Peridigm Summary Report: Lessons Learned in Development with Agile Components

David Littlewood, Michael Parks, Andy Salinger, and John Mitchell

September 16, 2011

Executive Summary

This report details efforts to deploy *Agile Components* for rapid development of a peridynamics code, *Peridigm*. The goal of Agile Components is to enable the efficient development of production-quality software by providing a well-defined, unifying interface to a powerful set of component-based software. Specifically, Agile Components facilitate interoperability among packages within the *Trilinos Project*, including data management, time integration, uncertainty quantification, and optimization. Development of the *Peridigm* code served as a testbed for Agile Components and resulted in a number of recommendations for future development. Agile Components successfully enabled rapid integration of *Trilinos* packages into *Peridigm*. A cost of this approach, however, was a set of restrictions on *Peridigm*'s architecture which impacted the ability to track history-dependent material data, dynamically modify the model discretization, and interject user-defined routines into the time integration algorithm. These restrictions resulted in modifications to the Agile Components approach, as implemented in *Peridigm*, and in a set of recommendations for future Agile Components development. Specific recommendations include improved handling of material states, a more flexible flow control model, and improved documentation. A demonstration mini-application, *SimpleODE*, was developed at the onset of this project and is offered as a potential supplement to Agile Components documentation.

1 The Agile Components Methodology

Agile Components is a strategic effort at Sandia to maximize both developer efficiency and the impact of our computational science libraries. The intent is to align the many software development efforts at Sandia, in particular those tied to the *Trilinos Project*, for the modeling and simulation of physical systems (primarily PDEs). The strategic goals of the Agile Components methodology are:

- Enable rapid development of new production codes,
- Embed these codes with transformational design, analysis, and decision-support capabilities (*e.g.*, embedded uncertainty quantificaiton (UQ), sensitivity analysis, and optimization),
- Reduce redundancy.

The Agile Components methodology utilizes software components and software quality tools from the *Trilinos Project*, a Sandia effort to develop algorithms and enabling technologies within an object-oriented software framework for the solution of large-scale, complex multi-physics research and engineering problems [2]. Some *Trilinos* capabilities are shown in Figure 1.

The Agile Components methodology focuses on delivering capabilities of *Trilinos* software components through a set of well-designed interfaces. As a practical example, a schematic of the *Albany* Agile Components code is shown in Figure 2, which depicts several of the capabilities from Figure 1 connected through interfaces.

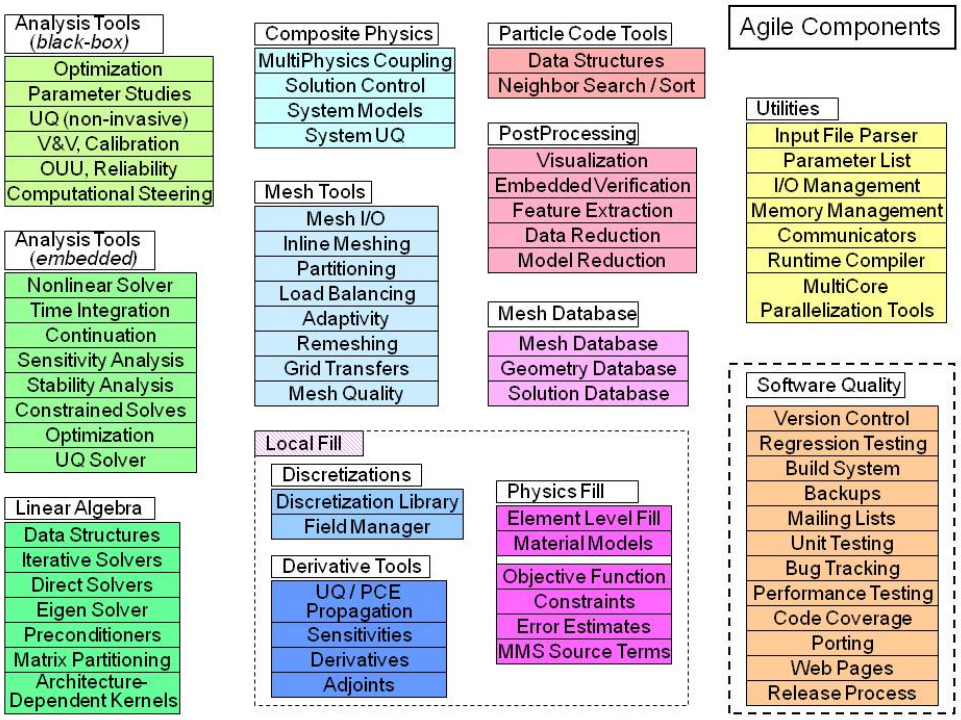

Figure 1: Capabilities delivered by the Agile Components methodology via *Trilinos* components.

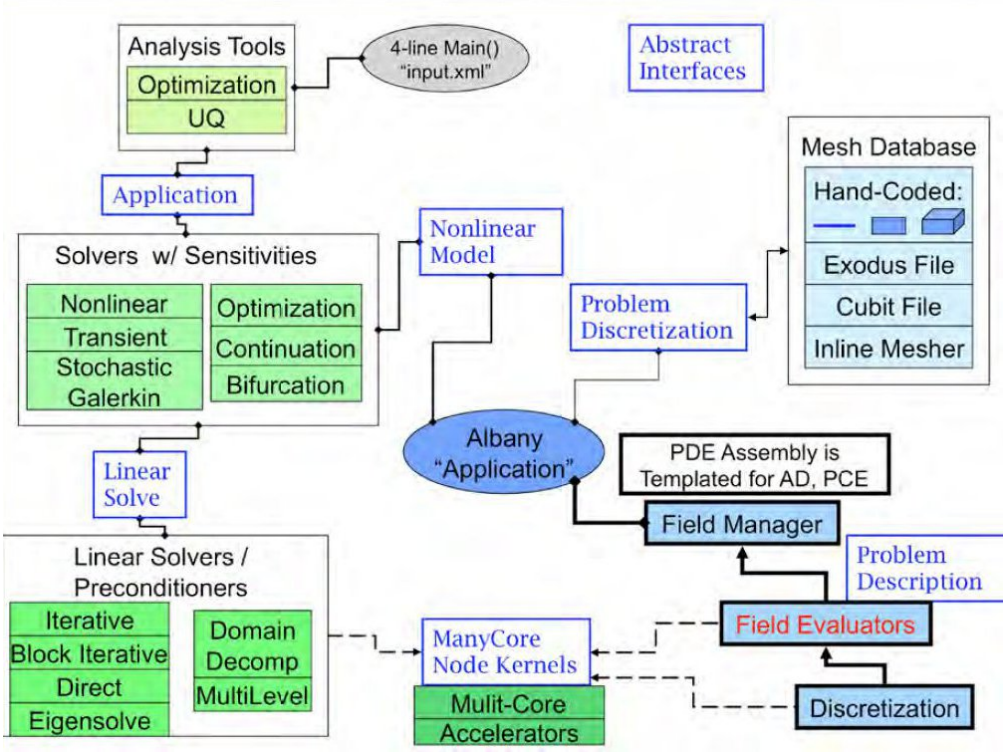

Figure 2: The component-based *Albany* code: A deployment of capabilities from Figure 1 via Agile Components interfaces to form a demonstration application.

These design of these interfaces is key to delivering capabilities through Agile Components to an end-user application. Each software component in *Trilinos* implements its own interface, requiring a component-based code to adhere to the different interfaces of each of the *Trilinos* software components it consumes, as depicted in Figure 3. This increases time to deployment, and requires code to be refactored every time a component interface changes. Figure 4 illustrates the Agile Components approach to *Trilinos* solver components; application codes are presented with a single interface with which to consume solver capabilities. In addition to supplying a unified interface to component functionality, this interface can provide metadata analysis utilities such as embedded UQ, sensitivity analysis, and optimization. Applications codes adhering to an Agile Components interface receive these additional capabilities with without additional burden on the developer.

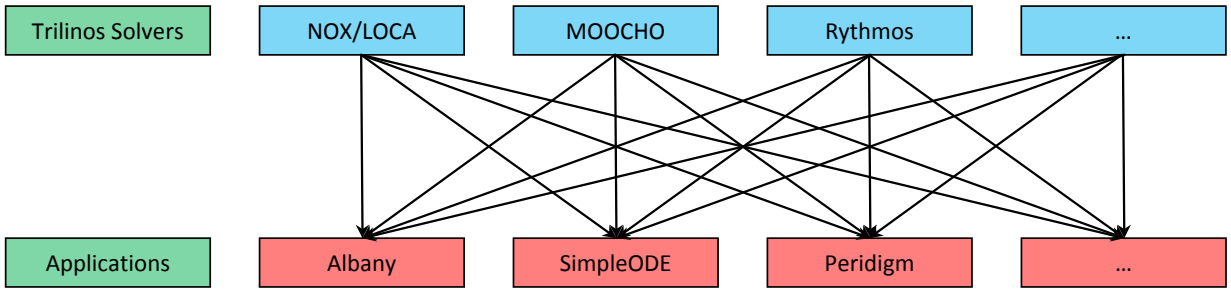

Figure 3: Direct interfaces between *Trilinos* components and application codes. Each solver component implements its own interface that must be utilized by the application code.

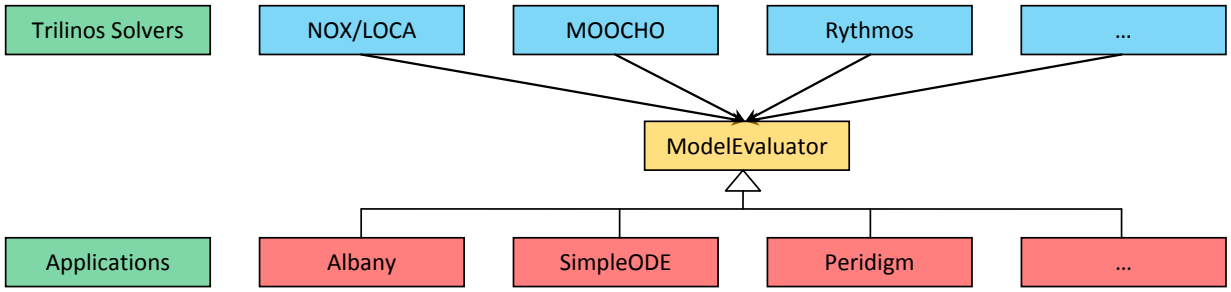

Figure 4: *ModelEvaluator* interface between *Trilinos* components and application codes. A common interface facilitates use, deployment, and adoption of solver components.

Remark 1.1. **The primary roles of Agile Components are to (1) deliver interfaces to software components to facilitate their use, deployment, and adoption, and (2) deliver metadata analysis tools (embedded UQ, sensitivity analysis, optimization, etc.)**

1.1 The Agile Components Interface Paradigm: The *ModelEvaluator* Class

To achieve the strategic goals of the Agile Components methodology, a sufficiently general interface is required. The *ModelEvaluator* interface has been proposed as the interface utilized by solver components and implemented by model developers.

The leftmost column in Figure 5 lists the mathematical problems for which solvers exist within *Trilinos*. These solvers address mathematical problems without specific knowledge of the right-hand-side, $\mathbf{f}$. Model developers provide specific instances of $\mathbf{f}$. The *ModelEvaluator* interface represents a contract between the solver and the model. The *ModelEvaluator* supports the solution of nonlinear equations, stability analysis, explicit first-order ODEs, implicit first-order ODEs, explicit first-order ODE forward sensitivities, implicit

first-order ODE forward sensitivities, constrained optimization, unconstrained optimization, and ODE constrained optimization.

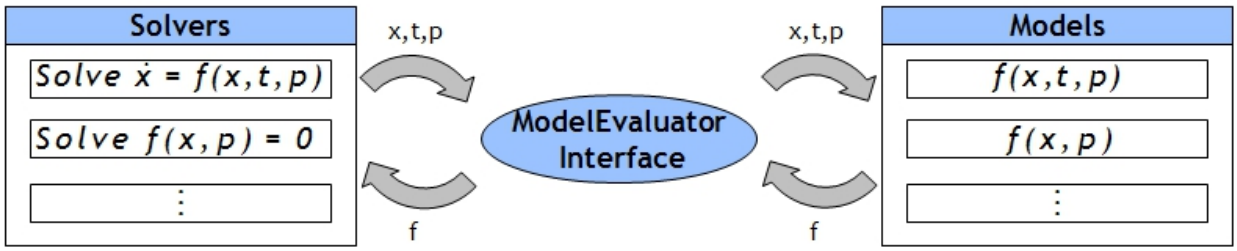

Figure 5: Diagram of the Agile Components *ModelEvaluator* interface.

Specifically, the contract enforced by the *ModelEvaluator* assumes that the models **f** are stateless, pure mathematical functions. This is a reasonable assumption, as an optimizer will want to evaluate a model for many different inputs without regard to the order in which the inputs are evaluated. An important property of pure mathematical functions is that they are *referentially transparent*. A referentially transparent function **f** always provides the same output when given the same input, as it contains no state.

Definition 1.1. An expression is referentially transparent if it can be replaced with its value without changing the behavior of a program [1].

2 Practical Experiences with Agile Components

The Agile Components methodology has been deployed for the development of several application codes, including the multiphysics finite element code *Albany*, the peridynamics code *Peridigm*, and the demonstration mini-application, *SimpleODE*. An overview of the development process for each of these codes is given below, with an emphasis on the use of *Trilinos* Agile Components.

Remark 2.1. **Models that naturally fit into the assumptions and implicit contract enforced by the *ModelEvaluator* interface can naturally and relatively effortlessly utilize this interface for rapid development. Models that violate the assumptions of this interface can not.**

2.1 *SimpleODE*, an Agile Components Demonstration MiniApplication

During the course of the development of *Peridigm*, we created *SimpleODE*, a MiniApp demonstrating the usage of Agile Components on the simplest problem we could conceive:

$$\dot{x} = -cx, \tag{2.1}$$

with $c > 0$ a constant. The principle purpose of *SimpleODE* is to provide a template for other developers wishing to create their own agile component code. Unlike *Albany* (c.f. §2.2), a full-blown Agile Components application, *SimpleODE* contains all of the Agile Components infrastructure without a complex application model to obscure that infrastructure.

We present the execution control flow of *SimpleODE*, as it will be useful to our discussion later. Referring to Figure 6, we see that control flow begins with `main()` making a call to a *ModelEvaluator*, which wraps a Rythmos solver. The Rythmos solver, in turn, makes calls to a provided instance of a *ModelEvaluator* that merely returns $-cx$ when given x. Note that the first model evaluator encapsulates the entire problem.

The problem solved by *SimpleODE* fits completely and naturally into the assumptions and contract enforced by the model evaluator interface, as the function $\mathbf{f}(x) = -cx$ is a pure mathematical function, and thus referentially transparent.

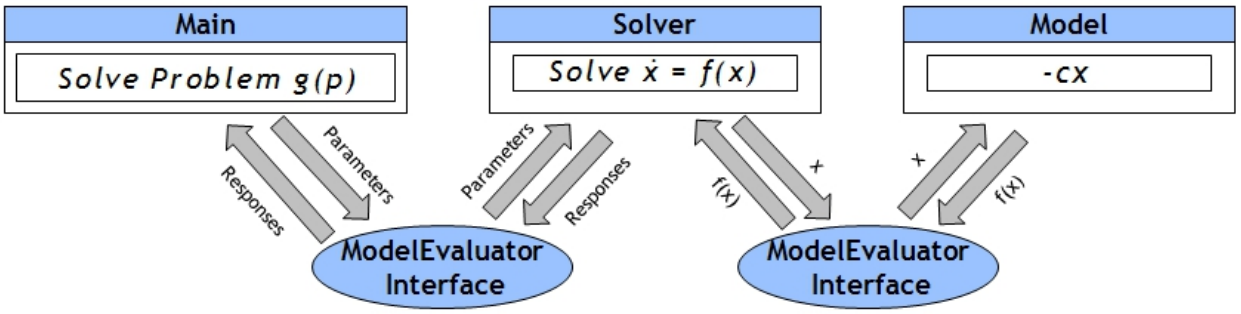

Figure 6: Diagram of the control flow in *SimpleODE*, showing two use cases of *ModelEvaluator*. Execution begins with *main()*, which calls a *ModelEvaluator*-wrapped Rythmos solver to solve an abstract problem $g(p)$. The Rythmos solver integrates an ODE in time, with the solver making calls to a separate *ModelEvaluator*-wrapped model.

2.2 *Albany*, an Agile Components Demonstration and Prototype Code

The *Albany* code [7] is the original code designed to drive and demonstrate the Agile Components vision. *Albany* serves as an early adopter and ripener of new algorithm libraries, a place to design and mature interfaces, and as a prototype *Trilinos*-based application with a functioning software quality environment.

Trilinos, in its role as a delivery vehicle for software libraries, has rapidly expanded beyond solvers to include numerous other computational science libraries. *Albany* has helped mediate this growth by being an early adopter of many of these new capabilities. *Albany* has been a place to mature the usability, robustness, flexibility, and scalability of these algorithms. It has also been a place to devise a larger domain model, so that the aggregate of capabilities does not have gaps or overlap.

A large fraction of the new capabilities vetted in *Albany* are tailored to partial differential equations and finite element methods, and are not directly relevant to *Peridigm*. However, the use of the model evaluator as the application abstraction in Figure 2 and the wrapping of the Rythmos and NOX solvers within this abstraction, as described in §2.1, occurred within *Albany*. Much of this work was concurrent with, and was influenced by, *Peridigm* development.

In its role as a prototype code, *Albany* has a small physics set as part of the main code base. This includes heat transfer, incompressible fluid flow, and a thermo-electrical model. These are meant to be sufficient for testing the full infrastructure (from input file parsing to uncertainty quantification). The embedded uncertainty quantification research program of Phipps uses *Albany*, and these physics sets, to demonstrate and evaluate their algorithms. In addition, application-oriented projects have put physics sets within Albany. We describe those projects briefly in the following subsections.

2.2.1 *Albany* QCAD: Quantum Device Simulation Tool

The QCAD project [4] is building a quantum device simulation tool within *Albany*. The goal is to be able to evaluate and design structures built from the tools of the semiconductor industry (doped semiconductors, oxide layers, gates) to isolate quantum dots. The primary computational science focus has been on development of a nonlinear Poisson solver to solve for the electrical potential in the device as a function of design and applied voltage. As a second step, a Schrödinger solver has been developed for the quantum effects in the region of the device with few free electrons, and coupled to the nonlinear Poisson solver.

The QCAD application was an ideal fit for the current state of the Agile Components vision and the pieces already assembled and matured in the Albany infrastructure. The application's models are stateless, referentially transparent pure mathematical functions, satisfying the assumptions of the *ModelEvaluator* interface. It could make full use of the finite element mesh database, discretization algorithms, automatic

differentiation technology, rapid insertion of new physics modules, nonlinear solvers, eigensolver, and optimization algorithms. As a result, this project is on its third year milestones at the end of one year, as is already being used by customers as a production code. Much of the infrastructure improvements motivated by QCAD involve post-processing capabilities, much of which reside in *Albany*, but some have migrated back to *Trilinos* in terms of bug fixes and enhancements to the SIERRA toolkit (STK) and Sandia Engineering Analysis Code Access System (SEACAS) I/O tools.

2.2.2 *Albany* LCM: Computation Mechanics Research Code

The Laboratory for Computational Mechanics (LCM) project [5] is meant to be a research and development platform for mechanics, particularly failure and fracture models. Its serves as a platform to rapidly and flexibly test new ideas and algorithms, with subsequent migration into the associated production codes. As an open source code, it also serves as a vehicle for external collaborations.

The LCM is written as a (configure-time) physics set within the *Albany* code. As with QCAD, the LCM project has made excellent progress in just one year, being an excellent fit with much of the finite-element based technology in *Albany*. However, the LCM project also presents some of the same issues as the *Peridigm* application that exposes assumptions or gaps in the current Agile Components instantiation. This includes the potential statefulness of the model, where the current residual evaluation depends on the history and not just the current state. In the case of solid mechanics, this can be the residual stress field or a model for material damage. The LCM and *Peridigm* projects have leveraged each others experience in dealing with state data. As with *Peridigm*, the time-dependent LCM applications are second order in time. One implicit and one explicit version of second-order time integrators have been written into Piro in *Trilinos*, though currently require a "hack" to the model evaluator interface.

2.3 *Peridigm*, A Peridynamics Code

The *Peridigm* code, through consequence of its computational structure and material models, violates several assumptions of the *ModelEvaluator* interface. We first discuss the mathematical formulation of the peridynamic equation of motion, followed by specific details of each development issue encountered, concluding with specific recommendations for future Agile Components development.

In the peridynamic theory, the deformation at a point depends collectively on all points interacting with that point. Using the notation of [8], we write the peridynamic equation of motion as

$$\rho(\mathbf{x})\ddot{\mathbf{u}}(\mathbf{x},t) = \int_{\mathcal{H}_\mathbf{x}} \left\{\underline{\mathbf{T}}\left[\mathbf{x},t\right]\left\langle\mathbf{x}'-\mathbf{x}\right\rangle - \underline{\mathbf{T}}\left[\mathbf{x}',t\right]\left\langle\mathbf{x}-\mathbf{x}'\right\rangle\right\} dV_{\mathbf{x}'} + \mathbf{b}(\mathbf{x},t), \tag{2.2}$$

where ρ represents the mass density, $\underline{\mathbf{T}}$ the force vector state, and $\mathbf{b}$ an external body force density. A point $\mathbf{x}$ interacts with all the points $\mathbf{x}'$ within the neighborhood $\mathcal{H}_\mathbf{x}$, assumed to be a spherical region of radius $\delta > 0$ centered at $\mathbf{x}$. δ is called the *horizon*, and is analogous to the cutoff radius used in molecular dynamics. For more on the peridynamic equation of motion, see see [8].

Peridigm solves (2.2) with appropriate initial and boundary conditions. Based upon *Trilinos* components, *Peridigm* inherits standard software practices and tools used in *Trilinos* development, some of which are depicted graphically in Figure 7. *Peridigm* provides parallel simulation, peridynamic material models such as the linear peridynamic solid (LPS) model [8], a peridynamic plasticity model [3], and a peridynamic viscoelastic model. *Peridigm* provides both explicit and implicit time integration, and performs multi-material simulations. It has the capability to dynamically load balance running simulations to achieve better parallel performance. It is capable of reading Exodus meshes as input or internally generating meshes of simple geometric solids. Output is to VTK-format files. The user can define their own "compute" styles to output quantities of interest to them.

Remark 2.2. **Differences between peridynamics and PDE-based finite element models have motivated new development in Agile Components. Key driving factors were the need for history-dependant material models and load rebalance within *Peridigm*.**

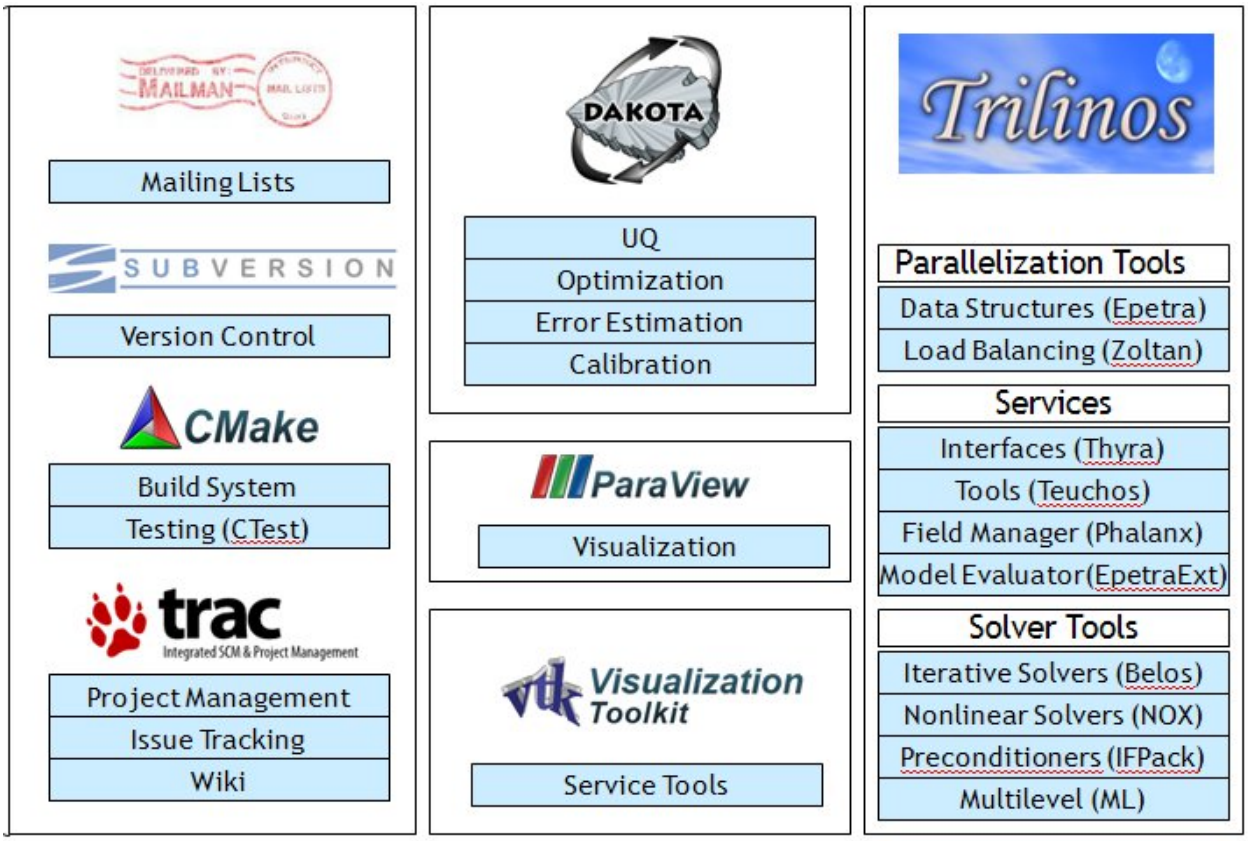

Figure 7: Graphical representation of the software components, interfaces, tools, and quality practices that comprise *Peridigm*.

2.3.1 Handling Model State

The *ModelEvaluator* interface assumes stateless models. When implementing *Peridigm*, we encountered two significant issues regarding the handling of model state. The first arises because many material models of interest contain state in the form of history dependence, and these models violate the assumption made by the *ModelEvaluator* of a stateless model. The second issue was discovered when attempting to implement load rebalance. Each *ModelEvaluator*-derived class creates and stores its own parallel data distribution across processors, which is a form of state data. This behavior is ostensibly in violation of the *ModelEvaluator* contract that they be stateless. This only manifests as a problem when attempting to resistribute data across processors, which is necessary when preforming either load rebalance, or mesh adaptivity.

History Dependance in Material Models Practical peridynamic material models contain state in the form of history dependence. This history dependence is not unique to Peridynamics – perhaps 99% of the material models of interest in practical computational solid mechanics applications are not stateless. Within *Peridigm*, specifically, all models contain a fracture rule governing the breaking of peridynamic bonds, which is a history-dependent phenomenon. Additionally, *Peridigm* also contains a peridynamic plasticity model [3], which uses history-dependent state.

To address the *ModelEvaluator* assumption of statelessness, we utilized an *observer* design pattern.

Definition 2.1. The observer pattern is a software design pattern in which an object (called the subject)

maintains a list of its dependents (called observers) and notifies them of any state changes, usually by calling one of their methods.

Specifically, we implemented our material models as a *ModelEvaluator*-derived class, storing the model state within the class. This is ostensibly a violation of the *ModelEvaluator* contract, but does not manifest for explicit or implicit time integration. An observer was called from the time integrator to inform the material model that a time step had been completed, and that the material model should update its internal state.

While this approach is effective for time integration, it's not clear that this remedy would work for more complicated scenarios, such as adaptive time stepping or optimization.

Load Rebalance and Mesh Adaptivity Peridynamics is commonly used for modeling fragmentation problems, where material initially contained in a relatively small spatial region becomes spread over a much larger spatial region by the end of the simulation. In order to preserve parallel load balance, it is periodically necessary to redistribute data across processors. In the *Trilinos* framework, data assignment to processors is encapsulated within a *map* class, such as `Epetra::Map` or `Epetra::BlockMap`. To perform a rebalance, *Peridigm* asks the Zoltan package to determine a near-optimal data distribution across processors in the current configuration, which can be used to determine a new map. Given the current map (containing the current data distribution across processors) and the new map (containing the desired data distribution across processors), an import/export operation can be called to migrate data between processors.

Mesh adaptivity, or adaptive mesh refinement, also involves redistributing data. During the course of a computation one may desire to adaptively refine a mesh, for example, to reduce error in localized regions of the computational domain. This dynamically introduces more degrees of freedom into the problem, necessitating that one migrate data from an old map (the coarse mesh) to a new map (the fine mesh).

Load rebalance and mesh adaptivity have proven problematic with the *ModelEvaluator* interface. *ModelEvaluator*-derived classes have the property that they produce their own input parameter and output response data structures, as can be seen in this code segment taken from *SimpleODE*:

```
SimpleODE::SolverFactory slvrfctry(xml_file_name, appComm);
Teuchos::RCP<EpetraExt::ModelEvaluator> App = slvrfctry.create();
EpetraExt::ModelEvaluator::InArgs params_in = App->createInArgs();
EpetraExt::ModelEvaluator::OutArgs responses_out = App->createOutArgs();
```

This behavior is beneficial in that it makes the model self-contained. However, each model makes its own maps and control its data distribution across processors, as seen here, where the ModelEvaluator-derived class "App" returns a map:

```
Teuchos::RCP<Epetra_Vector> g = Teuchos::rcp(new Epetra_Vector(*App->get_g_map(0)));
```

This property of *ModelEvaluator*-derived classes presented an issue when trying to perform a rebalance with *Peridigm*. The maps are a form of state data contained within the *ModelEvaluator*-derived classes, ostensibly in violation of the *ModelEvaluator* contract that they be stateless. Unlike the issue with history-dependant material models, discussion with other *Trilinos* developers did not reveal a path forward. One idea proposed was to generate a "meta-ModelEvaluator" that would transform a *ModelEvaluator*-derived class (using an old map) to a *ModelEvaluator*-derived class (using a new map). This idea was not pursued as it would require a substantial coding effort, in part because each *ModelEvaluator*-derived class would require an associated expert-developed meta-ModelEvaluator class to effect a rebalance. An alternative approach is to destroy the existing *ModelEvaluator* objects, creating new ones in their place. This process is equivalent to the process that would need to occur, for example, if a simulation running on N_1 processors was checkpointed and then restarted on N_2 processors.

Faced with this obstacle, to proceed *Peridigm* development, we instead developed a *ModelEvaluator*-like class (`Peridigm::ModelEvaluator`) that mimics the interface of the *ModelEvaluator* used in the Agile Components frameworks, but does not create maps, etc.

2.3.2 Data Distribution

Another issues encountered during *Peridigm* development was the requirement by the *ModelEvaluator* interface that `Epetra::Map` and `Epetra::Vector` data structures be used, rather than the more general `Epetra::BlockMap` and `Epetra::MultiVector` data structures. In the Epetra inheritance diagrams, an `Epetra::Vector` inherits from an `Epetra::MultiVector`, and an `Epetra::Map` inherits from an `Epetra::BlockMap`, meaning that the block map and multivector data structures are more general.

In peridynamics, we frequently have multiple degrees of freedom per node (for example, all 3D vector fields have three degrees of freedom per node). The `Epetra::BlockMap` and `Epetra::MultiVector` data structures were designed for this use case, and are more natural to use within material model evaluation routines. To utilize the *ModelEvaluator* interface, it was necessary to use the `Epetra::Vector` and `Epetra::Map` data structures, which required mapping from `Epetra::MultiVector`s to `Epetra::Vector`s and back every timestep. In addition to requiring extra computation, this also required us to keep twice as many map data structures as were really needed (one `Epetra::BlockMap` and one `Epetra::Map` for each field variable).

2.3.3 Time Integration

When initially developing *Peridigm*, we attempted to use Rythmos, the lone time integration package in *Trilinos*. Unfortunately, none of the explicit integrators in Rythmos are for second-order ODEs, and all of the explicit integrators in Rythmos are unconditionally unstable when applied to the peridynamic equation of motion when it is reformulated as a first-order ODE. This is not a shortcoming of the Agile Components approach, but simply an observation that the solver tools needed by *Peridigm* were not initially present in any *Trilinos* package.

Since that time, a second-order explicit integrator has been added to the PIRO package within *Trilinos*.

2.3.4 Control Flow

Another issue encountered during *Peridigm* development was managing the execution control flow. The standard Agile Components approach is to cast the problem to be solved as a single *ModelEvaluator* instance, and evaluate that model to return the solution. This is depicted in Figure 6.

However, standard expectations from developers of computational solid mechanics codes are that one has the ability to intervene and modify simulation data at arbitrary points during the simulation. As a practical example, it is standard practice to execute contact resolution algorithms between atomic steps of an explicit time integration algorithm to ensure that solids do not interpenetrate each other.

This issue lead to a reformulation of the control flow within *Peridigm*. Originally mimicking the control flow of *SimpleODE* in Figure 6, the control flow of *Peridigm* now has only one instance of a *ModelEvaluator* object (that encapsulates the material model), and a `main()` routine that allows the simulation data to potentially be modified between atomic steps of time integration routines. The rationale for this decision is that it is not the providence of the material model nor the solver to perform tasks such as contact resolution or rebalance. Such tasks naturally fall to a `main()` routine, at the the highest level of control in the program.

3 Other Remarks

The Phalanx package for evaluating field variables was among the more recently developed *Trilinos* components utilized in `Peridigm`. We found Phalanx relatively easily did what we needed. We expect this is the case because Phalanx was ripped from a production code, rather than being built from the ground up to be placed into production codes. Although perhaps initially counterintuitive, this "backwards" development model would appear to be highly effective at delivering mature capabilities.

4 Recommendations

Although some development issues were encountered deploying the Agile Components methodology in *Peridigm*, we are supportive of the goals of Agile Components, and view this report as an opportunity to further broaden the scope of applications suitable to use in the Agile Components framework. For models satisfying the assumptions of the *ModelEvaluator* interface, the Agile Components methodology has proven to facilitate rapid development of powerful application code. For applications outside these target use cases, the potential rapid development capabilities that Agile Components may provide are simply too tantalizing to overlook. To that end, we make the following recommendations which we feel will improve the impact of Agile Components efforts, and broaden its scope to new applications.

Documentation Attempting to learn the Agile Components methodology by reading *Albany* code is somewhat like drinking from a fire hydrant. Although Agile Components developers are always happy to explain any concept, this approach is not scalable if we want the Agile Components methodology to be widely adopted beyond the current expert users. We recommend improved documentation along with an easy-to-understand demonstration code, such as *SimpleODE*. This will greatly shorten the learning curve for new users, and help them to understand how their application fits within the Agile Components methodology. *SimpleODE* may be useful as a starting template for new users to develop their own application based upon Agile Components.

Graceful Handling of Model State We recommend that all service modules, such as *ModelEvaluator*-derived classes, be made referentially transparent (*c.f.* Defn. 1.1). This would eliminate any and all issues with regard to rebalance and mesh adaptivity, the biggest obstacles encountered. The consequence of this is that *ModelEvaluator*-derived classes must no longer make their own maps, data, etc., but that these must be passed in externally. Referring to the control flow diagram of Figure 6, we see that it is not the role of the mathematical model **f** nor of the solver to create these data structures, meaning that they must be created at the level of `main()` (by the application developer) and passed in. This requires a mechanism for the developer to gracefully and naturally hand in model state as input, have the *ModelEvaluator* class potentially transform that state, and then hand that state back out. The *ModelEvaluator* becomes less self-contained, but at the benefit of increased usability and flexibility. This recommendation comes at the cost of a substantial refactor of all agile component based code, and may not be feasible.

Data Distribution If possible, we recommend moving the *ModelEvaluator* interface and *ModelEvaluator*-derived solvers to support BlockMap and MultiVector data structures. A Map is a special case of a BlockMap, and a Vector is special case of a MultiVector, so this would not impact any current use cases.

Control Flow The control flow of Figure 6 could be preserved so long as the application developer has the opportunity and freedom to intervene during solver execution and modify simulation data in essentially arbitrary ways. One way to achieve this is by inserting callbacks between atomic steps in all *ModelEvaluator*-derived solvers. The application developer would then have complete freedom to specify how simulation data should be transformed between atomic solver steps, without the solver itself ever having to know anything about contact resolution, rebalance, etc.[1] For a specific example, see Algorithm 1 with callbacks put in between steps of the velocity Verlet explicit time integrator.[2]

References

[1] R. Helm, E. Gamma, J. Vlissides, and R. Johnson, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[1] Such use of callbacks is common practice within Sandia's LAMMPS molecular dynamics code [6]. The code makes callbacks to user-defined functions between atomic steps in time integration or minimization processes, and has been used to great effect.

[2] This is exactly the callback structure used in LAMMPS.

Algorithm 1 Velocity Verlet with Callbacks

1: initial_integrate()
2: $\mathbf{v}_i^{n+1/2} = \mathbf{v}_i^n + \frac{\Delta t}{2\rho_i}\mathbf{f}_i^n$
3: $\mathbf{y}_i^{n+1} = \mathbf{y}_i^n + \Delta t \mathbf{v}_i^{n+1/2}$
4: post_force()
5: $\mathbf{v}_i^{n+1} = \mathbf{v}_i^{n+1/2} + \frac{\Delta t}{2\rho_i}\mathbf{f}_i^{n+1}$
6: final_integrate()

[2] M. Heroux, R. Bartlett, V. H. R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams, *An Overview of Trilinos*, Tech. Report SAND2003-2927, Sandia National Laboratories, 2003.

[3] J. A. Mitchell, *A nonlocal, ordinary, state-based plasticity model for peridynamics*, Tech. Report SAND2011-3166, Sandia National Laboratories, May 2011.

[4] R. Muller, Gao, Nielsen, A. Salinger, and Young, *Qcad web page*. `https://development.sandia.gov/Albany/qcad.html`.

[5] J. Ostien, J. Fould, A. Mota, and A. Salinger, *Lcm web page*. `https://development.sandia.gov/Albany/lcm.html`.

[6] S. Plimpton, *Fast parallel algorithms for short-range molecular dynamics*, J. Comp. Phys., 117 (1995), pp. 1–19.

[7] A. Salinger, E. Phipps, and J. Ostien, *Albany web page*. `https://development.sandia.gov/Albany/`.

[8] S. A. Silling, M. Epton, O. Weckner, J. Xu, and E. Askari, *Peridynamic states and constitutive modeling*, J. Elasticity, 88 (2007), pp. 151–184.