

Managing Complexity in Simulations of Land Surface and Near-surface Processes,[☆]

Ethan T. Coon^{a,*}, J. David Moulton^b, Scott L. Painter^c

^a*Computational Earth Sciences, Los Alamos National Laboratory*

^b*Applied Mathematics and Plasma Physics, Los Alamos National Laboratory*

^c*Climate Change Science Institute, Environmental Sciences Division, Oak Ridge National Laboratory*

Abstract

Increasing computing power and the growing role of simulation in Earth systems science have led to an increase in the number and complexity of processes in modern simulators. We present a multiphysics framework that specifies interfaces for coupled processes and automates weak and strong coupling strategies to manage this complexity. Process management is enabled by viewing the system of equations as a tree, where individual equations are associated with leaf nodes and coupling strategies with internal nodes. A dynamically generated dependency graph connects a variable to its dependencies, streamlining and automating model evaluation, easing model development, and ensuring models are modular and flexible. Additionally, the dependency graph is used to ensure that data requirements are consistent between all processes in a given simulation. Here we discuss the design and implementation of these concepts within the *Arcos* framework, and demonstrate their use for verification testing and hypothesis evaluation in numerical experiments.

Keywords: Multiphysics, framework, directed acyclic graph, land surface modeling, thermal hydrology

*Corresponding author

Software Availability

Name of Software	Arctic Terrestrial Simulator (ATS)
Developers	Ethan Coon
Contact	ecoon _at_ lanl.gov
Year First Available	2014
Hardware Required	Flexible, laptops to supercomputers
Software Required	Linux/OSX, Amanzi and its dependencies (see following entry for Amanzi)
Software Availability	Source and documentation available at https://software.lanl.gov/ats
Cost	Free, Open Source (three-clause BSD license)

Name of Software	Amanzi
Developers	Multi-institution team led by J. D. Moulton, see https://software.lanl.gov/ascem/amanzi for more information.
Contact	moulton _at_ lanl.gov
Year First Available	2014
Hardware Required	Flexible, laptops to supercomputers
Software Required	Linux/OSX with C++/C/Fortran compilers, Message Passing Interface (MPI), CMake tools. A bootstrap configuration tool is provided that automatically downloads and builds additional third party libraries (TPLs).
Software Availability	Source and documentation is available at https://software.lanl.gov/ascem/amanzi
Cost	Free, Open Source (three-clause BSD license)

1. Introduction

Increasing computational power has enabled not only the increase of spatial and temporal resolution of simulations in the Earth sciences, but also an increase in complexity. More physical processes are included, with a higher degree of physical fidelity in each process representation. This growth in complexity represents a fundamental shift for multiphysics simulators from the old paradigm, in which a small number of process models are coupled together in a limited number of pre-defined combinations, to a new paradigm, in which many, varying processes are coupled in nearly arbitrary permutations. In moving from a few to many process models, identifying which processes are critical to simulate a given phenomena is often a fundamental part of the science, requiring flexible coupling strategies and eliminating a statically configured coupling strategy as an option.

The requirements of two complex environmental simulation codes with significant subsurface focus, the *Arctic Terrestrial Simulator (ATS)* and *Amanzi*, exemplify the challenges inherent in emerging Earth Science applications. The

need to simulate the hydrologic response of Arctic tundra to warming temperatures, and the fate of carbon stored in Arctic soils have motivated development of *ATS*. *ATS* performs coupled simulations of soil thermal hydrology with freeze/thaw cycles, topological evolution due to melting subsurface ice, surface flows with freezing/thawing, snow processes, surface energy balance, soil biogeochemistry, and vegetation processes <16>. This is a computationally challenging application because of the number and complexity of the controlling physical processes and constitutive models, very strong nonlinearities that necessitate implicit coupling, the need for complex unstructured meshes requiring advanced discretizations, and different process representations on different subdomains ranging from three-dimensional (subsurface processes), to two-dimensional (surface processes), to one-dimensional (snow models), to zero-dimension (surface energy balance and carbon decomposition models). This application stresses the need for a flexible and dynamic multiphysics framework. There is significant uncertainty in the physical phenomena necessary to accurately predict carbon emissions, and error introduced by coupling strategies has not yet been quantified. Developing software explicitly configured for all of the possible combinations for processes and process couplings would be extremely difficult and error prone. Instead, there is a strong need for approaches that allow for a flexible, hierarchically organized, and dynamically configured model.

Similar considerations apply to the Environmental Management program within the Department of Energy, which oversees the remediation and closure of DOE sites storing legacy waste from the development of nuclear weapons and related technologies. Although, at a high-level these sites have flow and reactive transport in common, there are significant differences in the heterogeneous subsurface environment, complex biogeochemistry, and external forcing that affects contaminant transport. In addition, a variety of engineered systems such as tanks, flow barriers, and injection wells are often used and can be modeled with varying levels of abstraction and fidelity. Thus, to underpin scientifically defensible decisions and strategies for cleanup and remediation, risk and performance assessments use a graded and iterative approach. This approach first establishes simplified models and then iteratively enhances geometric and process level complexity to identify and characterize the key processes and assumptions that are needed to efficiently reach a defensible decision. This need led to the development of the Advanced Simulation Capability for Environmental Management (ASCEM) program, and its simulator *Amanzi*. Moreover, it required that *Amanzi* support a flexible and extensible design that would support run time selection of process models, and interoperability with established biogeochemical reaction capabilities <14>. An important feature of this design is that it significantly improves the ability of developers to meet software quality assurance requirements because it enables testing components both in isolation and in integrated settings.

The requirements for *ATS* and *Amanzi* and the broader shift toward greater process complexity in land surface and subsurface simulations represent both a challenge and an opportunity for basic research in advanced computational scientific software. Although in traditional multiphysics efforts, data, data depen-

dependencies, and model coupling strategies were managed by an “omniscient coder,” new paradigm applications require a more abstract view. Specifically, pursuing this new level of abstraction creates the opportunity for a well-designed programming model to gather information about the physical system, informing automated methods for exposing concurrency within the application, thus helping to bridge the productivity gap between application software and extreme-scale machines. Additionally, flexibility gained by following abstract interfaces and frameworks significantly improves the efficiency of developing and running new simulations to test scientific hypotheses, advance understanding and make predictions.

In fact, the increasing interest in process-rich simulations with ever higher-fidelity is a common challenge across disciplines that have embraced simulation. Thus, strategies for managing complexity in process-rich simulations are beginning to appear in various applications and include data managers <19; 13>, dependency graphs<1; 6; 11> , and process couplers <7; 18; 17; 8>. Data managers are more common in simulations considering multiple domains, such as atmosphere-ocean-land climate models, or in engineering applications with solid and fluid domains. Tools such as the Data Transfer Kit<19>, the Model Coupling Toolkit<13>, and OASIS<18> focus on mapping fields between differing meshes, in parallel, at domain interfaces. However, as most codes deal with a single discretization assumed from the start of development, fields are typically collocated at a single entity.

Dependency graphs are seeing more and more attention as a potential tool for extreme-scale computing. These graphs have been used, for instance, to manage fine-grained concurrency in dense linear algebra calculations in PLASMA<1> and coarse-grained concurrency in finite-element simulations in Phalanx<15>. Similarly, task graphs, which form dependency graphs for evaluation processes, have been used to exploit concurrency in task-based parallelism strategies. This approach has been used in several codes and programming models, including Uintah<6>, Charm++<11>, and Legion<5>.

Generalized process couplers, which provide interfaces for components to implement and thereby be coupled to other components, are a common strategy in the Earth sciences, and have been approached in an ad hoc way in many codes. However, several efforts deserve note for their generality for complex problems. In particular, climate codes often must couple modules from ocean, atmosphere, sea ice, and land processes into a common framework. Efforts toward this problem include the CESM flux coupler CPL7<7>, the Earth System Modeling Framework <10>, and the Flexible Model System <3>. CSDMS<17> uses similar approaches for surface processes. In these efforts, interfaces for coupling are required of each component, and once a component implements that interface it may be plugged into every other system component. Typically, these are limited to sequential coupling. Recently, feature modeling, a form of meta-analysis about these types of couplers, has been performed on several couplers in the literature by Dunlap et al. <8>. These types of efforts result in clean, modular code, albeit at a relatively coarse granularity generally focused on coupling through fluxes.

Although these efforts have contributed capability and experience that are valuable for geoenvironmental modeling, none alone are sufficient for meeting the challenges presented by emerging simulation needs in ecohydrological (e.g. ATS) and environmental management applications (e.g. Amanzi). In this work, we present a conceptual model for managing complexity in multiphysics codes using a process tree, a dependency graph, and a data manager. Our approach builds on the previous ideas and extends them in multiple directions. It uses the process coupler approach at a finer granularity to manage coupling among ecohydrological process models represented in the same subdomain, not simply coupling among subdomains. In addition, dependency graphs and the associated evaluation process are used extensively to manage and organize shared data requirements among different process representations. Management of data location with respect to a mesh, dependencies of data and the evaluation of the models defining these dependencies, and flexibility within coupling of the various processes are abstracted into a framework and leveraged by physical modules. While we focus here on the strategy and implications and capability enabled by the strategy, we also discuss one implementation, *Arcos*, in order to make some of the concepts more concrete, and to demonstrate the feasibility of such a strategy for real problems and codes.

The remainder of this paper is organized as follows. In Section 2 we highlight the challenges that motivate the need for this conceptual model and guide the design. The realization of these concepts in both conceptual strategy and implementation, is presented in Section 3. This discussion begins with the data manager (Section 3.1), connecting this concept to the dependency graph (Section 3.2), and concluding with automated coupling (Section 3.3). To highlight the powerful capabilities implied by this strategy, we present two demonstration cases in Section 4: Section 4.1 shows how the strategy enables testing, and Section 4.2 shows a more complete example of how the *Arcos* framework is used in practice. Finally, in Section 5 we discuss qualitative advantages and disadvantages of using such a strategy, and in Section 6 we provide some conclusions and comment on future work.

2. Overview of the conceptual model

Our approach to develop a flexible and extensible multiphysics framework is built around a series of concepts that enable the capabilities discussed in the following sections. To motivate these concepts and understand their synergy, we present a high-level view of their roles and interactions. Specifically, the conceptual model of the proposed framework is based around two graphs, and the interactions between these graphs (Figure 1).

The first graph, the process tree, describes the hierarchical coupling among the various equations or systems of equations (Figure 1a). Each node in the tree is an equation or system of equations. A leaf node (shown in brown) is a single equation, be it a conservation equation PDE, evolution equation ODE, or other physical algebraic constraint. Then each node couples the subtree below it, and represents that system of equations. Each node presents a common interface,

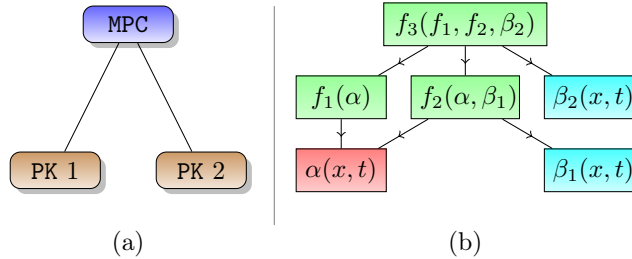


Figure 1: Basic graph structures of *Arcos*, including (a), the process tree, and (b), the dependency graph.

which we call the *process kernel*, or PK, interface, to the nodes above it. This interface includes both physical methods, such as advancing a single timestep, and more administrative methods, such as callback hooks to calculate diagnostics for a visualization or observation. Internal nodes are called *multi-process coupler* nodes, or MPCs (shown in blue). MPCs couple the equations in the subtree they root, and may often be fully automated with no knowledge of the PKs (or other MPCs) below them in the tree. Critically, MPCs themselves implement the PK interface, and may be used as a PK, allowing MPCs to couple other systems hierarchically. This hierarchical representation of models is natural to the coupled physical system, and implicitly exists in most existing codes. By making it explicit, general, and dynamically formed, we can automate much of the coupling while still allowing flexibility for domain scientists to customize each part of the coupler.

Each PK and MPC does little actual physical work. Instead, PKs provide the administrative glue of the equation, dealing with time discretizations and solvers. Much of the work in a PK is delegated to *evaluators*, which apply *models* to data stored in *fields*. Each evaluator manages one of three types of variables:

1. Independent variables are user-provided functions of spatial and temporal coordinates used to provide boundary terms, source terms, or other input equations.
2. Primary variables are solved for within a PK, and their evaluators track changes applied in the course of solving the equation.
3. Secondary variables are evaluated functions of other variables. A model for a secondary variable can be anything from a constitutive relation (evaluate a model for density as a function of temperature and pressure on cells in the subsurface) to a discrete operator (apply a divergence operator to pressure given a mesh and discretization) to a summation operator (add the divergence of Darcy fluxes to a source term to determine the mass balance).

These evaluators are stored in a *dependency graph*, which is a directed, acyclic

graph (DAG) describing the functional relationship of each variable in the system of equations (see Figure 1b). Leaf nodes in the dependency graph are independent variables (shown in cyan) or primary variables (shown in red), and are potentially functions of both time and space. All other nodes are secondary variables (shown in green), and directed edges from a secondary variable point to the variables that they depend upon.

The evaluator is much like a functor or function; it stores no actual data, only metadata and a few parameters or constants. Instead, it accesses data using a data manager, which controls access for both read-only and read/write of a field. This abstracts physical equations (which must be written by domain scientists) from data layout, transfer, and storage (which should be written by computational scientists).

The combination of a data manager and a dependency graph enables dynamic definition of each variable’s model and data, and splits complex equations into manageable chunks. It also allows lazy evaluation, where variables in the graph are recalculated only if their dependencies have changed, resulting in a managed, automated evaluation process with fewer bugs and inefficiencies. The combination of a dependency graph and a process tree provide two overlapping representations of the mathematical equations – one at the variable level and one at the equation level. These views are made explicit by representing both in code, allowing an abstract yet formalized representation of the mathematical equations in code. This formalization provides an interface between physics implementing nodes of these graphs (by domain scientists) and the evaluation/solution and use of these graphs (by computational scientists). This coexistence enables automated, simple coupling (implemented independently of the physics) and complex, custom, physics-driven coupling (implemented by collaborations between domain scientists and computational scientists).

3. Design decisions and implementation within *Arcos*

The concepts outlined in the preceding section are implemented within the *Arcos* framework, as diagrammed in Figure 2. In this section, we discuss each concept in more detail, highlighting the capability that it enables and identifying cases where functionality can be automated. The design decisions and implementation details of *Arcos* are described, and simplified examples are provided to explain targeted features.

Throughout, we use code snippets extensively to describe interfaces and functionality. Code snippets, and *Arcos*, are written in C++, but the concepts and framework could easily be implemented in any general purpose language. In these snippets, and the remainder of the text, class names are written in monospaced font, as in `MyExampleClass`. Several code details are ignored for conciseness of the snippets. Namespaces are generally ignored, including both the `arcos` namespace and the `std` namespace. Additionally, all `public`, `protected`, and `virtual` statements are ignored, except where necessary to make a point. Finally, interfaces are simplified to remove reference counted

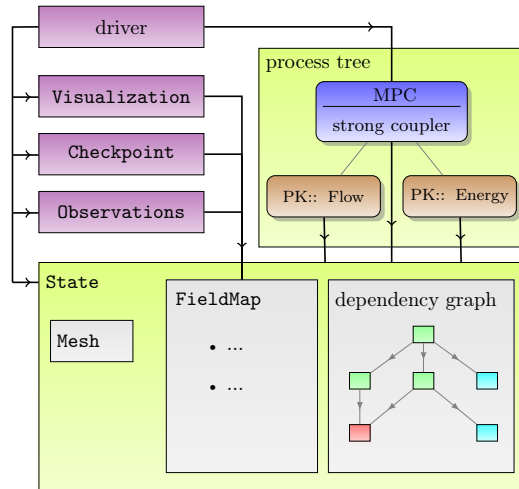


Figure 2: Overview of components of *Arcos* (boxed in yellow), and of simulators using *Arcos* (boxed in red) and their interactions.

pointers, and some administrative methods are ignored to focus less on implementation and more on concept. As *ATS* and *Amanzi* are open source, actual interface implementations can be seen there.

Arcos relies heavily on object-oriented programming for implementing the concepts described in Section 2. Nearly all of the code in *Arcos* is either a purely virtual interface, where implementations are provided by derived classes (as in the case of the PK or `Evaluator` interface), or partially implemented base classes (as in the case of MPCs), where user-provided classes inherit from these bases and implement the remaining, physics-specific capability. The exception is data manager class called `State`, which is a fully functional object implementation.

3.1. Data management

One difficulty in many-physics simulations is the compatibility of data between various processes. To manage these compatibility requirements, we use an object called `State`. Specifically, `State` stores a collection of `Fields` that represent both the data and assorted metadata required to describe the data. Implementations of the PK class call into `State` to access and evaluate residual equations for nonlinear solvers, boundary and initial data, and coupling terms like fluxes between domains. These `Fields` are classified as one of three types:

1. Independent variables are input functions and data that are defined by the user.
2. Primary variables are data that are solved for within a PK, which represents a single differential equation.

3. Secondary variables are data that is calculated from other data (of any type). These include both algebraic equations like constitutive models, differential quantities like gradients of primary variables, and integral quantities like total water content.

Ensuring that the expectations and assumptions of each PK and Evaluator about each Field are consistent is the primary duty of State.

3.1.1. Fields store metadata

The expectations about data are collected, accumulated, and checked for consistency as metadata. We formally define a Field’s metadata as a complete set of the following descriptors:

- mesh (surface, subsurface, a single column of the subsurface, a boundary region of the surface or subsurface)
- mesh entity or entities (e.g., cell, face, and node)
- number of degrees of freedom for each entity
- data layout
- communication pattern (needed for parallel communication in domain decomposition)

Rarely does the domain scientist need to fully specify this metadata. For instance, a physical process such as a flow solver may let the discretization choose the location (e.g., cell and/or face) of the pressure variable, and a secondary variable such as an equation of state may be able to evaluate density anywhere both pressure and temperature exist. Therefore, we allow each PK and Evaluator to place requirements on data, and then check for consistency and validity of those requirements once all have been completed.

Metadata is specified by two kinds of processes. One and only one kernel is called the “owner” of a field; this kernel defines the set of all possible metadata, and is the only kernel that gains write access to the data. Every other process simply places requirements on the field; if those requirements are not met, the code can throw a useful error message. We note that operator split strategies, in which different terms in the same equation may naturally be stored in different PKs, are enabled within this restriction by including an explicit Field for the solution at “intermediate steps”.

For instance, consider a PK for Richards equation:

$$\frac{\partial \Theta}{\partial t} - \nabla \cdot \left(\frac{\rho}{\mu} k_r K (\nabla p + \rho \mathbf{g}) \right) = Q \quad (1)$$

where water content Θ is conserved, and its time evolution is given by a fluxes due to gradients of pressure p and gravity g along with a source term Q . The diffusive coefficient for fluxes is given by water density ρ , viscosity μ , a linear tensor absolute permeability K and a nonlinear scalar relative permeability k_r .

The discretized flow PK may need a Field for the integrated water content on a cell at a given time:

```
S->RequireField("water_content")
->SetMesh("subsurface")
->AddComponent("cell", CELL, 1);
```

Here `S` is the `State` object. The following calls indicate the need for a field named `"water_content"`, on the mesh `"subsurface"`, which includes a single vector named `"cell"` on the mesh entity `CELL` (an enum within the mesh framework). Each of the string literals are keys into dictionaries of objects stored within `State`, and can then be accessed through methods in `State`, i.e.

```
const Mesh& subsurf_mesh =
    *S->GetMesh("subsurface");
const CompositeVector& wc =
    *S->GetFieldData("water_content");
```

Here we use `*` to dereference a reference-counted smart pointer, and access a custom vector object designed for block vectors.

Additionally, the flow process may require a field for pressure, the primary variable solved for within that process:

```
vector<string> names(2);
names[0] = "cell"; names[1] = "face";
vector<Entity_ID> locs(2);
locs[0] = CELL; locs[1] = FACE;
vector<int> n_dofs(2);
n_dofs[0] = 1; n_dofs[1] = 1;

S->RequireField("pressure", "flow")
->SetGhosted()
->SetMesh("subsurface")
->SetComponents(names, locs, n_dofs);
```

Here we add several additional requirements. First, the additional argument `"flow"` to `RequireField()` indicates that this object, the flow PK, will own the pressure field. This argument allows non-`const` access to the pressure field, and indicates that this object will be the only object to change the values stored in the pressure field. Additionally, we require pressure to have ghost entities for use by a discretization, and specify the field on both cells and faces (as required by our spatial discretization). Here, the use of `SetComponents()` as opposed to `AddComponents()` indicates exactly what will be specified for pressure – since this object “owns” pressure, only it knows where pressure will be defined. Any other object that calls `RequireField("pressure")` can only use `AddComponents()`, and if those components requested are not specified here, `State` throws an exception indicating incompatibility of data requirements.

3.1.2. Temporal models within *State*

One important design decision regarding **State** is the temporal model for using **State**. *Arcos* is designed to support two models. In the singleton model of **State**, there is one and only one state which stores all data for all times. This is more convenient for more complex coupling strategies such as subcycling, but requires more explicit management to advance the **State** at the completion of each timestep. In the time-slice model of **State**, each **State** exists at one and only one time, and multiple copies are constructed for each stage of a timestepping algorithm. This model is more convenient for globally implicit coupling. For instance, a simulation running many processes coupled via globally implicit coupling using a backward Euler time algorithm must deal with the potential issue of one of its component processes erroring. This error may occur both before some processes have started and after others have completed. Therefore two **States** are created – one for the old time and one for the new time. If one process fails, the top driver can simply copy the old step’s **State** back into the new timestep’s **State** and re-attempt the step with a smaller timestep. Once all processes have successfully advanced to the new time, the two **States** are re-used for the next timestep.

3.2. Dependency graphs for model evaluation

A managed evaluation process, in which each variable will only be updated if necessary, and at most once per change to its dependencies, is accomplished by the dependency graph. Each node of the dependency graph is a (**Field**, **Evaluator**) pair in which the **Evaluator** owns the **Field**, and (directed) edges of the graph are oriented from a node to its dependencies, as specified by the **Evaluator**. In this way, we can ensure that a well-posed equation’s dependency graph is also a directed, acyclic graph (DAG). Leaf nodes (those with no outgoing connections) consist of primary variables (specified and required by the PK), and independent variables (specified by the user). Interior nodes are secondary variables. Root nodes (those with no incoming connections) are defined by the PKs, and are typically terms in residuals which get passed off to time integration methods. Given that each interior or root node specifies its dependencies, the formation of the dependency tree is done dynamically, and the evaluation process is then automated.

Continuing the example for water content, a flow PK, in addition to requiring **Field** metadata for water content, may also require an **Evaluator** of the same name:

```
S->RequireFieldEvaluator("water_content");
```

This calls on a generic factory in **State** to search the input specification’s **Evaluator** section for an **Evaluator** spec named **"water_content"**, and to construct an **Evaluator** based upon that spec. Within that spec, an evaluator type and dependency list is provided (or default values for the dependency

names are provided in the derived `Evaluator`). Being more specific, let us assume a physical model for water content Θ in cell e as:

$$\Theta = \phi \rho(T, p) s(p) |e| \quad (2)$$

where ϕ is the porosity, ρ is the density as a function of pressure p and temperature T , s is the saturation as a function of pressure, and $|e|$ is the volume of cell e . Typically, to avoid the duplication of data already managed by the mesh, quantities such as $|e|$ are not stored in `State`. However, the `State` can store this data in situations where it is beneficial or required, such as deforming meshes. Here ϕ , p , and T are independent or primary variables (leaf nodes) and ρ and s are secondary variables (internal nodes). Note that the type of variable of each of its dependencies is unknown by the water content `Evaluator`; they need only implement the `Evaluator` interface. Dependencies of the water content `Evaluator`, expressed as edges in the DAG, are ϕ , ρ , and s . In the implementation of the water content `Evaluator` shown below in Listing 1, specifications for where water content is defined are set and requirements for the dependencies (both `Evaluator` and `Field`) are listed in a method that checks for metadata compatibility. In this way, ownership of the water content `Field` is claimed and

```
class WCEvaluator :
    public SecondaryVariableEvaluator {
...
    void EnsureCompatibility(State& S) {
        ...
        S.RequireField("water_content", "water_content")
        ->SetMesh("subsurface")
        ->SetComponent("cell", CELL, 1);

        S.RequireField("density")
        ->SetMesh("subsurface")
        ->AddComponent("cell", CELL, 1);
        S.RequireFieldEvaluator("density");

        ...
    }
...
};
```

Listing 1: The `Evaluator` for the water content, which is a cell-based field on the mesh, shows the steps that are needed to register the `Evaluator`'s requirements with the meta data stored in the `State`.

requirements made on water content's `Field` are checked for consistency and recursively passed on to the dependencies. We note that many of the calls made in these consistency checks are redundant. However, this process is done once at simulation setup, and is not a performance bottleneck. The full DAG rooted

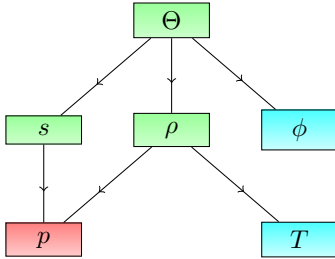


Figure 3: A dependency graph of the algebraic equation for water content, Eq. (2). The primary variable p (pressure) is shown in red, and the independent variables T and ϕ (temperature and porosity) are shown in blue. The secondary variables are shown in green, and not only include the saturation and density (s and ρ), but the water content itself (Θ).

by the water content `Evaluator` is shown in Figure 3.

The dependency graph is a powerful tool for exposing concurrency and managing order of execution for complex, multiphysics simulations. For instance, graph search algorithms can be used to identify physical processes which can be evaluated concurrently, such as s and ρ , or to define an order of evaluation in which each node is guaranteed to be calculated only after its dependencies have been calculated. This also enables lazy evaluation; a kernel is only evaluated when its `Field` is needed, and when its dependencies' values have changed.

3.2.1. Lazy evaluation of the dependency graph

To support lazy evaluation, each `Evaluator` implements the following (public) interface:

```

class Evaluator {
  virtual void
  EnsureCompatibility(State& S) = 0;

  virtual bool
  IsDependency(Key dependency) = 0;

  virtual bool
  HasFieldChanged(State& S, Key requestor) = 0;

  virtual bool
  HasFieldDerivativeChanged(State& S,
                             Key wrt, Key requestor) = 0;
}
  
```

Listing 2: To support lazy (i.e., just-in-time) evaluation of variables, an `Evaluator` provides implementations of the pure virtual methods shown here.

The first method in Listing 2, `EnsureCompatibility()`, simply checks that all dependencies provide everything necessary for computing the specified `Field`. Key elements of a compatibility method were presented in Listing 1, and highlighted its connection to the `Evaluator`'s DAG. The second method simply checks if a `dependency` is in the subgraph rooted at this node. The next two methods play a fundamental role in lazy evaluation as they determine if the `Field`, and its derivative, need to be updated (evaluated) before they can be used by another method closer to the root of the DAG. Specifically, the implementation of `HasFieldChanged()` must return `true` if the `Evaluator`'s `Field` has changed since the last time this method was called by the requesting `Evaluator` (otherwise `false`). Similarly, the implementation of `HasFieldDerivativeChanged()` must return `true` if the derivative of the `Evaluator`'s `Field` *with respect to* a variable named `wrt` has changed. Default implementations of this interface are provided in the base classes for each type of `Evaluator`.

Note that these change monitoring methods do not provide access to the results, which in the case of the `WCEvaluator` sketched above are the water content `Field` and its derivatives. To access these fields, for instance, the PK first calls the change monitoring methods:

```
S->GetFieldEvaluator("water_content")
->HasFieldChanged(*S, "flow_pk");
S->GetFieldEvaluator("water_content")
->HasFieldDerivativeChanged(*S, "temperature",
                           "flow_pk");
```

Then if a change has occurred, the PK uses the `Evaluator`'s `GetFieldData()` method,

```
const CompositeVector& wc =
*S->GetFieldData("water_content");
const CompositeVector& dwc_dT =
*S->GetFieldData("Dwater_content_Dtemperature");
```

These change monitoring interfaces are implemented in different ways for each type of variable. A `PrimaryVariableEvaluator` uses the following model for implementing these interfaces. It exposes the additional interface,

```
class PrimaryVariableEvaluator :
    public Evaluator {
    void MarkAsChanged();
}
```

which is used by PKs and solvers to mark when the primary variable has been updated in the course of an iteration or time step. The `PrimaryVariableEvaluator`

stores a list of requestors which is cleared every time `MarkAsChanged()` is called. When a call to `HasFieldChanged()` is made, the `PrimaryVariableEvaluator` checks the list of requestors to determine if the request has been made previously; if not it adds the requestor to the list of requestors and returns `true`; otherwise it returns `false`.

An `IndependentVariableEvaluator` acts similarly, but it only needs to check if the time in `State` has changed. If the time is different from the last time a function was calculated, it is re-evaluated and its list of requestors is cleared; no `MarkAsChanged()` call is necessary.

Although, `SecondaryVariableEvaluators` also maintain a list of requestors, secondary variables correspond to internal nodes in the DAG, and require an additional (protected) interface:

```
class SecondaryVariableEvaluator :
    public Evaluator {
    virtual void
    UpdateField_() = 0;

    virtual void
    UpdateFieldPartialDerivative_(Key wrt) = 0;
}
```

These additional methods are implemented by the user, and evaluate the physical equations. When `HasFieldChanged()` is called, it loops through its dependencies, recursively calling their `HasFieldChanged()` methods. If any dependency has changed, the requestor list is cleared and the `Evaluator`'s `UpdateField_()` method is called. Recursively traversing the DAG in this way ensures that all fields are updated as required. If no dependencies return `true`, then the requestor's key is checked against the list of requestor and the response is returned; no calculation needs to be done.

The `SecondaryVariableEvaluator` class allows most of the specific example of water content shown in Listing 1 to be abstracted away. Algebraic relationships such as our water content evaluator example need only provide the physical update methods above, the name of the variable evaluated, and the name of all needed dependencies. The location required of `'water_content'` is inferred by the requirement call in the PK using water content; it can be evaluated anywhere that the dependencies also exist. The entirety of the `Evaluator` interface shown in Listing 2 is then automated from just this information.

3.2.2. Automated chain rule for algebraic relations

Additionally, the DAG facilitates the automation of the chain rule in simple cases. Consider the case above, where the derivative of water content with respect to temperature is requested. The result is given by:

$$\frac{d\Theta}{dT} = \phi|e|s(p) \frac{d\rho}{dT} \quad (3)$$

More generally, we use the chain rule:

$$\frac{d\Theta}{dT} = \frac{\partial\Theta}{\partial T} + \frac{\partial\Theta}{\partial\phi} \frac{d\phi}{dT} + \frac{\partial\Theta}{\partial s} \frac{ds}{dT} + \frac{\partial\Theta}{\partial\rho} \frac{d\rho}{dT} \quad (4)$$

To calculate this, the `SecondaryVariableEvaluator` can check whether temperature is a dependency of each of its dependencies (ignoring that term if not, as in the case of s, ϕ), calculate the total derivative of that dependency with respect to temperature using `HasFieldDerivativeChanged()` recursively, multiply it by the partial derivative as calculated using `UpdateFieldPartialDerivative_()`, and add it into the total result.

This capability is extremely powerful; often if a constitutive model changes, it changes its dependencies. For traditional multiphysics codes, this change in dependencies typically means re-engineering much of the code to calculate derivatives. For *Arcos*, this simply means that the graph structure changes, and derivatives are automatically updated correctly. The user need only implement the much simpler partial derivatives with respect to each of its dependencies; total derivatives with respect to primary variables are automatically formed.

3.3. Process models and automated coupling

Process models are organized hierarchically in a dynamically constructed, user-specified tree of equations called a *process tree*. Nodes in the tree (a PK) typically inherit from one of a few common base classes, including,

- `PKPhysicalBase`, which collects mesh information and sets up data for primary variables,
- `PKBDFBase`, which sets up an interface for implicit time integration methods, such as the Backward Difference Formulae (BDF).
- `PKExplicitBase`, which sets up an interface for explicit time integration methods, such as the Runge-Kutta methods.

Alternatively, they may implement the PK interface in a completely custom way, as may be necessary for approaches not based on the method of lines. Internal nodes are MPCs, several of which are provided in *Arcos*. Note that each MPC class *is a* PK.

For a simple example, consider a thermal hydrology problem. Leaf nodes of the tree are the equations for conservation of mass (flow) and energy. The root node is the MPC that couples the two equations; see Figure 4. In a typical application development cycle, the flow PK may be developed separately from the energy PK, and by different domain scientists. Let us assume that, as is common for conservation equations, each PK is to be integrated using an implicit, backward Euler time discretization, requiring a nonlinear solver. Each PK then inherits from the appropriate base class, `PKBDFBase`, a PK with additional methods required for nonlinear solvers. The base class provides a single method implementation for time integration, and specifies an interface for use by a time integrator and nonlinear solver: The `Advance()` method calls a time integrator

```

class PKBDFBase :
public PK {
    // PK interface
    bool Advance(double dt);

    // BDF interface
    virtual void
    Residual(double t_old, double t_new,
             const TreeVector& u_old,
             const TreeVector& u_new,
             TreeVector& res) = 0;

    virtual void
    UpdatePreconditioner(double t,
                        const TreeVector& u,
                        double dt) = 0;

    virtual void
    ApplyPreconditioner(const TreeVector& u,
                       TreeVector& PCu) = 0;

    ...
};

```

Listing 3: The PKBDFBase class for implicit time integration is shown, along with the interfaces to the critical methods. Each PK must provide implementations of these methods.

and nonlinear solver to advance the timestep, using the physics callbacks required by the virtual interface. The limited set of callbacks shown in Listing 3 was shortened for clarity. Other callbacks include error norms, globalization, and checks for solution admissibility. This callback approach is used in many other libraries, including PETSc^{<4>} and Trilinos^{<9>}, and could easily wrap an existing nonlinear solver context from these libraries to create a leaf node. Each of these callbacks work on a `TreeVector`, which is a hierarchical block vector that mirrors the process tree. For our example case, the flow and energy PK nodes have their own `TreeVector`, which simply contains a pointer to the primary variable’s `Field` (e.g., the pressure field described in Section 3.1.1). The MPC’s `TreeVector` stores pointers to its children’s `TreeVectors`, following an identical tree structure to the process tree in Figure 4.

From this structure, it is easy to see how automated coupling is enabled. `WeakMPCs` are sequential couplers, and rely upon each child PK’s `Advance()` method. Two independent time integrators are created, one for each child `PKBDFBase`. The `WeakMPC`’s `Advance()` method calls each child `Advance()` method in turn, in an order specified by the user. Subcycling and other strate-

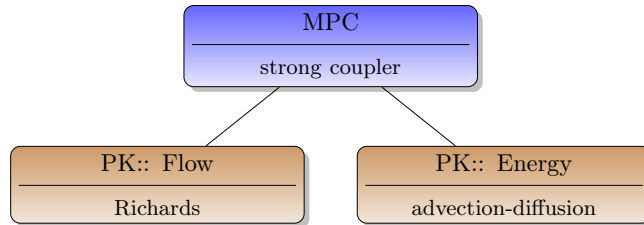


Figure 4: Simple process tree for coupled thermal-hydrology.

gies are easily implemented within the `WeakMPC` concept.

A `StrongMPC` is an implicit coupler, resulting in an implicit coupling of all processes in the subtree it roots, and itself *is a* `PKBDFBase`. For this subtree, only one time integrator is called. The `StrongMPC` must also implement the BDF virtual interface, which it does by calling the child methods with each respective child vector. This implies that, by default, `ApplyPreconditioner()` uses a block-diagonal preconditioner, decoupling the problem.

This approach, while valid, is rarely sufficient to ensure efficient convergence rates of the coupled nonlinear problem. Therefore, *Arcos* encourages treating `StrongMPC` as itself a base class for a smarter coupler. For example, in this problem we may implement a derived class `ThermalHydroStrongMPC`, which inherits from `StrongMPC`. All functionality uses the base class functionality, except the `UpdatePreconditioner()` and `ApplyPreconditioner()` methods. These methods instead create and assemble a better preconditioner, using the diagonal parts from the child PKs, and adding off-diagonal terms as needed. These off-diagonal terms also leverage the DAG to determine derivatives.

In this way, coupling is easily tested as one part of the scientific process. First, fully automated sequential coupling is used. Then, a fully automated `StrongMPC` is used. While this may result in a slow calculation, it requires no additional work by the physics developers related to coupling terms. The comparison of the resulting simulations is a direct measure of the coupling error introduced by sequential coupling. If this error is deemed unacceptable, and the performance of the automated implicit coupler is deemed unacceptable, only then is a physics-specific coupler written. This coupler can overload any or all callback functions, implementing custom preconditioners, solver globalizations such as clipping or limiting, and other physics-driven custom algorithms. In this way, *Arcos* enables rapid development of new simulations, while maintaining enough flexibility to allow domain scientists complete control over coupling strategies.

3.4. Framework usage

Given these components, and referring back to Figure 2, we shortly summarize how an application like *Amanzi* or *ATS* use *Arcos* by discussing the control flow of these applications, as would be implemented in a typical `Driver`.

```

// 1. Construction

// construct a State
State S;

// construct the PK tree by constructing the
// root node, this recursively constructs the
// entire tree
PK root;

// 2. Setup
// In the PK's Setup() method, all Field
// (Section 3.1.1) and FieldEvaluators
// (Section 3.1.2) are "required" in the
// State. This recurses the PK tree calling
// all PKs' Setup() methods.
root.Setup(S);

// In State's Setup method, EnsureCompatibility_()
// of all Evaluators are called, propagating
// Field meta-data through the dependency
// graph as discussed in Section 3.2. Then
// all Field data is allocated.
S.Setup();

// 3. Initialize
// In the PK's Initialize() method, all
// initial conditions for primary variables
// are set, recursively through the PK tree.
root.Initialize(S)

// In the State's Initialize() method, all
// independent and secondary variables are
// initialized.
S.Initialize()

// 4. Set up timestepping
double dt, t_begin, t_end;
S.set_time(t_begin);

// Construct a second State for the "new"
// time.
State S_next(S);

// 5. Advance the simulation

```

```

while (S_next.time() <= t_end) {
    S_next.set_time(S.time() + dt);

    // In the PK's Advance() method, PKs use
    // time integrators and Evaluators to
    // solve for the State at the new time
    // in S_next, as discussed in Section 3.2.
    root.Advance(S, S_next);

    // update the timestep
    *S = *S_next;
}

```

4. Framework demonstration and an example

In this section, we demonstrate how *ATS* uses *Arcos* in two real-world examples. The first focuses on the process of model development, specifically verification testing of complex, coupled systems. The second demonstrates the final graphs and usage of *Arcos* in simulations studying the evolution of the active layer under varying climates.

4.1. Verification testing for coupled systems

Testing is a critical component in the development of simulation software, and in scientific simulation in general. Two types of testing are of interest here: verification testing and integration testing. Verification involves comparing a code's output to a "trusted" solution such as an analytical solution, a constructed solution, or output from a previously verified code. Typically, verification involves significant simplification from the target simulations because of difficulty in obtaining analytical solutions. Integration testing focuses on scenarios that more closely resemble the target simulations to ensure robustness and physical "reasonableness" of the solution. For multiphysics codes, analytical solutions are often not available and integration testing involves comparing to previous versions of the code (regression) or to other codes (benchmarking) or simply "reality checks" on the reasonableness of the solution. A third type of testing - confidence building or "validation" - involves comparing to observations. Building confidence in the model is more focused on the process representation itself, which is an important science activity but outside the scope of this paper. Verification and integration testing are challenging tasks for single- and few-process codes, but become increasingly important and more difficult as the number of processes and possible ways to couple processes together increases. A natural way to approach that testing is to first target specific limiting cases and then incrementally add complexity. For traditional multiphysics codes that explicitly define the combinations of processes and constitutive relationships that may be coupled, that natural systematic approach can become difficult as the number of processes representations increases. In particular, a broad test

convergence may require many more combinations of submodels to be accommodated. If these are explicitly enumerated, as in traditional multiphysics codes, code modifications may be required to add new tests, thus increasing code complexity and generally making it difficult to manage the testing process. The dynamic nature of both the dependency graph and process tree make it possible to build an extensive set of tests of increasing complexity, often with little or no code modifications. This is a powerful advantage of our complexity management approach.

To demonstrate this approach to systematic testing, consider a coupled thermal-hydrological model using Richards' equation for fluid flow and an advection-diffusion equation for energy conservation:

$$\frac{d\Theta}{dt} + \nabla \cdot \frac{\rho}{\mu} k_r K \nabla p = 0 \quad (5)$$

$$\frac{dE}{dt} + \nabla \cdot \kappa \nabla T + \nabla \cdot uq = 0 \quad (6)$$

$$\Theta = \phi \rho s \quad (7)$$

$$E = \phi \rho s u + (1 - \phi) \rho_{soil} u_{soil} \quad (8)$$

Here Richards' equation describing mass conservation of the fluid is shown in Eq. (5), and is solved for pressure p . The mass of the fluid in the pore space is given in Eq. (7) and is the continuous field corresponding to the water content of a mesh cell introduced in the preceding section (Eq. (2)). Similarly, Eq. (6) describes energy conservation in the medium and is solved for the temperature T . Energy E is given in Eq. (8), and is a function of, amongst other things, the internal energy of liquid, u , and soil, u_{soil} . Constitutive equations for density and viscosity, and water retention models for saturation and relative permeability are chosen to complete the system of equations.

In real applications, these constitutive models can be highly nonlinear and lead to problem dependent strong and weak coupling, making the simulation code difficult to test. Thus, a natural approach to building confidence in the capabilities of the simulation code, is to test various simplified models targeting specific limiting cases and coupling scenarios. For instance, in the limit of isothermal, steady-state saturated flow of an incompressible (constant density) fluid, the relative permeability $k_r = 1$, and the pressure solution is linear. Therefore, a common first step in testing is to remove all reference to temperature, provide constants for density and viscosity, and solving the resulting system.

Once this simple problem has been solved successfully, we begin to systematically add complexity, continuing to test at every step. For instance, returning to unsaturated conditions, the relative permeability depends on saturation, and a parameterized model provides the relationship between saturation and pressure. This case is quite common and very challenging tests can be created with prescribed analytical solutions as well as for benchmarking with other codes. It is described by the DAG shown in Figure 5(left), where the absence of the temperature T clearly indicates it is an isothermal model.

Next, equations of state for density and viscosity could be added, and to ease

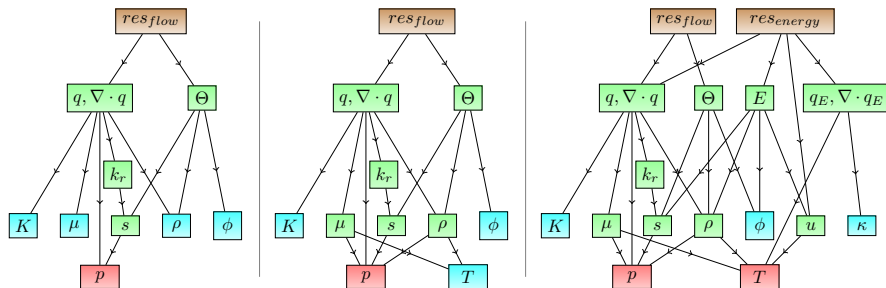


Figure 5: Dependency graphs for the three tests of thermal hydrology are shown with primary variables in red, independent variables in blue, and secondary variables in green. The first test (left) explicitly removes all dependence on temperature. The second (center) solves the isothermal problem with specified, constant temperature. The third (right) solves the coupled system of equations. Neither driver nor flow PK code changes or is even aware of the change.

into a non-isothermal model we add temperature as a prescribed independent variable. This scenario leads to the DAG in Figure 5(middle), where temperature (leaf node) is now present and shaded blue to indicate it is an independent variable. The dependence of density and viscosity on both pressure and temperatures is clearly shown, as well. Finally, once we have explored the complexity of each of the processes independently, we can begin to test the complete thermal-hydrological model. The DAG for this model is shown in Figure 5(right), where it is apparent both the flow and energy residual nodes (the root nodes) are present and shaded brown. In addition, the temperature (leaf node) is now shaded pink to indicate it is a primary variable. As with the more complex single-physics models, analytic solutions to this full nonlinear physical system do not exist for simple boundary conditions on finite domains, but analytic solutions may be prescribed, and the corresponding source terms and boundary conditions can be computed to enable effective verification testing. In addition, to build confidence in the simulation of for real-world scenarios, benchmarking with other codes can be employed.

This multi-step testing process is easily implemented within the *Arcos* framework. As terms may be turned on and off dynamically, and constitutive models for secondary variables replaced by independent variables with user-defined constants defined in an input specification, simplified models are constructed. As each test is run, the dependency graph and process tree are dynamically constructed, and the code driving the simulation and controlling visualization need not change. Each test is run using the same executable but different input files. As the flow PK needs only that **Evaluators** and **Fields** exist for water content and divergence of fluxes, and that the **Fields** meet the requirements specified, it executes identical code paths for all three tests. No **if** statements or special cases are required at any point in the flow PK, ensuring that test coverage is complete for coupled runs.

4.2. Active layer evolution in *ATS*

Finally, to demonstrate the usability of *Arcos* for typical simulations, we discuss an example problem, that of predicting active layer thickness in current-day permafrost in polygonal ground near Barrow, Alaska. In this problem, we solve equations for coupled surface and subsurface, energy and flow, along with a surface energy balance including snow. The subsurface model is a three-phase (ice/water/air) Richards-like equation, with immobile (ice), mobile (water) and infinitely mobile (air) phases. Coupled to the flow equation is an advection-diffusion equation for energy. This combined system is as described in Karra, Painter, and Lichtner <12> but without water vapor diffusion. On the surface, to capture snowmelt runoff, we couple this system to a diffusion wave approximation for flow, modified to include freezing of ponded water, and an advection-diffusion equation for energy transport on the surface. This surface is coupled to the subsurface in a pressure and temperature-continuous coupling. Finally, the forcing of climate conditions such as precipitation, incoming short-wave radiation, air temperature, wind speed, and relative humidity drive a simple surface energy balance which includes snow, as described in Atchley, Painter, Harp, Coon, Wilson, Liljedahl, and Romanovsky <2>.

This model is solved on a two-dimensional transect of a single polygon in Barrow, Alaska covering approximately 24 meters of horizontal extent and 40 meters of vertical depth. The topography of the surface is taken from LIDAR data of the actual site, processed to approximately quarter-meter resolution. In this problem, we drive the system by providing forcing data, i.e. independent variable inputs to the surface balance, based on meteorological data from the Barrow Environmental Observatory from January 1, 2012 – January 1, 2013. Initial conditions are given by a spinup process driving the model using 10-year averaged historical data until annual steady-state. Boundary conditions in the subsurface are given by zero flux of energy and mass. On the surface domain, boundary conditions are specified head, as provided by water table data from gauges in troughs near the transect locations. Surface energy boundary conditions are no diffusive flux of energy, and incoming water is assumed to be advected into the domain at the same temperature as the existing water. These conditions are consistent with viewing the troughs (the boundaries shown here) as natural symmetry points across polygons. Then a year of simulation time is run to predict the active layer thickness, i.e. the maximum thaw depth in the summer season. Ice and liquid saturation, along with temperature, are shown at September 5, 2012, in Figure 6. Scientific results of this and similar simulations are in preparation for a future paper, but here we discuss the role of *Arcos* in enabling scientific results in complex simulations such as these.

Each of the processes discussed above is implemented as a separate PK, as diagrammed in Figure 7. Each PK requires its primary variable evaluator, along with evaluators for the major components, such as water content, advection and diffusion terms, source terms, etc. To demonstrate what code a physics developer and user of *Arcos* would provide, we include examples of components from this problem in Appendix A. Each of the five physical PKs, many Evaluators, and couplers can be seen in *ATS* source repository. Simulations are performed

Sep 05, 2012

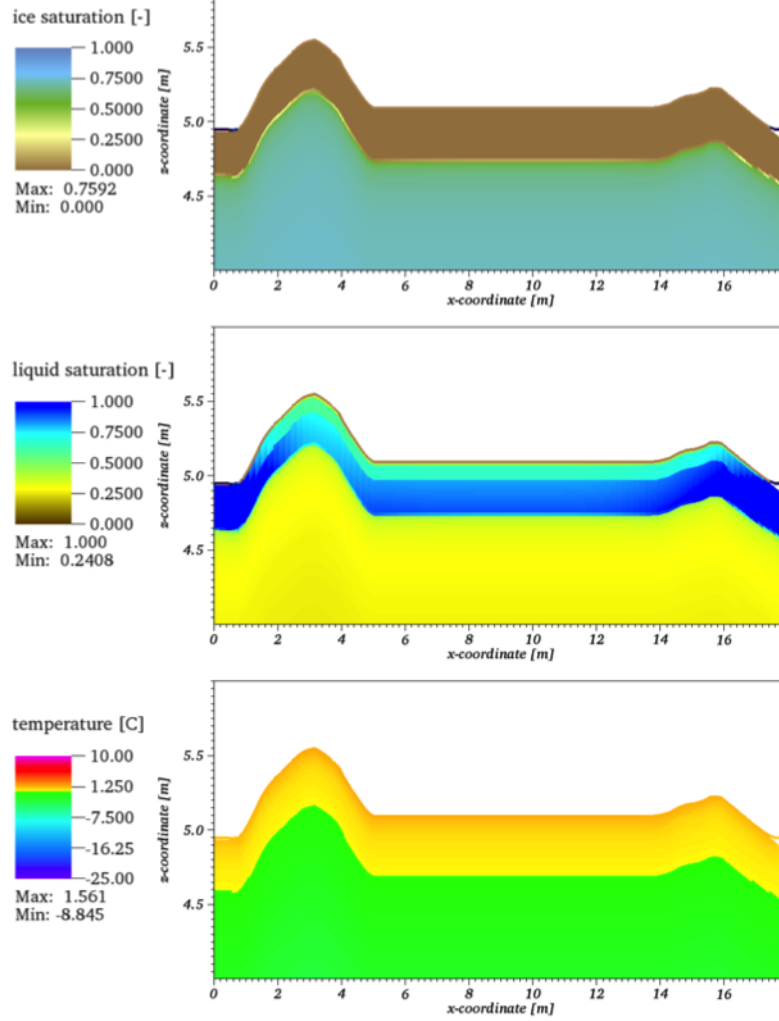


Figure 6: From top to bottom, ice saturation, liquid saturation, and temperature on September 5, the approximate time of maximal thaw depth.

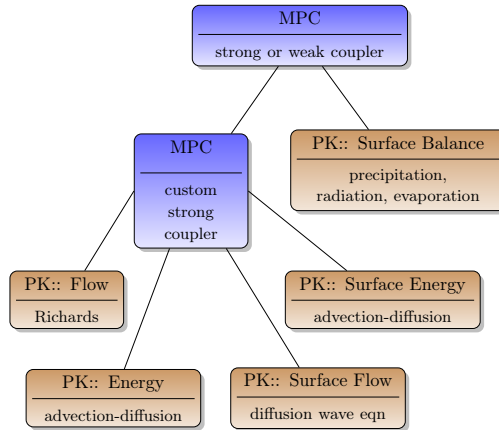


Figure 7: Complete process tree for simulations of coupled surface/subsurface thermal hydrology, driven by a surface energy balance model, as in *ATS*.

using *Amanzi* v0.83 and *ATS* v0.83. The input file for the example shown here is `test13` in the *ATS* test collection, run as preserved in revision 61 of the `ats-tests` repository.

In the full run, *Arcos* is used to form the physics representation of this coupled system. The PK tree for this run is shown in Figure 7, along with the coupling strategy. The final dependency graph, including all primary, independent, and secondary variables is shown in Figure 8.

Coupling for this process-rich set of models was a significant part of the scientific discovery; early approaches attempted to couple flow and energy in the subsurface in one MPC, flow and energy on the surface in a second, and use a `WeakMPC` to couple between the two domains. This approach was found insufficient at capturing transitions between a saturated and unsaturated near-surface region, leading to the need for a custom coupler for all four thermal-hydrology processes. Additionally, the coupling of the surface energy balance PK to the surface/subsurface thermal hydrology (the root node in the process tree of Figure 7) has been an open question throughout numerical experiments. In some problems, weak coupling is sufficient. In others, the coupling between surface temperature (the primary variable in the surface energy equation) and latent and sensible heat calculations in the surface energy balance have been stiff, causing temporal oscillation as the surface temperature adjusts to the energy balance fluxes. The ability to swap coupling strategies dynamically, with no change to the process kernels involved, has been invaluable as we explore these questions at the nexus of physics and algorithm. The formality of a multiphysics strategy such as *Arcos* has been critical for the development and evaluation of complex models such as this example.

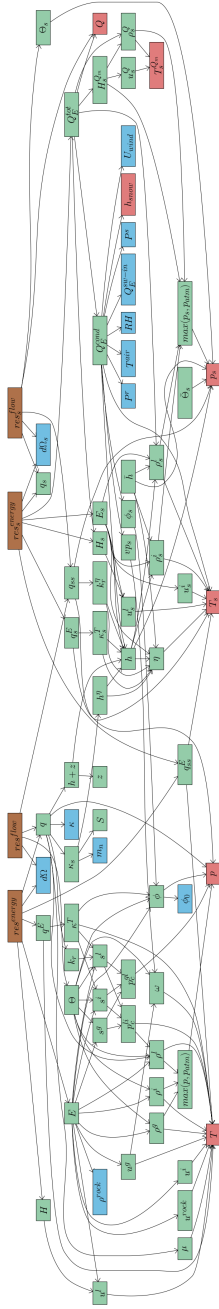


Figure 8: Dependency graph for the simulations shown in Figure 7. Tightly coupled processes naturally are seen as well-connected subgraphs. For this simulation, the left subgraph is surface processes, while the right is surface processes.

5. Discussion

This strategy for model expression and software design has many advantages and a few disadvantages to computational development and productivity.

Advantages of the concept are a more structured design concept to improve flexibility, reliability, systematic testing, and rapid development of models. Using this strategy results in more flexible code components which can be combined dynamically to form new models quickly, improving developer productivity. Different combinations of components still use the same physics implementation for each coupled component, improving code reuse and therefore reliability. These advantages gain even more importance as the number and complexity of processes (and/or developers) increase. Additionally, dependency graphs express the physical system in a form that is directly useful in “coarse-task” parallelism, where multiple evaluators may be evaluated concurrently. This last advantage, while relatively unexplored, offers a path to improve performance of complex model systems on new computer architectures.

There are, however, several disadvantages of using such a framework. The first, and arguably biggest, is that it requires the developer to cede some control and change their approach to software development through a significant learning curve. The developer is forced to implement small blocks of code with significant boilerplate code. This strategy, while having advantages in large code bases with multiple developers with differing expertises, can seem like a significant burden on smaller code efforts with few developers. However, once developers have worked through this learning curve, we find that productivity greatly increases, even for small teams, as new models are quickly introduced and coupled incrementally.

Additionally, there are performance concerns, in that the overhead of traversing, maintaining, and evaluating these graphs can add indirection and therefore wall-clock time and memory requirements to the simulation. This overhead could potentially be significant, especially in small problems (1D, etc), where each `Evaluator` does relatively little work, or in cases where intermediate steps are stored in memory that otherwise would have been discarded. This work is done at high levels (i.e. never in inner loops), and so the run-time overhead is amortized over the evaluation of many cells, especially on larger runs. Some of the run-time overhead is alleviated by making it easier to write correct code – lazy evaluation of the dependency graph often eliminates duplication of work (i.e. evaluating density twice for the same inputs in a coupled simulation). While not yet considered, enabling task-based parallelism could serve to more than make up for these performance losses. Memory concerns could be addressed by tools to fuse multiple `Evaluators` into a single `Evaluator`.

Finally, there are several other minor concerns, including the significant expansion of the code base by the introduction of many, smaller classes. This is both a disadvantage, requiring more work to build and link the application code, and an advantage, as it encourages and enables unit testing, enables parallel build systems for faster code compilation, and other best practices. Incorporating legacy code into a framework like Arcos can require significant refactoring

to abstract functions from the data they act on; this is a challenging area of ongoing research (see, for example, Wang, Xu, Thornton, King, Steed, Gu, and Schuchart <20>).

On the whole, we find the developer productivity improvements due to code testing and flexibility through abstraction to greatly outweigh any negatives, but more focused models with fewer processes and model components might find these disadvantages not worth the advantages.

6. Conclusions

Increasing fidelity of physical models used in simulation is greatly increasing the complexity of multiphysics simulators. To investigate the relative importance of various processes and the nature of coupling among those processes, flexible tools for managing simulation complexity are becoming more important.

We described an approach for managing complexity in multiphysics simulations aimed at land surface and subsurface applications, and discussed the design decisions and implementation details of the *Arcos* framework. The approach prescribes a metadata-driven data manager for ensuring consistency of simulation and input data descriptions. Evaluation of models that depend on the data is managed by a dependency graph, a DAG with variables at nodes and dependencies of variables as edges. Processes are managed hierarchically in a process tree, providing a natural mechanism to organize the many equations in today’s multiphysics simulations. Classes for each of independent, primary, and secondary variables are provided, automating the dynamic formation of the dependency graph. Base classes implementing key automated couplers are provided to act on a process tree describing the coupled system. These base classes may be inherited to create custom couplers, thus eliminating much of the startup costs for a new combination of processes while still allowing domain scientists control over the coupling strategies.

This framework enables powerful capabilities for multiphysics systems. Data requirements are propagated recursively into the dependency graph. Lazy evaluation of the dependency graph eliminates the need for writing bug-prone code to manage the evaluation of models. This graph also assists in the construction of Jacobians by implementing the chain rule for propagating derivatives through the various models. Additional graph algorithms could potentially be developed to expose concurrency at the coarsest level, the physical system, providing a potentially important tool for extreme-scale computing. The flexibility inherent in the dynamically constructed graphs significantly reduces the effort required to swap constitutive models and coupling strategies. It also enables a rigorous testing strategy, thus building confidence in implementations of multiphysics simulations. This framework has been leveraged extensively by *ATS* and *Amanzi* to enable flexible, complex simulation control for ecosystem hydrology and environmental management.

Continuing work is focusing on separating the “domain,” or physical locality where a model resides, from the mesh. Ecosystem applications often involve

processes that are represented on a subset of the full mesh, such as a one-dimensional column of cells in a three-dimensional mesh. For example, root water uptake is typically calculated on a column of cells, but applied as a sink in a hydrology model on the entire three-dimensional subsurface mesh. In these cases, multiple views of the same data may be necessary. Capability supporting these types of single-data, multiple-view problems is an extension of the approach.

Appendix A. Example Physics Modules

Here we provide a simplified subset of the full, actual source code for implementing Richard's equation 1 within the *Arcos* framework. The entire code for the subsurface flow PK can be seen in *ATS*'s source repository in `src/pks/flow/richards/`.

First, we express in a mix of psuedo-code and actual code, the setup, initialization, and part of the residual equation steps of the subsurface flow PK, named Richards.

```

void
Richards::Setup(State* S) {
    // require primary variable pressure
    // -- specify data layout
    vector<string> names(2);
    names[0] = "cell"; names[1] = "face";
    vector<Entity_ID> locs(2);
    locs[0] = CELL; locs[1] = FACE;
    vector<int> n_dofs(2);
    n_dofs[0] = 1; n_dofs[1] = 1;

    S->RequireField("pressure", "flow")
    ->SetGhosted()
    ->SetMesh("subsurface")
    ->SetComponents(names, locs, n_dofs);

    // -- make a child node, add to dependency graph
    PrimaryVariableFieldEvaluator* pressure_eval =
        new PrimaryVariableFieldEvaluator("pressure");
    S->SetFieldEvaluator("pressure", pressure_eval);

    // require water content and evaluator
    S->RequireField("water_content")
    ->SetMesh("subsurface")
    ->AddComponent("cell", CELL, 1);
    S->RequireFieldEvaluator("water_content");

    // require diffusion coefficient, typically

```

```

    // the product relative permeability *
    // density / viscosity
    S->RequireField("permeability_coefficient")
    ->SetMesh("subsurface")
    ->AddComponent("face", FACE, 1);
    S->RequireFieldEvaluator("permeability_coefficient");

    // require the absolute permeability,
    // assuming a diagonal tensor
    S->RequireField("absolute_permeability")
    ->SetMesh("subsurface")
    ->AddComponent("cell", CELL, spatial_dimension);
    S->RequireFieldEvaluator("absolute_permeability");

}

void
Richards::Initialize(State* S) {
    CompositeVector* pressure =
        S->GetFieldData("pressure", "pressure");
    // Read pressure initial conditions from
    // an input file.
    pressure = ...
}

void
Residual(double t_old, double t_new,
          const TreeVector& u_old,
          const TreeVector& u_new,
          TreeVector& res) {
    // Ensure that the old pressure
    // provided is "pressure" in the
    // State at the "old" time.
    assert(u_old->Data() ==
           S_old->GetFieldData("pressure"));

    // Ensure that the new pressure
    // provided is "pressure" in the
    // State at the "new" time.
    assert(u_new->Data() ==
           S_new->GetFieldData("pressure"));

    // Evaluate the time derivative
    // -- update the conserved quantity
    S_old->HasFieldChanged("water_content", "richards_pk");
}

```

```

S_new_->HasFieldChanged("water_content", "richards_pk");

// -- time difference (pseudo-code)
double dt = t_new - t_old;
*res->Data() = 1./dt *
    (S_new_->GetFieldData("water_content")
     - S_old_->GetFieldData("water_content"));

// Continue evaluating flux terms, source terms, etc.
...
}

```

Additionally, we here include a simple yet fully-functional water content evaluator which evaluates the conserved quantity in Richards' equation.

Note that, to change from a standard Richards equation to one which includes ice, we need only provide a new water content `Evaluator` and a new saturation evaluator which includes ice saturation. The Richards PK itself does not change, and every other `Evaluator` stays the same and is reused.

Acknowledgements

This work was supported by the Los Alamos National Laboratory LDRD project 20110068DR, the Department of Energy's Office of Science Next Generation Ecosystem Experiment (NGEE) Arctic project, and the Department of Energy's Office of Science Interoperable Design for Extreme-scale Application Software (IDEAS). LA-UR-14-25386.

- [1] Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) Project, 2014. URL <http://icl.cs.utk.edu/plasma/>.
- [2] A. L. Atchley, S. L. Painter, D. R. Harp, E. T. Coon, C. J. Wilson, A. K. Liljedahl, and V. E. Romanovsky. Using field observations to inform thermal hydrology models of permafrost dynamics with ATS (v0.83). *Geosci. Model Dev. Discuss.*, 8:3235–3292, 2015. doi: 10.5194/gmdd-8-3235-2015.
- [3] V Balaji. FMS: the GFDL Flexible Modeling System, 2005.
- [4] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, and Hong Zhang. PETSc Web page. <http://www.mcs.anl.gov/petsc>, 2014. URL <http://www.mcs.anl.gov/petsc>.

- [5] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, page 66. IEEE Computer Society Press, 2012.
- [6] Martin Berzins, Qingyu Meng, John Schmidt, and James C Sutherland. DAG-based software frameworks for PDEs. In *Euro-Par 2011: Parallel Processing Workshops*, pages 324–333. Springer, 2012.
- [7] Anthony P Craig, Mariana Vertenstein, and Robert Jacob. A new flexible coupler for earth system modeling developed for CCSM4 and CESM1. *Int. J. High. Perform. C.*, page 1094342011428141, 2011.
- [8] Rocky Dunlap, Spencer Rugaber, and Leo Mark. A feature model of coupling technologies for Earth system models. *Comput. Geosci.*, 53:13–20, 2013.
- [9] Michael Heroux, Roscoe Bartlett, Vicki Howle Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams. An Overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.
- [10] C. Hill, C. DeLuca, V. Balaji, M. Suarez, and A. da Silva. Architecture of the Earth System Modeling Framework. *IEEE Comput. Sci. Eng.*, 6(1): 18–28, 2004.
- [11] Laxmikant V Kale, Eric Bohm, Celso L Mendes, Terry Wilmarth, and Gengbin Zheng. Programming petascale applications with Charm++ and AMPI. *Petascale Computing: Algorithms and Applications*, 1:421–441, 2007.
- [12] S Karra, SL Painter, and PC Lichtner. Three-phase numerical model for subsurface hydrology in permafrost-affected regions. *Cryosphere*, 8(1):149–185, 2014.
- [13] Jay Larson, Robert Jacob, and Everest Ong. The model coupling toolkit: A new Fortran90 toolkit for building multiphysics parallel coupled models. *Int. J. High. Perform. C.*, 19(3):277–292, 2005.
- [14] JD Moulton, Meza JC, and M et al Day. High-level design of Amanzi, the multi-process high performance computing simulator. Technical report, DOE-EM, Washington, DC, 2012.
- [15] Patrick K Notz, Roger P Pawlowski, and James C Sutherland. Graph-based software design for managing complexity and enabling concurrency in multiphysics PDE software. *Acm. T. Math. Software*, 39(1):1, 2012.

- [16] SL Painter, JD Moulton, and CJ Wilson. Modeling challenges for predicting hydrologic response to degrading permafrost. *Hydrogeol. J.*, pages 1–4, 2013.
- [17] Scott D Peckham, Eric WH Hutton, and Boyana Norris. A component-based approach to integrated modeling in the geosciences: The design of CSDMS. *Comput. Geosci.*, 53:3–12, 2013.
- [18] René Redler, Sophie Valcke, and Hubert Ritzdorf. OASIS4—a coupling software for next generation earth system modelling. *Geosci. Model Dev.*, 3(1):87–104, 2010.
- [19] SR Slattery, PPH Wilson, and RP Pawlowski. The Data Transfer Kit: A geometric rendezvous-based tool for multiphysics data transfer. In *International Conference on Mathematics & Computational Methods Applied to Nuclear Science & Engineering*, pages 5–9, 2013.
- [20] Dali Wang, Yang Xu, Peter Thornton, Anthony King, Chad Steed, Lianhong Gu, and Joseph Schuchart. A functional test platform for the community land model. *Environ. Model. Softw.*, 55:25–31, 2014.

```

// richards_water_content.hh
#include "secondary_variable_evaluator.hh"

class RichardsWaterContent :
    public SecondaryVariableEvaluator {

public:
    RichardsWaterContent();

    // Required methods from SecondaryVariableFieldEvaluator
    virtual void EvaluateField_(State* S,
        CompositeVector* result);
    virtual void EvaluateFieldPartialDerivative_(State* S,
        Key wrt_key, CompositeVector* result);
}

```

```

// richards_water_content.cc
#include "richards_water_content.hh"

RichardsWaterContent::RichardsWaterContent() {
    // name my_key, the variable to be evaluated
    my_key_ = "water_content";

    // set up dependencies
    dependencies_.insert("porosity");
    dependencies_.insert("saturation_liquid");
    dependencies_.insert("density_liquid");
    dependencies_.insert("cell_volume");
}

void
RichardsWaterContent::EvaluateField_(State* S,
    CompositeVector* result) {
    // Collect dependencies from State.
    // Note that these have already recursively
    // been updated if needed.
    CompositeVector& phi = *S->GetFieldData("porosity");
    CompositeVector& s = *S->GetFieldData("saturation_liquid");
    CompositeVector& rho = *S->GetFieldData("density_liquid");
    CompositeVector& vol = *S->GetFieldData("cell_volume");

    // Evaluate the water content
    *result = phi * s * rho * vol;
}

void EvaluateFieldPartialDerivative_(State* S,
    Key wrt_key, CompositeVector* result) {
    // Collect dependencies from State.
    CompositeVector& phi = *S->GetFieldData("porosity");
    CompositeVector& s = *S->GetFieldData("saturation_liquid");
    CompositeVector& rho = *S->GetFieldData("density_liquid");
    CompositeVector& vol = *S->GetFieldData("cell_volume");

    // Evaluate partial derivatives

```