2014

EMELI 1.0: An Experimental Smart Modeling Framework For Automatic Coupling Of Self-Describing Models

Scott Dale Peckham

EMELI 1.0: AN EXPERIMENTAL SMART MODELING FRAMEWORK FOR AUTOMATIC COUPLING OF SELF-DESCRIBING MODELS

SCOTT D. PECKHAM (1)
(1): INSTAAR, University of Colorado, 1560 30th St., Boulder, CO 80020, USA

EMELI (Experimental Modeling Environment for Linking and Interoperability) is a modeling framework written in Python that was designed to explore the possibility of "smart modeling frameworks." As defined here, a smart modeling framework is one that makes it easy for users to couple reusable component models to create new, composite models through the use of a standardized model interface and standardized model metadata. Users make selections from a repository of component models that each provide a CSDMS Basic Model Interface (BMI) for self-description and model control. EMELI then (1) creates a framework object that serves as a container for the component models, (2) instantiates the selected component models as objects in the framework, (3) checks whether the chosen component models are compatible and together provide a complete composite model (i.e. whether every component model can get the variables it needs from one of the other models in the selected set) and then (4) runs the model, automatically passing required variables (or references) between the coupled components as necessary and automatically adjusting for differences between the component models, such as time-stepping scheme and units. EMELI demonstrates an attractive mechanism for coupling heterogeneous models after they have undergone a relatively small amount of additional preparation while also helping to prevent inappropriate couplings.

INTRODUCTION

Geoscientists develop and use a great variety of computer models to study the physical processes that take place on the Earth and other planets, and to make predictions and solve problems for the benefit of society. These models are built using mathematics and physics that represent our best scientific understanding of these processes. Model developers make choices based on their needs, abilities, resources and judgment, such as which programming language, computational grid, time-stepping scheme, simplifying assumptions, algorithms, numerical methods (i.e. equation solving schemes), input data sets and variable names to use in their model. As a result, there are now a large number of heterogeneous models that are actively used by geoscientists, both at academic institutions and federal agencies. Each solves a particular type of problem and has its own set of input variables (required data that it must get from another model or data set) and output variables (data it can compute using physical laws and approximations). The breadth of these models means that by coupling them together in various ways, their combined capabilities can provide predictions for a much larger set of science problems. These models often complement one another, with one model providing an output variable that another model needs as an input variable.

Unfortunately, the heterogeneity present in this large set of models makes it difficult to couple them and to thereby take advantage of their combined capabilities. It can take a large amount of

programmer time and effort just to couple two nontrivial models in order to realize such benefits. In order to perform the coupling operation, a software engineer must typically spend time learning the unique "anatomy" or idiosyncrasies of each model, and this typically requires the involvement of the model authors. Great care is required so as not to introduce new bugs into either model or to adversely affect their performance. In addition, there are pairs of models that are not appropriate for coupling because some of their underlying assumptions or simplifications are incompatible. Model developers often express concern about their models being used incorrectly by non-experts that do not understand the important but low-level details and limitations.

Many of these difficulties associated with model coupling and code reuse have been addressed in a robust manner by a large, NSF-funded project called CSDMS (Community Surface Dynamics Modeling System). One of the goals of the CSDMS project was to assemble a large repository of freely available, open-source models and tools, and this growing collection now consists of 225 models and tools. Taken together, these are able to solve a wide variety of problems in the domain of earth surface process dynamics. Another key goal of CSDMS was to create a component-based modeling framework that would make it easy to couple models from the repository in a plug-and-play manner to solve new problems (Peckham et al., 2013). While technically challenging, CSDMS achieved the latter goal by building upon and combining several well-established and open-source software tools, including several from the Common Component Architecture (CCA) toolchain (i.e. Babel, Bocca and Ccaffeine) and by developing an innovative model interface standard called BMI (Basic Model Interface) along with a supporting model metadata standard called the CSDMS Standard Names (Peckham, 2014). BMI has proven to be an elegant solution to the problem of preparing a model for reuse in a plug-and-play modeling framework and is described in more detail in a separate section.

The EMELI framework described in this paper was developed to explore the extent to which plug-and-play modeling with reusable model components could be automated by using a simple but standardized model interface and standardized model metadata. It is "lighter weight" than the CCA-compliant Ccaffeine framework that underpins the CSDMS modeling framework. EMELI 1.0 can only couple model components written in Python and it does not support parallel computation. However, a future version of EMELI could use the same language interoperability tool used by the CSDMS framework, called Babel, to create Python language bindings for models written in languages other than Python (i.e. C, C++, any Fortran, Java). Like the CSDMS framework, EMELI is designed for coupling model components that have been augmented with a Basic Model Interface (BMI). EMELI also uses service components to reconcile differences between model components, such as the time-stepping scheme and units used for variables. However, EMELI differs from the current CSDMS framework in that the BMI-enabled model components do not need to be wrapped to create a new set of objects that each provide a (framework-specific) Component Model Interface (CMI). Instead, the capabilities of the CMI interface are provided by EMELI directly. EMELI also checks whether each component to be coupled is able to obtain the specific input variables that it needs from one of the other components in the set that a user has selected for coupling, which is something the current CSDMS framework does not do. Experiments with EMELI are intended to inform the future development of the CSDMS modeling framework. For example, the next version of EMELI will check whether models are compatible or appropriate for coupling by making use of CSDMS Standard Assumption Names, a large set of standardized terms for describing the assumptions and constraints that define a model. A future version of EMELI may even take the place of Ccaffeine in the CSDMS framework.

BACKGROUND: THE BASIC MODEL INTERFACE (BMI)

Toward the beginning of the CSDMS project, several different model-coupling frameworks were evaluated (Peckham, 2008) against a set of design criteria that had been articulated by the academic modeling community (Syvitski et al., 2004; Goodall et al., 2008). The projects that were the most mature and that came closest to meeting these criteria were the Earth System Modeling Framework or ESMF (Hill et al., 2004), the Common Component Architecture or CCA (Bernholdt et al., 2006), the Open Model Interface or OpenMI (Gregersen et al., 2007) and the Object Modeling System or OMS (David et al., 2002). Each of these projects has had the goal of making it easier to reuse and connect models written by different authors, but each serves a somewhat different modeling community with somewhat different design criteria. Interestingly, all of these projects have identified the need for some simple refactoring of model source code to provide separate functions for (1) initializing the model (i.e. open files, allocate memory, initialize variables, etc.), (2) updating the model by one time step (i.e. advancing all of the models computed variables) and (3) finalizing the model (i.e. closing files, cleanup and reporting). In CSDMS these are referred to as **model control functions,** and they are necessary in order for a modeling framework to bypass a model's own time loop for the purpose of coupling. Fortunately, many models already have functions similar to this or can be modified fairly easily to provide them.

Based on this analysis, it was decided that the CCA toolchain (e.g. Babel, Bocca and Ccaffeine) provided important infrastructure that was needed by the CSDMS project, such as support for high-performance computing (HPC) and efficient interoperability of code written in different languages. However, as a general system for component-based software development, CCA did not specify any particular component interface and left this job to users of the CCA tools. The CSDMS team therefore experimented with a variant of the OpenMI interface to serve as a standardized model interface for model coupling. After working to "componentize" many different open-source models with this type of interface and facing a variety of technical and social challenges, the essence of the problem gradually came into sharper focus. This ultimately led to the breakthrough idea of a *two-interface* (or two-level) wrapping process, consisting of a simple, framework-agnostic standardized interface called the Basic Model Interface (BMI) that model developers would be asked to implement (with help from CSDMS staff) and another, more sophisticated and framework-specific (or framework-aware) interface called the Component Model Interface (CMI) that would adapt any BMI-enabled model for use in the CSDMS modeling framework.

This really was a breakthrough for a number of reasons. It addressed the needs of model developers by: (1) requiring minimal effort, (2) being noninvasive (no introduction of dependencies on CSDMS data structures or code and no interference with the developer's design), (3) allowing the model to continue to be used as before in a stand-alone manner, (4) not requiring any new code intended to accommodate the needs of other models (unlike OpenMI 1.4), (5) being **framework agnostic**; it requires no modeling-framework specific knowledge (e.g. the CCA concept of *ports*) and developers do not "code to" the framework, (6) requiring the developer to do only those things that would be necessary for the model to be used in *any* modeling framework and (7) providing added value, such as the ability to couple to other models and to write the model's output variables to standardized NetCDF files. It therefore removed many of the barriers that were discouraging to model developers (and added enticements). However, it also addressed the needs of the CSDMS software engineering team by: (1) allowing the same piece of code to provide a "CMI wrapper" for any BMI-enabled model, (2) making it possible to largely automate the process of converting models to CSDMS components, (3) dramatically reducing code maintenance time, (4) requiring minimal additional effort to bring a new version of the same model (e.g. with enhancements or bug fixes) into the framework, (5) not significantly impacting model performance and (6) allowing the CSDMS framework to automatically call service components when needed to accommodate differences

between models such as programming language, computational grid, time-stepping scheme, variable names and units. Service components provide additional added value such as output to NetCDF files, time interpolation, unit conversion and spatial regridding.

The essence of BMI is that it provides functions that: (1) give the caller (e.g. framework) **complete, fine-grained control** of the model and (2) make a model **self-describing**, so that the framework can retrieve any information it needs about the model to facilitate coupling. The BMI functions can be organized into five groups called: Model control functions, Model information functions, Variable information functions, Variable getter and setter functions and Grid information functions. BMI is described in more detail by Peckham et al. (2013) and on the BMI pages of the CSDMS website. BMI is somewhat analogous to XML in the sense that while it may first appear to be deceptively simple, it is actually quite powerful and does exactly what it needs to do. Since BMI is framework agnostic and provides model control and self-description, it should be straightforward to wrap a BMI-enabled model to provide a framework-specific component interface other than CMI, allowing the model to used in other frameworks like ESMF, OpenMI or OMS. The EMELI and CSDMS frameworks are both designed to work with model components that have a BMI interface.

WHAT DOES EMELI DO?

(1) Before starting EMELI, users make selections from a repository of component models that each have a BMI interface. These choices are provided in the form of a small text file called a "provider file", where each line contains the name of a component type or "port type" followed by the name of a particular component of that type that exists in the repository. These component types are usually associated with a particular physical process, such as infiltration, evaporation or snowmelt. The repository is likely to contain several components of a given "type". While this use of component types helps users to know which components are interchangeable and is used by the CSDMS framework, EMELI has shown that it is not strictly necessary and it adds some extra complexity. This will likely be handled differently in a future version of EMELI.

(2) An instance of EMELI is created and its *run_model()* method is called. The remaining steps take place within the *run_model()* method.

(3) EMELI calls the *read_repository()* method in order to get information about all of the components that are (locally) available for potential coupling. This information includes the information - such as paths, module names and configuration filenames – that is needed to instantiate and configure each model component.

(3) EMELI calls the *read_provider_file()* method, which reads the user's provider file to determine which components have been selected from the repository for (potential) coupling to create a composite model.

(4) EMELI creates a Python dictionary called "comp_set" where the keys are "port names" (i.e. a component type) and the values are instances of the BMI-enabled model components that are listed in the user's provider file.

(5) EMELI calls the *initialize_comp_set()* method. First, the find_var_users_and_providers() method is called, which calls the BMI functions *get_input_var_names()* and *get_output_var_names()* for each component in comp_set to get the names for the (1) input variables that each model needs from another component and (2) the output variables that it is able provide to another component. These two BMI functions return lists of CSDMS Standard Variable Names that the model developer previously identified as equivalent to the abbreviated names used in the model code. (This mechanism addresses the problem of **semantic mediation**,

which is discussed in a separate section below.) Next, the *check_var_users_and_providers()* method is called to check if (1) any component requires an input variable that cannot be provided by one of the other components in the comp_set, or if (2) there is a required input variable that is provided by **more** than one component in the comp_set. EMELI exits with a warning message in both of these cases. By analogy to the mathematical requirement of needing N equations to solve for N unknowns, these cases can be described as being *underdetermined* or *overdetermined*. (Note: This is also where a future version of EMELI will compare the underlying assumptions of each component model in the comp_set to check whether the components are compatible and appropriate for coupling.)

(6) Continuing with the *initialize_comp_set()* method, EMELI calls the BMI function *initialize()* for each component in the comp_set, in the order in which they are listed in the provider file. For each output variable that a given component is able to provide, EMELI creates a list of which, if any, of the other components in the comp_set require this variable as an input variable. It then calls the *connect()* method, which first calls the *get_values()* method to get initialized values from the component and then calls the *set_values()* method to set these values in all other components that need this variable. Note that components may need the initial value of a variable before they can initialize their own variables. The *connect()* method also determines whether values need to be regridded or have their units converted before the call to the *set_values()* method. (EMELI does not yet contain a service component for regridding, but the ESMF regridder used by CSDMS now provides a Python API that would allow it to be used.) EMELI's *get_values()* and *set_values()* methods make calls to the BMI getter and setter methods of the components in the comp_set.

(7) EMELI creates an instance of a "time interpolator" service component and calls its *initialize()* method. This service component is described in a subsequent section.

(8) EMELI starts a time loop. In each pass through the loop, EMELI compares the clock time to the internal time of each component in the comp_set (in the order listed in the provider_file). If the clock time exceeds a model component's internal time, then EMELI: (1) calls the *get_required_vars()* method to get the latest values of variables that the model component needs from other components, (2) calls the model component's BMI *update()* function (which may use these updated variables from others) and (3) calls a method of the time interpolator component that updates the interpolation variables for every variable that the model component provides to other components in the comp_set. The *get_required_vars()* method gets interpolated values from the time interpolator, which in turn makes calls to the BMI getters of the model components that provide the needed variables. It then calls the *set_values()* method which in turn calls the BMI setter function of the model that needs the variables.

(9) EMELI checks the stopping condition of the model component that the user's provider file identified as the driver to determine if the model run is finished. If so, it calls the *finalize_all()* method which calls *get_required_vars()* once more and then calls the BMI finalize function for every component in the comp_set. Otherwise, it calls the *update_time()* method and the model run continues.

AUTOMATIC SEMANTIC MEDIATION

Since the model components in a repository have usually been developed by different people, often from different science domains, they each use different names and abbreviations for the input and output variables in their source code. For example, one model might be able to compute the volumetric flow rate at a river mouth, calling it "streamflow", and another model might need this same variable to do its calculations, calling it "discharge" (or just "Q", the standard abbreviation used by hydrologists). The two models might also use different units for this variable. However, as long as each model component in the repository has a BMI interface,

EMELI is able to understand the semantic equivalence, "see" the possibility for sharing this variable, automatically link the two models and even adjust for the possibility that they use different units. This is because most of the BMI functions require a "long variable name" argument that is required to be a CSDMS Standard Variable Name. As part of implementing the BMI interface for a model, the developer finds the equivalent CSDMS Standard Variable Name for each of the input and output (i.e. shared) variables that appear in the model source code. The developer does **not** change the names (usually just abbreviations) as they appear in the source code, but instead simply provides a mapping (e.g. a Python dictionary) from the names used internally to longer, standardized names. This only requires minimal changes to the source code, in the form of new BMI functions that augment those of the original source code. Two of the BMI functions, called *get_input_var_names()* and *get_output_var_names()*, simply return lists of the standardized names for the model's input and output variables, respectively. EMELI uses these to see what every component in a set needs or can provide to others in the set.

AUTOMATIC TIME INTERPOLATION

Since each model component in a repository has its own time step and time-stepping scheme (e.g. fixed or adaptive), EMELI automatically calls a service component to perform time interpolation when necessary for all variables that are passed between components. EMELI gets time-related information for each component in the comp_set by calling its BMI functions *get_current_time()* and *get_time_units()*.

The time interpolation component creates a Python dictionary called "time_interp_vars" where the keys are "long variable names" (i.e. CSDMS Standard Variable Names) for any variables that must be passed between components in the comp_set, and the values are objects (instances of a class called time_interp_data) with the data and methods needed to compute interpolated values at any requested time. The data part consists of two successive model times (for the model component that provides the variable of interest) separated by one model time step along with the two values (of any data type) that the variable takes at those two times. From this data (retrieved from the BMI-enabled model), the *update()* method can compute interpolated values for any intermediate time that may be needed by other components. If the time interpolator component receives a request (from EMELI) for a variable at a time beyond the internal time of the model component that provides the variable, it calls that model's BMI *update()* method and updates the data for the interpolation interval endpoints. The time interpolator also converts the time units, if necessary, between the model component that is providing the variable and the one that is using it. It currently supports linear interpolation and no interpolation (stairsteps).

AUTOMATIC COMPATIBILITY CHECKING

Although this hasn't been implemented yet, the intent is for a future version of EMELI's *initialize_comp_set()* method to check whether all of the components in the comp_set are compatible and appropriate for coupling. CSDMS has developed a prototype XML schema called a Model Metadata File (MMF) designed to store detailed information about all of the "assumptions" made by a given model, including named equations, simplifying assumptions, coordinate system conventions and so on. These assumptions are recorded as standardized strings in the MMF, drawn from the (also experimental) set of CSDMS Standard Assumption Names.

LINKING TOPOFLOW HYDROLOGIC MODEL COMPONENTS WITH EMELI

TopoFlow is a spatial, hydrologic model that provides multiple options for modeling each of the physical processes that contribute to the hydrologic cycle in a watershed (Peckham, 2009). The original version was written in Interactive Data Language (IDL) and provides a wizard-style,

point-and-click graphical user interface (GUI). In support of the CSDMS project, TopoFlow was converted from IDL to Python/NumPy with help from a program called I2PY (Peckham, 2010). It was then broken into 16 separate, process model components that can run as stand-alone models or be used as easily replaceable, plug-and-play components within the CSDMS or EMELI modeling frameworks. For a given process like snowmelt, the interchangeable component options typically range from a fairly simple method that requires less input data, (e.g. degree-day) to more sophisticated methods that require more input data (energy-balance). The components available for each hydrologic process are as follows:

Flow routing:	Kinematic wave, Diffusive wave and Dynamic wave (D8-based)
Meteorology:	Shortwave and longwave radiation calculators following Dingman (2001)
Snowmelt:	Degree-day and energy balance
Evaporation:	Priestley-Taylor, energy balance and read from file
Infiltration:	Green-Ampt, Smith-Parlange 3-parameter, Richards 1D
Saturated Zone:	Shallow, surface parallel layers (up to 3)
Diversions:	Flow fraction method for sources, sinks and canals
Icemelt:	GC2D
Driver:	TopoFlow (main)

Each of the TopoFlow components is open-source and has a complete BMI interface (with standard names), a tabbed-dialog GUI for use in the CSDMS Modeling Tool (CMT) and complete, HTML-based documentation on the CSDMS wiki. All of the TopoFlow components are now available as a single Python package that also includes EMELI as well as many supporting utilities and components from the Erode model (a D8-based landscape evolution model). This set of components serves as an example repository for use with EMELI. Users select components from this repository by creating provider files as explained previously. Some of the TopoFlow process components (e.g. snowmelt and icemelt) have much slower time steps than others (e.g. channel flow). The package therefore serves as a demonstration of both EMELI and its time-interpolation service component.

As an example of automatic semantic mediation, TopoFlow contains a snowmelt component and an evaporation component that are both based on a full energy balance. The snowmelt component requires many different input variables, including: air temperature, land surface temperature, air relative humidity, air pressure at the land surface, net flux of longwave and shortwave radiation incident on the land surface and wind speed at a reference height of 10 meters. In the model source code these are called: T_air, T_surf, RH, p0, Qn_LW and Qn_SW and uz, but these names are mapped to equivalent CSDMS Standard Names in the BMI functions. This particular set of input variables are among the output variables that are provided by the TopoFlow meteorology component. By calling BMI functions, EMELI "sees" this and *automatically* ensures that these variables are all passed from the meteorology component to the energy balance snowmelt component via dynamic linking at runtime. EMELI does the same thing for *all* of the variables that must be passed between TopoFlow components.

CONCLUSIONS AND FUTURE WORK

EMELI 1.0 provides a "smart" modeling framework that makes use of a standardized model interface (BMI), standardized model metadata (the CSDMS Standard Names) and service components (e.g. a time interpolation component) in order to greatly simplify the reuse and sharing of heterogeneous model components. It is written in Python and easily extended to facilitate experimentation so that different ways of using standardized interfaces and metadata can be evaluated in terms of performance, robustness, maintenance, elegance and ease of use. EMELI currently provides automatic semantic mediation and includes a service component for automatic time interpolation. A future version will include additional service components to

provide automatic spatial regridding and unit conversion when needed to reconcile differences between model components. In addition, Babel will be used to create Python language bindings when needed to support model components written in a language other than Python. EMELI will also be used to further refine and test the CSDMS Standard Assumption Names and Model Metadata File format. This represents a new frontier in the use of standardized model metadata to first find model components that can be used together and then to automatically check and quantify (with a report) the degree to which they are compatible for coupling.

ACKNOWLEDGMENTS

Funding for this work by a subaward from the University of Alaska at Fairbanks is gratefully acknowledged in addition to many helpful discussions with Bob Bolton and Eric Hutton.

REFERENCES

Bernholdt, D.E., B.A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T.L. Dahlgren, K. Damevski, W.R. Elwasif, T.G.W. Epperly, M. Govindaraju, D.S. Katz, J.A. Kohl, M. Krishnan, G. Kumfert, J.W. Larson, S. Lefantzi, M.J. Lewis, A.D. Malony, L.C. McInnes, J. Nieplocha, B. Norris, S.G. Parker, J. Ray, S. Shende, T.L. Windus and S. Zhou (2006) A component architecture for high- performance scientific computing, *Intl. J. High Performance Computing Applications, ACTS Collection Special Issue*, 20(2), 163-202.

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 Tech. Transfer*, Lewis Publisher, CRC Press, pp. 317–331. (Ch. 15).

Dingman, S.L. (2001) *Physical Hydrology*, Prentice Hall, 656 pp. (Appendix E)

Goodall, J., D.G. Tarboton, S.D. Peckham, R. Hooper (2008) New software architecture for integrated water modeling, *EOS, Transactions*, 89(43), p. 420, American Geophys. Union.

Gregersen, J.B., P.J.A. Gijsbers and S.J.P. Westen (2007) OpenMI: Open modelling interface, *J. Hydroinformatics*, 09.3, 175–191.

Hill, C., C. DeLuca, V. Balaji, M. Suarez, and A. da Silva (2004) The architecture of the Earth System Modeling Framework, *Computing in Science and Engineering*, 6(1), 18-28.

Peckham, S.D. (2014) The CSDMS Standard Names: Cross-domain naming conventions for describing process models, data sets and their associated variables, International Environmental Modelling and Software Society (iEMSs), Proceedings of the 7th Intl. Congress on Env. Modelling and Software, San Diego, California, USA, Daniel P. Ames, Nigel W. T. Quinn, Andrea E. Rizzoli (Eds.) http://www.iemss.org/society/index.php/iemss-2014-proceedings

Peckham, S.D., E.W.H. Hutton and B. Norris (2013) A component-based approach to integrated modeling in the geosciences: The Design of CSDMS, *Computers & Geosciences*, special issue: Modeling for Environmental Change, 53-12, http://dx.doi.org/10.1016/j.cageo. 2012.04.002.

Peckham, S.D. (2010) The I2PY 0.2 User's Guide, 23 pp (a significant extension of I2PY 0.1 written in 2005 by Christopher J. Stawarz), http://csdms.colorado.edu/trac/i2py.

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

Peckham, S.D. (2008) Evaluation of model coupling frameworks for use by the Community Surface Dynamics Modeling System (CSDMS), In: *Proceedings of MODFLOW and MORE 2008: Ground Water and Public Policy Conference*, May 18-21, 2008, Golden, CO, 535p, Eds. E.P Poeter, M.C. Hill and C. Zheng.

Syvitski, J.P.M., G. Tucker, D. Seber, S. Peckham, S. Seitzinger, T. Pfeffer, A. Voinov, R. Slingerland, B. Goran (2004) Community Surface Dynamics Modeling System (CSDMS) Implementation Plan, based on two NSF-sponsored workshops (Boulder 2002 and Minneapolis 2004). http://csdms.colorado.edu/wiki/CSDMS_docs.