

A Component-Based Approach to Integrated Modeling in the Geosciences: The Design of CSDMS

Scott Peckham, Eric Hutton

CSDMS, University of Colorado, 1560 30th Street, UCB 450, Boulder, CO 80309, USA

Boyana Norris

*Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S.
Cass Ave., Argonne, IL 60439, USA*

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.

Keywords:

component software, CCA, CSDMS, modeling, code generation

Email addresses: `Scott.Peckham@colorado.edu` (Scott Peckham),
`Eric.Hutton@colorado.edu` (Eric Hutton), `norris@mcs.anl.gov` (Boyana Norris)

1. Introduction

The Community Surface Dynamics Modeling System (CSDMS) project [12] 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, OpenMI and OMS) and seeks to evaluate different

26 approaches in pursuit of those that are optimal. As with any design problem,
27 myriad factors must be considered in determining what is optimal, including
28 how various choices affect users and developers. Other key factors are per-
29 formance, ease of maintenance, ease of use, flexibility, portability, stability,
30 encapsulation, and software longevity.

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

47 Component programming builds on the fundamental concepts of object-
48 oriented programming, with the main difference being the introduction or
49 presence of a runtime *framework*. Components are generally implemented as
50 classes in an object-oriented language and are essentially “black boxes” that

51 encapsulate some useful bit of functionality. The purpose of a framework is
52 to provide an environment in which components can be linked together to
53 form applications. The framework also provides *services* to all components,
54 such as the linking mechanism itself. The following are some key advantages
55 of component-based programming:

- 56 • Components can be written in different languages and still communi-
57 cate (via language interoperability).
- 58 • Components are precompiled units that can be replaced, added to, or
59 deleted from an application at runtime via dynamic linking.
- 60 • Components can easily be moved to a remote location (different ad-
61 dress space) without recompiling other parts of the application (via
62 RMI/RPC support).
- 63 • Components can have multiple different interfaces.
- 64 • Components can be “stateful”; component data is retained between
65 method calls over its lifetime.
- 66 • Components can be customized at runtime with configuration param-
67 eters.
- 68 • Components facilitate code reuse and rapid comparison of different
69 implementations.
- 70 • Components facilitate efficient cooperation between groups, each doing
71 what it does best.

72 **2. Background**

73 Here we review the role that the Common Component Architecture (CCA)
74 has played in component programming. We then discuss three foundational
75 tools that underpin CSDMS: Babel, Ccaffeine, and Bocca.

76 *2.1. The Common Component Architecture*

77 The Common Component Architecture [3] is a *component architecture*
78 *standard* adopted by federal agencies (largely the Department of Energy and
79 its national laboratories) and academic researchers to allow software compo-
80 nents to be integrated for enhanced functionality on high-performance com-
81 puting systems. The CCA Forum is a grassroots organization that started in
82 1998 to promote component technology standards and code reuse for HPC.
83 CCA defines standards necessary for interoperation of components developed
84 in different frameworks. Components that adhere to these standards can be
85 ported with relative ease to another CCA-compliant framework. While var-
86 ious other component architecture standards exist in the commercial sector
87 (e.g., CORBA, COM, .Net, and JavaBeans), CCA was created to fulfill the
88 needs of scientific, high-performance, open-source computing that are un-
89 met by these other standards. For example, scientific software requires full
90 support for complex numbers, dynamically dimensioned multidimensional
91 arrays, Fortran (and other languages), and multiple processor systems. Arm-
92 strong et al. [3] explain the motivation for creating CCA by discussing the
93 pros and cons of other component-based frameworks with regard to scien-
94 tific, high-performance computing. Many of the papers in our cited references
95 have been written by CCA Forum members and are helpful for learning more

96 about the CCA. The CCA Forum has also prepared a set of tutorials called
97 “A Hands-On Guide to the Common Component Architecture” [11].

98 A variety of different frameworks, such as Ccaffeine [1], CCAT/XCAT [25],
99 SciRUN [15], and Decaf [26], adhere to the CCA standard. Ccaffeine was de-
100 signed to support both serial and parallel computing, and SciRUN and XCAT
101 were designed to support distributed computing. Decaf [26] was designed by
102 the developers of Babel primarily as a means of studying the technical as-
103 pects of the CCA standard itself. The important point is that each of these
104 frameworks adheres to the same standard, and this facilitates reuse of CCA
105 components in different computational settings.

106 *2.2. Language Interoperability with Babel*

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

112 Babel [29, 14] is an open-source, language interoperability tool (consist-
113 ing of a compiler and runtime) that automatically generates the “glue code”
114 necessary for components written in different computer languages to com-
115 municate. As illustrated in Fig. 1, Babel currently supports C, C++, For-
116 tran (77, 90, 95, and 2003), Java, and Python. Babel is much more than a
117 “least common denominator” solution; it even enables passing of variables
118 with data types that may not normally be supported by the target language
119 (e.g., objects and complex numbers). Babel was designed to support *scien-*
120 *tific, high-performance* computing and is used as an integral part of CCA-

121 compliant frameworks. It won an R&D 100 design award in 2006 for “the
122 world’s most rapid communication among many programming languages in
123 a single application.” It has been shown to outperform similar technologies
124 such as CORBA and Microsoft’s COM and .NET.

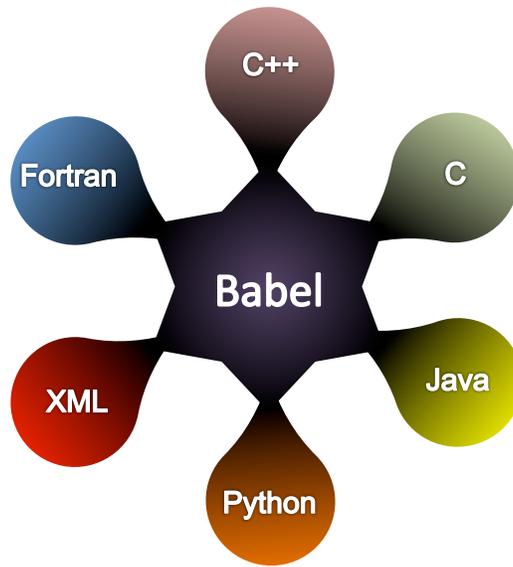


Figure 1: Language interoperability provided by Babel.

125 In order to create the glue code needed for two components written in dif-
126 ferent programming languages to exchange information, Babel needs to know
127 only about the interfaces of the two components. Babel therefore accepts as
128 input a description of an interface in either of two “language-neutral” forms,
129 XML (eXtensible Markup Language) or SIDL (Scientific Interface Defini-
130 tion Language). The SIDL language (somewhat similar to CORBA’s IDL)
131 was developed specifically for the Babel project. Its sole purpose is to pro-
132 vide a concise description of a scientific software component interface. This
133 interface description includes complete information about a component’s in-

134 terface, such as the data types of all arguments and return values for each of
135 the component’s methods (or member functions). SIDL has a complete set
136 of fundamental data types to support scientific computing, from Booleans to
137 double-precision complex numbers. It also supports more sophisticated data
138 types such as enumerations, strings, objects, structs, and dynamic multi-
139 dimensional arrays. A description of SIDL syntax and grammar can be found
140 in “Appendix B: SIDL Grammar” in the Babel User’s Guide [14]. Complete
141 details on how to represent a SIDL interface in XML are given in “Appendix
142 C: Extensible Markup Language (XML)” of the same guide.

143 2.3. The Ccaffeine Framework

144 Ccaffeine [1] is the most widely used CCA framework, providing the run-
145 time environment for sequential or parallel component applications. Us-
146 ing Ccaffeine, component-based applications can run on diverse platforms,
147 including laptops, desktops, clusters, and leadership-class supercomputers.
148 Ccaffeine provides some rudimentary MPI communicator services, although
149 individual components are responsible for managing parallelism internally
150 (e.g., communicating with other distributed components). A CCA frame-
151 work provides *services*, which include component instantiation and destruc-
152 tion, connecting and disconnecting of ports, handling of input parameters,
153 and control of MPI communicators. Ccaffeine was designed primarily to
154 support single-component multiple-data (SCMD) programming, although it
155 can support multiple-component multiple-data (MCMD) applications that
156 implement more dynamic management of parallel resources.

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

- 158 • The framework loads the dynamic library for the component. Static
159 linking options are also available.
- 160 • The component is instantiated. The framework calls the `setServices`
161 method on the component, passing a handle to itself as an argument.
- 162 • User-specified connections to other components' ports are established
163 by the framework.
- 164 • If the component provides a `gov.cca.ports.Go` port (similar to a
165 “main” subroutine), its `go()` method can be invoked to initiate com-
166 putation.
- 167 • Connections can be made and broken throughout the life of the com-
168 ponent.
- 169 • All component ports are disconnected, and the framework calls `re-`
170 `leaseServices` prior to calling the component's destructor.

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

178 2.4. Component Development with Bocca

179 Bocca [2] is a tool in the CCA tool chain that was designed to help users
180 create, edit, and manage a set of SIDL-based entities, including the CCA

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

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

199 While Bocca automatically generates dynamic libraries, a separate tool
200 can be used to create *stand-alone executables* for projects by automatically
201 bundling all required libraries on a given platform. Examples of using Bocca
202 are available in the set of tutorials called “A Hands-On Guide to the Common
203 Component Architecture,” written by the CCA Forum members [11].

204 3. Component Design Issues

205 In this section we overview our CSDMS infrastructure design criteria and
206 implementation approach.

207 3.1. Design Criteria

208 The technical goals of CSDMS include the following.

- 209 • Support for *multiple operating systems* (especially Linux, Mac OS X,
210 and Windows).
- 211 • *Language interoperability* to support code contributions written in pro-
212 cedural languages (e.g., C or Fortran) as well as object-oriented lan-
213 guages (e.g., Java, C++, and Python).
- 214 • Support for both *structured and unstructured grids*, requiring a spatial
215 regridding tool.
- 216 • *Platform-independent graphical user interfaces (GUIs) and visualiza-*
217 *tion capabilities* where appropriate.
- 218 • Use of well-established, open-source *software standards* whenever pos-
219 sible (e.g., CCA, SIDL, OGC, MPI, NetCDF, OpenDAP, and XML).
- 220 • Use of *open-source tools* that are mature and have well-established com-
221 munities, avoiding dependencies on proprietary software (e.g., Win-
222 dows, C#, and Matlab) whenever possible.
- 223 • Support for both *serial and parallel computation* (multiprocessor, via
224 MPI standard).

- 225 • *Interoperability with other coupling frameworks.* Because code reuse is a
226 fundamental tenet of component-based modeling, the effort required to
227 use a component in another framework should be kept to a minimum.
- 228 • *Robustness and ease of maintenance.* The modeling system will
229 clearly have many software dependencies, and this software infrastruc-
230 ture must be updated on a regular basis.
- 231 • *Use of HPC tools and libraries.* If the modeling system runs on HPC
232 architectures, it should strive to use parallel tools and models (e.g.,
233 VisIt, PETSc, and the ESMF regriding tool).
- 234 • *Familiarity.* Model developers and contributors should not be required
235 to make major changes to their existing development processes and
236 habits.

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

247 3.2. *Interface vs. Implementation*

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

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

270 The distinction between *interface* and *implementation* is an important
271 theme in computer science. The word pair *declaration* and *definition* is used

272 in a similar way. A function (or class) declaration tells what the function
273 does (and how to interact with or use it) but not how it works. To see how
274 the function actually works, we must look at how it has been defined or
275 implemented. C and C++ programmers are familiar with this idea, which
276 is similar to declaring variables, functions, classes, and other data types in a
277 header file with the file name extension `.h` or `.hpp`, and then defining their
278 implementations in a separate file with extension `.c` or `.cpp`.

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

283 While the tools in the CCA tool chain are powerful and general, they do
284 not provide a ready interface for linking geoscience models (or any domain-
285 specific models). In CCA terminology, a *port* is essentially a synonym for
286 interface, and a distinction is made between ports that a given component
287 uses (*uses ports*), and those that it provides (*provides ports*) to other com-
288 ponents. This approach provides a means of bidirectional information ex-
289 change between components, unlike dataflow-based approaches that support
290 only unidirectional links between components.

291 Each scientific modeling community that wishes to make use of the CCA
292 tools must design or select the component interface best suited to the kinds
293 of models they wish to link together. This is a big job that involves both
294 social and technical issues and requires a significant time investment. The
295 component interface we have developed for CSDMS is described in Sections
296 4 and 5.

297 *3.3. Granularity*

298 While components may represent any level of granularity, from a sim-
299 ple function to a complete hydrologic model, the optimum level appears to
300 be that of a particular physical process, such as infiltration, evaporation, or
301 snowmelt. This is the level at which researchers typically want to swap out
302 one method of modeling a process for another. A simpler method of param-
303 eterizing a process may apply only to simplified special cases or may be used
304 because there is insufficient input data to drive a more complex method. A
305 different numerical method may solve the same equations with greater accu-
306 racy, stability, or efficiency; it may or may not use multiple processors. Even
307 the same method of modeling a given process may exhibit improved perfor-
308 mance when coded in a different programming language. Physical processes
309 often act within a domain that shares a physically important boundary with
310 other domains (e.g., coastline and ocean-atmosphere). The fluxes between
311 these domains are often of key interest and help to determine granularity.

312 Some models are written in such a way that decomposing them into sepa-
313 rate process components is not appropriate because of some special aspect of
314 the model’s design or because decomposition would result in a loss of perfor-
315 mance (e.g., speed, accuracy, or stability). For example, *multiphysics mod-*
316 *els*—such as Penn State Integrated Hydrologic Model (PIHM)—represent
317 many physical processes as one large, coupled set of ODEs that are then
318 solved as a matrix problem on a supercomputer.

319 *3.4. Interchangeability and Autoconnection*

320 A key goal of component-based modeling is to create a collection of com-
321 ponents that can be coupled together to create new and useful composite

322 models. This goal can be achieved by providing every component with the
323 same interface. A secondary goal, however, is for the coupling process to be
324 as automatic as possible, that is, to require as little input as possible from
325 users. In order to achieve this goal, components must somehow be grouped
326 into categories according to the functionality they provide. This grouping
327 must be readily apparent to both a user and the framework (or system)
328 so that it is clear whether a given pair of components are *interchangeable*.
329 But what should it mean for two components to be interchangeable? Do
330 they really need to use identical input variables and provide identical output
331 variables?

332 To clarify this issue, consider the physical process of infiltration within
333 a hydrologic model. An infiltration component must provide the infiltration
334 rate at the surface, because it represents a loss term in the overall hydro-
335 logic budget. If the domain of the infiltration component is restricted to
336 the unsaturated zone, above the water table, then it may also need to pro-
337 vide a vertical flow rate at the water table boundary. Thus, any infiltration
338 component must provide fluxes at the (top and bottom) boundaries of its
339 domain. To do so, it needs variables such as flow depth and rainfall rate that
340 are outside its domain and computed by another component. Hydrologists
341 use a variety of different methods and approximations to compute surface
342 infiltration rate. The Richards 3D method, for example, is a more rigor-
343 ous approach that tracks four state variables throughout the domain. The
344 Green-Ampt method, on the other hand, makes a number of simplifying as-
345 sumptions, computes a smaller set of state variables, and does not resolve
346 the vertical flow dynamics to the same level of detail (i.e., piston flow, sharp

347 wetting front). As a result, the Richards 3D and Green-Ampt infiltration
348 components use a different set of input variables and provide a different set
349 of output variables. Nevertheless, *they both provide certain, key outputs* such
350 as surface infiltration rate and can therefore be used “interchangeably” in a
351 hydrologic model.

352 Autoconnection of components is important from a user’s point of view.
353 Components typically require many input variables and produce many out-
354 put variables. Users quickly become frustrated when they need to manually
355 create all of these pairings or connections, especially when using more than
356 just two or three components at a time. CSDMS currently employs an ap-
357 proach to autoconnection that involves providing interfaces (i.e., CCA ports)
358 with different names to reflect their intended use (or interchangeability), even
359 though they use the same interface internally.

360 **4. The Basic Model Interface**

361 Most Earth-science models initialize a set of state variables (often as 1D,
362 2D, or 3D arrays) and then execute of series of time steps that advance the
363 variables forward in time according to physical laws (e.g., mass conservation)
364 or some other set of rules. Hence, the underlying source code tends to follow a
365 standard pattern that consists of three main parts. The first part comprises
366 all source code prior to the start of the time loop and serves to set up or
367 *initialize* the model. The second part comprises all source code *within* the
368 time loop and is where the main functionality of the model is implemented
369 to update state variables with each time step. The third part consists of all
370 source code after the end of the time loop and serves to tear down, or *finalize*,

371 the model. Note that root-finding and relaxation algorithms follow a similar
372 pattern even if the iterations do not represent timestepping.

373 All the model coupling projects that we are aware of (including CSDMS,
374 ESMF, OMS, and OpenMI) have developed a component interface that pro-
375 vides initialize, update, and finalize functions. This approach provides a
376 caller with fine-grained control that is essential for model coupling and al-
377 lows the caller to bypass the model’s own time loop.

378 The streamlined approach developed by CSDMS for converting mod-
379 els into plug-and-play components utilizes two new interfaces in a two-level
380 wrapping process. Recall that a component *interface* is simply a named set
381 of functions (called methods) that have been defined completely in terms of
382 their names, arguments, and return values. The first-level interface, called
383 the Basic Model Interface (BMI), is the only one that model contributors are
384 required to implement. The purpose of the BMI is to ensure that the model
385 “fits into” the second-level wrapper, called the CSDMS Component Model
386 Interface (CMI), discussed in the next section. By design, the BMI uses only
387 simple data types and is easy to implement in all of the languages supported
388 by CSDMS. However, it provides all of the information that is needed at the
389 CMI level for plug-and-play modeling.

390 Table 1 summarizes the key BMI functions. BMI.initialize() typically
391 reads data from a configuration file, initializes variables, opens output files,
392 and stores data in a “handle” for non-object-oriented languages. BMI.update()
393 advances the state variables by one model time step. BMI.finalize() typically
394 frees resources and closes files. Several “getter” and “setter” functions, whose
395 details differ between languages, allow the CMI wrapper to retrieve data from

396 and load data into the model. The CMI wrapper must be able to retrieve
 397 anything it needs for model coupling, such as lists of input and output vari-
 398 able names; any variable’s units, rank or data type; and a description of the
 399 model’s grid or mesh. The BMI functions that must be provided to fully
 400 describe a model’s computational grid (not shown) depend on the type of
 401 grid, e.g. uniform, orthogonal curvilinear, or unstructured.

Table 1. Summary of BMI functions.
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.
array<string> get_output_var_names () Return a list of model’s output vars.
string get_attribute (string att_name) Return mesh_type (raster/ugrid), time_step_type (fixed/adaptive), etc.
string get_var_type (string long_var_name) Return variable’s type, e.g. ‘double’, ‘int’
string get_var_units (string long_var_name) Return variable’s units, e.g. ‘meters’ (UD_units standard)
int get_var_rank (string long_var_name) Return variable’s number of dimensions (0 for scalars)
(continued on next page)

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'
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> 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> get_2d_double_at_indices (string long_var_name, array<int> 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> 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> indices, array<double,2> array) # Set values at indices in a 2D array of type double.
Notes: (1) Each BMI function also has a "handle" argument, not shown. (2) long_var_name must comply with the NetCDF CF conventions. (3) Models with 3D arrays provide BMI.get_3d_double(), etc. (4) 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> 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 () # ('driver' or 'nondriver')
void set_mode (string mode)
array<string> get_input_var_names () Return a list of model's input vars (CF convention long_names).
array<string> get_output_var_names () Return a list of model's output vars (CF convention long_names).
string get_var_units (string long_var_name)
string get_attribute (string att_name) Return mesh_type (raster/ugrid), time_step_type (fixed/adaptive), etc.
void setServices () # (used by CCA framework)
void releaseServices () # (used by CCA framework)

402 5. The CSDMS Component Model Interface

403 A model that provides the BMI functions is ready to be converted to a
404 CSDMS plug-and-play component using automated tools that provide it with
405 the CSDMS Component Model Interface. Therefore, model contributors do
406 not need to know anything about the CMI or technical CCA framework con-
407 cepts such as ports. The functions in the CMI allow CSDMS components to
408 communicate and share data with other CSDMS components even if they are

409 written in a different language, use a different grid, use different units, or use
410 a different timestepping scheme. A model wrapped as a CSDMS component
411 becomes an object and can maintain its state variables between method calls,
412 regardless of whether it was written in an object-oriented language.

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

425 In CCA jargon, the “IMPL file” is where all the CMI functions are imple-
426 mented. In our design, the CMI functions make calls only to the interfaces
427 of the entities just discussed. Anything that is needed from the model is
428 obtained through standardized BMI function calls. This means that essen-
429 tially the same IMPL file can be used for all components written in a given
430 language, such as Fortran.

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

434 services. The BMI functions do not call any external entity.

435 Babel is used during compilation to create language bindings, or “glue
436 code”, that allow the component to share data with other CSDMS compo-
437 nents that may have been written in another language. In most cases, CS-
438 DMS components access each other’s state variables through references (pass-
439 by-reference) rather than copying data (pass-by-copy), as this approach re-
440 sults in higher performance. However, since Babel supports Remote Method
441 Invocation (RMI), it is not necessary for CSDMS components to reside in
442 the same address space and they could even be coupled across a network.

443 Table 2 summarizes the key CMI functions. `CMI.initialize()` must get
444 and save the CCA ports it needs to call other components. It then calls
445 `CMI.initialize()` on those components, if necessary, and finally calls `BMI.initialize()`.
446 `CMI.run_for()` first calls `CMI.run_for()` on the components it needs data from
447 and retrieves that data. It then makes as many calls to `BMI.update()`
448 as necessary to span a time interval. Because of this fine-grained control,
449 `CMI.run_for()` can then call service components to do other tasks after each
450 update, such as write data to NetCDF files at a specified interval, advance
451 a progress bar, or check a stopping rule. `CMI.finalize()` releases CCA ports
452 and calls `BMI.finalize()`. `CMI.get_values()` takes a long variable name argu-
453 ment (following the NetCDF CF conventions) and makes calls to various BMI
454 methods to retrieve the requested data. It then uses service components, if
455 necessary, to convert units to those needed by the caller or perform spatial
456 regridding to the caller’s grid. `CMI.set_values()` does unit conversion and
457 regridding, if necessary, before loading variables into the model’s state using
458 BMI methods. A more detailed explanation of the CMI will be provided in

459 a subsequent paper.

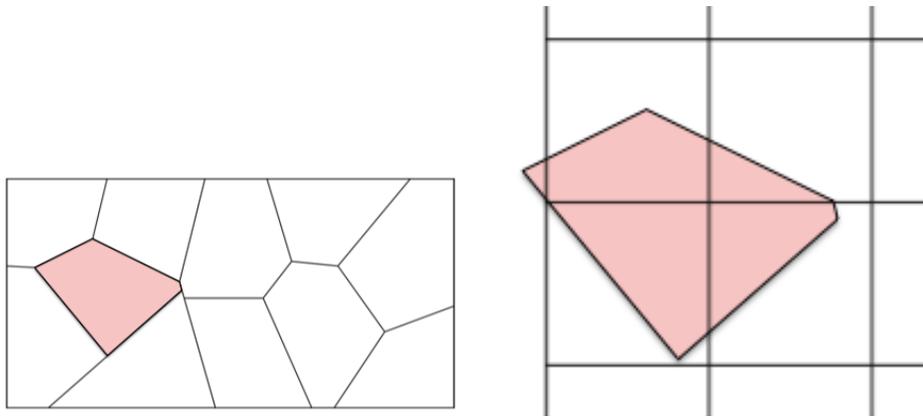
460 **6. Service Components: Tools That Any Component Can Use**

461 It is helpful for certain low-level tools or utilities to be encapsulated in
462 special components called *service components* that can be automatically in-
463 stantiated by a CCA framework on startup. Unlike other components, which
464 users may assemble graphically into larger applications, users do not inter-
465 act with service components directly. However, CMI functions can call the
466 methods of service components through *service ports*. The use of service
467 components allows CSDMS to maintain code for a shared functionality in
468 a single place and to make that functionality available to all components
469 regardless of the implementation language. Any CCA component can be
470 “promoted” to a service component by registering it as such using a CCA
471 port called *gov.cca.ports.ServiceRegistry*.

472 CMI functions call service components for tasks like spatial regridding,
473 as explained previously. An example regridding scenario is shown in Fig-
474 ure 2. CSDMS uses regridding tools from both the ESMF and OpenMI
475 projects. The ESMF regridder is implemented in Fortran and uses multiple
476 processors. It also supports sophisticated options such as mass-conservative
477 interpolation.

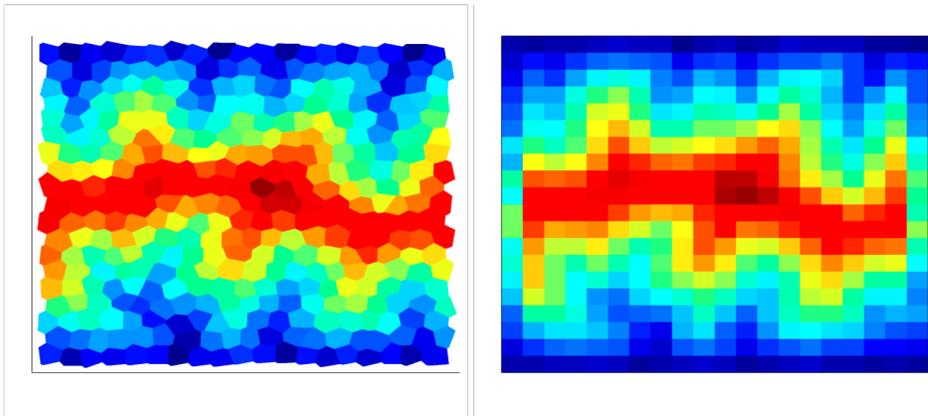
478 **7. Component Wrapping Issues**

479 For a small or simple model, little effort may be needed to rewrite the
480 model in a preferred language and with a particular interface. Rewriting
481 large models, however, is both time-consuming and error prone. In addition,



(a) Voronoi cells.

(b) Intersecting raster and Voronoi cells.



(c) Voronoi cells before regriding.

(d) After regriding to raster cells.

Figure 2: Regriding example.

482 most large models are under continual development, and a rewritten version
 483 will not see the benefits of future improvements. Thus, for code reuse to
 484 be practical, we need a *language interoperability tool* (such as Babel), so
 485 that components do not need to be converted to a different language, and
 486 a *wrapping procedure* to provide existing code with a new calling interface.
 487 Wrapping applies to the “outside” (interface) of a component vs. its “inside”

488 (implementation), so it tends to be a noninvasive and practical way to convert
489 existing models into components.

490 *7.1. Wrapping for Object-Oriented Languages*

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

498 *7.2. Wrapping for Procedural Languages*

499 Languages such as C or Fortran (up to 2003) do not provide object-
500 oriented primitives for encapsulating data and functionality. Because component-
501 based programming requires such encapsulation, the CCA provides a means
502 to produce object-oriented software even in languages that do not support it
503 directly. Specifically, all of the model’s data and functions must be bundled
504 and a suitable namespace created. A model contributor provides implemen-
505 tations for each of the BMI functions. In the case of Fortran, the model’s
506 variables are encapsulated in named modules that are accessed via “use mod-
507 ule_name.” The model is then compiled as a set of shared library functions
508 using Babel.

509 When the model is wrapped with a CMI interface, the data and functions
510 are made accessible to all the other CMI functions by using a “handle” that
511 provides a reference to the model object. This handle is passed as the first

512 argument to each of the interface functions so that they can operate on a
513 particular instance of a model. For example, in C this handle could simply
514 be a pointer to the object, and in Fortran the handle could be an index into
515 a table of opaque objects in a system table. Model handles are allocated and
516 deallocated in the CMI initialize and finalize functions, respectively.

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

524 **8. The CSDMS Modeling Tool**

525 As explained in Section 2.3, Ccaffeine is a CCA-compliant framework
526 for connecting components to create applications. From a user's point of
527 view, Ccaffeine is a low-level tool that executes a sequence of commands
528 in a Ccaffeine script. The commands in the Ccaffeine scripting language
529 are fairly simple, but many people prefer using a GUI because it provides
530 a natural, visual representation of the connected components as boxes with
531 buttons connected by wires. It can also prevent common scripting errors
532 and offer many other convenient features. The CCA Forum developed such
533 a GUI, called Ccafe-GUI, that presented components as boxes in a palette
534 that can be moved into an arena (workspace) and connected by wires. It
535 also allows component configurations and settings to be saved in BLD files

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

540 While Ccafe-GUI was certainly useful, the CSDMS project realized that it
541 could be improved and extended in numerous ways to make it more powerful
542 and more user-friendly. In addition, these changes not only would serve the
543 CSDMS community but could be shared back with the CCA community.
544 The new version, called CMT (CSDMS Modeling Tool), works with any
545 CCA-compliant components, not just CSDMS components. Significant new
546 features of CMT 1.6 include the following.

- 547 • Integration with a powerful visualization tool called VisIt (see below).
- 548 • New, “wireless” paradigm for connecting components (see below).
- 549 • A login dialog that prompts users for remote server login information.
- 550 • Job management tools that are able to submit jobs to processors of a
551 cluster.
- 552 • “Launch and go”: launch a model run on a remote server and then
553 shut down the GUI (the model continues running remotely).
- 554 • New file menu entry: “Import Example Configuration.”
- 555 • A help menu with numerous help documents and links to websites.
- 556 • Ability to submit bug reports to CSDMS.

- 557 • Ability to do file transfers to and from a remote server.
- 558 • Help button in tabbed dialogs to launch component-specific HTML
559 help.
- 560 • Support for droplists and mouse-over help in tabbed dialogs.
- 561 • Support for custom project lists (e.g., projects not yet ready for re-
562 lease).
- 563 • A separate “driver palette” above the component palette.
- 564 • Support for numerous user preferences, many relating to appearance.
- 565 • Extensive cross-platform testing and “bulletproofing.”

566 CMT provides integrated visualization by using VisIt. VisIt [47] is an
567 open-source, interactive, parallel visualization and graphical analysis tool for
568 viewing scientific data. It was developed by the U.S. Department of Energy
569 Advanced Simulation and Computing Initiative to visualize and analyze the
570 results of simulations ranging from kilobytes to terabytes. VisIt was designed
571 so that users can install a client version on their PC that works together with
572 a server version installed on a high-performance computer or cluster. The
573 server version uses multiple processors to speed rendering of large data sets
574 and then sends graphical output back to the client version. VisIt supports
575 about five dozen file formats and provides a rich set of visualization features,
576 including the ability to make movies from time-varying databases. CSDMS
577 uses a service component to provide other components with the ability to
578 write their output to NetCDF files that can be visualized with VisIt. Output

579 can be 0D, 1D, 2D, or 3D data evolving in time, such as a time series (e.g.,
580 a hydrograph), a profile series (e.g., a soil moisture profile), a 2D grid stack
581 (e.g., water depth), a 3D cube stack, or a scatter plot of XYZ triples.

582 Another innovative feature of CMT 1.6 is that it can toggle between the
583 original, *wired* mode and a new *wireless* mode. CSDMS found that displaying
584 connections between components as wires (i.e., red lines) did not scale well to
585 configurations with several components and multiple ports. In wireless mode,
586 a component dragged from the palette to the arena appears to broadcast what
587 it can provide (i.e., CCA provides ports) to other components in the arena
588 (using a concentric circle animation). Any components in the arena that
589 need to use that kind of port get linked automatically to the new one, as
590 indicated using unique, matching colors (Fig. 3).

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

600 **9. Providing Components with a Uniform Help System and GUI**

601 Within a framework where it is easy to connect, interchange, and run
602 coupled models, some users may treat components as black boxes and ignore

603 the underlying assumptions that a component was built upon. They may
604 even couple two components for which coupling does not make sense. To
605 combat this problem, each CSDMS component is bundled with an HTML
606 help document, easily accessible through the CMT, that describes the model
607 it wraps. These documents are standardized to include the following.

- 608 • Extended model description (along with references)
- 609 • Listing and brief description of the component's uses and provides ports
- 610 • Main equations of the model
- 611 • Sample input and output
- 612 • Acknowledgment of the model developer(s)

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

625 10. Conclusions

626 CSDMS has developed a component-based approach to integrated mod-
627 eling that draws on the combined power of several open-source tools such
628 as Babel, Bocca, Ccaffeine, the ESMF regriding tool, and the VisIt visu-
629 alization tool. This noninvasive approach includes the new BMI and CMI
630 interfaces. CSDMS also draws on the combined knowledge and creative effort
631 of a large community of Earth-surface dynamics modelers and computer sci-
632 entists. Using a variety of tools, standards and protocols, CSDMS converts
633 a heterogeneous set of open-source, user-contributed models into a suite of
634 plug-and-play modeling components that can be reused in many different
635 contexts. The CSDMS model repository currently contains more than 160
636 models and tools. Of those, 50 have been converted into components that
637 can be used in coupled modeling scenarios with Ccaffeine or the CMT. An
638 up-to-date list is maintained at csdms.colorado.edu. As with the model repos-
639 itory as a whole, CSDMS components cover the breadth of surface dynamics
640 systems.

641 All the software that underlies CSDMS is installed and maintained on
642 its high-performance cluster. CSDMS members have accounts on this clus-
643 ter and access its resources using a lightweight, Java-based client application
644 called the CSDMS Modeling Tool (CMT) that runs on virtually any desk-
645 top or laptop computer. This approach is a type of *community cloud* since
646 it provides remote access to numerous resources. This centralized cloud ap-
647 proach offers many advantages including (1) simplified maintenance, (2) more
648 reliable performance, (3) automated backups, (4) remote storage and com-
649 putation (user's PC remains free), (5) ability for many components (such

650 as ROMS) and tools (such as VisIt and ESMF's regridder) to use parallel
651 computation, (6) need for users to install only a lightweight client on their
652 PC, (7) less need for technical support, and (8) ability to submit and run
653 multiple jobs.

654 **Acknowledgments**

655 CSDMS gratefully acknowledges major funding through a cooperative
656 agreement with the National Science Foundation (EAR 0621695). Addi-
657 tional work was supported by the Office of Advanced Scientific Computing
658 Research, Office of Science, U.S. Dept. of Energy, under Contracts DE-AC02-
659 06CH11357 and DE-FC-0206-ER-25774.

660 **References**

- 661 [1] Allan, B., Armstrong, R., Lefantzi, S., Ray, J., Walsh, E., Wolfe, P.,
662 2010. Ccaffeine – a CCA component framework for parallel computing.
663 <http://www.cca-forum.org/ccafe/>.
- 664 [2] Allan, B. A., Norris, B., Elwasif, W. R., Armstrong, R. C., Dec. 2008.
665 Managing scientific software complexity with Bocca and CCA. *Scientific*
666 *Programming* 16 (4), 315–327.
- 667 [3] Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes,
668 L., Parker, S., Smolinski, B., 1999. Toward a Common Component Ar-
669 chitecture for high-performance scientific computing. In: *Proc. 8th IEEE*
670 *Int. Symp. on High Performance Distributed Computing*.

- 671 [4] Ashton, A., Kettner, A. J., Hutton, E. W. H., 2011. Progress in coupling
672 between coastline and fluvial dynamics. *Computers & Geosciences* (this
673 issue).
- 674 [5] Ashton, A., Murray, A. B., Arnoult, O., 2001. Formation of coastline
675 features by large-scale instabilities induced by high-angle waves. *Nature*
676 414, 296–300.
- 677 [6] Balay, S., Brown, J., , Buschelman, K., Eijkhout, V., Gropp, W. D.,
678 Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., Zhang,
679 H., 2010. PETSc users manual. Tech. Rep. ANL-95/11 - Revision 3.1,
680 Argonne National Laboratory.
- 681 [7] Balay, S., Brown, J., Buschelman, K., Gropp, W. D., Kaushik, D., Kne-
682 pley, M. G., McInnes, L. C., Smith, B. F., Zhang, H., 2011. PETSc web
683 page. [Http://www.mcs.anl.gov/petsc](http://www.mcs.anl.gov/petsc).
- 684 [8] Balay, S., Gropp, W. D., McInnes, L. C., Smith, B. F., 1997. Effi-
685 cient management of parallelism in object oriented numerical software
686 libraries. In: Arge, E., Bruaset, A. M., Langtangen, H. P. (Eds.), *Modern*
687 *Software Tools in Scientific Computing*. Birkhäuser Press, pp. 163–202.
- 688 [9] Bernholdt D. (PI), 2010. TASCs Center.
689 <http://www.scidac.gov/compsci/TASCs.html>.
- 690 [10] Buttler, H., AprilJune 2005. A guide to the python uni-
691 verse for esri users. *ArcUser Mag.* Available online at
692 <http://www.esri.com/news/arcuser/>.

- 693 [11] CCA Forum, 2010. A hands-on guide to the Common Component Ar-
694 chitecture. <http://www.cca-forum.org/tutorials/>.
- 695 [12] CSDMS, 2011. Community Surface Dynamics Modeling System (CS-
696 DMS). <http://csdms.colorado.edu>.
- 697 [13] CUAHSI, 2011. Consortium of Universities for the Advancement of the
698 Hydrological Sciences Inc. . <http://www.cuahsi.org>.
- 699 [14] Dahlgren, T., Epperly, T., Kumfert, G., Leek, J., 2007. Babel User's
700 Guide. CASC, Lawrence Livermore National Laboratory, UCRL-SM-
701 230026, Livermore, CA.
- 702 [15] de St. Germain, J. D., Morris, A., Parker, S. G., Malony, A. D., Shende,
703 S., May 15-17 2002. Integrating performance analysis in the Uintah
704 software development cycle. In: Proceedings of the 4th International
705 Symposium on High Performance Computing (ISHPC-IV). pp. 190–206.
706 URL <http://www.sci.utah.edu/publications/dav00/ishpc2002.pdf>
- 707 [16] Diachin L. (PI), 2011. Center for Interoperable Tech-
708 nologies for Advanced Petascale Simulations (ITAPS).
709 <http://www.scidac.gov/math/ITAPS.html>.
- 710 [17] EJB, 2011. Enterprise Java Beans Specification.
711 <http://java.sun.com/products/ejb/docs.html>.
- 712 [18] ESMF Joint Specification Team, 2011. Earth System Modeling Frame-
713 work (ESMF) Website. <http://www.earthsystemmodeling.org/>.

- 714 [19] FRAMES, 2011. Framework for Risk Analysis of Multi-Media Environ-
715 mental Systems (FRAMES). <http://mepas.pnl.gov/FRAMESV1/>.
- 716 [20] Hill, C., DeLuca, C., Balaji, V., Suarez, M., da Silva, A., ESMF Joint
717 Specification Team, 2004. The architecture of the Earth System Model-
718 ing Framework. *Computing in Science and Engineering* 6, 18–28.
- 719 [21] Hutton, E. W. H., Syvitski, J. P. M., 2008. Sedflux-2.0: An advanced
720 process-response model that generates three-dimensional stratigraphy.
721 *Computers & Geosciences* 34 (10), 1319–1337.
- 722 [22] Kessler, M. A., Anderson, R. S., Briner, J. P., 2008. Fjord insertion
723 into continental margins driven by topographic steering of ice. *Nature*
724 *Geoscience* 1, 365–369.
- 725 [23] Kettner, A. J., Syvitski, J. P. M., 2008. Hydrotrend version 3.0: a
726 climate-driven hydrological transport model that simulates discharge
727 and sediment load leaving a river system. *Computers & Geosciences*
728 34 (10), 1170–1183.
- 729 [24] Keyes D. (PI), 2011. Towards Optimal Petascale Simulations (TOPS)
730 Center. <http://tops-scidac.org/>.
- 731 [25] Krishnan, S., Gannon, D., April 2004. XCAT3: A framework for CCA
732 components as OGSA services. In: *Proceedings of the 9th International*
733 *Workshop on High-Level Parallel Programming Models and Supportive*
734 *Environments (HIPS 2004)*. IEEE Computer Society, pp. 90–97.
- 735 [26] Kumfert, G., April 2003. Understanding the CCA Specification Using

- 736 Decaf. Lawrence Livermore National Laboratory.
737 URL <http://www.llnl.gov/CASC/components/docs/decaf.pdf>
- 738 [27] Larson, J. W., 2009. Ten organising principles for coupling in multi-
739 physics and multiscale models. ANZIAM Journal 47, C1090–C1111.
- 740 [28] Larson, J. W., Norris, B., 2007. Component specification for parallel
741 coupling infrastructure. In: Gervasi, O., Gavrilova, M. L. (Eds.), Pro-
742 ceedings of the International Conference on Computational Science and
743 its Applications (ICCSA 2007). Vol. 4707 of Lecture Notes in Computer
744 Science. Springer-Verlag, pp. 56–68.
- 745 [29] Lawrence Livermore National Laboratory, 2011. Babel.
746 <http://www.llnl.gov/CASC/components/babel.html>.
- 747 [30] Lucas R. (PI), 2011. Performance Engineering Research Institute
748 (PERI). <http://www.peri-scidac.org>.
- 749 [31] MathWorks, 2011. MATLAB - The Language of Technical Computing.
750 <http://www.mathworks.com/products/matlab/>.
- 751 [32] Message Passing Interface Forum, 1998. MPI2: A message passing in-
752 terface standard. High Performance Computing Applications 12, 1–299.
- 753 [33] NET, 2011. Microsoft .NET Framework.
754 <http://www.microsoft.com/net/>.
- 755 [34] NetCDF, 2011. NetCDF. <http://www.unidata.ucar.edu/packages/netcdf>.
- 756 [35] OMP, 2011. Object Modeling System v3.0.
757 <http://www.javaforge.com/project/oms>.

- 758 [36] Ong, E. T., Larson, J. W., Norris, B., Jacob, R. L., Tobis, M., Steder,
759 M., 2008. A multilingual programming model for coupled systems. In-
760 ternational Journal for Multiscale Computational Engineering 6, 39–51.
- 761 [37] Open Geospatial Consortium, 2011. OGC Standards and Specifications.
762 <http://www.opengeospatial.org/>.
- 763 [38] Parker, G., 2011. 1d sediment transport morphodynam-
764 ics with applications to rivers and turbidity currents.
765 http://vtchl.uiuc.edu/people/parkerg/morphodynamic_e-book.htm.
- 766 [39] Peckham, S., 2008. Geomorphometry and spatial hydrologic modeling.
767 Vol. 33 of Geomorphometry: Concepts, Software and Applications. De-
768 velopments in Soil Science. Elsevier, Ch. 25, pp. 579–602.
- 769 [40] Peckham, S. D., Goodall, J. L., 2011. Driving plug-and-play compo-
770 nents with data from web services: A demonstration of interoperability
771 between CSDMS and CUAHSI-HIS. Computers & Geosciences (this is-
772 sue).
- 773 [41] Rosen, L., 2004. Open Source Licensing: Software Freedom and Intellec-
774 tual Property Law. Prentice Hall, <http://rosenlaw.com/oslbook.htm>.
- 775 [42] T. Oliphant et al., 2011. Scientific Computing Tools for Python –
776 NumPy. <http://numpy.scipy.org/>.
- 777 [43] The MCT Development Team, 2006. Model Coupling Toolkit (MCT)
778 Web Site. <http://www.mcs.anl.gov/mct/>.

- 779 [44] The OpenMI Association, 2011. The Open Modeling Interface
780 (OpenMI). <http://www.openmi.org>.
- 781 [45] Tucker, G. E., Lancaster, S. T., Gasparini, N. M., Bras, R. L., 2001. The
782 Channel-Hillslope Integrated Landscape Development (CHILD) Model.
783 Academic/Plenum Publishers, pp. 349–388.
- 784 [46] United States Department of Energy, 2011. SciDAC Initiative homepage.
785 <http://scidac.gov/>.
- 786 [47] VisIt, 2011. VisIt. <http://wci.llnl.gov/codes/visit>.
- 787 [48] XML, 2011. Extensible Markup Language (XML).
788 <http://www.w3.org/XML/>.

789

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory (“Argonne”). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.

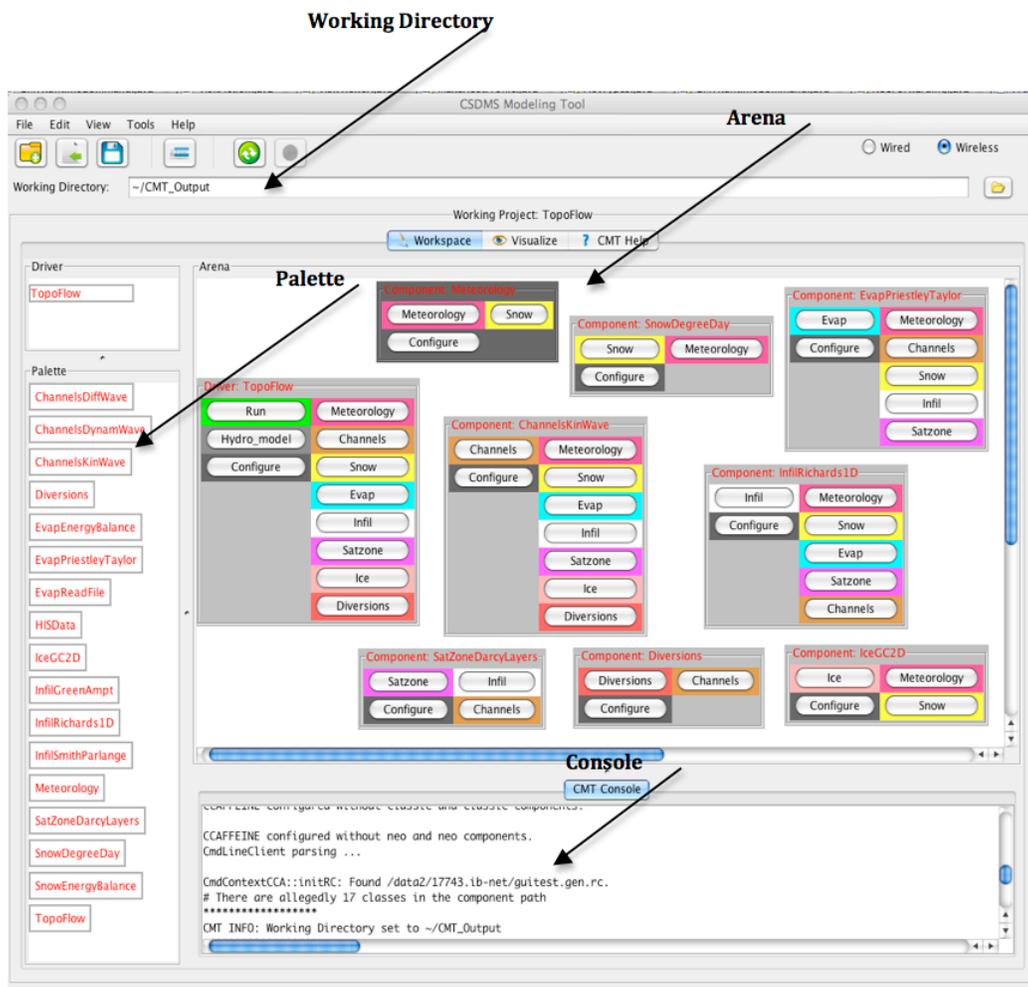


Figure 3: Hydrologic model assembled in the CMT by dragging interchangeable process components from the palette to the arena.