A Component-Based Approach to Integrated Modeling in the Geosciences

Scott Peckham, Eric Hutton

CSDMS, University of Colorado, Boulder, CO, USA

Boyana Norris

Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA

Abstract

1. Introduction

The Community Surface Dynamics Modeling System (CSDMS) Project is an NSF-funded, international effort to develop a suite of modular numerical models able to simulate the evolution of landscapes and sedimentary basins, on time scales ranging from individual events to many millions of years. To enable simulation of complex scenarios, CSDMS provides comprehensive software and educational infrastructure that enables the independently developed models to be integrated into coupled numerical simulations. To achieve this, CSDMS has adopted a component software development model and created a suite of tools that make creation of components as automated and effortless as possible.

The Community Surface Dynamics Modeling System (CSDMS) is community software project that is funded through a cooperative agreement with the National Science Foundation (NSF). A major goal of this project is to serve a diverse community of surface dynamics modelers by providing resources to promote the sharing and re-use of high-quality, open-source modeling software. In support of this goal, the CSDMS Integration Office maintains a searchable inventory of contributed models and employs state-of-the-art software tools that make it possible to convert stand-alone models into flexible, "plug-and-play components that can be assembled into larger applications. The CSDMS project also has a mandate from the NSF to provide a migration pathway for surface dynamics modelers towards high-performance computing (HPC) and provides a 720-core supercomputer for use by its members.

The CSDMS project is happy to accept open-source code contributions from the modeling community in any programming language and in whatever form it happens to be in. One of our key goals is to create an inventory of what models are available. We use an online questionnaire to collect basic information about different open-source models and we make this information available to anyone

who visits our website at csdms.colorado.edu. We can also serve as a repository for model source code, but in many cases our website instead redirects visitors to another website which may be a website maintained by the model developer or a source code repository like SourceForge, JavaForge or Google Code. Online source code repositories (or project hosting sites) like these are free and provide developers with a number of useful tools for managing collaborative software development projects. CSDMS does not aim to compete with the services that these repositories provide (e.g. version control, issue tracking, wikis, online chat and web hosting).

Another key goal of the CSDMS project is to create a collection of open-source, earth-science modeling components that are designed so that they are relatively easy to reuse in new modeling projects. CSDMS has studied this problem and has examined a number of different technologies for addressing it. We have learned that there are certain fundamental design principles that are common to all of these model-coupling technologies. That is, there is a certain minimum amount of code refactoring that is necessary in order for a model to be usable as a *plug-and-play* component.

1.1. Component Programming Concepts

Component-based programming is all about bringing the advantages of "plug and play" technology into the realm of software. When you buy a new peripheral for your computer, such as a mouse or printer, the goal is to be able to simply plug it into the right kind of port (e.g., a USB, serial or parallel port) and have it work, right out of the box. In order for this to be possible, however, there has to be some kind of published standard that the makers of peripheral devices can design against. For example, most computers nowadays have USB ports, and the USB (Universal Serial Bus) standard is well-documented. A computer's USB port can always be expected to provide certain capabilities, such as the ability to transmit data at a particular speed and the ability to provide a 5-volt supply of power with a maximum current of 500 mA. The result of this "standardization" is that it is usually pretty easy to buy a new device, plug it into your computer's USB port and start using it. Software "plug-ins" work in a similar manner, relying on the existence of certain types of "ports" that have certain, well-documented structure or capabilities. In software, as in hardware, the term *component* refers to a unit that delivers a particular type of functionality and that can be "plugged in".

Component programming builds upon the fundamental concepts of object-oriented programming, with the main difference being the introduction or presence of a *framework*. Components are generally implemented as classes in an object-oriented language, and are essentially "black boxes" that encapsulate some useful bit of functionality. They are often present in the form of a library file, that is, a shared object (so file in Unix) or a dynamically linked library (dll file in Windows or dylib file in Mac OS X). The purpose of a framework is to provide an environment in which components can be linked together to form applications. The framework provides a number of *services* that are accessible to all components, such as the linking mechanism itself. Often, a framework will

also provide a uniform method of trapping or handling exceptions (i.e., errors), keeping in mind that each component will throw exceptions according to the rules of the language that it is written in. In some frameworks, (e.g., CCA's Ccaffeine) there is a mechanism by which any component can be promoted to a framework service. It is therefore preloaded by the framework so that it is available to all components without the need to explicitly "import" or "link" to the service component.

One thing that often distinguishes components from ordinary subroutines, software modules or classes is that they are able to communicate with other components that may be written in a different programming language. This problem is referred to as *language interoperability*. In order for this to be possible, the framework must provide some kind of language interoperability tool that can create the necessary "glue code" between the components. For a CCA-compliant framework, that tool is Babel, and the supported languages are C, C++, Fortran (all years), Java and Python. Babel is described in more detail in a subsequent section. For Microsoft's .NET framework, that tool is CLR (Common Language Runtime) which is an implementation of an open standard called CLI (Common Language Infrastructure), also developed by Microsoft. Some of the supported languages are C# (a spin-off of Java), Visual Basic, C++/CLI, IronLisp, IronPython and IronRuby. CLR runs a form of bytecode called CIL (Common Intermediate Language). Note that CLI does not appear to support Fortran, Java, standard C++ or standard Python. The Java-based frameworks used by Sun Microsystems are JavaBeans and Enterprise JavaBeans (EJB). In the words of Armstrong et al. [**?**]: *"Neither JavaBeans nor EJB directly addresses the issue of language interoperability, and therefore neither is appropriate for the scientific computing environment. Both JavaBeans and EJB assume that all components are written in the Java language. Although the Java Native Interface [34] library supports interoperability with C and C++, using the Java virtual machine to mediate communication between components would incur an intolerable performance penalty on every inter-component function call.* While in recent years the performance of Java codes has improved steadily through just-in-time (JIT) compilation into native code, Java is not yet available on key high-performance platforms such as the IBM Blue Gene/L and Blue Gene/P supercomputers.

Some key advantages of component-based programming are that components:

- can be written in different languages and still communicate (via language interoperability).
- can be replaced, added to or deleted from an application at run-time via dynamic linking (as precompiled units).
- can easily be moved to a remote location (different address space) without recompiling other parts of the application (via RMI/RPC support).
- can have multiple different interfaces and can "have state" or be "stateful," which simply means that data stored in the fields of an "component

object" (class instance) retain their values between method calls for the lifetime of the object.

- can be customized with configuration parameters when an application is built.
- provide a clear specification of inputs needed from other components in the system.
- have the potential to encapsulate parallelism better.
- allow multicasting calls that do not need return values (i.e. sending data to multiple components simultaneously).
- provide clean separation of functionality (for components, this is mandatory vs. optional)
- facilitate code re-use and rapid comparison of different implementations, etc.
- facilitate efficient cooperation between groups, each doing what they do best.
- promote economy of scale through development of community standards.

2. Background

2.1. The Common Component Architecture (CCA)

Common Component Architecture (CCA) [**?**] is a *component architecture standard* adopted by federal agencies (largely the Department of Energy and its national labs) and academics to allow software components to be combined and integrated for enhanced functionality on high-performance computing systems. The CCA Forum is a grassroots organization that started in 1998 to promote component technology standards (and code re-use) for HPC. CCA defines standards necessary for the interoperation of components developed in the context of different frameworks. That is, software components that adhere to these standards can be ported with relative ease to another CCA-compliant framework. While there are a variety of other component architecture standards in the commercial sector (e.g., CORBA, COM, .Net, JavaBeans, etc.) CCA was created to fulfill the needs of scientific, high-performance, open-source computing that are unmet by these other standards. For example, scientific software needs full support for complex numbers, dynamically dimensioned multidimensional arrays, Fortran (and other languages) and multiple processor systems. Armstrong et al. explain the motivation to create CCA by discussing the pros and cons of other component-based frameworks with regard to scientific, high-performance computing.

There are a number of large DOE projects, many associated with the SciDAC program (Scientific Discovery through Advanced Computing), that are devoted

to the development of component technology for high-performance computing systems. Most of these are heavily invested in the CCA standard (or are moving toward it) and employ computer scientists and applied mathematicians. Some examples include:

- TASCS = The Center for Technology for Advanced Scientific Computing Software, which focuses on CCA and its associated tools [**?**]
- CASC = Center for Applied Scientific Computing, which is home to CCA's Babel tool [**?**]
- ITAPS = The Interoperable Technologies for Advanced Petascale Simulation [**?**], which focuses on meshing and discretization components, formerly TSTT
- PERI = Performance Engineering Research Institute, which focuses on HPC quality of service and performance issues [**?**]
- TOPS = Terascale Optimal PDE Solvers, which focuses on PDE solver components [**?**]
- PETSc = Portable, Extensible Toolkit for Scientific Computation, which focuses on linear and nonlinear PDE solvers for HPC, using MPI [**? ? ?**]

There are a variety of different frameworks that adhere to the CCA component architecture standard, such as Ccaffeine, CCAT/XCAT, SciRUN and Decaf. A framework can be CCA-compliant and still be tailored to the needs of a particular computing environment. For example, Ccaffeine was designed to support parallel computing and XCAT was designed to support distributed computing. Decaf was designed by the developers of Babel (Kumfert, 2003) primarily as a means of studying the technical aspects of the CCA standard itself. The important thing is that each of these frameworks adheres to the same standard, which makes it much easier to re-use a (CCA) component in another computational setting. The key idea is to try to isolate the components themselves, as much as possible, from the details of the computational environment in which they are deployed. If this is not done, then we fail to achieve one of the main goals of component programming, which is code re-use.

CCA has been shown to be interoperable with ESMF [**?**] and MCT [**? ? ? ?**]. CSDMS has also demonstrated that it is interoperable with a Java version of OpenMI. Many of the papers in our cited references have been written by CCA Forum members and are helpful for learning more about CCA. The CCA Forum has also prepared a set of tutorials called "A Hands-On Guide to the Common Component Architecture" [**?**].

2.2. What is Babel?

Babel is an open-source, language interoperability tool (and compiler) that automatically generates the "glue code" that is necessary in order for components written in different computer languages to communicate. As illustrated in

Fig. **??**, Babel currently supports C, C++, Fortran (77, 90, 95 and 2003), Java and Python. Babel is much more than a "least common denominator" solution; it even enables passing of variables with data types that may not normally be supported by the target language (e.g., objects, complex numbers). Babel was designed to support *scientific*, *high-performance* computing and is one of the key tools in the CCA tool chain. It won an R&D 100 design award in 2006 for "The world's most rapid communication among many programming languages in a single application." It has been shown to outperform similar technologies such as CORBA and Microsoft's COM and .NET.

In order to create the glue code that is needed in order for two components written in different programming languages to communicate (i.e. pass data between them), Babel only needs to know about the interfaces of the two components. It does not need any implementation details. Babel was therefore designed so that it can ingest a description of an interface in either of two fairly "language neutral" forms, XML (eXtensible Markup Language) or SIDL (Scientific Interface Definition Language). The SIDL language (somewhat similar to the CORBA's IDL) was developed for the Babel project. Its sole purpose is to provide a concise description of a scientific software component interface. This interface description includes complete information about a component's interface, such as the data types of all arguments and return values for each of the component's methods (or member functions). SIDL has a complete set of fundamental data types to support scientific computing, from booleans to double precision complex numbers. It also supports more sophisticated data types such as enumerations, strings, objects, and dynamic multi-dimensional arrays. The syntax of SIDL is very similar to Java. A complete description of SIDL syntax and grammar can be found in "Appendix B: SIDL Grammar" in the Babel User's Guide [**?**]. Complete details on how to represent a SIDL interface in XML are given in "Appendix C: Extensible Markup Language (XML)" of the same user's guide.

2.3. The Ccaffeine Framework

Ccaffeine [**?**] is the most widely used CCA framework, providing the runtime environment for sequential or parallel components applications. Using Ccaffeine, component-based applications can run on a variety of platforms, including laptops, desktops, clusters, and leadership-class supercomputers. Ccaffeine provides some rudimentary MPI communicator services, although individual components are responsible for managing parallelism internally, e.g., communicating data to and from other distributed components. A CCA framework provides so-called *services*, which include component instantiation and destruction, connecting and disconnecting ports, handling of input parameters, and control of MPI communicators. Ccaffeine was primarily designed to support the single-component multiple-data (SCMD) programming style, although it can support multiple-component multiple-data (MCMD) applications that implement more dynamic management of parallel resources. The CCA specification also includes an event service description, but it is not fully implemented in Ccaffeine yet. Multiple interfaces to configuring and executing component

Figure 1: Language interoperability provided by Babel.

applications within the Ccaffeine framework exist, including a simple scripting language, a graphical user interface, and the ability to take over some of the operations normally handled by the frameworks, such as component instantiation and port connections.

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

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

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

2.4. What is Bocca?

Bocca [**?**] is a tool in the CCA tool chain that was designed to help users create, edit, and manage a set of SIDL-based entities, including CCA components and ports, that are associated with a particular project. Once you have prepared a set of CCA-compliant components and ports, you can then use a CCA-compliant framework like Ccaffeine (see above) to actually link components from this set together to create applications or composite models.

Bocca was developed to address usability concerns and reduce the development effort required for implementing multi-language component applications. Bocca was specifically designed to free users from mundane, time-consuming, low-level tasks so they can focus on the scientific aspects of their applications. It can be viewed as a development environment tool that allows application developers to perform rapid component prototyping while maintaining robust software- engineering practices suitable to HPC environments. Bocca provides project management and a comprehensive build environment for creating and managing applications composed of CCA components. Bocca operates in a language-agnostic way by automatically invoking the lower-level Babel tool. A set of Bocca commands required to create a component project can be saved as a shell script, so that the project can be rapidly rebuilt, if necessary. Various aspects of an existing component project can also be modified by typing Bocca commands interactively at a Unix command prompt. While Bocca automatically generates dynamic libraries, a separate tool can be used to create *stand-alone executables* for projects by automatically bundling all required libraries on a given platform. Examples of using Bocca are available in the set of tutorials called "A Hands-On Guide to the Common Component Architecture", written by the CCA Forum members [**?**].

2.5. Other Component-Based Modeling Projects

- OMS (Object Modeling System)
- OpenMI (Open Modeling Interface)
- ESMF (Earth System Modeling Framework)
- EPAs FRAMES ??
- CUAHSI-HIS ?? (Consortium of Universities for the Advancement of Hydrologic Science, Inc. - Hydrologic Information System)
- iemhub.org

3. Problem Definition – Component-based, Plug-and-play Modeling

3.1. Why is it Often Difficult to Link Models?

Linking together models that were not specifically designed from the outset to be linkable is often surprisingly difficult and a brute force approach to the problem often requires a large time investment. The main reason for this is that

there are a lot of ways in which two models may differ. The following list of possible differences helps to illustrate this point.

- Written in different languages (conversion is time-consuming and error-prone).
- The person doing the linking may not be the author of either model and the code is often not well-documented or easy to understand.
- Models may have different dimensionality (1D, 2D or 3D)
- Models may use different types of grids (e.g., rectangles, triangles,Voronoi cells)
- Each model has its own time loop or "clock".
- The numerical scheme may be either explicit or implicit.

The type of coupling required poses its own challenges. Some common types of model coupling are:

- Layered = A vertical stack of grids that may represent:
  different domains (e.g atm-ocean, atm-surf-subsurf, sat-unsat),
  subdivision of a domain (e.g stratified flow, stratigraphy),
  different processes (e.g., precip, snowmelt, infil, seepage, ET)
  A good example is a distributed hydrologic model.
- Nested = Usually a high-resolution (and maybe 3D) model that is embedded within (and may be driven by) a lower-resolution model.
  (e.g., regional winds/waves driving coastal currents, or a 3D channel flow model within a landscape model)
- Boundary-coupled = Model coupling across a natural (possibly moving) boundary, such as a coastline. Usually fluxes must be shared across the boundary.

3.2. Attributes of Earth Surface Process Models

The earth surface process modeling community has *lots* of models but it is difficult to couple or reconfigure them to solve new problems. This is because they are an inhomogeneous set in the sense that:

- They are written in *many different languages*, some object-oriented, some procedural, some compiled, some interpreted, some proprietary, some open-source, etc. These languages dont all offer the same data types and other features, so special tools are required to create "glue code" necessary to make function calls across the *language barrier*.
- They werent designed to "talk" to each other and dont follow any particular set of conventions.

- We cant rewrite all of them (rewriting/debugging is too costly)
- Some models use explicit timestepping, some implicit.
- Earth surface process models generally have a *geographic* context, and are therefore often used in conjunction with GIS (Geographic Information System) tools.
- Models generally consist of one or more arrays (1D, 2D or 3D) that are being advanced in time according to differential equations or other rules. (i.e. we are not modeling molecular dynamics)
- Models use different input and output file formats.
- Many earth surface process models are *open-source*. Even many models that were originally sold commercially are now available as open-source code (e.g., Delt3D - Flow, Mor, etc.; EDF model code)

3.3. Design Criteria

Some of the "functional specs" or "technical goals" of component-based modeling are:

- Support for *multiple operating systems* (especially Linux, Mac OS X and Windows)
- *Language interoperability* to support code contributions written in C and Fortran as well as object-oriented languages (e.g., Java, C++, Python)
- Support for both legacy code (non-protocol) and more structured code submissions ("procedural" and object-oriented)
- Support for both structured and unstructured grids. As a result, there is a need for a spatial regridding tool.
- *Platform-independent GUIs and graphics* where useful
- Use well-established, open-source *software standards* whenever possible (e.g., CCA, SIDL, OGC, MPI, netCDF, OpenDAP, XUL, etc.)
- Make use of *open-source tools* that are mature and have well-established communities. Avoid dependencies on proprietary software whenever possible (e.g., Windows, C#, MatLab.
- Model developers/contributors should not be required to make major changes to how they work. That is, to the extent possible, they should not be required to (1) convert to another programming language, (2) use invasive changes to their code (e.g., use specified data structures, libraries or classes). It is desirable for the author to retain "ownership" of the code, make continual improvements to it and for someone to be able to componentize future, improved versions with minimal additional effort.

However, some degree of code refactoring (e.g., breaking code into functions or adding a few new functions) and ensuring that the code compiles with an open-source compiler are considered reasonable requirements. It is also expected that many developers will take advantage of various built-in tools if it is straight-forward and beneficial to do so.

- Support for *parallel computation* (multi-processor, via MPI standard)
- *Interoperability with other coupling frameworks.* Since code re-use is a fundamental tenet of component-based modeling, the effort required to use a component in another framework should be kept to a minimum.
- The modeling system should be easy to maintain and as robust as possible. It is recognized that the system will have many software dependencies and that this software infrastructure will be updated on a regular basis.
- If the modeling system runs on a multi-processor machine, it should strive to use tools that can take advantage of this. (e.g., VisIt, PETSc, ESMF regridding tool)

3.4. Interface vs. Implementation

The word *interface* may be the most overloaded word in computer science. In each case, however, it adheres to the standard, English meaning of the word that has to do with a boundary between two things and what happens at the boundary. The overloading has to do with the large numbers of pairs of things that we could be talking about. Many people hear the word interface and immediately think of the interface between a human and a computer program, which is typically either a command-line interface (CLI) or a graphical user interface (GUI). While this is a very interesting and complex subject in itself, this is usually not what computer scientists are talking about. Instead, they tend to be interested in other types of interface, such as the one between a pair of software components, or between a component and a framework, or between a developer and a set of utilities (i.e. an API or a software development kit (SDK)).

Within the present context of component programming, we are primarily interested in the interfaces between components. In this context, the word *interface* has a very specific meaning, essentially the same as how it is used in the Java programming language. (You can therefore learn more about component interfaces in a Java textbook.) It is a user-defined entity/type, very similar to an abstract class. It does not have any data fields, but instead is a named set of methods or member functions, each defined completely with regard to argument types and return types but without any actual implementation. A CCA *port* is simply this type of interface. Interfaces are the name of the game when it comes to the question of re-usability or "plug and play". Once an interface has been defined, one can then ask the question: Does this component have interface A? To answer the question we merely have to look at the methods (or member functions) that the component has with regard to their names,

argument types and return types. If a component does "have" a given interface, then it is said to *expose* or *implement* that interface, meaning that it contains an actual *implementation* for each of those methods. It is perfectly fine if the component has additional methods beyond the ones that comprise a particular interface. Because of this, it is possible (and frequently useful) for a single component to expose multiple, different interfaces or ports. For example, this may allow it to be used in a greater variety of settings. There is a good analogy in computer hardware, where a computer or peripheral may actually have a number of different ports (e.g., USB, serial, parallel, ethernet) to enable them to communicate with a wider variety of other components.

The distinction between *interface* and *implementation* is an important theme in computer science. The word pair *declaration* and *definition* is used in a similar way. A function (or class) declaration tells us what the function does (and how to interact with or use it) but not how it works. To see how the function actually works, we need to look at how it has been defined or implemented. C and C++ programmers will be familiar with this idea, where variables, functions, classes, etc. are often declared in a header file with the filename extension .h or .hpp (that is, data types of all arguments and return values are given) whereas they are defined in a separate file with extension .c or .cpp. Of course, most of the gadgets that we use every day (from iPods to cars) are like this. We need to understand their interfaces in order to use them (and interfaces are often standardized across vendors), but often we have no idea what is happening inside or how they actually work, which may be quite complex.

It is important to realize that the CCA standard and the tools in the CCA tool chain are powerful and quite general but they do not provide us with an interface for linking models. In CCA terminology, the term *port* is essentially a synonym for interface and a distinction is made between ports that a given component uses (*uses ports*) and those that it provides (*provides ports*) to other components. Each scientific modeling community that wishes to make use of the CCA tools is responsible for designing or selecting component interfaces (or ports) that are best suited to the kinds of models that they wish to link together. This is a big job in and of itself that involves social as well as technical issues and typically requires a significant time investment. In some disciplines, such as molecular biology or fusion research, the models may look quite different from ours. Ours tend to follow the pattern of a 1D, 2D or 3D array of values (often multiple, coupled arrays) advancing forward in time. However, our models can still be quite different from each other with regard to their dimensionality or the type of computational grid that they use (e.g., rectangles, triangles or polygons), or whether they are implicit or explicit in time.

3.5. Granularity

While components may represent any level of granularity from a simple function to a complete hydrologic model, the optimum level appears to be that of a particular physical process, such as infiltration, evaporation or snowmelt. It is at this level of granularity that researchers are most often interested in "swapping out one method of modeling a process for another. A simpler method

of parameterizing a process may only apply to simplified special cases or may be used simply because there is insufficient input data to drive a more complex model. A different numerical method may solve the same governing equations with greater accuracy, stability or efficiency, and may or may not use multiple processors. Even the same method of modeling a given process may exhibit improved performance when coded in a different programming language. But the physical process level of granularity is also natural for other reasons. Specific physical processes often act within a domain that shares a physically important boundary with other domains (e.g., coastline, ocean-atmosphere) and the fluxes between these domains are often of key interest. In addition, experience shows that this level of granularity corresponds to GUIs and HTML help pages that are more manageable for users.

A judgement call is frequently needed to decide whether a new feature should be provided in a separate component or as a configuration setting in an existing component. For example, a kinematic wave channel-routing component may provide both Mannings formula and the law of the wall as different options to parameterize frictional momentum loss. Each of these options requires is own set of input parameters (e.g., Mannings n or the roughness parameter, z0). We could even think of frictional momentum loss as a separate physical process,under which we would have separate Manning and law of the wall components. Usually, the amount of code associated with the option and usability considerations (including user expectations) can be used to make these decisions.

Some models are written in such a way that decomposing them into separate process components is not really appropriate. This may be because of some special aspect of the models design or because doing so would result in an unacceptable loss of performance (e.g., speed, accuracy or stability). For example, *multiphysics models* — such as PIHM (Penn State? Integrated Hydrologic Model) — represent many physical processes as one large, coupled set of ODEs that are then solved as a matrix problem on a supercomputer. Other models involve several physical processes that operate in the same domain and are relatively tightly coupled within the governing equations. ROMS (Regional Ocean Modeling System) is an example of such a model, in which it may not be practical to model processes such as tides, currents, passive scalar transport (e.g., T and S) and sediment transport within separate components. In such cases, however, it may still make sense to wrap the entire model as a component so that it may interact with other models (e.g., an atmospheric model, like WRF, or a wave model, like SWAN) or be used to drive another model (e.g., a Lagrangian transport model, like LTRANS).

4. Designing a Modeling Interface

While interface functions are often easy to implement, they allow a caller to have fine-grained control over our model, and therefore use it in clever ways as part of something bigger. In essence, this set of methods is like a handheld remote control for our model. Compiling models in this form as shared objects

(.so, Unix) or dynamically-linked libraries (.dll, Windows) is one way that they can then be used as plug-ins.

4.1. The "IRF" Interface Functions

One "universal truth" of component-based programming is that in order for a model to be used as a component in another model, its interface must allow complete control to be handed to an external caller. Most earth-science models have to be initialized in some manner and then use time stepping or another form of stepping in order to compute a result. While time-stepping models are the most familiar, many other problems such as root-finding and relaxation methods employ some type of iteration or stepping. For maximum plug-and-play flexibility, it is necessary to make the actions that take place during a single step directly accessible to a caller.

To see why this is so, consider two time-stepping models. Suppose that Model A melts snow, routes the runoff to a lake, and increases the depth of the lake while Model B computes lake-level lowering due to evaporative loss. Each model initializes the lake depth, has its own time loop and changes the lake depth. If each of these models is written in the traditional manner, then combining them into a single model means that whatever happens inside the time loop of Model A must be pasted into Model B's time loop or vice versa. There can only be one time loop. This illustrates, in its simplest form, a very common problem that is encountered when linking models

Now imagine that we restructure the source code of both models slightly so that they each have their own Initialize(), Run_Until() and Finalize() subroutines. The Initialize routine contains all of the code that came before the time loop in the original model, the Run_Until() routine contains the code that was *inside* the time loop (and returns all updated variables) and the Finalize() routine contains the code that came after the time loop. We can also write one additional routine, perhaps called Run_Model() or Main(), which simply calls Initialize(), starts a time loop which calls Run_Until() and then calls Finalize(). Calling Run_Model() reproduces the functionality of the original model, so we have made a fairly simple, one-time change to our two models and yet retained the ability to use them in "stand-alone mode". Future enhancements to the model simply insert new code into this new set of four subroutines. However, this simple change means that, in effect, we have converted each model into an object with a standard set of four member functions or methods. Now, it is trivial to write a new model that combines the computations of Model A and B. This new model first calls the Initialize() methods of Model A and B, then starts a time loop, then calls the Run_Until() methods of Model A and B, and finally calls the Finalize() methods of Model A and B. For models written in an object-oriented language, these four subroutines would be methods (or member functions) of a class, but for other languages, like Fortran, it is enough to simply break the model into these subroutines.

For lack of a better term, we refer to this Initialize(), Run_Until(), Finalize() pattern as an "IRF interface". All of the model coupling projects that we are aware of use this pattern as part of their component interface, including

the Earth Surface Modeling Framework (ESMF), the Object Modeling System (OMS), the Open Modeling Interface (OpenMI) and the Community Surface Dynamics Modeling System (CSDMS). An IRF interface is also used as part of MPI (Message Passing Interface) for communication between processes in high-performance computers. However, these are really only the core functions of a model coupling interface. A complete interface will also require several other functions, such as ones that enable another component to request data from the component (a getter) or change data values (a setter) in the component. These are typically called within the Initialize() or Run_Until() methods.

4.2. The Getter and Setter Interface Functions

The Get_Value() and Set_Value() methods can be general in terms of supporting different grid/mesh types, but it should be possible to bypass that generality and use simple, raster-based grids to keep things simple and efficient when the generality is not needed.

The Get_Value() and Set_Value() methods should optionally allow specification (via indices) of which individual elements within an array that are to be obtained or modified. We often need to manipulate just a few values and we dont want to transfer copies of entire arrays (which may be quite large) unless absolutely necessary.

Each component should understand what variables will be requested from it and if those represent some function of its state variables (e.g., a sum or product) then that computation should be done by the component and offered as an output variable rather than passing several state variables that must then be combined in some way by the caller.

To support dynamically-typed languages like Python, additional interface functions may be required in order to query whether the variable is currently a scalar or a vector (1D array) or a grid.

4.3. Self-Descriptive Interface Functions

It is also helpful to have two additional methods, perhaps called Get_Input_Exchange_Item_List() and Get_Output_Exchange_Item_List(), that a caller can use to query what type of data the component is able to use as input or compute as output.

4.4. Framework Interface Functions

In order for a component to be "CCA compliant," it must implement some additional member functions that allow it to be instantiated by and communicate with a CCA-compliant framework. These are: __init__, getServices, releaseServices, .

4.5. The Auto-Connection Problem

A key goal of component-based modeling is to create a collection of components that can be coupled together to create new and useful composite models. This particular goal is achieved by simply providing every component with the

same interface, and this is the approach that is employed by OpenMI. A secondary goal, however, is for the coupling process to be as automatic as possible, that is, to require as little input as possible from users. To achieve this goal, we need some way to group components into categories according to the functionality that they provide. This grouping must be readily apparent to both a user and the framework (or system) so that it is clear whether a particular pair of components are *interchangeable* or not. But what should it mean for two components to be interchangeable? Do they really need to use identical input variables and provide identical output variables? Our experience shows that this definition of interchangeable is unnecessarily strict. It also shows that the concept of interchangeability is closely related to the idea of granularity.

To bring these issues into sharper focus, consider the physical process of infiltration which plays a key role in hydrologic models. As part of a larger hydrologic model, the main purpose of an infiltration component is to compute the infiltration rate at the surface because it represents a loss term in the overall hydrologic budget. If the domain of the infiltration component is restricted to the unsaturated zone, above the water table, then it may also need to provide a vertical flow rate at the water table boundary. So the main job of the infiltration component is to provide fluxes at the (top and bottom) boundaries of its domain. To do this job, it needs variables such as flow depth and rainfall rate that are outside of its domain and computed by another component. Hydrologists use a variety of different methods and approximations to compute surface infiltration rate. The Richards 3D method, for example, is a more rigorous approach that tracks four different state variables throughout the domain, while the Green-Ampt method, makes a number of simplifying assumptions so that it computes a smaller set of state variables and does not resolve the vertical flow dynamics to the same level of detail (i.e. piston flow, sharp wetting front). As a result, the Richards 3D and Green-Ampt infiltration components use a different set of input variables and provide a different set of output variables. Despite this, however, they both provide the surface infiltration rate as one of their outputs and can therefore be used "interchangeably in a hydrologic model as an " infiltration component.

The infiltration process example is a good one in that it illustrates several key points that are transferable to other situations. It is often the case that a model, such as a hydrologic model, breaks the larger problem domain into a set of subdomains where one or more processes are relevant. The boundaries of these subdomains are often physical interfaces, such as surface/subsurface, unsaturated/saturated zone, atmosphere/ocean, ocean/seafloor, land/water. Moreover, the variables that are of interest in the larger model often depend on the fluxes across these subdomain boundaries.

Within a group of interchangeable components (e.g., infiltration components), there are many other implementation differences that a modeler may wish to explore, beyond just how a physical process is parameterized. For example, performance and accuracy often depend on things like (1) numerical scheme (explicit vs. implicit, order of accuracy, stability), (2) data types used (float vs. double) (3) number of processors (parallel vs. serial), (3) approxima-

tions used, (4) the programming language or (5) coding errors.

Auto-connection of components is important from a users point of view. It is common for components to require many input variables and produce many output variables. Users quickly become frustrated when they need to manually create all of these pairings/connections, especially when using more than just 2 or 3 components at a time. The OpenMI project does not support the concept of auto-connection or interchangeable components. When using the graphical Configuration Editor provided in its SDK, users are presented with droplists of input and output variables and must select the ones to be paired. This requires expertise and is made more difficult because there is so far no ontological or semantic scheme that makes it clear whether two variable names actually refer to the same thing.

The CSDMS project currently employs an approach to auto-connection that involves providing interfaces (i.e. CCA ports) with different names to reflect their intended use (or interchangeability), even though the interfaces are the same internally.

5. The Current CSDMS Component Interface

A concise definition, referring back to previous section

5.1. The IRF Interface

The body of a numerical model in divided into three sections: set up, execution, and teardown. The set-up phase occurs before time stepping begins and initializes the model. The execution phase is the guts of the model and consists of everything *within* the main time loop of the model. The teardown phase occurs after time stepping completes and acts to clean up the model simulation. In the case of a time-independent model, the model calculations can be thought of as a time-stepping model with a single time step.

5.1.1. Model handle

For object-oriented languages (Python, C++, Java, for example), the model's state is encapsulated as private data within a class, and the interface functions are member functions of that class. For languages that are not object-oriented, the model's state is contained in a data structure that is exposed to the application programmer as an opaque object. The memory within the object is not directly accessible to the user but is accessed via a handle, which exists in user space. This handle is passed as the first argument to each of the interface functions so that they can operate on a particular instance of a model.

In C, this handle could simply be a pointer to the object.

Model handles are allocated and dellaocated through the initialize and finalize interface functions, respectively. For allocate calls, the initialize functions are passed an OUT argument that will contain a valid reference to the object. In the case of deallocation, the finalize function accepts an INOUT variable that provides a reference to the object to be destroyed and sets the object to an invalid state.

5.1.2. Initialize (Model Set Up)

Before a model enters into its time-stepping loop, it will usually execute a set of commands necessary to set up the subsequent model simulation. This is the initialization step — the lines of code before the time loop. The initialize step puts a model into a valid state that is ready to be executed. Mostly this will be initializing variables or grids that will subsequently be used within the execution step. Temporary files that the execution step will read from or write to should also be opened here. Any user interface, whether it be graphical or command line, should be left outside of the initialization step. User interfaces should be kept outside of the IRF modeling interface and within the model application. This allows new application developers to attach their own user interface to their application and, more importantly, it avoids conflicting user interfaces competing with one another when multiple models are linked to one another within a single application.

CMI_INITIALIZE (handle, filename)

OUT	handle	handle to the CMI object
IN	filename	path to configuration file

A typical interface for model initialization:

```
int CMI_Initialize (CMI_Handle *handle, char *filename)
```

The initialization function takes the name of a file as input and constructs an object that holds the state of the model. A null filename indicates that there is no input file, otherwise this is the main input file used by the model. Input file names or paths should not be hardcoded within the implementation of the initialize step. This is necessary to protect against the case where two models run in the same directory and are both looking for files of the same name.

5.1.3. Run (Model Execution)

A model's execute step should run the model for a particular amount of simulation time. Generally speaking, this will be the lines of code that are *within* the time loop. The CMI_RUN_UNTIL interface function advances the model from its current state to a future state. For time-independent models the run step simply executes the model calculation and updates the model state so that future calls will not require executing the calculations again. Encapsulating the only the code *within* the time loop allows an application to run the model to an intermediate state. This is necessary to allow an application to query the model's state for the purposes of (for instance) printing output or passing state data to another model.

Many models contain code within the time loop that prints output files as the model advances in time. Ideally this code should be pulled out of the run step and be controlled by the application. However, in existing models this code

is often deeply embedded within the model and would be difficult to extract. If this code is not removed, the application developer must be aware that by executing RUN_UNTIL output files that they were not aware of may be generated and may conflict with output files from their application or another model's run step. If not removed, the output files should be well documented and allow for a naming convention that will reduce the possiblility of naming conflicts. For example, output files could be written in a user-defined folder or have a name prefix that reflects the name of the model.

CMI_RUN_UNTIL (handle, stop_time)

IN	handle	handle to the CMI object
IN	stop_time	simulation time to run model until

Sample interface:

```
int CMI_Run_until (CMI_Handle handle, double stop_time);
```

5.1.4. Finalize (Model Termination)

The finalize step cleans up after the model is no longer needed. The main purpose of this step to make sure that all resources your model acquired through its life have been freed. Most often this will be freeing allocated memory, but could also be freeing file or network handles. Following this step, the model should be left in an invalid state such that its run step can no longer be called until it has been initialized again.

CMI_FINALIZE (handle)

INOUT	handle	handle to the CMI object

Sample interface:

```
int CMI_Finalize (CMI_Handle handle);
```

5.2. Value Getters and Setters

Although a model that exposes the IRF interface described above is of great use, there are still a couple of interface functions missing to make it even more useful in integrating with other models. For one model to meaningfully communicate with other models, it must be able to exchange values with those other models. Furthermore, each model should provide a means for another model to change particular values of itself. One way of doing this is through a getter and setter interface. As we have seen, a barebones IRF interface is certainly valuable but augmenting this interface with getter and setter methods will make the model more useful in linking with others.

5.2.1. Value Getters

If an application wishes to access values from a model, the model should implement some kind of getter method (or methods). Limiting access to the model's state to be through a set of functions allows control of what data the model shares with other programs and how it shares that data. There are two ways to transfer the data. The first is to give the calling program a copy of the data. The second is to give the actual data that is being used by the model (in C, this would mean passing a pointer to a value). The first has the advantage that it hides implementation details of the model from the calling program and also limits what the calling program can do to the model. However, the downside of the first method is that communication will be slower (and could be significantly so, depending on the size of the data being transferred).

CMI_Get_Value() (handle, value_str, buffer, count, datatype)

IN	handle	handle to the CMI object
IN	value_str	name of the value to get
OUT	buffer	initial address of the destination values
IN	count	number of elements in buffer
IN	datatype	data type of each buffer element

Sample interface:

```
int CMI_Get_Value (CMI_Handle handle, char *value_str,
                   void *buffer, int count, CMI_Datatype datatype);
```

5.2.2. Value Setters

For a program to be able to set data within a model, the interface should implement a setter for those values. Variables within a model should be accessed and changed through interface methods. This ensures that users of the interface are not able to change values that the interface implementor doesn't want them to. This also detaches the programmer using your interface from your model implementation, thus freeing the model developer to change details of his or her model without an application programmer having to make any changes.

The setter can also perform tasks other than just setting data. For instance, it might be useful if the setter checked to make sure that the new data is valid. After the setter method sets the data it should ensure that the model is still in a valid state.

CMI_Set_Value() (handle, value_str, buffer, count, datatype)

	IN	handle	handle to the CMI object
	IN	value_str	name of the value to get
782	IN	buffer	initial address of the source values
	IN	count	number of elements in buffer
	IN	datatype	data type of each buffer element

783 Sample interface:

```
784 int CMI_Set_value (CMI_Handle handle, char* value_str,
785                    void* buffer, int count, CMI_Datatype datatype);
```

786 6. Component Wrapping Issues

787 *6.1. Code Reuse and the Case for Wrapping*

788 Using computer models to simulate, predict and understand earth surface
789 processes is not a new idea. This means that many models exist, some of
790 which are fairly sophisticated, comprehensive and well-tested. The difficulty
791 with reusing these models in new contexts or linking them to other models
792 typically has less to do with how they are implemented and more to do with
793 the interface through which they are called. (And to some extent, the language
794 they are written in.) For a small or simple model, it may take little effort
795 to rewrite the model in a preferred language and with a particular interface.
796 However, rewriting large models is very time-consuming and error prone. In
797 addition, most large models are under continual development and a rewritten
798 version will not see the benefits of future improvements. Given these facts,
799 it becomes clear that in order for code reuse to be practical, we need (1) a
800 *language interoperability tool*, so that components dont need to be converted
801 to a different language and (2) a *wrapping procedure* that allows us to provide
802 existing code with a new calling interface. As suggested by its name, and the
803 fact that it applies to the "outside (interface) of a component vs. its "inside
804 (implementation), wrapping tends to be noninvasive and is a practical way to
805 convert existing models into components.

806 *6.2. Wrapping for Object-Oriented Languages*

807 As mentioned previously, component-based programming is essentially object-
808 oriented programming with the addition of a framework. If a model has been
809 written as a class, then it is relatively straight-forward to modify the definition
810 of this class so that it exposes a particular model-coupling interface. This could
811 be done by adding new methods (member functions) that call existing ones or
812 by modifying the existing methods. Each function in the interface has access
813 to all of the state variables (data members) without passing them explicitly, as
814 well as all of the other interface functions. In object-oriented languages it is
815 common to distinguish between *private methods* that are intended for internal
816 use by the model object and *public methods* that are to be used by callers and

which may comprise one or more interfaces. (Some languages, like Java, make this part of a methods declaration.)

In order for this model object to be used as a component in a CCA-compliant framework like Ccaffeine, it must also be "wrapped by a CCA implementation file (or IMPL file). The CCA tool chain has tools like Babel and Bocca (described previously) that are used to auto-generate an IMPL-file template. For a model that is written in an object-oriented and Babel-supported language (e.g., C++, Python or Java), the IMPL file needs to do little more than add interface functions like setServices and releaseServices that allow the component to communicate with and be instantiated by the framework. The interface functions that are used for inter-component communication (i.e. passing data and IRF) can simply be inherited from the model class. Inheritance is a standard mechanism in object-oriented languages that allows one interface (set of methods) to be extended or overridden by another. Note that the IMPL file may have its own Initialize() function that first gets the required CCA ports and then calls the Initialize() function in the models interface. But the function that gets the CCA ports can simply be another function in the models interface that is only used in this context. Similarly, the IMPL file may have a Finalize() function that calls the Finalize() function of the model and then calls a function to release the CCA ports that are no longer needed. It is desirable to keep the IMPL files as clean as possible, which means adding some CCA-specific functions to the models interface. For example, a CSDMS component would have (1) functions to get and release the required CCA ports, (2) a function to create a tabbed-dialog (using CCAs so-called parameter ports) and (3) a function that prints a language-specific traceback to stdout if an exception occurs during a model run.

6.3. Wrapping for Procedural Languages

Languages such as C or Fortran (up to 2003) do not provide object-oriented primitives for encapsulation of data and functionality. Because component-based programming requires such encapsulation, the CCA provides a means to produce object-oriented software even in languages that do not support it directly. We briefly overview the mechanism for creating components based on functionality implemented in a procedural language (e.g., an existing library or model).

A class in OO terminology encapsulates some set of related functions and associated data. To wrap a set of library functions, one can create a SIDL interface or class that contains a set of methods whose implementations call the legacy functions. The new interface does not have to mirror existing functions exactly, presenting a nonintrusive opportunity for redesigning the publicly accessible interfaces presented by legacy software. The creation of class or component wrappers also enables the careful definition of namespaces, thus reducing potential conflicts when integrating with other classes or components. The SIDL definitions are processed by Babel to generate IMPL files in the language of the code being wrapped. The calls to the legacy library can then be added either manually or by a tool, depending on how closely the SIDL interface follows the original library interface.

Function argument types that appear in the SIDL definition can be handled in two ways: by using a SIDL type or by specifying them as *opaque*. SIDL already supports most basic types and different kinds of arrays found in the target languages. Any user defined types (e.g., structs in C, derived types in Fortran) must have SIDL definitions or be passed as opaques. Because opaques are not accessible from components implemented in a different language, they are rarely used. Model state variables that must be shared among components can be handled in a couple of different ways. They can be encapsulated in a SIDL class and accessed through get/set methods (e.g., as described in Sec. **??**. Recently Babel has added support for defining *structs* in SIDL, whose data members can be accessed directly from multiple languages.

6.4. Automated Wrapping via Annotation

Once a model has been written so that it provides entry points to its functionality (whether it be an IRF interface, or otherwise) it can be wrapped as a component to be used within a modeling framework. The precise steps needed to do this depend on the framework. However, if the model contains sufficiently descriptive metadata, it should be easily imported into a modeling framework.

Because modeling frameworks may change over time, it is important to provide metadata within a model so that it does not tie itself to any one framework. If a framework injects too much of itself into a model, the model becomes reliant on that framework. It should be that the framework relies on its set of models, not the other way around. To help with this problem, a key step is to annotate the model source code so that the required metadata stays with the model. Using special keywords within comment blocks, a programmer is able to provide basic metadata for a model and its variables that is closely tied to the model but doesnt affect how the model itself is written. For example, metadata for a variable could follow its declaration in a comment that describes its units, valid range of values and whether it is used for input or output. Another annotation could identify a particular function as being the models initialize, run, or finalize step. This type of annotation makes it possible to write utilities that parse the source code, extract the metadata and then automatically generate whatever component interface is required for compatibility with other models. In fact, this metadata could be automatically extracted and used for a wide range of purposes such as generating documentation, or providing an overview of the state of a communitys models.

6.5. Guidelines for Model Developers

There are several relatively simple things that developers can do so that it becomes much easier to create a reuseable, plug-and-play component from their model source code. Given the large number of models that are contributed to the CSDMS project, it is much more efficient for model developers to follow these guidelines and thereby “meet us halfway than for CSDMS staff to make these changes after code has been contributed. This can be thought of as a form of “load balancing.

6.5.1. Programming Language and License

- Write code in a Babel-supported language (C, C++, Fortran, Java, Python).
- If code is in MatLab or IDL, use tools like I2PY to convert it to Python. Python (with the numpy, scipy and matplotlib packages) provides a free work-alike to MatLab with similar performance.
- Make sure that code can be compiled with an open-source compiler (e.g. gcc, gfortran).
- Specify what type of open-source license applies to your code. Rosen (2004) is a good, online and open-source book that explains open source licensing in detail. CSDMS requires that contributions have an open source license type that is compliant with the standard set forth by the Open Source Initiative (OSI).

6.5.2. Model Interface

- Refactor the code to have the basic IRF interface (see above).
- If code is in C or Fortran, add a model name prefix to all interface functions to establish a namespace (e.g. ROMS_Initialize()) . C code can alternately be compiled as C++.
- Write Initialize() and Run_Until() functions that will work whether the component is used as a driver or *nondriver*.
- Provide getter and setter functions (see above).
- Provide functions that describe input and output *exchange items* (see above).
- Use descriptive function names (e.g. Update_This_Variable).

6.5.3. State Variables

- Decide on an appropriate set of state variables to be maintained by the component and made available to callers.
- Attempt to minimize data transfer between components (as discussed above).
- Use descriptive variable names.
- Carefully track each variables units.

6.5.4. Input and Output Files

- Do not hardwire configuration settings in the code; read them from a configuration file (text).
- Do not use hardwired input filenames.
- Read configuration settings from text files (often in Initialize()). Do not prompt for command-line input. If a model has a GUI, write code so it can be bypassed; use the GUI to create a configuration file.
- Design code to allow separate input and output directories that are read from the configuration file. (This allows many users to use the same input data without making copies (e.g. test cases).) It is frequently helpful to include a *case prefix* (scenario) and a *site prefix* (geographic name) and use them to construct default output filenames.
- Establish a namespace for configuration files (e.g. ROMS_input.txt vs. input.txt).
- If large arrays are to be stored in files, save them as binary vs. text. (e.g. this is the case with netCDF)
- Provide self-test functions or unit tests and test data. One self-test could simply be a "sanity check that uses trivial (perhaps hard-coded) input data. When analytic solutions are available, these make excellent self-tests because they can also be used to check the accuracy and stability of the numerical methods.

6.5.5. Documentation

- Help CSDMS to provide a standardized, HTML help page.
- Help CSDMS to provide a standaridized, tabbed-dialog GUI.
- Make liberal use of comments in the code.

7. The CSDMS Modeling Tool (CMT)

As explained in section **??**, Ccaffeine is a CCA-compliant framework for connecting components to create applications. From a users point of view, Ccaffeine itself is a low-level tool that executes a sequence of commands in a Ccaffeine script. The (natural language) commands in Caffeines scripting language are fairly straightforward so it is not difficult for a programmer to write one of these scripts. For many people, however, using a graphical user interface (GUI) is preferable because it means they dont have to learn the syntax of the scripting language. A GUI also provides users with a natural, visual representation of the connected components as boxes with buttons connected by wires. It can also prevent common scripting errors and offer a variety of other

convenience features. The CCA Forum developed such a GUI, called Ccafe-GUI, that presented components as boxes in a palette that could be moved into an arena (workspace) and connected by wires. It also allowed component configurations and settings to be saved in BLD files and instantly reloaded later. Another key feature of this GUI is that, as a lightweight and platform-independent tool written in Java, it could be installed and used on any computer with Java support to create a Ccaffeine script. This script could then be sent to a remote, possibly high-performance computer for execution.

While the Ccafe-GUI was certainly useful, the CSDMS project realized that it could be improved and extended it in numerous ways to make it more powerful and more user-friendly. In addition, these changes would not only serve the CSDMS community but could be shared back with the CCA community. That is, the new GUI works with any CCA-compliant components, not just CSDMS components. The new version is called CMT (CSDMS Modeling Tool). Significant new features of CMT 1.5 include:

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

As mentioned above, the CMT provides integrated visualization using VisIt. VisIt (http://wci.llnl.gov/codes/visit) is an open-source, interactive, parallel visualization and graphical analysis tool for viewing scientific data. It was developed by the Department of Energy (DOE) Advanced Simulation and Computing Initiative (ASCI) to visualize and analyze the results of simulations ranging from kilobytes to terabytes. VisIt was designed so that users can install a client version on their PC that works together with a server version installed on a high-performance computer or cluster. The server version makes use of multiple processors to speed up rendering of large data sets and then sends graphical output back to the client version. VisIt supports about five dozen file formats and provides a rich set of visualization features, including the ability to make movies from time-varying databases. The CMT provides help on using VisIt in its Help menu. CSDMS uses a service component to provide other components with the ability to write their output to netCDF files that can be visualized with VisIt. Output can be 0D, 1D, 2D or 3D data evolving in time, such as (1) a time series (e.g. a hydrograph), (2) a *profile series* (e.g. a soil moisture profile), (3) a 2D *grid stack* (e.g. water depth), (4) a 3D *cube stack* or (5) a scatter plot of XYZ triples.

Another innovative feature of CMT 1.5 is that it allows users to toggle between the original, *wired* mode and a new *wireless* mode. CSDMS found that displaying connections between components with the use of wires (i.e. red lines) did not scale well to configurations that contained several components with multiple ports. In wireless mode, a component that is dragged from the palette to the arena appears to broadcast what it can provide (i.e. CCA provides ports) to the other components in the arena (using a concentric circle animation). Any components in the arena that need to use that kind of port get automatically linked to the new one and this is indicated through the use of unique, matching colors. In cases where two components in the arena have the same *uses port* but need to be connected to different providers, wires can still be used.

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

8. Providing Components with a Uniform Help System and GUI

Beyond the usual software engineering definition of a component, a useful component will be one that also comes bundled with metadata that that describes the component and the underlying model that it is build around. While creating a component as described in the preceding sections is important, it is

Figure 2: CMT screenshot.

of equal importance to have a well documented component that an end user is able to easily use.

With a plug-and-play framework where users easily connect, interchange, and run coupled models, there is a tendency for a user to treat components as black boxes and ignore the details of the foundation that each component was built upon. For instance, if a user is unaware of the assumptions that underlie a model, that user may couple two components that do not make sense coupling because of the physics of each model, or attempt to use a component in a situation that it was not intended to be used in. To combat this problem components are bundled with HTML help documents, which are easily accessible through the CMT, that describe the component and the model that it wraps. These documents include:

- an extended model description (along with references)

- listing and brief description of the components uses and provides ports
- the main equations of the model
- sample input and output
- acknowledgement of the model developer(s)

A complete component also comes with metadata supplied in a more structured format. Components include XML description files that describe their user-editable input variables. These description files contain a series of XML elements that contain detailed information about each variable including a default value, range of acceptable values, short and long descriptions, units, and data type.

```
<entry name=velocity>
 <label>River velocity</label>
 <help>Depth-averaged velocity at the river mouth</help>
 <default>2</default>
 <type>Float</type>
 <range>
   <min>0</min>
   <max>5</max>
 </range>
 <units>m/s</units>
</entry>
```

Using this XML description, the CMT automatically generates a graphical user interface (in the form of tabbed dialogs) for each CSDMS component. Despite each model's input files being drastically different, this provides CMT users with a uniform interface across all components. Furthermore, the GUI checks user input for errors and provides easily accessible help within the same environment — none of which is available in the batch interface of most models.

Nearly every model gathers initial settings from an input file and then runs without user intervention. Ultimately, any user interface that wraps a model must generate this input file for the component to read as part of its initialization step. The above XML description along with a template input file allows this to happen. Once input is gathered from the user, a model-specific input file is created based upon a template input file provided with each component. A valid input file is created based on `$`-based substitutions in this template file. Instead of actual values, the template file contains substitution placeholders of the form `$identifier`. Each identifier corresponds to an entry name in the XML description file and, upon substitution, is replaced by the value gathered from an external user interface (the CMT GUI, for instance).

9. Framework Services: "Built-in" Tools That Any Component Can Use

There are certain low-level tools or utilities that developers (e.g., CSDMS staff) may wish to make available so that any component (or component developer) can use them without requiring any action from a user. These tools can be encapsulated within special components called *service components* that are automatically instantiated by a CCA framework on startup. The services/methods provided by these components are then called *framework services*. Unlike other components, which users may assemble graphically into larger applications, users do not interact with service components directly. However, a component developer can make calls to the methods of service components through *service ports*. The use of service components allows developers to maintain code for a shared functionality in a single place and to make that functionality available to all components regardless of the language they are written in (or which address space they are in). CSDMS uses service components for tasks such as (1) providing component output variables in a form needed by another component (e.g., spatial regridding, interpolation in time, unit conversion) and (2) writing component output to a standard format such as netCDF.

Any CCA component can be "promoted to a service component. A developer simply needs to add lines to its setServices() method that register it as a framework service. CCA provides a special port for this called *gov.cca.ports.ServiceRegistry* with three methods called: addService(), addSingletonService() and removeService(). If a developer then wants another component to be able to use this framework service, a call to the gov.cca.Services.getPort() method must be added within its setServices() method. (A similar call must be added in order to use CCA parameter ports and ports provided by other types of components.) Note that the setServices() method is defined as part of the gov.cca.Component interface.

CCA components are designed for use within a CCA-compliant framework (like Ccaffeine) and may make use of service components. But what if we want to use these components outside of a CCA framework? One option is to encapsulate a set of functionality (e.g., a service component) in a SIDL class and then "promote this class to (SIDL) component status through inheritance and by adding only framework-specific methods like setServices(). (Note that a CCA framework is the entity that calls a components setServices() method as described in Sec. **??**.) This approach can be used to provide both component and non-component versions of the class. Compiling the non-component version within a bocca project generates a library file that we can link against or, in the case of Python, a module that we can import.

10. Conclusions

- Optimal granularity is the physical process level.
- Advantages of "community cloud" approach

- Advantages of Python support
  Bocca, VisIt, ArcGIS scripting, Matlab work-alike, etc.
- Pros and cons of building on open-source
- CCA need not be invasive
- Use of standards like netCDF, XML (or XUL), CCA, SIDL, HTML, etc.
  Mention Ugrid Interoperability group working on a netCDF standard for unstructured grids (ugrids)
  Mention OGC, GDAL, CUAHSI-HIS (& WML), etc. ?
- Using toolkits like PETSc whenever possible
- Multiple paths to parallelism
- Components that retrieve data from web services (cite other paper, this issue)
- Automated wrapping via annotation ?

Acknowledgements

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