dims(3)
! mw- and mx- are Matlab-Fortran API

file_in = matOpen(trim(file_name), 'r')
if (file_in .eq. 0) then
  call display( 'Can''t open '// trim(file_name))
  stop
end if
pGlobal = matGetVariable(file_in, 'g')
! a mwPointer to variable 'g' in the mat file

pm = mxGetField(pGlobal, index, 'DM')
! above line get us a pointer to g.DM
call bridgeDM(pm, index, 'g')
! above line bridges sub-object g.DM

pm = mxGetField(pGlobal, index, 'VDZ')
call bridgeVDZ(pm,1)

pm = mxGetField(pGlobal, index, 'Soil')
call bridgeSoils(pm)

pm = mxGetField(pGlobal, index, 'GW')
ndim = mxGetNumberOfDimensions(pm)
CALL mxCopyPtrToPtrArray(mxGetDimensions(pm),dims,ndim)
allocate(g%GW(dims(2)))
! there can be multiple GW. i.e., g.GW(i) i>=1
do i = 1, dims(2)
  call bridgeGW(pm, i)
end do
! and so on for many fields. omitted here...
```

```
SUBROUTINE bridgeGW(GWM,index)
use Vdata
use matRW
IMPLICIT none
mwIndex index
mwPointer :: nG(2), m(2),nn(3)
mwpointer :: GWM,MAPP
mwpointer :: h,EB,ES,E,D,K,Wm,W2,ST,T,DR,DLDR
mwpointer :: hNEW,DZ,Seep,Qperc,WTL
mwpointer :: A1B,A2,A3,A4,A5,pm,dd,dt,tt
mwpointer :: topH,topK,topD,botH,botK,botD,topBound
mwpointer :: STO,NLC,STATE
EB = mxGetPr(mxGetField(GWM,index,'EB'))
MAPP = mxGetPr(mxGetField(GWM,index,'MAPP'))
ES = mxGetPr(mxGetField(GWM,index,'ES'))
! ... and so on for many fields. Omitted here ......

CALL linkGW(index,m(1),m(2),nn,%val(dd),%val(h),%val(EB),%val(ES)   &
&     ,%val(E),%val(D),%val(K),%val(Wm),%val(W2),%val(ST),%val(T)   &
&     ,%val(hNew),%val(DZ),%val(Seep) ,%val(DLDR),%val(DR)          &
&     ,%val(dt),%val(Qperc),%val(A1B),%val(A2),%val(A3)             &
&     ,%val(A4),%val(A5),%val(tt),%val(MAPP),%val(STO),             &
&     %val(NLC),%val(botK),%val(botD),%val(STATE),%val(topBound),nn)

END SUBROUTINE bridgeGW

SUBROUTINE linkGW(J,ny,nx,nn,dd,h,EB,ES,E,D,K,W1,W2,ST,T, &
&     hNew,DZ,Seep,DLDR,DR,dt,Qperc,A1B,A2,A3,A4,A5,tt,   &
&     MAPP,STO,NLC,botK,botD,STATE,topBound)
USE Vdata
mwIndex J
mwPointer nc
mwPointer :: ny,nx,nn(3)
REAL*8,DIMENSION(ny,nx),Target::EB,ES,E,D,K,W1,   &
&       W2,ST,T,hNew,DZ,Seep,DLDR,DR,Qperc,A1B,    &
&       A2,A3,A4,A5,h,STO,botK,botD,topBound
real*8,dimension(ny,nx),target:: MAPP
REAL*8,Target:: dd(2),dt,tt,STATE
real*8,dimension(nn(1),nn(2),nn(3)),target:: NLC
g%GW(J)%ny = ny
g%GW(J)%nx = nx
g%GW(J)%EB => EB
g%GW(J)%ES => ES
g%GW(J)%E => E
! ... and so on for many fields. Omitted here ...
```

(b)

```
call buildMatTable(file_name)
! one line reads the file into gd_base
mat => gd_base .G. file_name
! mat is of type(gd), pointer. The above gets us the
! root-level pointer. .G. is an overloaded operator
! to obtain sub-structure

! The following is extracted from codes
! generated automatically by a script, autoLink
gdG => gd_rt .G. 'g'
! gdG is of type(gd), pointer.
! This gets us variable 'g' in the mat file

DO i=1, countRepNum(gdG,'GW')
    call getptr(gdG,'GW',gd_ptr,i)  ! This gets us g.GW
    call getptr(gd_ptr,'DZ',g%GW(i)%DZ)
    call getptr(gd_ptr,'EB',g%GW(i)%EB)
    ! ... and so on for other fields. Omitted here ...
```

(c)

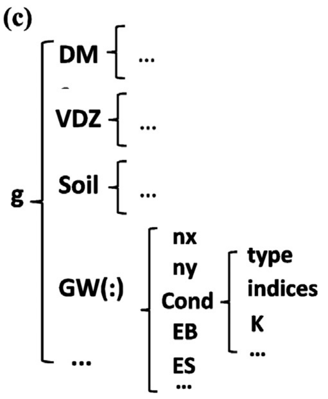


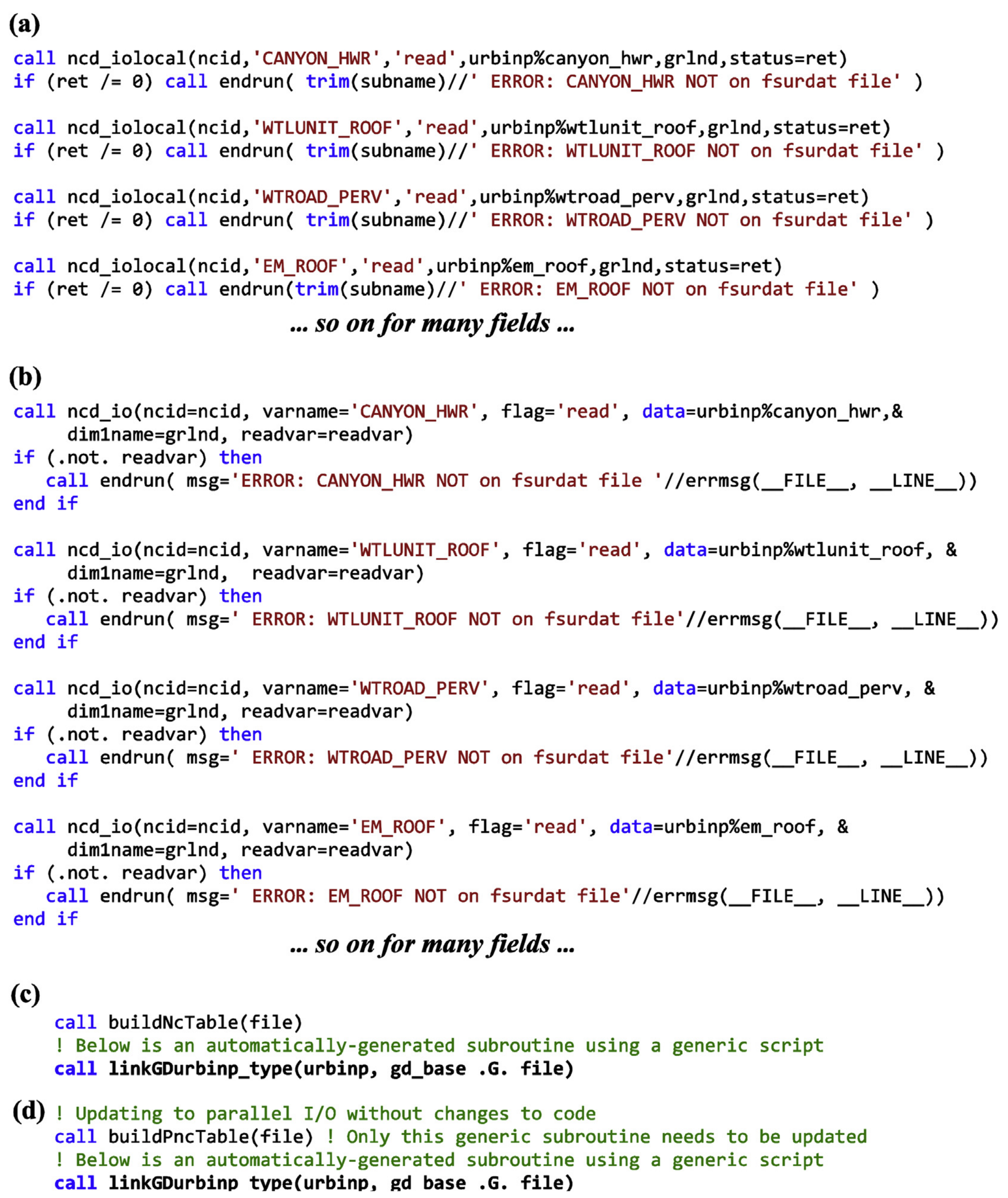
Fig. 3. (a) A counter example, from an earlier version of PAWS, using MATLAB-Fortran API to read a. mat file in a non-reflective manner (see fortranBridge\counterExample). The data is hierarchical. 'g' and 'r' are root-level variables and 'g%DM', 'g%GW(i)', are 2nd level structures. Both a "bridge-" and a "link-" step are needed. Fields and data structures were hard-coded. These redundant codes amounted to 3000 lines; (b) The reflective solution: with a simple statement (*buildMatTable*), the data is read into a central *gd* metaObject, *gd_base*, which can be accessed with a simple syntax. We also used a script to automatically generate the linking codes. Inline comments are provided for paper clarifications; (c) Parts of the data structure from PAWS + CLM that is demonstrated in this figure. GW() it is a structures array. EB and ES are both 2D numerical matrices.

3.4.4. *Interfacing modules using reflection*

Reflection can greatly facilitate language-interoperation. For example, CHOMBO (Adams et al., 2015; Trebotich et al., 2014) is a high-performance, scalable C++ numerical package supporting block-structured Adaptive Mesh Refinement (AMR). Although many users desire AMR, adapting CHOMBO for earth system modeling is challenging due to the barrier between C++ and Fortran. Although an interface exists (Cornford et al., 2013), hard-coding will be needed to handle the thousands of variables passing through the interface. We envision if a *gd* structure is created on each side of the interface, then data passing and methods invocation can be managed with ease by controlling only the *generic* interface between the *gd* structures in C++ and Fortran (Fig. 6).

3.4.5. *Generic computational tasks*

There are also other computational tasks that can be generically tackled by *gd*. For example, instead of writing case-specific codes, we can write a generic subroutine, recursively using *gd*, to check for the existence of NaN's and report their locations in the hierarchy (Example: checkNaN). These functions should improve the efficiency of model development. For another example, GMs often need to code up mass balance checkers. Since there is a uniform format for this task (change in state variables = integration of fluxes over time), it should be possible to code a generic mass balance checker, with a dictionary of state variables and corresponding fluxes, to define a mass balance check.



(a)

```
call ncd_iolocal(ncid,'CANYON_HWR','read',urbinp%canyon_hwr,grlnd,status=ret)
if (ret /= 0) call endrun( trim(subname)//' ERROR: CANYON_HWR NOT on fsurdat file' )

call ncd_iolocal(ncid,'WTLUNIT_ROOF','read',urbinp%wtlunit_roof,grlnd,status=ret)
if (ret /= 0) call endrun( trim(subname)//' ERROR: WTLUNIT_ROOF NOT on fsurdat file' )

call ncd_iolocal(ncid,'WTROAD_PERV','read',urbinp%wtroad_perv,grlnd,status=ret)
if (ret /= 0) call endrun( trim(subname)//' ERROR: WTROAD_PERV NOT on fsurdat file' )

call ncd_iolocal(ncid,'EM_ROOF','read',urbinp%em_roof,grlnd,status=ret)
if (ret /= 0) call endrun(trim(subname)//' ERROR: EM_ROOF NOT on fsurdat file' )
```

... so on for many fields ...

(b)

```
call ncd_io(ncid=ncid, varname='CANYON_HWR', flag='read', data=urbinp%canyon_hwr,&
     dim1name=grlnd, readvar=readvar)
if (.not. readvar) then
   call endrun( msg='ERROR: CANYON_HWR NOT on fsurdat file '//errmsg(__FILE__, __LINE__))
end if

call ncd_io(ncid=ncid, varname='WTLUNIT_ROOF', flag='read', data=urbinp%wtlunit_roof, &
     dim1name=grlnd,  readvar=readvar)
if (.not. readvar) then
   call endrun( msg=' ERROR: WTLUNIT_ROOF NOT on fsurdat file'//errmsg(__FILE__, __LINE__))
end if

call ncd_io(ncid=ncid, varname='WTROAD_PERV', flag='read', data=urbinp%wtroad_perv, &
     dim1name=grlnd, readvar=readvar)
if (.not. readvar) then
   call endrun( msg=' ERROR: WTROAD_PERV NOT on fsurdat file'//errmsg(__FILE__, __LINE__))
end if

call ncd_io(ncid=ncid, varname='EM_ROOF', flag='read', data=urbinp%em_roof, &
     dim1name=grlnd, readvar=readvar)
if (.not. readvar) then
   call endrun( msg=' ERROR: EM_ROOF NOT on fsurdat file'//errmsg(__FILE__, __LINE__))
end if
```

... so on for many fields ...

(c)

```
call buildNcTable(file)
! Below is an automatically-generated subroutine using a generic script
call linkGDurbinp_type(urbinp, gd_base .G. file)
```

(d)

```
! Updating to parallel I/O without changes to code
call buildPncTable(file) ! Only this generic subroutine needs to be updated
! Below is an automatically-generated subroutine using a generic script
call linkGDurbinp_type(urbinp, gd_base .G. file)
```

Fig. 4. (a) reading NetCDF files for urban inputs in CLM4.0, using the serial version of NetCDF. This kind of I/O statements occupy many thousand lines throughout the model; (b) when upgrading to CLM4.5, the procedure was changed to parallel I/O, necessitating changes to all calling statements; (c) a reflection-based solution: similar to the matTable, we automatically read NetCDF files for access, and establishing the DMS using automatically-generated subroutines; (d) migrating to parallel I/O using *gd* does not incur any changes to model codes, enabling *separation of concerns*. We only need to update the generic procedures in *buildNcTable* to invoke parallel NetCDF methods.

3.5. Limitations and setup time

The storage overhead per *gd* item is estimated to be 3kBytes (10 Mb for 3500 items). Thus, it is important not to use *gd* to store data on discretized element level. For example, for a solution vector *dat(ny,nx)* in 2D, we should not store each element *dat(i,j)* using a separate *gd* object. Instead, we should use *gd* only to store a 2D pointer to *dat*, which can then be used in language-native ways. The number of items in the data structure, no matter how complex it is, is small for modern computers. In our buildMatTable test case, linking 6267 fields in the .mat file using *gd* takes only 0.008s. Therefore, as long as we do not operate on the element-by-element level, *i.e.*, call the linking code on each of ny*nx elements, but works on the numeric-array level, the overhead of fields linking is almost negligible.

Although reflection is a very powerful tool, it has its limitations and potential pitfalls. We pay a small performance penalty when looking up items, although this is trivial in most cases as indicated above. In the context of GMs, the biggest potential danger is the modification of values without using specific model data members. Thus, it is recommended to use reflection only through a small number of well-understood, carefully-controlled functions, whose use is clearly summarized for other users to inspect. Also, as error checking is deferred to runtime, if there is insufficient error handling and reporting with the reflective code, it can cause confusion for end users (*gd* does provide error messages reporting type/dimension mismatches, which turns out to be great help during our debugging). Therefore, we advised against widespread use of reflection for entry-level modelers, and also programs with stable and simple data structures. On the flip side, *gd* can be used to implement validity checks, e.g., check for NaNs, out-of-bounds errors, etc, so the overall advantage should outweigh the disadvantage.

(a)

```
SUBROUTINE CopyCrossSection(cross_section_ptr, cross_section)
  ! Argument variables
  TYPE(cross_section_ptr_t), INTENT(IN) :: cross_section_ptr
  TYPE(cross_section_t), INTENT(OUT) :: cross_section
  ! Local variables
  INTEGER(INT_KIND) :: i, ncoords

  ! Copy cross section name

  cross_section%name = cross_section_ptr%name

  ! Copy cross section station

  cross_section%station = cross_section_ptr%station

  ! Copy indices of bank top and toe

  cross_section%left_bank_top = cross_section_ptr%left_bank_top
  cross_section%left_bank_toe = cross_section_ptr%left_bank_toe
  cross_section%right_bank_top = cross_section_ptr%right_bank_top
  cross_section%right_bank_toe = cross_section_ptr%right_bank_toe
```

... so on for many fields ...

(b)

```
if ( masterproc ) write (iulog,*) 'Setting initial data

do c = bounds%begc,bounds%endc

  ! To detect first time-step
  this%fsat_bef_col       (c) = spval
  this%annsum_counter_col (c) = spval
  this%totcolch4_col      (c) = spval

  ! To detect first year
  this%annavg_somhr_col(c)   = spval
  this%annavg_finrw_col(c)   = spval

  ! To detect file input
  this%qflx_surf_lag_col  (c)   = spval
  this%finundated_lag_col (c)   = spval
  this%layer_sat_lag_col  (c,:) = spval
  this%conc_ch4_sat_col   (c,:) = spval
  this%conc_ch4_unsat_col (c,:) = spval
  this%conc_o2_sat_col    (c,:) = spval
  this%conc_o2_unsat_col  (c,:) = spval
  this%o2stress_sat_col   (c,:) = spval
```

... so on for ~170 fields ...

(c)

```
call gdPartialCopy(gd_cross_section_ptr,
       gd_cross_section,.false.)
```

(d)

```
call gdSetVal(gdMain,'R', spval, .false.)
```

Fig. 5. Comparison of non-reflective and reflective programming for batch operations. (a) Batch copy in the bank erosion and sediment transport model CONCEPTS. There are many places such copy occurs in this model; (b) Batch set value in CLM4.5. This operation occupies 170 lines for the methane (ch4) module alone, and thousands of lines throughout the model. (c) Achieve the batch-copy using a generic copy subroutine, which examines common fields between two *gd* structure and copy between them; (d) Achieve the batch-set-value by a single call to a subroutine that sets Real*8 values (as specified by 'R') to a uniform value, regardless of dimension. This code can recursively traverse the hierarchical tree if the last argument is set to *. true.*.

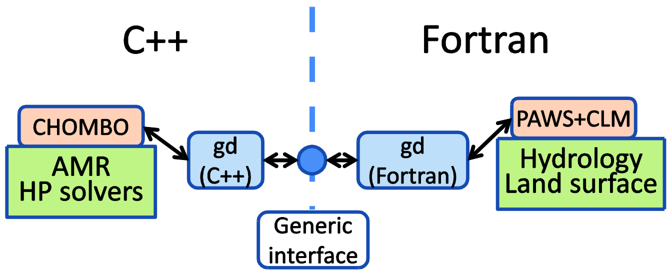


Fig. 6. Envisioned reflective inter-language coupling mechanisms: a generic way to couple CHOMBO (C++) and PAWS + CLM (Fortran) using *gd*. Only a generic interface between *gd* (C++) and *gd* (Fortran) needs to be implemented. Once coded, this generic interface can be used in other Fortran-C coupled applications.

The steps involved in setting up the DMS (Fig. 2) are also much cheaper to learn, implement and maintain because they are generically written and thus are plug-and-play. In our experience, we were able to set up DMS for PAWS in one day (this step previously took ~10 days to accomplish, and required much maintenance), and set up three cases of I/O update (Example: ioUpdate) all within one day. We also employed *gd* to build new input data format for CONCEPTS (Langendoen and Simon, 2008) in a week. Writing a code to read the content of a Matlab file takes 2 min.

4. On the definition of reflection

Some readers may argue that reflection must be achieved at the fundamental programming language level: the exploration of data structure must be supported without explicit data population by the programmer, via certain Application Programming Interface (API). If we restrict the definition of reflection so narrowly, it is then not possible to be accomplished with either Fortran or C. Since even many Fortran 2008 standards are not fully implemented in many compilers, it is difficult to envision compiler-level support for reflection will be usable in a matter of years. Our approach, on the other hand, is fully covered under Fortran 90 standards, so it is portable across a wide range of systems.

Moreover, detailed investigation of literature suggests that practice of reflection was never restricted to a language-default feature. As described above (Section 2), in Java-based work (Chiba, 2006, 2000), a language-supported first-class metaObject serves as the fundamental basis for reflection in Java. Our approach is philosophically similar to this one, only that this base-level object (*gd*) needs to be implemented by us. With companion software to extract metadata from source code (Section 3.2), this suite of functionality is suited to carry out many tasks in the scope of structural reflection with sufficient generality.

The concepts of dynamic control of program structure and data are indeed alien to many Fortran/C programmers and especially Geoscientific modelers due to education reasons. Many modelers were educated to think of predefined procedures and perhaps object-oriented programming. Many educational resources will list Fortran as a procedural language, while *gd* demonstrates that the power offered by currently supported language standards is still under-utilized, and substantial potential can be realized by unconventional thinking.

5. Conclusions

Reflection can significantly reduce the amount of error-prone hard-coding. With a layer of generic data structure, the reflective code can be generically-written, detached from the case-specific environment. Such a design leads to a higher level of code reusability. Incorporating a reflective data structure in Geoscientific models may also separate concerns and improve flexibility. Contrary to previous thought, reflection can be implemented without additional dependencies in native-Fortran language. The example library *gd* is a standard Fortran module (and a C++ class) that are easily incorporated on any platform. The DMS proposed in the system can create a separate layer of data control that enables efficient tackling of cross-cutting concerns as well as insulating computational codes from I/O operations. It also enables generic handling of inter-model coupling. There are potential issues on data encapsulation, which needs to be managed carefully.

6. Software availability

The Fortran version of the module *gd*, the *matTable* library, the Fortran type scanner (*GeneralTypeScanner*) and *autoLink* can be freely downloaded from the corresponding author's website (http://water.engr.psu.edu/shen/software_page.html) under the title “Integrated generic data package”.

Acknowledgments

This work was accomplished while conducting research supported by Office of Biological and Environmental Research of the US Department of Energy under Contract No. DE-SC0010620. We thank the anonymous reviewers for their constructive comments.

Appendix A. Supplementary data

Supplementary data related to this article can be found at https://doi.org/10.1016/j.cageo.2017.09.014.

References

Adams, M., Colella, P., Graves, D.T., Johnson, J.N., Keen, N.D., Ligocki, T.J., Martin, D.F., McCorquodale, P.W., Modiano, D., Schwartz, P.O., Sternberg, T.D., van Straalen, B., 2015. Chombo Software Package for AMR Applications - Design Document [WWW Document]. Lawrence Berkeley Natl. Lab. Tech. Rep. LBNL-6616E. https://commons.lbl.gov/display/chombo/Chombo+-+Software+for+Adaptive+Solutions+of+Partial+Differential+Equations. (Accessed 4 August 2015).

Argent, R.M., Perraud, J.-M., Rahman, J.M., Grayson, R.B., Podger, G.M., 2009. A new approach to water quality modelling and environmental decision support systems. Environ. Model. Softw. 24, 809–818. https://doi.org/10.1016/j.envsoft.2008.12.010.

Chiba, S., 2000. Load-time Structural Reflection in Java, Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/3-540-45102-1.

Chiba, S., 2006. Program transformation with reflection and aspect-oriented programming. In: Lämmel, R., Saraiva, J., Visser, J. (Eds.), Generative and Transformational Techniques in Software Engineering, Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 65–94. https://doi.org/10.1007/11877028.

Collins, W.D., Craig, A.P., Truesdale, J.E., Di Vittorio, A.V., Jones, A.D., Bond-Lamberty, B., Calvin, K.V., Edmonds, J.A., Kim, S.H., Thomson, A.M., Patel, P., Zhou, Y., Mao, J., Shi, X., Thornton, P.E., Chini, L.P., Hurtt, G.C., 2015. The integrated Earth system model version 1: formulation and functionality. Geosci. Model Dev. 8, 2203–2219. https://doi.org/10.5194/gmd-8-2203-2015.

Cornford, S.L., Martin, D.F., Graves, D.T., Ranken, D.F., Le Brocq, A.M., Gladstone, R.M., Payne, A.J., Ng, E.G., Lipscomb, W.H., 2013. Adaptive mesh, finite volume modeling of marine ice sheets. J. Comput. Phys. 232, 529–549. https://doi.org/10.1016/j.jcp.2012.08.037.

David, O., Ascough, J.C., Lloyd, W., Green, T.R., Rojas, K.W., Leavesley, G.H., Ahuja, L.R., 2013. A software engineering perspective on environmental modeling framework design: the object modeling system. Environ. Model. Softw. 39, 201–213. https://doi.org/10.1016/j.envsoft.2012.03.006.

Elrad, T., Filman, R.E., Bader, A., 2001. Aspect-oriented programming: introduction. Commun. ACM 44, 29–32. https://doi.org/10.1145/383845.383853.

Fang, K., Shen, C., Fisher, J.B., Niu, J., 2016. Improving Budyko curve-based estimates of long-term water partitioning using hydrologic signatures from GRACE. Water Resour. Res. 52, 5537–5554. https://doi.org/10.1002/2016WR018748.

Forman, I.R., Forman, N., 2004. Java Reflection in Action.

Holzworth, D.P., Huth, N.I., de Voil, P.G., 2010. Simplifying environmental model reuse. Environ. Model. Softw. 25, 269–275. https://doi.org/10.1016/j.envsoft.2008.10.018.

Holzworth, D.P., Huth, N.I., deVoil, P.G., Zurcher, E.J., Herrmann, N.I., McLean, G., Chenu, K., van Oosterom, E.J., Snow, V., Murphy, C., Moore, A.D., Brown, H., Whish, J.P.M., Verrall, S., Fainges, J., Bell, L.W., Peake, A.S., Poulton, P.L., Hochman, Z., Thorburn, P.J., Gaydon, D.S., Dalgliesh, N.P., Rodriguez, D., Cox, H., Chapman, S., Doherty, A., Teixeira, E., Sharp, J., Cichota, R., Vogeler, I., Li, F.Y., Wang, E., Hammer, G.L., Robertson, M.J., Dimes, J.P., Whitbread, A.M., Hunt, J., van Rees, H., McClelland, T., Carberry, P.S., Hargreaves, J.N.G., MacLeod, N., McDonald, C., Harsdorf, J., Wedgwood, S., Keating, B.A., 2014. APSIM – evolution towards a new generation of agricultural systems simulation. Environ. Model. Softw. 62, 327–350. https://doi.org/10.1016/j.envsoft.2014.07.009.

Jacques, Demers, F., 1996. A tutorial on behavioral reflection and its implementation. In: Reflect. 96’ Conf. Proc.

Janssen, S., Athanasiadis, I.N., Bezlepkina, I., Knapen, R., Li, H., Domínguez, I.P., Rizzoli, A.E., van Ittersum, M.K., 2011. Linking models for assessing agricultural land use change. Comput. Electron. Agric. 76, 148–160. https://doi.org/10.1016/j.compag.2010.10.011.

Ji, X., Shen, C., Riley, W.J., 2015. Temporal evolution of soil moisture statistical fractal and controls by soil texture and regional groundwater flow. Adv. Water Resour. 86, 155–169. https://doi.org/10.1016/j.advwatres.2015.09.027.

Kelly, R.A., Jakeman, A.J., Barreteau, O., Borsuk, M.E., ElSawah, S., Hamilton, S.H., Henriksen, H.J., Kuikka, S., Maier, H.R., Rizzoli, A.E., van Delden, H., Voinov, A.A., 2013. Selecting among five common modelling approaches for integrated environmental assessment and management. Environ. Model. Softw. 47, 159–181. https://doi.org/10.1016/j.envsoft.2013.05.005.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J., 1997. Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (Eds.), ECOOP’97 — Object-oriented Programming, Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 220–242. https://doi.org/10.1007/BFb0053371.

Langendoen, E.J., Alonso, C.V., 2008. Modeling the evolution of incised streams: I. Model formulation and validation of flow and streambed evolution components. J. Hydraul. Eng. 134, 749–762.

Langendoen, E.J., Simon, A., 2008. Modeling the evolution of incised streams. II: streambank erosion. J. Hydraul. Eng. 134 (7), 905–915. https://doi.org/10.1061/(ASCE)0733-9429(2008)134:7(905).

Laniak, G.F., Olchin, G., Goodall, J., Voinov, A., Hill, M., Glynn, P., Whelan, G., Geller, G., Quinn, N., Blind, M., Peckham, S., Reaney, S., Gaber, N., Kennedy, R., Hughes, A., 2013. Integrated environmental modeling: a vision and roadmap for the future. Environ. Model. Softw. 39, 3–23. https://doi.org/10.1016/j.envsoft.2012.09.006.

Lawrence, D.M., Oleson, K.W., Flanner, M.G., Thornton, P.E., Swenson, S.C., Lawrence, P.J., Zeng, X., Yang, Z.-L., Levis, S., Sakaguchi, K., Bonan, G.B., Slater, A.G., 2011. Parameterization improvements and functional and structural advances in version 4 of the community land model. J. Adv. Model. Earth Syst. 3 https://doi.org/10.1029/2011MS000045.

Markstrom, S.L., Niswonger, R.G., Regan, R.S., Prudic, D.E., Barlow, P.M., 2008. Gsflow—coupled ground-water and surface-water flow model based on the integration of the precipitation-runoff modeling system (PRMS) and the modular ground-water flow model (MODFLOW-2005) [WWW Document]. U.S. Geol. Surv. Chapter 1 sect. D, ground-water/surface-water B. 6, model. Tech http://pubs.usgs.gov/tm/tm6d1/.

Neitsch, S.L., Arnold, J.G., Kiniry, J.R., Williams, J.R., 2009. Soil and water Assessment Tool Theoretical Documentation Version 2009. US Department of Agriculture - Agricultural Research Service.

Niu, G.-Y., Yang, Z.-L., Mitchell, K.E., Chen, F., Ek, M.B., Barlage, M., Kumar, A., Manning, K., Niyogi, D., Rosero, E., Tewari, M., Xia, Y., 2011. The community Noah land surface model with multiparameterization options (Noah-MP): 1. Model description and evaluation with local-scale measurements. J. Geophys. Res. 116, D12109. https://doi.org/10.1029/2010JD015139.

Niu, J., Phanikumar, M.S., 2015. Modeling watershed-scale solute transport using an integrated, process-based hydrologic model with applications to cacterial fate and transport. J. Hydrol. 529, 35–48. https://doi.org/10.1016/j.jhydrol.2015.07.013.

Niu, J., Shen, C., Li, S.-G., Phanikumar, M.S., 2014. Quantifying storage changes in regional Great Lakes watersheds using a coupled subsurface-land surface process model and GRACE, MODIS products. Water Resour. Res. 50, 7359–7377. https://doi.org/10.1002/2014WR015589.

Oleson, K.W., Lawrence, D.M., Bonan, G.B., Drewniak, B., Huang, M., Koven, C.D., Levis, S., Li, F., Riley, W.J., Subin, Z.M., Swenson, S.C., Thornton, P.E., Bozbiyik, A., Fisher, R., Heald, C.L., Luzek, E., Lamarque, J.-F., Lawrence, P.J., Leung, L.R., Lipscomb, W.H., Muszala, S., Ricciuto, D.M., Sacks, W., Sun, Y., Tang, J., Yang, Z.-L., 2013. Technical Description of Version 4.5 of the Community Land Model (CLM). NCAR Technical Note, NCAR/TN - 503+STR, UCAR. Earth System Laboratory, Boulder, Colorado.

Pau,, G.S.H., Shen, C., Riley, W.J., Liu, Y., 2016. Accurate and efficient prediction of fine-resolution hydrologic and carbon dynamic simulations from coarse-resolution models. Water Resour. Res. 52 (2), 791–812.

Rahman, J.M., Seaton, S.P., Cuddy, S.M., 2004. Making frameworks more useable: using model introspection and metadata to develop model processing tools. Environ. Model. Softw. 19, 275–284. https://doi.org/10.1016/S1364-8152(03)00153-1.

Riley, W.J., Shen, C., 2014. Characterizing coarse-resolution watershed soil moisture heterogeneity using fine-scale simulations. Hydrol. Earth Syst. Sci. 18, 2463–2483. https://doi.org/10.5194/hess-18-2463-2014.

Rizzoli, A.E., Davis, J.R., Abel, D.J., 1998. Model and data integration and re-use in environmental decision support systems. Decis. Support Syst. 24, 127–144. https://doi.org/10.1016/S0167-9236(98)00068-2.

Shen, C., Niu, J., Fang, K., 2014. Quantifying the effects of data integration algorithms on the outcomes of a subsurface–land surface processes model. Environ. Model. Softw. 59, 146–161. https://doi.org/10.1016/j.envsoft.2014.05.006.

Shen, C., Niu, J., Phanikumar, M.S., 2013. Evaluating controls on coupled hydrologic and vegetation dynamics in a humid continental climate watershed using a subsurface - land surface processes model. Water Resour. Res. 49, 2552–2572. https://doi.org/10.1002/wrcr.20189.

Shen, C., Phanikumar, M.S., 2010. A process-based, distributed hydrologic model based on a large-scale method for surface–subsurface coupling. Adv. Water Resour. 33, 1524–1541. https://doi.org/10.1016/j.advwatres.2010.09.002.

Shen, C., Riley, W.J., Smithgall, K.M., Melack, J.M., Fang, K., 2016. The fan of influence of streams and channel feedbacks to simulated land surface water and carbon dynamics. Water Resour. Res. 52, 880–902. https://doi.org/10.1002/2015WR018086.

Smith, B.C., 1982. Procedural Reflection in Programming Languages. Ph.D. Dissertation. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Trebotich, D., Adams, M.F., Molins, S., Steefel, C.I., Shen, C., 2014. High-resolution simulation of pore-scale reactive transport processes associated with carbon sequestration. Comput. Sci. Eng. 16, 22–31. https://doi.org/10.1109/MCSE.2014.77.