Computers and Geosciences 110 (2018) 32–40

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
0098-3004/

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