Computers & Geosciences 53 (2013) 3–12

Contents lists available at SciVerse ScienceDirect

Computers & Geosciences

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

A component-based approach to integrated modeling in the geosciences: The design of CSDMS

Scott D. Peckham [a], Eric W.H. Hutton [a,*], Boyana Norris [b]

[a] CSDMS, University of Colorado, 1560 30th Street, UCB 450, Boulder, CO 80309, USA
[b] Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439, USA

ARTICLE INFO

Article history:
Received 20 April 2011
Received in revised form
29 March 2012
Accepted 2 April 2012
Available online 20 April 2012

Keywords:
Component software
CCA
CSDMS
Modeling
Code generation

ABSTRACT

Development of scientific modeling software increasingly requires the coupling of multiple, independently developed models. Component-based software engineering enables the integration of plug-and-play components, but significant additional challenges must be addressed in any specific domain in order to produce a usable development and simulation environment that also encourages contributions and adoption by entire communities. In this paper we describe the challenges in creating a coupling environment for Earth-surface process modeling and the innovative approach that we have developed to address them within the Community Surface Dynamics Modeling System.

1. Introduction

The Community Surface Dynamics Modeling System (CSDMS) project (CSDMS, 2011; Syvitski et al., 2011) is an NSF-funded, international effort to develop a suite of modular numerical models able to simulate a wide variety of Earth-surface processes, on time scales ranging from individual events to many millions of years. CSDMS maintains a large, searchable inventory of contributed models and promotes the sharing, reuse, and integration of open-source modeling software. It has adopted a component-based software development model and has created a suite of tools that make the creation of *plug-and-play* components from stand-alone models as automated and effortless as possible. Models or process modules that have been converted to component form are much more flexible and can be rapidly assembled into new configurations to solve a wider variety of scientific problems. The ease with which one component can be replaced by another also facilitates experimenting with different approaches to providing a particular type of functionality. CSDMS also has a mandate from the NSF to provide a migration pathway for surface dynamics modelers toward high-performance computing (HPC) and provides a 720-core supercomputer for use by its members. In addition, CSDMS provides educational infrastructure related to physically based modeling.

This paper presents key issues and design criteria for a component-based, integrated modeling system and then describes the design choices adopted by CSDMS to address them. CSDMS was not developed in isolation: it builds on and extends proven, open-source technology. The CSDMS project also maintains close collaborations with several other integrated modeling projects (such as ESMF, 2011; OpenMI, 2011; and OMS, David et al., 2002) and has evaluated different approaches in pursuit of those that are optimal (Peckham, 2008). As with any design problem, myriad factors must be considered in determining what is optimal, including how various choices affect users and developers. Other key factors are performance, ease of maintenance, ease of use, flexibility, portability, stability, encapsulation, and software longevity.

This paper is organized as follows. Section 2 provides background on component-based programming and issues that are critically important to component design, such as the distinction between interface and implementation, the role of granularity and what it means for two components to be interchangeable. Section 3 lists the key design criteria that drove the development of CSDMS. Section 4 describes key supporting tools such as Babel, Bocca and Ccaffeine. Sections 5 provides an overview of the Basic Model Interface (BMI) and Component Model Interface (CMI) that we developed to meet the specific needs of component-based modeling. Section 6 explains how service components assist with coupling by reconciling various differences between models. Section 7 discusses the problem of converting existing models to plug-and-play components. Sections 8 and 9 describe the CSDMS Modeling Tool (CMT) that makes it easy for users to build

* Corresponding author.
E-mail addresses: Scott.Peckham@colorado.edu (S.D. Peckham), Eric.Hutton@colorado.edu (E.W.H. Hutton), norris@mcs.anl.gov (B. Norris).

0098-3004/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cageo.2012.04.002

new models from components graphically and components are provided with uniform graphical user interfaces and HTML help pages. Finally, Section 10 summarizes the current status of the CSDMS project and remaining challenges.

2. Background

2.1. What is component-based programming?

Component-based programming brings the advantages of "plug and play" technology into the realm of software. When buying a new peripheral for a computer, such as a mouse or printer, one wishes to simply plug it into the correct kind of port (e.g., a USB, serial, or parallel port) and have it work, right out of the box. For this situation to be possible, however, a published standard is needed, against which the makers of peripheral devices can design their products. For example, most computers have universal serial bus (USB) ports, and the USB standard is well documented. A computer's USB port can always be expected to provide certain capabilities, such as the ability to transmit data at a particular speed and the ability to provide a 5 V supply of power with a maximum current of 500 mA. The benefit of this standardization is that one can buy a new device, plug it into a computer's USB port, and start using it. Software "plug-ins" work in a similar manner, relying on interfaces (ports) that have well-documented structure or capabilities. In software, as in hardware, the term *component* refers to a unit that delivers a certain functionality and that can be "plugged in."

Component programming builds on the fundamental concepts of object-oriented programming, with the main difference being the introduction or presence of a runtime *framework*. Components are generally implemented as classes in an object-oriented language and are essentially "black boxes" that encapsulate some useful bit of functionality. The purpose of a framework is to provide an environment in which components can be linked together to form applications. The framework also provides *services* to all components, such as the linking mechanism itself.

2.2. Interface vs. implementation

Many people hear the word *interface* and immediately think of the interface between a person and a computer program, such as a graphical user interface (GUI). Within the present context of component programming, however, the focus is on interfaces between components. The word interface then has a specific meaning, essentially the same as in the Java programming language. An interface is a user-defined entity or type, similar to an abstract class. It does not have any data fields; instead, it is a named set of methods or member functions, each defined completely with regard to argument types and return types but without any actual implementation.

Interfaces are key to reusability or "plug and play." Once an interface has been defined, one can ask: Does this component have interface A? To answer this question, we simply look at the methods (or member functions) that the component has with regard to their names, argument types, and return types. If a component has a given interface, then it is said to *expose*, or *implement*, that interface; in other words, it contains an *implementation* for each of those methods. The component may also have other methods beyond those that constitute a particular interface. Thus, a single component can expose multiple, different interfaces, allowing it to be used in a greater variety of settings. An analogy exists in computer hardware, where a computer or peripheral may have a number of different ports (e.g., USB, serial, parallel, and ethernet) to enable it to communicate with a wider variety of other components.

The distinction between *interface* and *implementation* is an important theme in computer science. The word pair *declaration* and *definition* is used in a similar way. A function (or class) declaration tells what the function does (and how to interact with or use it) but not how it works. To see how the function actually works, we must look at how it has been defined or implemented. C and C++ programmers are familiar with this idea, which is similar to declaring variables, functions, classes, and other data types in a header file with the file name extension .h or .hpp, and then defining their implementations in a separate file with extension .c or .cpp.

Most of the gadgets that we use every day (from iPods to cars) are like this. We must understand their interfaces in order to use them (and interfaces are often standardized across vendors), but we often have no idea what is happening inside or how they actually work, which may be quite complex.

2.3. Granularity

While components may represent any level of granularity, from a simple function to a complete hydrologic model, the optimum level for modeling appears to be that of a particular physical process, such as infiltration, evaporation, or snowmelt. Smaller components are tedious to work with and larger ones provide fewer options for coupling. This is also the level at which researchers typically want to swap out one method of modeling a process for another. A simpler method of parameterizing a process may apply only to simplified special cases or may be used because there is insufficient input data to drive a more complex method. A different numerical method may solve the same equations with greater accuracy, stability, or efficiency; it may or may not use multiple processors. Even the same method of modeling a given process may exhibit improved performance when coded in a different programming language. Physical processes often act within a domain that shares a physically important boundary with other domains (e.g., coastline and ocean-atmosphere). The fluxes between these domains are often of key interest and help to determine granularity.

Some models are written in such a way that decomposing them into separate process components is not appropriate because of some special aspect of the model's design or because decomposition would result in a loss of performance (e.g., speed, accuracy, or stability). For example, *multiphysics models*—such as Penn State Integrated Hydrologic Model (PIHM Qu and Duffy, 2007)—represent many physical processes as one large, coupled set of ODEs that are then solved as a matrix problem on a supercomputer.

2.4. Interchangeability and autoconnection

A key goal of component-based modeling is to create a collection of components that can be coupled together to create new and useful composite models. This goal can be achieved by providing every component with the same interface. A secondary goal, however, is for the coupling process to be as automatic as possible, that is, to require as little input as possible from users. In order to achieve this goal, components must somehow be grouped into categories according to the functionality they provide. This grouping must be readily apparent to both a user and the framework (or system), so that it is clear whether a given pair of components are *interchangeable*. But what should it mean for two components to be interchangeable? Do they really need to use identical input variables and provide identical output variables?

To clarify this issue, consider the physical process of infiltration within a hydrologic model. An infiltration component must provide the infiltration rate at the surface, because it represents a

loss term in the overall hydrologic budget. If the domain of the infiltration component is restricted to the unsaturated zone, above the water table, then it may also need to provide a vertical flow rate at the water table boundary. Thus, any infiltration component must provide fluxes at the (top and bottom) boundaries of its domain. To do so, it needs variables such as flow depth and rainfall rate that are outside its domain and computed by another component. Hydrologists use a variety of different methods and approximations to compute surface infiltration rate. The Richards 3D method, for example, is a more rigorous approach that tracks four state variables throughout the domain. The Green–Ampt method, on the other hand, makes a number of simplifying assumptions, computes a smaller set of state variables, and does not resolve the vertical flow dynamics to the same level of detail (i.e., piston flow, sharp wetting front). As a result, the Richards 3D and Green–Ampt infiltration components use a different set of input variables and provide a different set of output variables. Nevertheless, *they both provide certain, key outputs* such as surface infiltration rate and can therefore be used "interchangeably" in a hydrologic model.

Autoconnection of components is important from a user's point of view. Components typically require many input variables and produce many output variables. Users quickly become frustrated when they need to manually create all of these pairings or connections, especially when using more than just two or three components at a time. CSDMS currently employs an approach to autoconnection that involves providing interfaces (called ports) with different names to reflect their intended use (or interchangeability), even though they use the same interface internally.

2.5. Key benefits of component-based programming

- Components can be written in different languages and still communicate (via language interoperability).
- Components are precompiled units that can be replaced, added to, or deleted from an application at runtime via dynamic linking.
- Components can be moved to a remote location (different address spaces) without recompiling other parts of the application (via RMI/RPC support).
- Components can have multiple different interfaces.
- Components can be "stateful"; component data is retained between method calls over its lifetime.
- Components can be customized at runtime with configuration parameters.
- Components facilitate code reuse and rapid comparison of different implementations.
- Components facilitate efficient cooperation between groups, each doing what it does best.

3. Design criteria for CSDMS

Here we list some of the key design criteria that drove the development of CSDMS. Subsequent sections describe how each of these criteria are met.

- Support for *multiple operating systems* (especially Linux, Mac OS X, and Windows).
- *Language interoperability* to support code contributions written in procedural languages (e.g., C or Fortran) as well as object-oriented languages (e.g., Java, C++, and Python).
- Support for both *structured and unstructured grids*, requiring a spatial regridding tool.
- *Semantic mediation* and *unit conversion* to accommodate differences between model variable names and units.
- *Platform-independent graphical user interfaces* (*GUIs*) *and visualization capabilities* where appropriate.
- Use of well-established, open-source *software standards* whenever possible (e.g., CCA, SIDL, OGC, MPI, NetCDF, OpenDAP, and XML).
- Use of *open-source tools* that are mature and have well-established communities, avoiding dependencies on proprietary software (e.g., Windows, C#, and Matlab) whenever possible.
- Support for both *serial and parallel computation* (multiprocessor, via MPI standard).
- *Interoperability with other coupling frameworks*: Because code reuse is a fundamental tenet of component-based modeling, the effort required to use a component in another framework should be kept to a minimum.
- *Robustness and ease of maintenance*: The modeling system will clearly have many software dependencies, and this software infrastructure must be updated on a regular basis.
- Use of *HPC tools and libraries*: If the modeling system runs on HPC architectures, it should strive to use parallel tools and models (e.g., VisIt, PETSc, and the ESMF regridding tool).
- *Familiarity*: Model developers and contributors should not be required to make major changes to their existing development processes and habits.

With regard to the last criterion, developers should not be required to convert to another programming language or make pervasive changes to their code (e.g., use specified data structures, libraries, or classes). They should be able to retain "ownership" of the code and make continual improvements to it. Someone should be able to componentize future, improved versions with minimal additional effort. The developer will likely want to continue to use the code outside the framework. However, some degree of code refactoring (e.g., breaking code into functions or adding a few new functions) and ensuring that the code compiles with an open-source compiler are considered reasonable requirements.

4. Tools for component-based programming

Many different tools and frameworks for component-based modeling were evaluated (Peckham, 2008) by CSDMS against the design criteria given in the previous section. CSDMS selected the Common Component Architecture (CCA) and its associated tools as being best able to meet those criteria. Some of the deciding factors were support for language interoperability, parallel computation and multiple operating systems. Here we briefly review CCA and three foundational tools that underpin CSDMS: Babel, Ccaffeine, and Bocca.

4.1. The Common Component Architecture

The Common Component Architecture (Armstrong et al., 1999) is a *component architecture standard* adopted by federal agencies (largely the Department of Energy and its national laboratories) and academic researchers to allow software components to be integrated for enhanced functionality on high-performance computing systems. The CCA Forum is a grassroots organization that started in 1998 to promote component technology standards and code reuse for HPC. CCA defines standards necessary for interoperation of components developed in different frameworks. Components that adhere to these standards can be ported with relative ease to another CCA-compliant framework. While various other component architecture standards exist in the commercial sector (e.g., CORBA, COM, .Net, and JavaBeans), CCA was created to fulfill the needs of scientific, high-performance, open-source computing that are unmet by these other standards. For example, scientific software requires full support for complex numbers,

dynamically dimensioned multidimensional arrays, Fortran (and other languages), and multiple processor systems. Armstrong et al. (1999) explain the motivation for creating CCA by discussing the pros and cons of other component-based frameworks with regard to scientific, high-performance computing. Many of the papers in our cited references have been written by CCA Forum members and are helpful for learning more about the CCA. The CCA Forum has also prepared a set of tutorials called "A Hands-On Guide to the Common Component Architecture" (CCA Forum, 2010).

A variety of different frameworks, such as Ccaffeine (Allan et al., 2010), CCAT/XCAT (Krishnan and Gannon, 2004), SciRUN (Germain et al., 2002), and Decaf (Kumfert, 2003), adhere to the CCA standard. Ccaffeine was designed to support both serial and parallel computing, and SciRUN and XCAT were designed to support distributed computing. Decaf (Kumfert, 2003) was designed by the developers of Babel primarily as a means of studying the technical aspects of the CCA standard itself. The important point is that each of these frameworks adheres to the same standard, and this facilitates reuse of CCA components in different computational settings.

4.2. Language interoperability with Babel

One feature that often distinguishes components from ordinary subroutines, software modules, or classes is that they can communicate with other components that may be written in a different programming language. This capability is often provided within a component framework and is called *language interoperability*.

Babel (Lawrence Livermore National Laboratory, 2012a; Dahlgren et al., 2007) is an open-source, language interoperability tool (consisting of a compiler and runtime) that automatically generates the "glue code" necessary for components written in different computer languages to communicate. As illustrated in Fig. 1, Babel currently supports C, C++, Fortran (77, 90, 95, and 2003), Java, and Python. Babel is much more than a "least common denominator" solution; it even enables passing of variables with data types that may not normally be supported by the target language (e.g., objects and complex numbers). Babel was designed to support *scientific*, *high-performance* computing and is used as an integral part of CCA-compliant frameworks. It won an R&D 100 design award in 2006 for "the world's most rapid communication among many programming languages in a single application."

In order to create the glue code needed for two components written in different programming languages to exchange information, Babel needs to know only about the interfaces of the two components. Babel therefore accepts as input a description of an interface in either of two "language-neutral" forms, XML (eXtensible Markup Language) or SIDL (Scientific Interface Definition Language). The SIDL language (somewhat similar to CORBA's IDL) was developed specifically for the Babel project. Its sole purpose is to provide a concise description of a scientific software component interface. This interface description includes complete information about a component's interface, such as the data types of all arguments and return values for each of the component's methods (or member functions). SIDL has a complete set of fundamental data types to support scientific computing, from Booleans to double-precision complex numbers. It also supports more sophisticated data types such as enumerations, strings, objects, structs, and dynamic multi-dimensional arrays. A description of SIDL syntax and grammar can be found in "Appendix B: SIDL Grammar" in the Babel User's Guide (Dahlgren et al., 2007). Complete details on how to represent a SIDL interface in XML are given in "Appendix C: Extensible Markup Language (XML)" of the same guide.

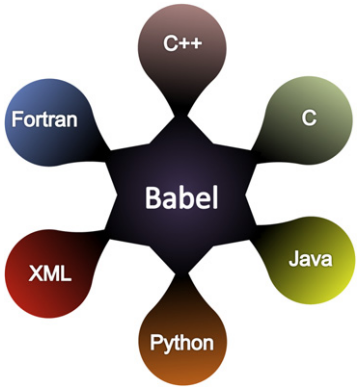

Fig. 1. Language interoperability provided by Babel.

4.3. The Ccaffeine framework

Ccaffeine (Allan et al., 2010) is the most widely used CCA framework, providing the runtime environment for sequential or parallel component applications. Using Ccaffeine, component-based applications can run on diverse platforms, including laptops, desktops, clusters, and leadership-class supercomputers. Ccaffeine provides some rudimentary MPI communicator services, although individual components are responsible for managing parallelism internally (e.g., communicating with other distributed components). A CCA framework provides *services*, which include component instantiation and destruction, connecting and disconnecting of ports, handling of input parameters, and control of MPI communicators. Ccaffeine was designed primarily to support single-component multiple-data (SCMD) programming, although it can support multiple-component multiple-data (MCMD) applications that implement more dynamic management of parallel resources.

A typical CCA component's execution consists of the following steps:

- The framework loads the dynamic library for the component. Static linking options are also available.
- The component is instantiated. The framework calls the `setServices` method on the component, passing a handle to itself as an argument.
- User-specified connections to other components' ports are established by the framework.
- If the component provides a `gov.cca.ports.Go` port (similar to a "main" subroutine), its `go()` method can be invoked to initiate computation.
- Connections can be made and broken throughout the life of the component.
- All component ports are disconnected, and the framework calls `releaseServices` prior to calling the component's destructor.

The handle to the framework services object, which all CCA components obtain shortly after instantiation, can be used to access various framework services throughout the component's execution. This represents the main difference between a class and a component: a component dynamically accesses another component's functionality through dynamically connecting ports (requiring the presence of a framework), whereas classes in object-oriented languages call methods directly on instances of other classes.

4.4. Component development with Bocca

Bocca (Allan et al., 2008) is a tool in the CCA tool chain that was designed to help users create, edit, and manage a set of SIDL-based

entities, including the CCA components and ports associated with a particular project. Once a set of CCA-compliant components and ports has been prepared, a CCA-compliant framework such as Ccaffeine can be used to link the components together to create applications or composite models.

Bocca was developed to address usability concerns and reduce the development effort required to implement multilanguage component applications. Bocca was designed specifically to free users from mundane, time-consuming, low-level tasks so they can focus on the scientific aspects of their applications. It can be viewed as a development environment tool that allows application developers to perform rapid component prototyping while maintaining robust software engineering practices suitable to HPC environments. Bocca provides project management and a comprehensive build environment for creating and managing applications composed of CCA components. Bocca operates in a language-agnostic way by automatically invoking the Babel compiler. A set of Bocca commands required to create a component project can be saved as a shell script, so the project can be rapidly rebuilt, if necessary. Various aspects of an existing component project can also be modified by typing Bocca commands interactively at a Unix command prompt.

While Bocca automatically generates dynamic libraries, a separate tool can be used to create *stand-alone executables* for projects by automatically bundling all required libraries on a given platform. Examples of using Bocca are available in the set of tutorials called "A Hands-On Guide to the Common Component Architecture," written by the CCA Forum (2010) members.

5. Interfaces for component-based modeling

While the tools in the CCA tool chain are powerful and general, they do not provide a ready-made interface for linking geoscience models (or any domain-specific models). In CCA terminology, a *port* is essentially a synonym for interface, and a distinction is made between ports that a given component uses (*uses ports*), and those that it provides (*provides ports*) to other components. This approach provides a means of two-way information exchange between components, unlike dataflow-based approaches that support only one-way links between components.

Each scientific modeling community that wishes to make use of the CCA tools must design or select the component interface best suited to the kinds of models they wish to link together. This is a big job that involves both social and technical issues and requires a significant time investment. We now describe two component interfaces that we have developed for CSDMS.

5.1. The Basic Model Interface

Most Earth-science models initialize a set of state variables (often as 1D, 2D, or 3D arrays) and then execute of series of time steps that advance the variables forward in time according to physical laws (e.g., mass conservation) or some other set of rules. Hence, the underlying source code tends to follow a standard pattern that consists of three main parts. The first part comprises all source code prior to the start of the time loop and serves to set up or *initialize* the model. The second part comprises all source code *within* the time loop and is where the main functionality of the model is implemented to update state variables with each time step. The third part consists of all source code after the end of the time loop and serves to tear down, or *finalize*, the model. Note that root-finding and relaxation algorithms follow a similar pattern even if the iterations do not represent timestepping.

CSDMS and all of the other model coupling projects that we are aware of (including ESMF, Hill et al., 2004; OMS, David et al., 2002; and OpenMI, Gregersen et al., 2007) have developed a component interface that provides initialize, update, and finalize functions. This approach provides a caller with fine-grained control that is essential for model coupling and allows the caller to bypass the model's own time loop.

The streamlined approach developed by CSDMS for converting models into plug-and-play components utilizes two new interfaces in a two-level wrapping process. Recall that a component *interface* is simply a named set of functions (called methods) that have been defined completely in terms of their names, arguments, and return values. The first-level interface, called the Basic Model Interface (BMI), is the only one that model contributors are required to implement. The purpose of the BMI is to provide model metadata that allows the model to "fit into" the second-level wrapper, called the CSDMS Component Model Interface (CMI), discussed in the next section. By design, the BMI uses only simple data types and is easy to implement in all languages supported by CSDMS. However, it provides all of the information that is needed at the CMI level for plug-and-play modeling. In fact, our goal was to design BMI so that a BMI-enabled model could be easily ingested into any modeling framework.

Table 1 summarizes the key BMI functions. `BMI.initialize()` typically reads data from a configuration file, initializes variables, opens output files, and stores data in a "handle" for non-object-oriented languages. `BMI.update()` advances the state variables by one model time step or does nothing if they do not vary with time. `BMI.finalize()` typically frees resources and closes files. Several "getter" and "setter" functions, whose details differ between languages, allow the CMI wrapper to retrieve data from and load data into the model. The CMI wrapper must be able to retrieve anything it needs for model coupling, such as lists of input and output variable names; any variable's units, rank or data type; and a description of the model's grid or mesh. The BMI functions that must be provided to fully describe a model's computational grid (not shown) depend on the type of grid, e.g., uniform, orthogonal curvilinear, or unstructured.

5.2. The CSDMS Component Model Interface

A model that provides the BMI functions is ready to be converted to a CSDMS plug-and-play component using automated tools that provide it with the CSDMS Component Model Interface. Therefore, model contributors do not need to know anything about the CMI or technical CCA framework concepts such as ports. The functions in the CMI allow CSDMS components to communicate and share data with other CSDMS components even if they are written in a different language, use a different grid, use different units, or use a different timestepping scheme. A model wrapped as a CSDMS component becomes an object and can maintain its state variables between method calls, regardless of whether it was written in an object-oriented language.

A CSDMS component must communicate with (make function calls to) five entities: (1) the model that it wraps, (2) other CSDMS components, (3) itself, (4) CSDMS service components and (5) the CSDMS/CCA framework. See Fig. 3. Each entity provides interface functions for this purpose. BMI functions are used to communicate with the underlying, wrapped model. CMI functions are used to communicate with other CSDMS components (and itself). Service components each have their own interface and provide a rich set of useful services, such as spatial regridding, interpolation in time, unit conversion, and writing of output to a standard file format (e.g., NetCDF). Every CSDMS component has a `setServices()` function that the framework calls to instantiate the component and give it a handle for calling framework services (e.g., `services.getPort()`).

Table 1
Summary of BMI functions. (See csdms.colorado.edu for details.)

opaque **initialize** (*string* config_file)
Returns a "handle". Initialize variables, open files, etc.
void **update** (*double* dt)
Advance state variables by one time step (model's own, if $dt=-1$)
void **finalize** ()
Free resources, close files, reporting, etc.
void **run_model** (*string* config_file)
Do a complete model run. (Not called by CMI.)
array⟨*string*⟩ **get_input_var_names** ()
Return a list of model's input vars (CF convention standard names).
array⟨*string*⟩ **get_output_var_names** ()
Return a list of model's output vars (CF convention standard names).
string **get_attribute** (*string* att_name)
Return static model attributes such as: mesh_type (uniform, ugrid…), time_step_type (fixed, adaptive...), time_units, model_name, author_name, version, etc.
string **get_var_type** (*string* long_var_name)
Return variable's type, e.g. 'uint8', 'int16', 'int32', 'float32', 'float64'
string **get_var_units** (*string* long_var_name)
Return variable's units, e.g., 'meters' (Unidata UDUNITS standard)
int **get_var_rank** (*string* long_var_name)
Return variable's number of dimensions (0 for scalars)
string **get_var_name** (*string* long_var_name)
Return model's internal, short variable name
double **get_time_step** ()
Return model's current timestep
string **get_time_units** ()
Return model's time units, e.g., 'seconds', 'years'
double **get_start_time** ()
Return model's start time
double **get_current_time** ()
Return model's current time
double **get_end_time** ()
Return model's end time
double **get_0d_double** (*string* long_var_name)
Return a scalar of type double
array⟨*double*,1⟩ **get_1d_double** (*string* long_var_name)
Return a 1D array of type double
array⟨*double*,2⟩ **get_2d_double** (*string* long_var_name)
Return a 2D array of type double
array⟨*double*,1⟩ **get_2d_double_at_indices** (*string* long_var_name, *array*⟨*int*,1⟩ indices)
Return a 1D array of values at specified indices in 2D array
void **set_0d_double** (*string* long_var_name, *double* scalar)
Set a scalar of type double
void **set_1d_double** (*string* long_var_name, *array*⟨*double*,1⟩ array)
Set a 1D array of type double
void **set_2d_double** (*string* long_var_name, *array*⟨*double*,2⟩ array)
Set a 2D array of type double
void **set_2d_double_at_indices** (*string* long_var_name, *array*⟨*int*,1⟩ indices, *array*⟨*double*⟩ values)
Set values at indices in a 2D array of type double.
Notes:
(1) When implementing in a non-object oriented language, BMI functions will have a "handle" (as returned by BMI.initialize) as the first argument.
(2) long_var_name must be a CF Convention (Lawrence Livermore National Laboratory, 2012b) standard name.
(3) Units are specified using UDUNITS (Unidata, 2012).
(4) Models with 3D arrays provide BMI.get_3d_double(), etc.
(5) Models with 2D integer arrays provide BMI.get_2d_int(), etc.

Table 2
Summary of CMI functions.

bool **initialize** (*string* config_file)
bool **run_for** (*double* time_interval, *string* time_units)
void **finalize** ()
bool **run_model** (*string* config_file, *string* stop_rule, *array*⟨*double*,1⟩ stop_vars)
array⟨⟩ **get_values** (*string* long_var_name)
void **set_values** (*string* long_var_name, *array*⟨⟩ values)
string **get_status** ()
See BMI set_status
void **set_status** (*string* status)
string **get_mode** ()
Return either 'driver' or 'nondriver'
void **set_mode** (*string* mode)
array⟨*string*⟩ **get_input_var_names** ()
Return a list of model's input vars (CF convention standard names).
array⟨*string*⟩ **get_output_var_names** ()
Return a list of model's output vars (CF convention standard names).
string **get_var_units** (*string* long_var_name)
See BMI get_var_units
string **get_attribute** (*string* att_name)
Return static attributes of the component or the wrapped model. See BMI get_attribute
void **setServices** ()
Used by the CCA framework
void **releaseServices** ()
Used by the CCA framework

In CCA jargon, the "IMPL file" is where all the CMI functions are implemented. In our design, the CMI functions make calls only to the interfaces of the entities just discussed. Anything that is needed from the model is obtained through standardized BMI function calls. This means that essentially the same IMPL file can be used for all components written in a given language, such as Fortran.

By design, our process to convert a model with a BMI interface to a plug-and-play component with a CMI interface is *noninvasive*. Model contributors do not insert any calls in their (BMI-compliant) code to CSDMS utilities or services. The BMI functions do not call any external entity.

Babel is used during compilation to create language bindings, or "glue code" that allow the component to share data with other CSDMS components that may have been written in another language. In most cases, CSDMS components access each other's state variables through references (pass-by-reference) rather than copying data (pass-by-copy), as this approach results in higher performance. However, since Babel supports Remote Method Invocation (RMI), it is not necessary for CSDMS components to reside in the same address space and they could even be coupled across a network.

Table 2 summarizes the key CMI functions. `CMI.initialize()` must get and save the CCA ports it needs to call other components. It then calls `CMI.initialize()` on those components, if necessary, and finally calls `BMI.initialize()`. `CMI.run_for()` first calls `CMI.run_for()` on the components it needs data from and retrieves that data. It then makes as many calls to `BMI.update()` as necessary to span a time interval. Because of this fine-grained control, `CMI.run_for()` can then call service components to do other tasks after each update, such as write data to NetCDF files at a specified interval, advance a progress bar, or check a stopping rule. `CMI.finalize()` releases CCA ports and calls `BMI.finalize()`. `CMI.get_values()` takes a long variable name argument and makes calls to various BMI methods to retrieve the requested data. It then uses service components, if necessary, to convert units to those needed by the caller or perform spatial regridding to the caller's grid. Semantic mediation is achieved because the BMI and CMI methods always use the standard names of the CF Convention (Lawrence Livermore National Laboratory, 2012b). `CMI.set_values()` does unit conversion (using UDUNITS) and regridding, if necessary, before loading variables into the model's state using BMI methods. See csdms.colorado.edu for additional details.

6. Service components: tools that any component can use

It is helpful for certain low-level tools or utilities to be encapsulated in special components called *service components* that can be automatically instantiated by a CCA framework on

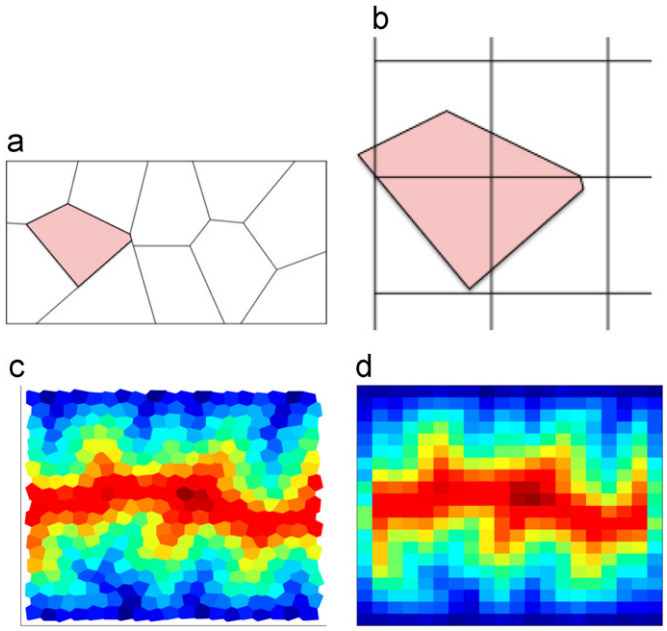

Fig. 2. Regridding example: (a) Voronoi cells; (b) intersecting raster and Voronoi cells; (c) Voronoi cells before regridding; and (d) after regridding to raster cells.

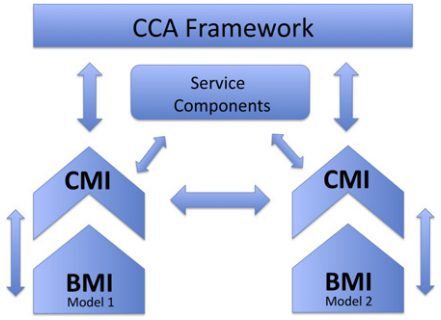

Fig. 3. Relationship between BMI, CMI, service components and framework.

startup. Unlike other components, which users may assemble graphically into larger applications, users do not interact with service components directly. However, CMI functions can call the methods of service components through *service ports*. The use of service components allows CSDMS to maintain code for a shared functionality in a single place and to make that functionality available to all components regardless of the implementation language. Any CCA component can be "promoted" to a service component by registering it as such using a CCA port called *gov.cca.ports.ServiceRegistry*.

CMI functions call service components for tasks like spatial regridding, temporal interpolation and unit conversion. An example regridding scenario is shown in Fig. 2. CSDMS uses regridding tools from both the ESMF and OpenMI projects. The ESMF regridder is implemented in C++ and uses multiple processors. It also supports sophisticated options such as mass-conservative interpolation.

7. Component wrapping issues

For a small or simple model, little effort may be needed to rewrite the model in a preferred language and with a particular interface. Rewriting large models, however, is both time-consuming and error prone. In addition, most large models are under continual development, and a rewritten version will not see the benefits of future improvements. Thus, for code reuse to be practical, we need a *language interoperability tool* (such as Babel), so that components do not need to be converted to a different language, and a *wrapping procedure* to provide existing code with a new calling interface. Wrapping applies to the "outside" (interface) of a component vs. its "inside" (implementation), so it tends to be a noninvasive and practical way to convert existing models into components.

7.1. Wrapping for object-oriented languages

Component-based programming is essentially object-oriented programming with the addition of a runtime framework. Thus, if a model has been written as a class with a BMI, it is straightforward to convert it to a component with a CMI. For object-oriented languages, all the model data is encapsulated in the model object, and the built-in use of namespaces makes compilation straightforward. The object-oriented languages supported by Babel are C++, Java, and Python.

7.2. Wrapping for procedural languages

Languages such as C or Fortran (up to 2003) do not provide object-oriented primitives for encapsulating data and functionality. Because component-based programming requires such encapsulation, the CCA provides a means to produce object-oriented software even in languages that do not support it directly. Specifically, all of the model's data and functions must be bundled and a suitable namespace created. A model contributor provides implementations for each of the BMI functions. In the case of Fortran, the model's variables are encapsulated in named modules that are accessed via "`use module_name`." The model is then compiled as a set of shared library functions using Babel.

When the model is wrapped with a CMI interface, the data and functions are made accessible to all the other CMI functions by using a "handle" that provides a reference to the model object. This handle is passed as the first argument to each of the interface functions so that they can operate on a particular instance of a model. For example, in C this handle could simply be a pointer to the object, and in Fortran the handle could be an index into a table of opaque objects in a system table. Model handles are allocated and deallaocated in the CMI initialize and finalize functions, respectively.

The creation of class or component wrappers also enables the careful definition of namespaces, thus reducing potential conflicts when integrating with other classes or components. In non-object-oriented languages, symbols (e.g., function names) are prefixed with the names of all enclosing packages and parent class. This approach greatly reduces the potential build-, link-, or runtime name conflicts that can result when multiple components define the same interfaces (e.g., the initialize, update, and finalize methods).

8. The CSDMS Modeling Tool

As explained in Section 4.3, Ccaffeine is a CCA-compliant framework for connecting components to create applications. From a user's point of view, Ccaffeine is a low-level tool that executes a sequence of commands in a Ccaffeine script. The commands in the Ccaffeine scripting language are fairly simple, but many people prefer using a GUI because it provides a natural, visual representation of the connected components as boxes with buttons connected by wires. It can also prevent common scripting errors and offer many other convenient features. The CCA Forum developed such a GUI, called Ccafe-GUI that presented components as boxes in a palette that can be moved into an arena (workspace) and connected by wires. It also allows component

configurations and settings to be saved in files and instantly reloaded later. As a lightweight and platform-independent tool written in Java, Ccafe-GUI can be installed and used on any computer with Java support to create a Ccaffeine script. This script can then be sent to a remote, possibly high-performance computer for execution.

While Ccafe-GUI was certainly useful, the CSDMS project realized that it could be improved and extended in numerous ways to make it more powerful and more user-friendly. In addition, these changes not only would serve the CSDMS community but could be shared back with the CCA community. The new version, called CMT (Component Modeling Tool), works with any CCA-compliant components, not just CSDMS components. CMT provides the following significant new features.

- Integration with a powerful visualization tool called VisIt (see below).
- New, "wireless" paradigm for connecting components (see below).
- A login dialog that prompts users for remote server login information.
- Job management tools for submitting jobs to a cluster and monitoring their progress.
- "Launch and go": launch a model run on a remote server and then shut down the GUI (the model continues running remotely).
- New file menu entry: "Import Example Configuration."
- A help menu with numerous help documents and links to websites.
- Ability to submit bug reports to CSDMS.
- Ability to do file transfers to and from a remote server.
- Help button in tabbed dialogs to launch component-specific HTML help.
- Support for droplists and mouse-over help in tabbed dialogs.
- Support for custom project lists (e.g., projects not yet ready for release).
- A separate "driver palette" above the component palette.
- Support for numerous user preferences, many relating to appearance.
- Extensive cross-platform testing and "bulletproofing."

CMT provides integrated visualization by using VisIt. VisIt (2011) is an open-source, interactive, parallel visualization and graphical analysis tool for viewing scientific data. It was developed by the U.S. Department of Energy Advanced Simulation and Computing Initiative to visualize and analyze the results of simulations ranging from kilobytes to terabytes. VisIt was designed so that users can install a client version on their PC that works together with a server version installed on a high-performance computer or cluster. The server version uses multiple processors to speed rendering of large data sets and then sends graphical output back to the client version. VisIt supports about five dozen file formats and provides a rich set of visualization features, including the ability to make movies from time-varying databases. CSDMS uses a service component to provide other components with the ability to write their output to NetCDF files that can be visualized with VisIt. Output can be 0D, 1D, 2D, or 3D data evolving in time, such as a time series (e.g., a hydrograph), a profile series (e.g., a soil moisture profile), a 2D grid stack (e.g., water depth), a 3D cube stack, or a scatter plot of XYZ triples.

Another innovative feature of CMT 1.6 is that it can toggle between the original, *wired* mode and a new *wireless* mode. CSDMS found that displaying connections between components as wires (i.e., red lines) did not scale well to configurations with several components and multiple ports. In wireless mode, a component dragged from the palette to the arena appears to broadcast what it can provide (i.e., CCA provides ports) to other components in the arena (using a concentric circle animation).

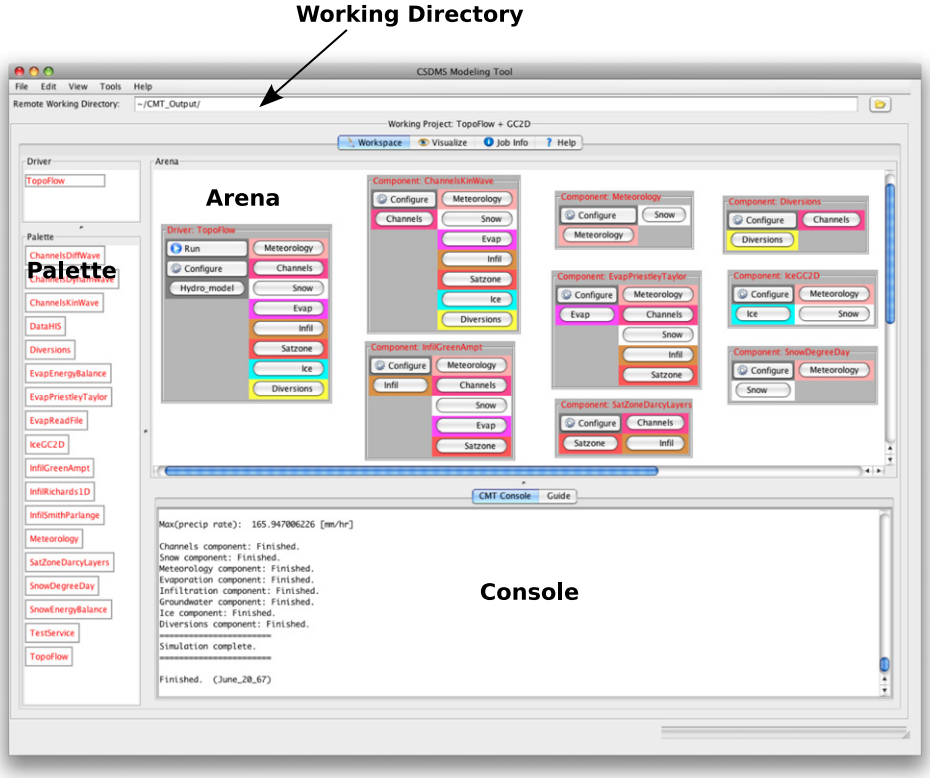

Fig. 4. Hydrologic model assembled in the CMT by dragging interchangeable process components from the palette to the arena. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this article.)

Any components in the arena that need to use that kind of port get linked automatically to the new one, as indicated using unique, matching colors (Fig. 4).

CSDMS continues to make usability improvements to the CMT and used it to teach a graduate-level course on surface process modeling at the University of Colorado, Boulder, in 2010. Several features of the CMT are ideal for teaching, including (1) the ability to save prebuilt component configurations and their settings in files (with extension BLD), (2) the Import Example Configuration feature, (3) a standardized HTML help page for each component, (4) a uniform, tabbed-dialog GUI for each component, (5) rapid comparison of different approaches by swapping one component for another, (6) the simple installation procedure, and (7) the ability to use remote resources.

9. Providing components with a uniform help system and GUI

Within a framework where it is easy to connect, interchange, and run coupled models, some users may treat components as black boxes and ignore the underlying assumptions that a component was built upon. They may even couple two components for which coupling does not make sense. To combat this problem, each CSDMS component is bundled with an HTML help document, easily accessible through the CMT that describes the model it wraps. These documents are standardized to include the following.

- Extended model description (along with references).
- Listing and brief description of the component's uses and provides ports.
- Main equations of the model.
- Sample input and output.
- Acknowledgment of the model developer(s).

CSDMS components also include XML files that describe their user-editable input variables. These files contain XML elements with detailed information about each variable including a default value, range of acceptable values, short and long descriptions, units, and data type. Using this XML file, the CMT automatically generates a graphical user interface for each component as a tabbed dialog. Despite significant differences between different models' input files, this provides CMT users with a uniform interface across all components. Furthermore, the GUI checks user input for errors and provides easily accessible help within the same environment. Input obtained from the GUI is automatically saved into whatever type of input or configuration file the model was designed to use. This process is done by replacing placeholders in an *input file template* with values from the GUI.

10. Summary

CSDMS has developed a component-based approach to integrated modeling that draws on the combined power of several open-source tools such as Babel, Bocca, Ccaffeine, the ESMF regridding tool, and the VisIt visualization tool. This noninvasive approach includes the new BMI and CMI interfaces. CSDMS also draws on the combined knowledge and creative effort of a large community of Earth-surface dynamics modelers and computer scientists. Using a variety of tools, standards and protocols, CSDMS converts a heterogeneous set of open-source, user-contributed models into a suite of plug-and-play modeling components that can be reused in many different contexts. The CSDMS model repository currently contains more than 160 models and tools. Of those, 55 have been converted into components that can be used in coupled modeling scenarios with Ccaffeine or the CMT. An up-to-date list is maintained at csdms.colorado.edu.

As with the model repository as a whole, CSDMS components cover the breadth of surface dynamics systems, such as (1) hydrology (e.g., the TopoFlow, Peckham, 2009; suite of 15 process models, HydroTrend, Ashton et al., 2013; Kettner and Syvitski, 2008; and a component to access CUAHSI-HIS data, Peckham and Goodall, 2013), (2) sediment transport (e.g., the 1D models of Parker, 2011), (3) landscape evolution (e.g., Erode, 2011; CHILD, Tucker et al., 2001; and MARSSIM, Howard, 1994), (4) geodynamics (e.g., Subside, Hutton and Syvitski, 2008), (5) glaciology (GC2D, Kessler et al., 2008), (6) coastal and marine (e.g., Ashton-Murray Coastal Evolution Model, Ashton et al., 2001; Avulsion, Ashton et al., 2013), and (7) stratigraphy (e.g., sedflux, Hutton and Syvitski, 2008).

All the software that underlies CSDMS is installed and maintained on its high-performance cluster. CSDMS members have accounts on this cluster and access its resources using a lightweight, Java-based client application called the CSDMS Component Modeling Tool (CMT) that runs on virtually any desktop or laptop computer. This approach is a type of *community cloud* because it provides remote access to numerous resources. This centralized cloud approach offers many advantages including (1) simplified maintenance, (2) more reliable performance, (3) automated backups, (4) remote storage and computation (user's PC remains free), (5) ability for many components (such as ROMS) and tools (such as VisIt and ESMF's regridder) to use parallel computation, (6) need for users to install only a lightweight client on their PC, (7) less need for technical support, and (8) ability to submit and run multiple jobs.

While the CF Convention (Lawrence Livermore National Laboratory, 2012b) provides a working solution for reconciling model variable names, semantic mediation remains one of the most challenging issues for model and data interoperability and more work is needed. The UDUNITS (Unidata, 2012) standard provides a working solution for unit conversion. Several other challenges remain, including methods for (1) tracking uncertainty, (2) model calibration, and (3) automated discovery and ingestion of models over a network.

Acknowledgments

CSDMS gratefully acknowledges major funding through a cooperative agreement with the National Science Foundation (EAR 0621695). Additional work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contracts DE-AC02-06CH11357 and DE-FC-0206-ER-25774. CSDMS is a community effort and we are especially grateful for the active involvement of many model contributors. We would also like to thank the entire CSDMS team with special thanks to our executive director, James Syvitski for his guidance and support and to Jisamma Kallumadikal for her programming work on the CMT.

References

Allan, B., Armstrong, R., Lefantzi, S., Ray, J., Walsh, E., Wolfe, P., 2010. Ccaffeine – a CCA component framework for parallel computing 〈http://www.cca-forum.org/ccafe/〉.

Allan, B.A., Norris, B., Elwasif, W.R., Armstrong, R.C., 2008. Managing scientific software complexity with Bocca and CCA. Scientific Programming 16 (December (4)), 315–327.

Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S., Smolinski, B., 1999. Toward a common component architecture for high-performance scientific computing. In: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing.

Ashton, A., Murray, A.B., Arnoult, O., 2001. Formation of coastline features by large-scale instabilities induced by high-angle waves. Nature 414, 296–300.

Ashton, A.D., Hutton, E.W., Kettner, A.J., Xing, F., Giosan, L., Kallumadikal, J., 2013. Progress in coupling models of coastline and fluvial dynamics. Computers & Geosciences 53, 21–29.

CCA Forum, 2010. A hands-on guide to the common component architecture ⟨http://www.cca-forum.org/tutorials/⟩.

CSDMS, 2011. Community Surface Dynamics Modeling System (CSDMS) ⟨http://csdms.colorado.edu⟩.

Dahlgren, T., Epperly, T., Kumfert, G., Leek, J., 2007. Babel User's Guide. CASC, Lawrence Livermore National Laboratory, UCRL-SM-230026, Livermore, CA.

David, O., Markstrom, S.L., Rojas, K.W., Ahuja, L.R., Schneider, I.W., 2002. The object modeling system. In: Ahuja, L.R., Ma, L., Howell, T. (Eds.), Agricultural System Models in Field Research and Technology Transfer. Lewis Publisher, CRC Press LLC, pp. 317–331. (Chapter 15).

de St.Germain, J.D., Morris, A., Parker, S.G., Malony, A.D., Shende, S., 2002. Integrating performance analysis in the Uintah software development cycle. In: Proceedings of the 4th International Symposium on High Performance Computing (ISHPC-IV), May 15–17, 2002. pp. 190–206 ⟨http://www.sci.utah.edu/publications/dav00/ishpc2002.pdf⟩.

Erode, 2011. Erode Landscape Evolution Model ⟨http://csdms.colorado.edu/wiki/Model_help:Erode-D8-Global⟩.

ESMF Joint Specification Team, 2011. Earth System Modeling Framework (ESMF) Web Site ⟨http://www.earthsystemmodeling.org/⟩.

Gregersen, J.B., Gijsbers, P.J., Westen, S.J.P., 2007. OpenMI: open modeling interface. Journal of Hydroinformatics 9 (3), 175–191. ⟨ http://www.iwaponline.com/jh/009/0175/0090175.pdf⟩.

Hill, C., DeLuca, C., Balaji, V., Suarez, M., da Silva, A., 2004. ESMF Joint Specification Team, 2004. The architecture of the earth system modeling framework. Computing in Science and Engineering 6, 18–28.

Howard, A.D., 1994. A detachment-limited model of drainage basin evolution. Water Resources Research 30 (7), 2261–2285.

Hutton, E.W.H., Syvitski, J.P.M., 2008. Sedflux-2.0: an advanced process-response model that generates three-dimensional stratigraphy. Computers & Geosciences 34 (10), 1319–1337.

Kessler, M.A., Anderson, R.S., Briner, J.P., 2008. Fjord insertion into continental margins driven by topographic steering of ice. Nature Geoscience 1, 365–369.

Kettner, A.J., Syvitski, J.P.M., 2008. Hydrotrend version 3.0: a climate-driven hydrological transport model that simulates discharge and sediment load leaving a river system. Computers & Geosciences 34 (10), 1170–1183.

Krishnan, S., Gannon, D., April 2004. XCAT3: a framework for CCA components as OGSA services. In: Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004). IEEE Computer Society, pp. 90–97.

Kumfert, G., April 2003. Understanding the CCA Specification using Decaf. Lawrence Livermore National Laboratory ⟨http://www.llnl.gov/CASC/components/docs/decaf.pdf⟩.

Lawrence Livermore National Laboratory, 2012a. Babel ⟨http://www.llnl.gov/CASC/components/babel.html⟩.

Lawrence Livermore National Laboratory, 2012b. CF Convention ⟨http://cf-pcmdi.llnl.gov/⟩.

Parker, G., 2011. 1D Sediment Transport Morphodynamics with Applications to Rivers and Turbidity Currents ⟨http://vtchl.uiuc.edu/people/parkerg/morphodynamic_e-book.htm⟩.

Peckham, S., 2009. Geomorphometry and spatial hydrologic modeling. In: Hengl, T., Reuter, H.I. (Eds.), Geomorphometry: Concepts Software and Applications, Developments in Soil Science, vol. 33. Elsevier, pp. 579–602. (Chapter 25).

Peckham, S.D., 2008. Evaluation of model coupling frameworks for use by the community surface dynamics modeling system (CSDMS). In: Poeter, E.P., Hill, M.C., Zheng, C. (Eds.), Proceedings of MODFLOW and MORE 2008: Ground Water and Public Policy Conference. Golden, Colorado, May 18–21, 2008, p. 535.

Peckham, S.D., Goodall, J.L., 2013. Driving plug-and-play components with data from web services: a demonstration of interoperability between CSDMS and CUAHSI-HIS. Computers & Geosciences 53, 154–161.

Qu, Y., Duffy, C.J., 2007. A semidiscrete finite volume formulation for multiprocess watershed simulation. Water Resources Research 43 (W08419), 1–18.

Syvitski, J.P., Hutton, E.W.H., Peckham, S.D., Slingerland, R., 2011. CSDMS – A modeling system to aid sedimentary research. The Sedimentary Record 9 (March (1)), 4–9.

The OpenMI Association, 2011. The Open Modeling Interface (OpenMI) ⟨http://www.openmi.org ⟩.

Tucker, G.E., Lancaster, S.T., Gasparini, N.M., Bras, R.L., 2001. The channel-hillslope integrated landscape development (child) model. In: Harmon, R.S., III, W.W.D. (Eds.), Landscape Erosion and Evolution Modeling. Academic/Plenum Publishers, pp. 349–388.

Unidata, 2012. UDUNITS ⟨http://www.unidata.ucar.edu/software/udunits/⟩.

VisIt, 2011. VisIt ⟨http://wci.llnl.gov/codes/visit⟩.