A problem solving environment for subsurface flow and transport phenomena

Matthew Farthing[a], David Sassen[b], Jan F. Prins[b], and Cass T. Miller[a]

[a]Center for the Advanced Study of the Environment,
Department of Environmental Sciences and Engineering,
University of North Carolina, Chapel Hill, North Carolina 27599-7431, USA

[b]Department of Computer Science,
University of North Carolina at Chapel Hill, North Carolina 27599-3175, USA

We describe a problem solving environment to transform a range of questions about subsurface flow and transport phenomena into numerical simulations. In our prototype implementation, the problem description is presented as a set of differential equations written in a coordinate-free operator form using LaTeX. The simulator is assembled from algorithmic skeletons and solvers represented as components following the Common Component Architecture standard, and executed under the control of the Ccaffeine runtime system. In this paper we describe the structure and operation of our prototype problem solving environment, and illustrate its operation on a simple problem.

1. INTRODUCTION

Subsurface systems hold vast fresh water reserves of vital importance to our lives above ground, but are susceptible to depletion and contamination. Hence questions about the management and remediation of subsurface systems are important for sustainability and regulatory reasons. The ability of computational models to characterize subsurface systems adequately and provide the information necessary for decision-making is currently limited at best, due to the complexity of the models and the computational demands placed by simulations of realistic multiphase systems. Fortunately, physical models are evolving as scientific understanding of subsurface systems improves, and numerical methods are advancing as researchers attempt to address the computational requirements of large-scale simulations.

The question that naturally arises is how to provide access to simulations of complex subsurface systems in the face of constantly changing models and numerical methods. Not only are the simulations subject to constant change, but also the sorts of questions that may be answered using the simulations can change with improved models. The goal of our *problem solving environment* (PSE) is to transform a range of questions about subsurface phenomena into numerical simulations. A PSE fixes a framework for problem specification, and constructs simulators drawn from a set of evolving algorithmic skeletons and numerical methods. The goal is that, over time, an increasing set of problems can be solved with an increasing range of solution methods targeting multiple computational environments, while largely retaining the way problems are expressed. By decoupling the development of algorithms and numerical methods from

specific applications, a problem solving environment can facilitate the introduction of new scientific advancements into models, and obtain better use and reuse of numerical methods as they become available.

Developing a PSE capable of fully realizing the above demands is a difficult task requiring many person-years of effort and necessitating research on many fronts. As a first step, we have focused our efforts on developing a prototype PSE flexible enough to produce efficient solvers for a restricted but significant and meaningful set of problems. Thus, we have selected several model subsurface problems to guide the development of the PSE. The model scenarios include a range of characteristics and difficulty in terms of the dimensionality, steady and transient behavior, single or coupled equations, and both linear and nonlinear problems. Using a broad range of physically meaningful problems helps insure that the PSE will include a sufficient and robust language for problem specification and will be flexible enough to include a set of advanced numerical techniques required for large-scale simulations. Our target scenarios include

1. batch sorption using a distributed first-order model
2. a batch model of the nitrogen cycle, including reactions and interphase mass transfer
3. transient groundwater flow in a confined aquifer
4. transient, variably saturated groundwater flow
5. non-reactive contaminant transport
6. nitrogen cycling in the vadose zone, including the solution of both groundwater flow and transport of multiple nitrogen species

1.1. Problem Solving Environments

Problem solving environments have been characterized as the integration of a problem-domain oriented specification language with numerical solvers, algorithm selection intelligence, and component software technology [10]. This definition can encompass many approaches ranging from the ELLPACK system for solving a large class of PDEs [5] to the MATLAB system. Restricting the problem domain can lead to a narrow but very practical PSE such as the Virtual Cell [15].

A PSE should be more than a toolkit of library routines providing a range of solution methods and functionality. Such an approach lacks a high-level problem specification language, and instead requires a user to construct a simulation using routines found in the library. However, excellent toolkits exist in many areas, such as linear system solvers [3, 8], and PDE solvers [5].

At the other hand, a software synthesis, or transformational, approach constructs a simulator directly from the high-level problem description given by the user. Producing a simulator using this approach involves the application of a set of transformations by which algorithms are introduced and data structures are refined from an abstract, high level into a concrete implementation to be executed on a given platform. The synthesis approach is complex, and is only feasible for a limited domain of problems. Examples include Ctadel [16] which automatically translates high-level descriptions of climate equations into optimized Fortran codes running on vector and parallel computers, and SciNapse [1] which solves a class of partial differential equations. One version of this system is specialized for financial modeling.

In our prototype PSE, we take a hybrid approach that offers the ability to refine a problem specification from an initial high-level description but is not required to generate the entire set of solution methods as would typically be done in a transformational approach. The key to realizing this goal is to provide sufficiently rich internal representations of the underlying mathematical problem as well as a component-based approach to representing the solvers.

Our notation for problem specification is a set of differential equations written in a coordinate-free operator form using LaTeX. In analogy to "literate programming" approaches [12] in which a written document describing the development of a program includes pieces of the program (which are extracted from such a document as input for a compiler), we expect a PSE problem specification to be developed in a written document culminating in the formal equations. Our current working documents follow this approach. Where needed, we use some LaTeX macro definitions to convey semantic information about equations beyond that needed by LaTeX for equation layout.

In an effort to provide the desired flexibility in constructing simulators, we have adopted a component-based approach for assembling numerical solution methods. In very broad terms, such an approach assembles applications out of existing software elements (components) that interact through well-defined interfaces. The specification of component interfaces is independent of the actual language used to implement the components. Moreover, components interact via a framework provided by a component architecture and so are more loosely coupled than one would commonly find within an object-oriented paradigm. We are using the Common Component Architecture (CCA) [4] under development by the CCA Forum based out of the DOE's SciDAC program, and targeted toward scientific computing.

In this paper we describe the structure and operation of our prototype PSE, and illustrate its operation on a simple problem.

2. ARCHITECTURE OF THE PSE

An overview of the PSE is shown in Figure 1. The boxes in this figure correspond to programs employed within the PSE to construct a component library and to generate a simulator.

On the right hand side of this figure, we illustrate the set of steps followed when integrating a new component such as a linear equation solver into the system. The component is expected to be written in a standard language such as C, C++, or Fortran. The component interfaces are described using the *Scientific Interface Definition Language* (SIDL) [7] to express the kinds of values required and the kinds of values generated. The interface description file is processed by the *Babel* processor [7] which generates interfaces that follow CCA conventions, and outputs a program in the selected language that has placeholders for the actual function to be performed by the component. The code implementing the function is inserted into this file, and a standard compiler can then be used to generate an object file that can be loaded dynamically (.so file). This object file becomes a component of the *component library*.

Another part of the integration of a new component is to add one or more rules to the *knowledge base*. The knowledge base is used by the PSE compiler to decide which algorithmic skeleton(s) and numerical methods can be used for a given problem [17, 9]. We have not settled on a form for these rules. The set of rules must be easily expandable to accommodate new forms of problems and additional solution methods, and must also provide a ranking of applicable solvers. Thus far we have a very limited set of problems and solvers and our selection

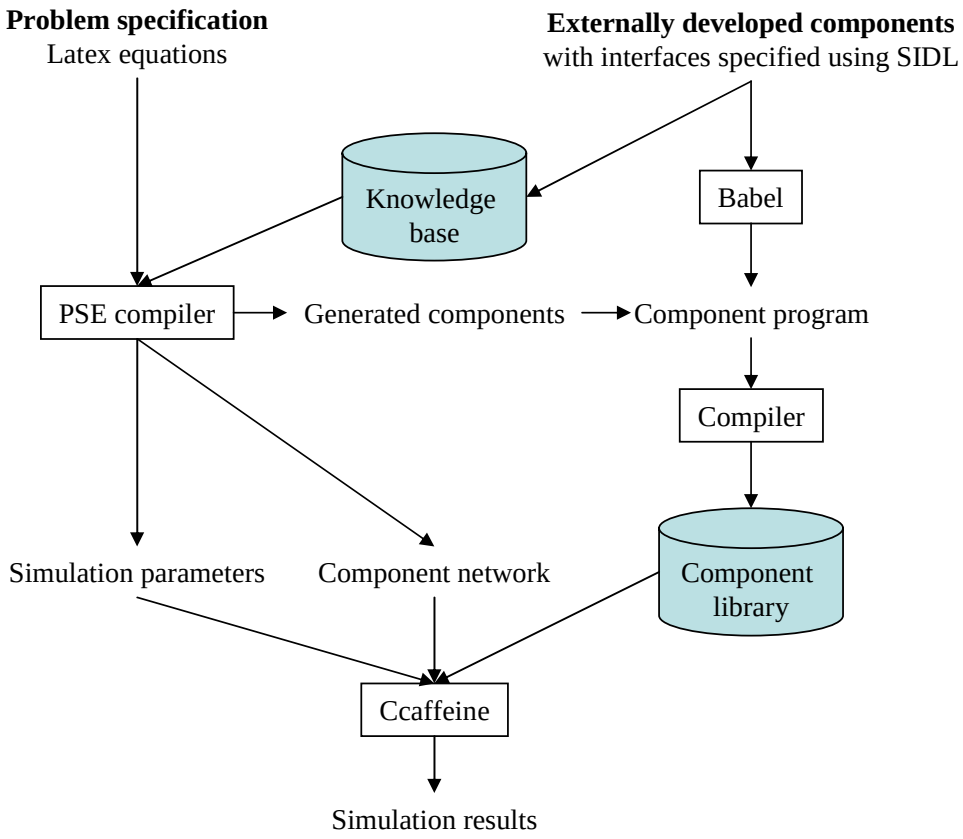

Figure 1. PSE architecture. The left hand side shows the steps involved in creating a simulation. The right hand side shows the steps involved in extending the PSE.

method is ad-hoc.

On the left side of the figure we illustrate the sequence of steps involved in constructing and running a simulation in response to a problem specification. The *PSE compiler* reads the problem specification in LaTeX form and computes a number of attributes and properties of the equations it encounters therein. Using these attributes in conjunction with the knowledge base, the PSE compiler determines a set of applicable algorithms and a relative ranking of the suitability of the possible choices. Once a particular algorithm is chosen, the PSE compiler determines what components it must generate. These would typically be right hand sides of a linear system, a Jacobian, or a residual function. These are generated from the equations and output as small functions to be inserted into CCA components. They need to be named components because in a problem that involves multiple solvers (e.g., one that involves diffusion as well as advection), each solver must be connected with the appropriate piece of the problem to be solved.

The compiler also outputs a *component network* which directs which components are to be used, where each gets its arguments, and what results are produced. This description is read by the *Ccaffeine* run-time system [6] which dynamically loads the necessary components from the component library and starts the simulation running.

The last part generated by the PSE compiler is a parameter file that is used by a standardized component to determine inputs for a given simulation run. By modifying parameters in this file, multiple simulation runs can be made without requiring repeated PSE compiler executions.

2.1. Choosing algorithms

A problem is presented to the PSE compiler as a list of equations and function definitions. Defined functions can be used in the equations. The PSE compiler expands all function applications so that each equation is fully specified. The resultant set of equations are analyzed individually and together. For each equation the following properties are determined.

Equation Type The compiler classifies whether a given equation is an algebraic (possibly nonlinear) equation, an ordinary differential equation (ODE), or a partial differential equation (PDE) based on the appearance of differentials in the input. By default the input is considered algebraic. ODEs are similarly considered implicit by default. However, the compiler will also attempt to determine if the equation is in explicit or semi-explicit form without performing symbolic manipulation.

Order The order is found simply by determining the highest order of any differential in the equation. The order of each differential is included in its definition.

Linearity The PSE compiler constructs a list of *differentiated variables* that appear somewhere in the system of equations in either partial or ordinary differentials. The PSE compiler analyzes each equation to find occurrences of differentiated variables in multiplication or exponentiation expressions. If a differentiated variable appears in such an expression, the equation is marked as nonlinear in that variable.

Once these attributes are computed for each of n equations, an $n \times n$ *coupling matrix* is constructed for the full set of equations. The coupling matrix has value 1 in position (i, j) if a variable that is differentiated in the i^{th} equation occurs in the j^{th} equation, and a 0 otherwise. Since a variable that is differentiated in an equation also occurs in that equation, the (i, i) entry in the matrix is 1 if and only if equation i is a differential equation. The coupling matrix provides important information about the global properties of the problem. For example, a diagonal matrix indicates a completely uncoupled system of differential equations.

The determination of other mathematical properties such as the stiffness of an ODE system or whether a given PDE is parabolic, hyperbolic, or elliptic is significantly more involved and depends, in part, on the dynamics of a simulation and the value of run-time parameters. Developing this capability is part of our ongoing work. Our approach will rely largely on assumptions about coefficient ranges and explicit user input on the allowed range for a given quantity.

2.2. Creating a solver

Once a problem specification has been analyzed, the PSE must then construct an efficient solver. To facilitate this process, we use the CCA paradigm. In CCA, high-level functional objects are represented as *components*.

Each component is treated as a black box, with the only knowledge about it being its name and its *ports*. In the following, we will use the convention of writing component names in Sans Serif and `port` names in `typewriter`. Each component has at least one *provide port*, which denotes what function it implements. Thus, a linear solver might provide the function:

```
Solve( in Matrix A, in Vector b, out Vector x)
```

This could be called a `LinearSolve` port. Any component that provides the `LinearSolve` port is expected to be able to take in a matrix $\mathbf{A}$, a vector $\mathbf{b}$, and provide the solution vector $\mathbf{x}$.

Components also have an arbitrary number of *use ports*. These represent functions in the component that have no actual implementation — at run time each use port must be provided with a component with a provide port of the same name as the use port. Thus, if we have a PDE solver as a component, with a `LinearSolve` use port, we can provide any component with a `LinearSolve` provide port. The idea is to be able to abstract away parts of the program. The PDE solver does not care whether the component that provides the `LinearSolve` is sequential or parallel, whether it is written in C++ or Fortran, or many other implementation details. It depends only on the `LinearSolve`-providing component correctly solving the linear system.

The PSE takes the attributes it has derived from the model and the type of solution being sought and chooses a top-level component, defined as a component that provides a `GoPort`. The CCA calls the `GoPort` function to begin the program. These components are, for example, ForwardInTimeSimulation, SteadyStateSimulation, InverseProblem, etc. Each top-level component has a different set of use ports. For example, ForwardInTimeSimulation requires an `Integrator` port, as well as several pre- and post-processing components. The PSE then chooses a component that provides an `Integrator` port, and that we know to to be appropriate for this problem. As a simple example, if the PSE knows that the system is made up of PDEs, it knows not to select an ODE integrator. This integrator will need other use ports, and the PSE selects appropriate components for those. In this manner, the PSE uses the attributes to select the appropriate component for each use port needed by each component.

Unfortunately, pre-written components cannot serve all our needs. At some point, there must be a connection between the actual mathematical model and the solver. Thus the PSE will write out certain low-level functions (generally objects representing the function to be integrated) that are inserted into interface wrappers generated by Babel. In this manner these problem specific functions become components that can be used in the solver.

2.3. Advantages of CCA

The advantages of this architecture for our needs are several. It provides a convenient abstraction of the solver. Each solver is represented as a network of components. This is useful for the PSE user, as it allows him or her to be able to quickly see and understand how the solver works, far more than if a monolithic code were output. It also enables components of the solver to be tested or profiled in isolation.

The CCA model is also useful for the PSE compiler. The abstraction of provide and use ports allow the PSE compiler to ignore the components that it has already chosen, or will later choose, when choosing a component. Since the ports serve as “contracts” to provide a certain service, theoretically the PSE need only consider one component at a time. That is, when choosing a component that provides an `Integrator` port, what component needs this port should be unimportant, as it should be programmed to deal only with the `Integrator` port, not with a specific component attached to the `Integrator` port. Similarly, other components used by the `Integrator`-providing component should not be important, since it has agreed to provide an integrating service regardless of what components are actually connected to its use ports.

The CCA-based component library also promotes portability and reuse of code. The compo-

nents externalize the algorithmic skeletons, so that the PSE compiler need not maintain these internally in code generation.

Moreover, if a person wanted to test their own code within a complex problem, they could not do so in general without attempting to understand the created program. As an example, consider a person who has created a new linear solver, or who has an old linear solver that they would like to use. By wrapping it in a component, the person can substitute it into a created component network and expect it to work the same without any modifications to the rest of the network. Also, the specific implementation of the CCA that we use, based on Babel [7] and Ccaffeine [6], allows components to be written in any of a number of languages, including C++, Fortran, Java, and Python.

Complex mathematical models often require parallel processing to solve them in a reasonable time period. The CCA is designed with this in mind — parallel processing is done by running the same component network on each node [2]. The components may then communicate "across" nodes, in which they may only communicate with their counterpart component on another node. So an integrator component on node A may not communicate with a linear solver component on node B. They may also communicate "within" a node, via the usual method of provide and use ports. The idea is again abstraction. Components need only know how to communicate with another component just like them.

Finally, we chose the CCA over other, similar component models such as CORBA or COM because those models are designed for business software, not for scientific applications. They are far less efficient in the implementation of interfaces, and they are not specifically designed for parallel computation. Further, they do not have support for Fortran, which is a common language in scientific applications.

3. EXAMPLE FORMULATION — BATCH DISTRIBUTED FIRST-ORDER SORPTION

As an example, we next step through a formulation for the first of our test-suite problems from section 1. We examine a batch system with mass exchange between solid and aqueous phases only — no reactions or sources. This is a simple problem that can be formulated several ways, including a small system of explicit ODEs. We proceed through several steps in the formulation to illustrate how we envision our PSE compiler functioning within a literate programming approach. Then we show the solver output by our prototype PSE.

3.1. Domain background

A general mass balance equation for species ι in phase α for the system under consideration is [14]

$$\frac{d}{dt}(\rho^{\alpha}\theta^{\alpha}\omega^{\iota\alpha}) = \mathcal{I}^{\iota\alpha} + \mathcal{R}^{\iota\alpha} + \mathcal{S}^{\iota\alpha} \tag{1}$$

where t is time, ι is a species index, θ is a volume fraction, ρ is mass density, ω is a mass fraction, $\mathcal{I}^{\iota\alpha}$ represents interphase mass exchange, $\mathcal{R}^{\iota\alpha}$ represents homogeneous (occurring in a single phase) chemical reactions, and $\mathcal{S}^{\iota\alpha}$ is a source term.

For the first example, we can neglect source terms and reactions and still maintain sufficient complexity. We also make a simplification and identify compartments with phases for the purposes of the mathematical formulation based on eqn (1). We assume that there is a single chemical species with a multi-compartment model to include heterogeneity in the solid phase.

We'll suppose that there is a "fast" compartment ($\alpha = s_f$) and a "slow" compartment ($\alpha = s_s$) exchanging mass with the aqueous phase but not each other. The fast compartment will be assumed to be at local equilibrium with the aqueous phase while the mass transfer for the slow compartment will be modeled using a first-order expression. The equilibrium solid-phase mass fractions will be modeled by a Freundlich isotherm [11]

$$\omega_e^{\iota\alpha} = K_f^{\iota\alpha}(C^{\iota a})^{n^{\iota\alpha}}, \quad \alpha = s_f, s_s \tag{2}$$

where $C^{\iota a} = \rho^a \omega^{\iota a}$.

The corresponding mass transfer relations for the slow and fast compartments with the aqueous phase can be expressed as

$$\mathcal{I}_{s_f}^{\iota a} = -\theta^{s_f}\rho^{s_f}\left(\frac{d\omega_e^{\iota s_f}}{dC^{\iota a}}\right)\frac{dC^{\iota a}}{dt} \tag{3}$$

$$\mathcal{I}_{s_s}^{\iota a} = -\theta^{s_s}\rho^{s_s}k_s^{\iota s_s}\left(\omega_e^{\iota s_s} - \omega^{\iota s_s}\right) \tag{4}$$

where $k_s^{\iota s_s}$ is a first-order mass transfer coefficient for the slow solid-phase compartment.

3.2. Formulation

We can write the mass balance equations for our single species as

$$\frac{d\omega^{\iota s_s}}{dt} = k_s^{\iota s_s}\left(\omega_e^{\iota s_s} - \omega^{\iota s_s}\right) \tag{5}$$

$$\frac{d\omega^{\iota s_f}}{dt} = \left(\frac{d\omega_e^{\iota s_f}}{dC^{\iota a}}\right)\frac{dC^{\iota a}}{dt} \tag{6}$$

$$\theta^a\frac{dC^{\iota a}}{dt} = -\theta^{s_f}\rho^{s_f}\left(\frac{d\omega_e^{\iota s_f}}{dC^{\iota a}}\right)\frac{dC^{\iota a}}{dt} - \theta^{s_s}\rho^{s_s}k_s^{\iota s_s}\left(\omega_e^{\iota s_s} - \omega^{\iota s_s}\right) \tag{7}$$

Eqn (6) is redundant. Introducing the retardation factor R_f and dropping the superscript ι, we have

$$\frac{d\omega^{s_s}}{dt} = k_s^{s_s}\left(\omega_e^{s_s} - \omega^{s_s}\right) \tag{8}$$

$$R_f(C^a)\frac{dC^a}{dt} = -\frac{\theta^{s_s}\rho^{s_s}}{\theta^a}k_s^{s_s}\left(\omega_e^{s_s} - \omega^{s_s}\right) \tag{9}$$

for $t \in (t^0, t^f]$ and

$$C^a(t^0) = C_0^a \tag{10}$$

$$\omega^{s_s}(t^0) = \omega_0^{s_s} \tag{11}$$

with the constitutive relations and definitions

$$R_f = 1 + \frac{\theta^{s_f}\rho^{s_f}}{\theta^a}\frac{d\omega_e^{s_f}}{dC^a} \tag{12}$$

$$\omega_e^{s_s} = K_f^{s_s}(C^a)^{n^{s_s}}$$

$$\omega_e^{s_f} = K_f^{s_f}(C^a)^{n^{s_f}}$$

$$\omega^{s_f} = \omega_e^{s_f} \tag{13}$$

and parameters $\rho^{s_s}, \rho^{s_f}, \theta^{\alpha}, K_f^{s_s}, K_f^{s_f}, n^{s_s}, n^{s_f}, k_s^{s_s}$ [14, 11].

Eqns (8) and (9) form an implicit ODE system. As long as R_f remains bounded and non-zero, it can easily be recast to an explicit ODE system that would be appropriate for a wider range of numerical ODE integrators. However, converting to an explicit ODE can increase the system's stiffness and make the problem harder to solve. The behavior of R_f depends on the mathematical form of $\omega_e^{s_f}$ and the parameter values $K_f^{s_f}$ and n^{s_f} chosen for a given problem. In the following, we consider only the explicit form of eqns (8) and (9) for simplicity.

The set of equations forms the first part of the input for the PSE compiler. They are written in LaTeX using a set of macros to disambiguate the mathematical meaning, where needed. For example, the definitions in eqn (12) are written as follows:

```
R_{f} &=& 1 +
\frac{\theta^{s_f}\rho^{s_f}}{\theta^{a}}\od{\omega^{s_f}_{e}}{C^{a}}\\
\omega^{s_s}_{e} &=& K^{s_s}_{f} (C^{a})^{n^{s_s}} \nonumber \\
\omega^{s_f}_{e} &=& K^{s_f}_{f} (C^{a})^{n^{s_f}} \nonumber
```

The PSE compiler ignores formatting commands like `\nonumber` and "&." Note the use of the `\od` macro to specify a (first order) differential.

The second part of the formulation includes the auxiliary conditions and a specification of what information is desired from the problem's solution. For example, taking $\mathbf{y}(t^k)$ to be $[\omega^{\iota s_s}, C^{\iota a}]^T$, one might write

$$\begin{aligned} &\text{Given } \mathbf{y}(t^0) = \mathbf{y}_0 \\ &\text{Find } \mathbf{y}(t^k) \text{ for } t^k \in [t^1, \ldots, t^f] \end{aligned} \tag{14}$$

to determine the species mass fractions in the slow compartment and species concentrations in the aqueous phase at each time step up to some parameter t^f.

3.3. Creating an algorithm

Starting with the specifications above and attributes determined by the compiler, the PSE must produce a useful classification for this problem that will allow it to generate a valid solution approach.

The formulation can be broken down conceptually into two pieces. The first is the set of equations or conservation laws that describe the evolution of the physical quantities together with the necessary constitutive relations like ω_e. These are shown in eqn (8) through eqn (13).

The second is the desired form of the solution. From eqn (14) the compiler can gather that the current problem requires a forward simulation of some transient phenomena. Since no spatial domain or keywords are included, the compiler assumes that the problem lacks spatial dependence. Of course, this needs to be consistent with the evolution equations. The following high-level algorithmic skeleton for the simulation is then selected.

```
initialize
for each time level
    preprocess
    solve evolution equation
    postprocess
finalize
```

This skeleton is called ForwardInTime (FT) and is a component in our library. Here, the `initialize` step includes reading in parameters, setting the value of the solution variables from the initial conditions, and initializing the solvers. The `postprocess` step is used to output the state of the system at each time step, as specified in eqn (14).

The FT algorithm above is very general and doesn't contain much information in itself. The next step in the analysis is to examine the governing equations associated with the problem to determine what actually must be done during each of the high level stages of the FT algorithm. At this point, the PSE compiler makes use of the attributes of the evolution equations listed above to choose a solver. For instance, an integrator capable of solving systems of nonlinear differential algebraic equations (DAEs) of index one can solve explicit ODEs, but one might expect that an explicit ODE solver would be better suited in this case since it is designed for equations that more closely match the formulation provided in the problem specification. This is not always the case, however, since the performance of a numerical solver depends on a host of factors beyond just the form of the equations themselves. Our prototype system bases its ranking of candidate solvers on matching the mathematical form of the analyzed system and includes the ability to override the compiler's selection through explicit instruction. One of the requirements for building a full-featured PSE, however, is to develop a knowledge base that allows for a more sophisticated ranking system incorporating expertise and domain-specific information about the performance of candidate solvers [17, 9].

Making the best choice from a collection of valid solvers is a difficult but happy task for the compiler. Unfortunately, determining a valid solution approach can be more daunting. The way the FT algorithm is structured, the `solve evolution equation` step is supposed to encompass the entire physical problem. While finding a numerical solver capable of solving systems of ODEs is no stretch, we can not rely on having numerical solvers at hand for entire systems of PDEs that are typical for most subsurface flow and transport problems. The PSE's internal representation of the problem specification must be sufficiently rich so that the compiler can decompose the system of evolution equations using a combination of algorithms and available solvers to build a valid solution approach.

To be somewhat more concrete, suppose the PSE only had scalar ODE solvers available. None of these would match the two dimensional system for our example. If the PSE contained an algorithm for decoupling systems of two equations that resembled

```
while ‖y^k − y^{k−1}‖ > ε
    solve equation 1 for y_1^{k+1} using y_2^k
    solve equation 2 for y_2^{k+1} using y_1^{k+1}
```

where k is an iteration index for the decoupling algorithm, the compiler could apply the decoupling algorithm with the scalar ODE solvers to find a solution to the problem. Of course, this approach would likely fail for the batch sorption system eqns (8)–(13), but might succeed in other cases for more loosely coupled problems.

3.4. Creating the component network

The FT algorithmic skeleton is represented in the component library as a CCA component with four use ports: one for a general preprocessor, one for a time-step preprocessor, one for an integrator, and a one for a time-step post-processor. The PSE compiler can associate these

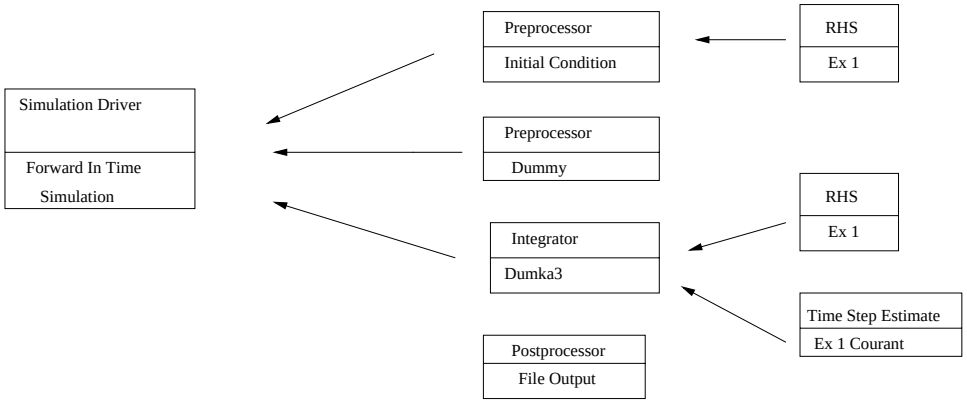

Figure 2. Component diagram for the example problem. The function provided by the component is shown at the top of each box, and the actual component instance is shown at the bottom. Arrows connect components to their users.

ports with any component that meets the CCA interface requirements. Figure 2 shows the components selected to solve the example problem.

The general preprocessor is used to set parameters and initial conditions. We have a component called InitialCondition that provides a uniform interface for obtaining parameters and values from data files, and that initializes the initial state of the system. The time-step preprocessor is run once each time step, before the integration. In this example, we have nothing we need to do, so the pre-processor component is a dummy component.

Since the PSE compiler has determined that an explicit ODE integrator is to be used, it has selected the Dumka3 [13] component. We chose this algorithm for inclusion in our component library for its simplicity and ease of wrapping into a component. The integrator component has two use ports, one to provide a right-hand-side for the problem, and one to provide an estimator for the largest possible stable time step. The right-hand-side component provides a way to set and read the state of the simulation, and provides the function to be integrated. This function is generated by the PSE compiler from the system of explicit ODEs, and is included into the right-hand-side skeleton to create a problem-specific component (here called Ex1) that is compiled and inserted into the component library. We also provide a component that estimates stable time steps to the Dumka3 integrator.

The post-processor component in this case simply reads the necessary quantities out of the right-hand-side and appends them to a file.

Figure 3 shows a Ccaffeine screenshot of the network generated by the PSE compiler. To run the simulation, this network is loaded dynamically by Ccaffeine, and executed by calling the `GoPort` in the FT component. Different simulation runs can be made, using different initial conditions and parameters, by simply modifying the parameter input and re-running the network.

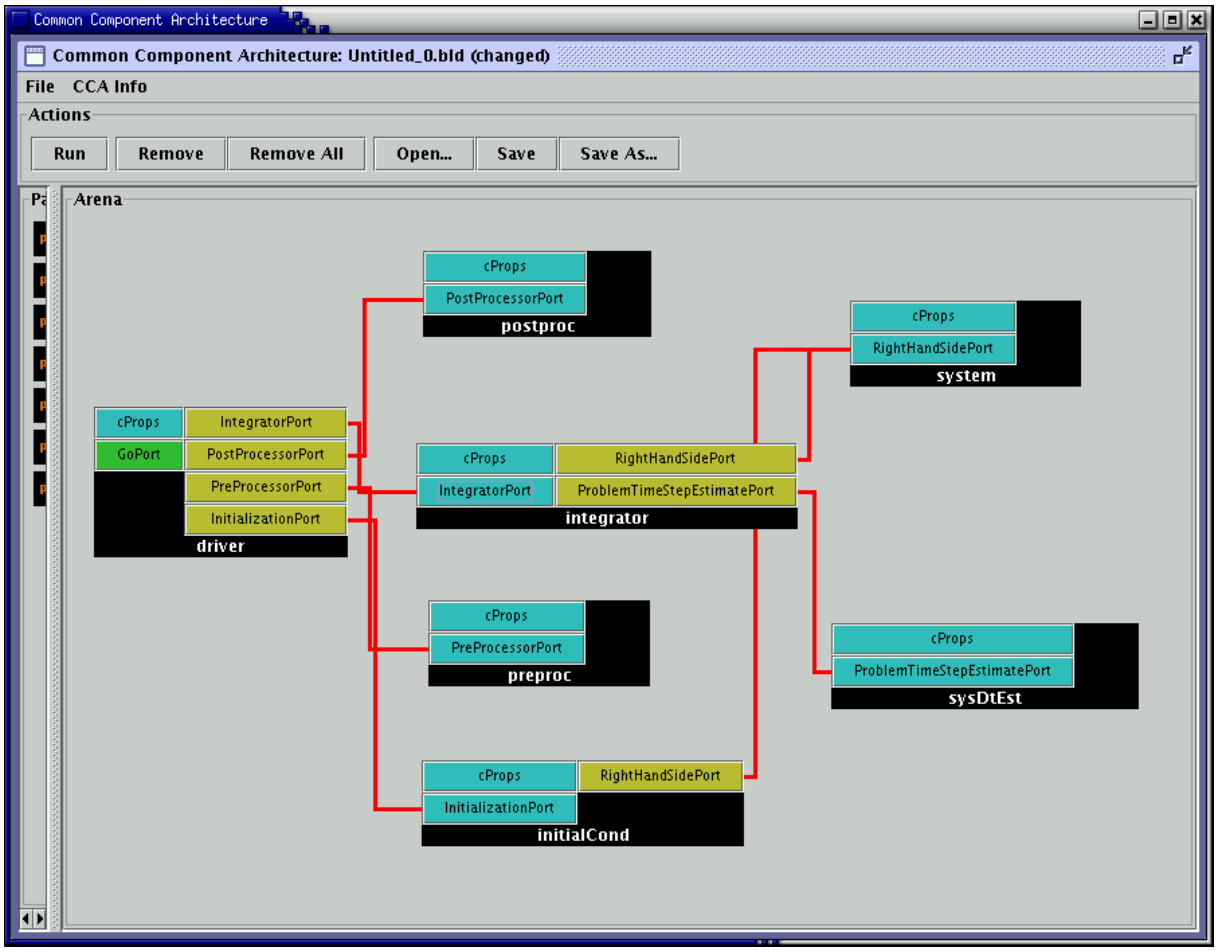

Figure 3. Component network generated for the sample problem, as viewed with the Ccaffeine network graphical tool. Each component is shown as a black box, with its name at the bottom. The use ports are shown on the right, and the provide ports are shown on the left. A line connects each use port with a provide port.

4. CONCLUSION

The objective of our PSE is to improve the current state of subsurface modeling by accelerating the process of developing simulators, facilitating the introduction of new scientific advancements into models, and encouraging the application of more realistic models. The key to this approach is to combine a high-level problem specification with a framework that is sufficiently flexible to include advanced solution methods suited to large-scale problems, incorporate legacy solvers, and introduce new methods as they become available. Finally, a succesful PSE must be able to generate simulators that can run effectively on a variety of sequential and parallel platforms.

Our prototype PSE thus far only handles a small subset of the intended class of problems, but has an architecture that we believe can accommodate each of these objectives on the larger range of problems.

ACKNOWLEDGEMENTS

Funding for this work was provided by the Department of Energy grant DE-FG02-02ER63369.

REFERENCES

1. Robert L. Akers, Elaine Kant, Curtis J. Randall, Stanly Steinberg, and Robert L. Young. Scinapse: A problem-solving environment for partial differential equations. *IEEE Journal of Computational Science and Engineering*, 4(3):32–42, 1997.
2. Benjamin A. Allan, Robert C. Armstrong, Alicia P. Wolfe, Jaideep Ray, David E. Bernholdt, and James A. Kohl. The CCA core specification in a distributed memory SPMD framework. *Concurrency and Computation: Practice and Experience*, 14:1–23, 2002.
3. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide (3rd ed.)*. SIAM, Philadelphia, PA, 1999.
4. David E. Bernholdt, Wael R. Elwasif, James A. Kohl, and Thomas G. W. Epperly. A component architecture for high-performance computing. In *Proceedings of the Workshop on Performance Optimization via High-Level Languages and Libraries (POHLL-02)*. ACM, 2002.
5. R. F. Boisvert and D. K. Kahaner. Deqsol and ellpack: Problem solving environments for partial differential equations. *Office of Naval Research Asian Office Scientific Information Bulletin (NAVSO P-3580)*, 16(1):7–19, 1991.
6. CCA Forum, http://www.cca-forum.org/ccafe/ccaffeine-man/. *Ccaffeine Reference Framework for CCA Components*, 2003.
7. Tamara Dahlgren, Thomas Epperly, and Gary Kumfert. *Babel Users' Guide*. Center for Applied Scientific Computing, Lawrence Livermore National Lab, http://www.llnl.gov/CASC/components/docs/users_guide.pdf, 2004.
8. John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. Flame: Formal linear algebra methods environment. *Transactions on Mathematical Software*, 27(4):422–455, December 2001.
9. E.N Houstis, A.C Catlin, J.R Rice, V.S. Verykios, N. Ramakrishnan, and C.E. Houstis. PYTHIA-II: a knowledge/database system for managing performance data and recommending scientific software. *ACM Transactions on Mathematical Software*, 26(2):227 – 253, June 2000.
10. E.N Houstis and J.R Rice. Future problem solving environments for computational science. *Mathematics and Computers in Simulation*, 54:243 –257, 2000.
11. J. F. Kanney, C. T. Miller, and D.A. Barry. Comparison of fully coupled approaches for approximating nonlinear transport and reaction problems. *Advances in Water Resources*, 26:353–372, 2003.
12. Donald Knuth. Literate programming. *The Computer Journal*, 27:97–111, 1984.
13. A. Medovikov. High order explicit methods for parabolic equations. *High order explicit methods for parabolic equations*, 38:372–390, 1998.
14. C. T. Miller, G. Christakos, P. T. Imhoff, J. F. McBride, J. A. Pedit, and J. A. Trangenstein. Multiphase flow and transport modeling in heterogeneous porous media: Challenges and approaches. *Advances in Water Resources*, 21(2):77–120, 1998.
15. J. Schaff, C. C. Fink, B. Slepchenko, J. H. Carson, and L. M. Loew. A general com-

putational framework for modeling cellular structure and function. *Biophysical Journal*, 73:1135–1146, 1997.

16. Robert van Engelen, Lex Wolters, and Gerard Cats. Tomorrow's weather forecast: Automatic code generation for atmospheric modeling. *IEEE Journal of Computational Science and Engineering*, 4(3):22–31, 1997.
17. S. Weerawarana, E.N Houstis, J.R. Rice, A. Joshi, and C.E. Houstis. PYTHIA: a knowledge-based system to select scientific algorithms. *ACM Transactions on Mathematical Software*, 22(4):447 – 468, December 1996.