International Journal for Multiscale Computational Engineering, 14 (4): 415–438 (2016)

ALBANY: USING COMPONENT-BASED DESIGN TO DEVELOP A FLEXIBLE, GENERIC MULTIPHYSICS ANALYSIS CODE

Andrew G. Salinger,[1,*] Roscoe A. Bartlett,[1] Andrew M. Bradley,[1] Qiushi Chen,[1,2] Irina P. Demeshko,[1] Xujiao Gao,[1] Glen A. Hansen,[1] Alejandro Mota,[1] Richard P. Muller,[1] Erik Nielsen,[1] Jakob T. Ostien,[1] Roger P. Pawlowski,[1] Mauro Perego,[1] Eric T. Phipps,[1] WaiChing Sun,[1,3] & Irina K. Tezaur[1]

[1]*Sandia National Laboratories, P.O. Box 5800, Albuquerque, New Mexico 87185-1318, USA*

[2]*Department of Civil Engineering, Clemson University*

[3]*Columbia University, Department of Civil Engineering and Engineering Mechanics*

[*]Address all correspondence to: Andrew G. Salinger, E-mail: agsalin@sandia.gov

Albany is a multiphysics code constructed by assembling a set of reusable, general components. It is an implicit, unstructured grid finite element code that hosts a set of advanced features that are readily combined within a single analysis run. Albany uses template-based generic programming methods to provide extensibility and flexibility; it employs a generic residual evaluation interface to support the easy addition and modification of physics. This interface is coupled to powerful automatic differentiation utilities that are used to implement efficient nonlinear solvers and preconditioners, and also to enable sensitivity analysis and embedded uncertainty quantification capabilities as part of the forward solve. The flexible application programming interfaces in Albany couple to two different adaptive mesh libraries; it internally employs generic integration machinery that supports tetrahedral, hexahedral, and hybrid meshes of user specified order. We present the overall design of Albany, and focus on the specifics of the integration of many of its advanced features. As Albany and the components that form it are openly available on the internet, it is our goal that the reader might find some of the design concepts useful in their own work. Albany results in a code that enables the rapid development of parallel, numerically efficient multiphysics software tools. In discussing the features and details of the integration of many of the components involved, we show the reader the wide variety of solution components that are available and what is possible when they are combined within a simulation capability.

KEY WORDS: *partial differential equations, finite element analysis, template-based generic programming*

1. INTRODUCTION

In this paper we present the Albany multiphysics code; a parallel, unstructured-grid, implicit, finite element application aimed at the solution of a wide variety of physics problems. Albany was developed to demonstrate the potential of a component-based approach to scientific application development. This approach, which we will discuss in detail in this paper, is to make broad use of template objects, computational science libraries, abstract interfaces, and software engineering tools. By making extensive use of flexible external capabilities, the application code development effort can remain focused on supporting physics models and analysis features but yet have full access to advanced supporting algorithms, each written by domain experts, where the costs of verification and maturation are amortized over many projects. While every application uses external libraries in its development, Albany takes module reuse to an extreme. We view the Albany code base as glue code that integrates dozens of external components and capabilities.

1543–1649/16/$35.00

Albany is an open-source development project; the code is available on GitHub (http://gahansen.github.io/Albany). All of the capabilities described here are supported using publicly available components, many of which come from the Trilinos (Heroux et al., 2005) multiphysics code development framework (http://trilinos.org). Some mesh capabilites are delivered by the Parallel Unstructured Mesh Infrastructure (PUMI) (Seol et al., 2012) code under development by the Scientific Computation Research Center (SCOREC) at Rensselaer Polytechnic Institute (RPI) and includes sophisticated parallel adaptation and load balancing capabilities. Albany has a generic discretization interface; it simultaneously supports PUMI and the Sierra ToolKit (STK) infrastructure (Edwards et al., 2010) that is the basis of the *Sierra Mechanics* analysis code suite (Stewart and Edwards, 2003) developed by Sandia. STK is a package within Trilinos.

By using the component-based strategy, Albany demonstrates the breadth of utility that can be assembled into a single application as well as the ease of integration of new models and capabilities into a code that follows this design approach. Albany supports a generic physics interface that is based on the evaluation of a residual function. While the abstraction of the residual function is a common design pattern in implicit codes that employ Jacobian-free solution methods, Albany's evaluation state is templated using a generic `ScalarT` type. This design feature directly supports the use of automatic differentiation [supplied by the Trilinos *Sacado* package (Phipps, 2015a)] to efficiently form an "analytic" Jacobian for Newton's method, in addition to providing capabilities for forming the Jacobian numerically using finite-differencing and the use of Jacobian-free approaches. The use of automatic differentiation can be significantly more efficient than a matrix-free method for some problems (Chen et al., 2014b). Furthermore, the combination of automatic differentiation with templated physics expressions directly supports the formation of all manner of derivative information, which leads to powerful sensitivity and other analysis capabilities [e.g., Chen et al. (2014a)], as well as *embedded uncertainty quantification* (UQ) that will be described later in the paper. The availability of mathematically correct sensitivities of physical quantities can substantially expedite the development of scientific software over the expense of deriving and coding them by hand. Finally, should a user prefer an alternative derivative form, such as an analytic Jacobian derived using a symbolic process, one can readily employ it as a template specialization to this generic machinery.

Albany uses the Trilinos *Shards* and *Intrepid* packages to represent the element topology of the mesh, to locate integration points, and subsequently integrate over the elements. The Trilinos *Phalanx* package is a local field evaluation kernel that serves as the basis of abstraction for physics evaluation. It uses template-based generic programming (TBGP) methods (Pawlowski et al., 2012a,b) to support arbitrary data and evaluation types, and manages the dependencies between field producers and consumers to allow expressing complex multiphysics coupling relationships in terms of a directed acyclic graph of simple operations. The physics *evaluator* functions typically operate on a *workset* of elements, and store integrated/evaluated quantities in local field arrays. Ultimately, when the operations are performed over all worksets, an assembled finite element discretization of the governing equations results.

Albany uses various Trilinos components to orchestrate the solution process. At a high level, the Library of Continuation Algorithms (*LOCA*) can be used to provide for continuation, displacement, and load stepping, bifurcation analysis, etc. The *Rythmos* package is used for time integration. Inside these enclosing stepping algorithms, most problems use some form of a Newton nonlinear solver, supplied by the Trilinos *NOX* package. The solution of the resulting linear system is managed by the *Stratimikos* package, which allows the user to select from a myriad of linear solvers and preconditioners available in Trilinos from the Albany input file. In addition to supporting embedded sensitivity analysis and UQ, Albany also couples to Dakota (Adams et al., 2009) using the Trilinos *TriKota* components to provide both coupled and black-box analysis and UQ capabilities using that package.

It is not our intent to directly compare the approach taken with Albany with other development strategies and specific examples of other analysis tools. Such a comparison would require a significant investment in time and resources if it were to be thorough enough to be of any value to the reader. Further, such comparisons are sufficiently subjective that the reader may not want to depend on the authors of the package to present such data and opinion. A natural first approach to design abstraction for implicit and semi-implicit methods is to separate the finite element assembly process from the linear and nonlinear solution algorithms. In this case, the assembly tools (e.g., basis function library and meshing library) are developed internally to the application code and the solvers are leveraged from external libraries such as PETSc (Balay et al., 2013) and Trilinos (Heroux et al., 2005). Examples of codes that fall into this category are the Differential Equations Analysis Library (`deal.II`) (Bangerth et al., 2007), libMesh (Kirk

et al., 2006), Life V (Prud'homme, 2007), Sierra (Stewart and Edwards, 2003), and Uintah (de St. Germain et al., 2000). Some projects additionally separate the assembly process into independently releasable components or rely on external components. Projects in this class include Albany, the FEniCS project (Logg et al., 2012), the MOOSE project (Gaston et al., 2009), and the Sundance rapid development system (Long et al., 2010). The FOOF framework (Yuan and Fish, 2015) provides a Fortran-based alternative to the above C++ codes, with a focus on applications for multiphysics systems [see, e.g., Michopoulos et al. (2005)], and further includes multiscale methods (Yu and Fish, 2002) to capture microscale heterogeneity.

In this paper, we discuss our experiences in developing a scientific application code using a collection of independently developed components. In Section 2 we provide details on how we define the component-based approach, what the scope of the current effort is, and what some of the advantages and disadvantages of this approach are.

In Section 3 we present details of the design of the Albany code. In particular, we describe the abstract interfaces between the major design elements of the code and how they were chosen to maintain modularity. Note that some of the functions live in Albany and some are contained in Trilinos. While there is not a unique or optimal design for how to combine different modules possessing abstract interfaces, the current architecture has not needed substantial refactoring as the number and variety of application areas supported by Albany has grown since inception.

In Section 4 we highlight two projects being hosted by Albany. The first is a computational mechanics research and development platform, supporting research in constitutive models, solution and preconditioning methods, discretizations, full coupling of mechanics to scalar equations, material models, and failure and fracture modeling. The second is a quantum device design and analysis capability, where nonlinear Poisson and coupled Schrodinger–Poisson systems are used for designing quantum dots, the building blocks of quantum computers. The success of these projects in quickly fielding new application codes with rich feature set and analysis capabilities indicates the strength of the component-based approach that is the basis for the Albany code.

2. THE COMPONENT-BASED APPLICATION DEVELOPMENT STRATEGY

The Albany code was written to demonstrate a component-based strategy for application code development. Component-based development is the process by which an application code is built primarily from modular pieces, such as independently developed software libraries, abstract class hierarchies, and template-based generic classes. The approach involves assembling components from four classes of software: libraries, interfaces, software quality tools, and demonstration applications, which form the foundation for the new code. There are several benefits to this approach, discussed in detail below.

The use of libraries and frameworks is common in scientific application development. Almost all scientific codes call specialized libraries, such as the BLAS, LAPACK, mesh databases, etc., to a degree. Some applications use libraries more strongly, calling linear and nonlinear solvers, preconditioners, etc., from widely available numerical analysis toolkits. The pattern for the use of libraries typically involves the use of a function call, passing basic or fundamental data types (integers, double precision values, or arrays of integer or double precision numbers) to interface with an archive or object file. These functions are called from the main code to perform operations on the function arguments. In component-based development, the application typically consists of "glue code" that connects together interfaces or instances of the components. In our case, this glue code accounts for less than 10% of the overall code that makes up the application. Further, the components are generic in nature, designed to support classes of applications to enable reuse for multiple projects. These components are often constructed using inheritance hierarchies or using template-based generic programming to better support such reuse. Finally, the interfaces between components less often use a functional form; they typically employ more advanced patterns (Gamma et al., 1995) such as factories, adapters, decorators, and/or observer patterns. Where library-based development typically passes lower-level information, such as arrays, lists, and simple structures of data to and from the library code, components will employ more advanced communication using generic abstractions like observers, factories, and evaluators.

A framework design is another form of application design, where a "driver" framework is used to connect with user application modules through published interfaces. In this case, reuse is achieved by employing the driver framework for several different applications, changing only the user code modules that provide the "personality" of the resulting application.

2.1 Computational Science Component Libraries

Generic components are generally grouped to form "component libraries," or just "libraries" for brevity. We recognize that overloading of the word "library" might cause confusion, we will explicitly state when we are referring to the conventional use of the term.

Figure 1 enumerates individual computational capabilities that can be deployed as independent libraries, and made available as building blocks for new application codes. The capabilities are grouped in a logical manner, which not only serves to organize the presentation but, as will be discussed later, also shows where opportunities exist for the definition of abstraction layers around clusters of related libraries. Many of the listed components shown in the figure represent a set of capabilities, in that there are multiple independent libraries that provide competing or complimentary capabilities in the listed capability area (e.g., preconditioners).

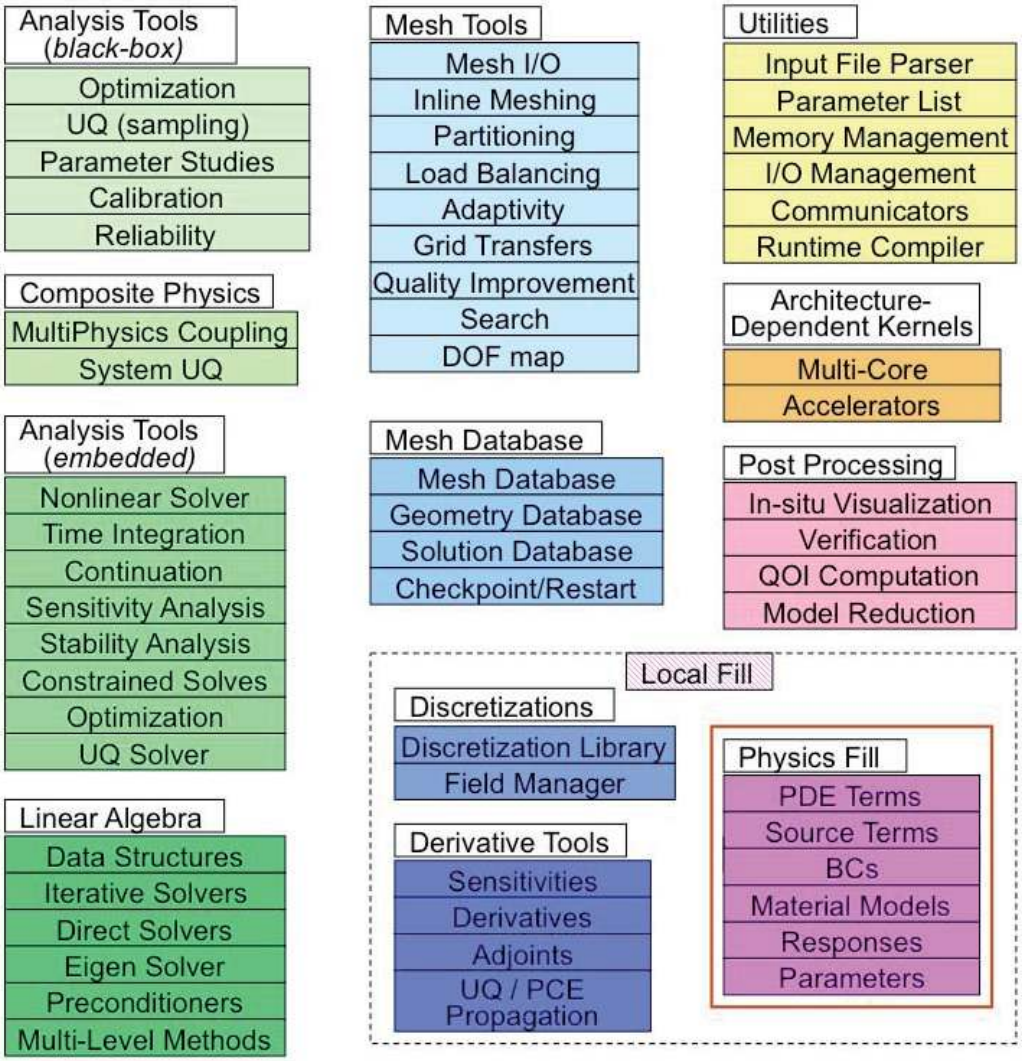

FIG. 1: Computational science capabilities that can be delivered using component libraries. Given the availability of these tools, the time required to construct an analysis code is dramatically reduced, with development time being concentrated in the outlined 'Physics Fill' box.

The granularity of the definition of an independent library is a software design decision, where the extremes (having all capabilities contained within one monolithic framework or having every C++ class an independent library) are obviously sub-optimal. For this description, which has a direct correlation to the development of independent packages in the Trilinos suite, a library is typically constructed by one to three domain experts. This level of effort is small enough that the lead developers can know and understand the entire code base of each package. In this definition, an existing library may be separated into a set of smaller independent pieces if the common usage involves only a subset of the capabilities contained in the original library. For instance, a library that contains both a GMRES linear solver and an ILU preconditioner would be split into two separate libraries since we would want to enable the use of the ILU preconditioner with any iterative linear solver.

2.2 Software Quality Tools

Software quality tools and processes are central to the productivity of code teams. The benefits of these tools increase significantly as project teams grow in the number of developers and become geographically distributed. In Fig. 2, we present a list of software quality tools which enhance productivity of a code project like Albany. With software quality toolsets, it is not necessary to use all the capabilities to realize benefits.

For computational science organizations, there is a significant benefit in sharing the same sets of tools and processes across many software projects. With a decrease in each project-specific learning curve, staff members have more agility to make an impact on multiple projects. For this reason, the Albany project has largely adopted the set of tools used by the Trilinos project.

2.3 Advantages and Disadvantages of Component-Based Code Design

With our experience in Albany and other application codes that use Trilinos libraries, we have noted significant advantages and some disadvantages in using the component-based approach to code design. These span both technical and social issues, and are influenced by the organizational culture, funding, and several other factors. A more extensive discussion of component-based design is presented in a technical report (Salinger, 2012).

Advantages of a component-based approach to application development include:

(i) The costs of writing, verifying, maturing, extending, and maintaining a library is amortized over multiple projects.

(ii) Shared support of an algorithmic capability over several projects allows the pooling of a critical mass of funding for a subject matter expert to develop significant expertise and software capabilities in a targeted area.

(iii) By using the algorithmic library, the application code development team can gain ongoing access to the expertise of the library developer.

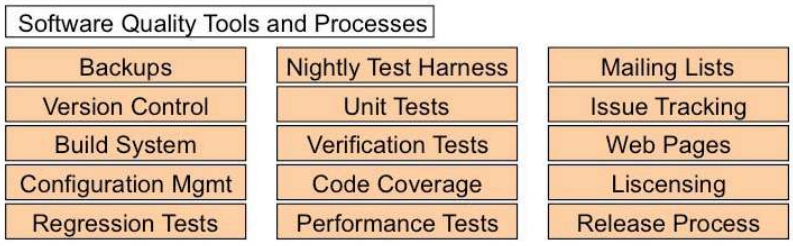

FIG. 2: In addition to computational science libraries, the rapid development of new application codes is also dependent on the availability and use of an effective set of software quality tools and processes. These support developer productivity, software quality, and the longevity of a project.

(iv) The use of general-purpose libraries developed externally to an application code forces the code to adopt a more modular design. This can result in more flexiblity and extensibility of the application in the long run.

(v) The use of libraries decreases the code base that must be maintained by the application team. The finer granularity of this approach creates natural divisions between code appropriate for open source release and code that must be protected (e.g., for intellectual property or export control reasons), decreasing the amount of code that requires protection.

i. The use of abstract interfaces around groups of related capabilities facilitates the implementation and investigation of alternative algorithms. Using an example from Trilinos, several direct and iterative solvers share the same interface and can be selected in Albany at run time from the input file.

ii. The effort to create abstract interfaces that support multiple concrete implementations improves the extensibility and flexibility of the code. Creating an abstract layer between the mesh database and the mesh data structures used in the PDE assembly enables us to flexibly use multiple mesh databases with minimal impact on the code.

In contrast with a monolithic application code that contains all required algorithms as part of the application, disadvantages of a component-based approach include:

(i) The use of numerous Third-Party Libraries can complicate the build process. It can be particularly difficult to keep track of what versions of libraries are compatible with each other.

We mitigate this complexity in three ways. First, we heavily use component libraries from Trilinos, which synchronizes the release of its numerous (>50) libraries. Albany also links to a collection of parallel unstructured mesh and adaptation component libraries contained within the Rensselaer Polytechnic Institute (RPI) Scientific Computation Research Center (SCOREC) toolset (Seol et al., 2012). Second, we employ a "CMake Superbuild" (i.e., the TriBITS build system of Bartlett et al. (2012)), that manages dependencies between the component libraries and the overall build complexity. Third, we employ (near) "continuous integration" (Booch, 1991) where the builds and tests of all the tightly coupled component libraries are performed frequently [at least nightly; see Brown and Canino-Koning (2015) and the Albany CDash site (Hansen et al., 2015)].

(ii) When debugging the application, developers on the application code development team may have difficulty tracking down issues in unfamiliar components and may not have ready access to the component developer.

(iii) Abstract interfaces that compartmentalize the code are difficult to design and require a different skill set to construct. Such skills may not be present on the application development team.

(iv) General purpose libraries with an improper interface design can lead to applications that do not perform optimally (e.g., performing unnecessary data copies) and have unnecessarily high memory requirements.

(v) The dependence on external components can significantly impact the deployment of an application to novel or advanced platforms. For example, the porting of a code from traditional CPU cores to general purpose graphics processing units (GPGPUs) requires that many components be rewritten to support that architecture. Even if some components support the architecture, it may not be possible to run the application on the new hardware until all, or a large subset of components provide that support.

The application development projects described in Section 4, together with the regression test suite and other development efforts using Albany, provide anecdotal experience that the component-based approach has net benefits. In particular, the ability to rapidly add new algorithms and capabilities by making use of pre-existing component libraries has been apparent in a wide variety of projects. The other strength of this approach is that it demonstrates an accelerated development process that builds on experience; the more that libraries are developed and matured, the more rapidly the next application using those libraries can be constructed and verified. This can yield significant strategic return on investment across a computational science organization but it may be of less immediate value within an individual code project.

3. ALBANY COMPONENT-BASED CODE DESIGN

The Albany code was developed to refine, demonstrate, and evaluate the component-based code design strategy. This section summarizes the design of the Albany driver application and some of the components that were used in its construction. We will also state remaining gaps in the component-based development strategy. Along the way, simple heat transfer and incompressible flow proxy applications were supplanted by independently funded application projects as the development drivers (see Section 4). We also discuss interface design aspects of Albany, detailing where we have placed abstract interfaces to gain access to general-purpose libraries and to maintain the flexibility and extensibility of a modular design.

Albany is designed to compute approximate solutions to coupled problems represented abstractly as

$$\mathcal{L}(\dot{u}(x,t), u(x,t)) = 0, \quad x \in \Omega, \quad t \in [0,T], \quad \dot{u}, u \in H, \tag{1}$$

where $\Omega \subset \mathbb{R}^d$ ($d = 1, 2, 3$) and $[0, T]$ are the spatial and temporal domains, $\mathcal{L}$ is a (possibly nonlinear) differential operator, H is a Hilbert space of functions upon which $\mathcal{L}$ is defined, u is the (unknown) PDE solution, and $\dot{u}$ its corresponding time derivative. Equation (1) is then discretized in space via the (generally unstructured grid) finite element method resulting in the finite-dimensional differential-algebraic (DAE) system

$$\mathbf{f}(\dot{\mathbf{u}}(t), \mathbf{u}(t), \mathbf{p}) = 0, \tag{2}$$

where $\mathbf{u} \in \mathbb{R}^n$ is the unknown solution vector, $\dot{\mathbf{u}} \in \mathbb{R}^n$ is its time derivative, $\mathbf{p} \in \mathbb{R}^m$ is a set of model parameters, and $\mathbf{f} : \mathbb{R}^{2n+m} \to \mathbb{R}^n$ is the DAE residual function. In Albany, we have focused on fully implicit solution algorithms which require evaluating and solving linear systems involving the Jacobian matrix

$$\alpha \frac{\partial \mathbf{f}}{\partial \dot{\mathbf{u}}} + \beta \frac{\partial \mathbf{f}}{\partial \mathbf{u}}, \tag{3}$$

and thus accurate and efficient evaluation of these derivatives is critical.

In addition to computing the approximate solution $\mathbf{u}(t)$ one is also often interested in evaluating functionals of the solution

$$\mathbf{s}(t) = \mathbf{g}(\mathbf{u}(t), \mathbf{p}), \tag{4}$$

which we call responses. Values of response functions at discrete time points are often targets of sensitivity and uncertainty analysis, as well as having utility as objective functions in optimization, design, and calibration problems. Many of these methods entail evaluation of derivatives of the responses $\mathbf{s}$ with respect to the model parameters $\mathbf{p}$, and often the performance of these methods is greatly improved when these derivatives are evaluated accurately. For steady-state problems, the response gradient can be computed via the formula

$$\frac{d\mathbf{s}}{d\mathbf{p}} = \frac{\partial \mathbf{g}}{\partial \mathbf{u}}(\mathbf{u}^*, \mathbf{p}) \frac{d\mathbf{u}^*}{d\mathbf{p}} + \frac{\partial \mathbf{g}}{\partial \mathbf{p}}(\mathbf{u}^*, \mathbf{p}) = -\frac{\partial \mathbf{g}}{\partial \mathbf{u}}(\mathbf{u}^*, \mathbf{p}) \left(\left(\frac{\partial \mathbf{f}}{\partial \mathbf{u}}(\mathbf{u}^*, \mathbf{p}) \right)^{-1} \frac{\partial \mathbf{f}}{\partial \mathbf{p}}(\mathbf{u}^*, \mathbf{p}) \right) + \frac{\partial \mathbf{g}}{\partial \mathbf{p}}(\mathbf{u}^*, \mathbf{p}), \tag{5}$$

where $\mathbf{u}^*$ satisfies $\mathbf{f}(\mathbf{u}^*, \mathbf{p}) = 0$. The necessity to quickly and accurately evaluate derivatives such as $\partial \mathbf{f}/\partial \mathbf{u}$ and $\partial \mathbf{f}/\partial \mathbf{p}$ (as well as other quantities such as polynomial chaos coefficients) needed by analysis algorithms, as well as to support an extensible interface for supplying these quantities to higher-level analysis algorithms, has dictated many of the code design decisions described below.

3.1 Overall Albany Code Design

At a high level, the code is separated into five main algorithmic domains separated by abstract interfaces, as shown in Fig. 3. These domains will each be discussed in detail in the following sections.

A key part of the Albany code is depicted as "Glue Code" in this figure, which is the driver code that integrates the components to provide the overall, physics-independent code capability. It depends on a discretization abstraction, which serves as a general interface to a mesh database and mesh services. As described below in Section 3.2, this

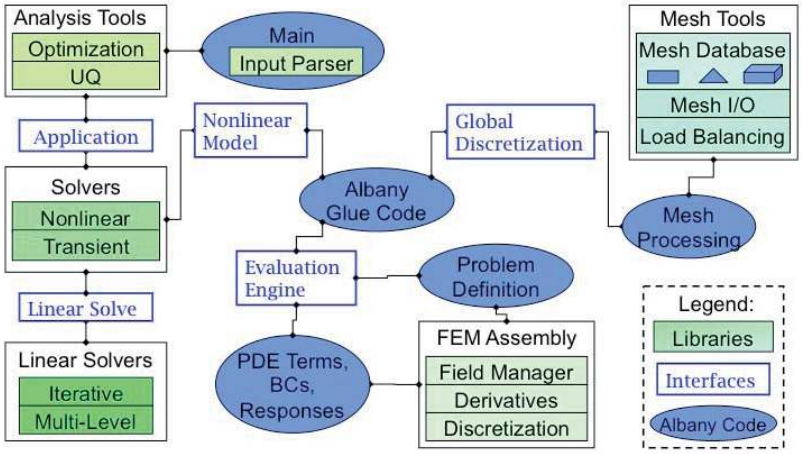

FIG. 3: Albany code is built largely from component libraries (colored boxes with black font) and abstract interfaces (clear boxes with blue font), and employs software quality tools (not shown). The bulk of the capabilities come from Trilinos libraries exposing abstract interfaces. The bulk of the coding effort for a new application involves writing PDE terms, boundary conditions, and responses.

interface deals with linear algebra objects and standard vectors, and is agnostic to the specific mesh database. The Glue Code also employs a problem class abstraction to construct the set of PDEs, boundary conditions, and response calculations. As described in Section 3.3, the assembly of these physics pieces comes down to the evaluation of a directed graph of computations of field data. The Glue Code then uses these pieces to satisfy the nonlinear model abstraction (e.g., computing a residual vector or Jacobian matrix).

With the nonlinear model interface satisfied, the full range of Trilinos solvers are available. This includes the embedded nonlinear analysis solvers such as nonlinear and transient methods described in Section 3.4. These embedded nonlinear solvers, in turn, call the linear solvers (see Section 3.5), which are the most feature-rich and mature set of general purpose libraries used in the code. Albany was designed to demonstrate how to design a code possessing analysis capabilities significantly beyond repeated forward simulation, so the nonlinear solver layer is not the top of the code hierarchy. A separate Analysis layer described in Section 3.6 wraps the solver layer, and performs parameter studies, optimization, and UQ; primarily using algorithms from the Dakota toolkit (Adams et al., 2009).

As presented in Fig. 2, there are many software tools and processes that can improve the productivity of a project. Albany has adopted the toolset from Trilinos to minimize the learning curve that Trilinos developers need to begin contributing to Albany. These include *git* for version control, *CMake* for configuration management, build, and porting, *CTest* for regression testing, and Doxygen for automatic documentation based on the class design and comments. We have adopted the mailing lists and webpage design from Trilinos as well. We currently have scripts run under a *cron* job that perform continuous integration with Trilinos and Dakota that do a fresh build and regression testing nightly on multiple machines with the results being placed on a CDash dashboard (Hansen et al., 2015).

3.2 Global Discretization Abstraction and Libraries

A critical component of any finite element code is the mesh framework, which defines the geometry, element topologies, connectivities, and boundary information, as well as the field information that lives on the mesh. As with many modern codes, Albany supports spatial adaptation, where the mesh may change by refining in certain areas, and perhaps coarsening in others, driven by evolving features and error indicators computed during the solution. A further complication involves the need to rebalance the workload between processors as the mesh is modified. Albany accesses the mesh database, adaptation and load balancing capabilities, together with functions used to transfer the solution information between mesh representations, using an abstract Global Discretization interface.

The global discretization abstraction, presented schematically in Fig. 4, gives the finite element assembly process access to all of the data distribution information required by the linear algebra objects. In all cases, mesh information is contained in an in-memory mesh database that is accessed through a specialization of the abstract Global Discretization interface class. These specializations, unique to each mesh library Albany supports, provides a set of common services. These include reading and writing mesh data files present on the file system through I/O routines, providing element topology and vertex coordinate information, and optionally mesh adaptation, load balancing, and solution transfer capabilities. Of note is that each mesh library is different internally and provides services in a unique way. The specialization of the abstract global discretization class may interpret or "fill in" missing or incompatible data representations when required.

We note that the placement of the abstract Global Discretization interface is above the location where a general interface to mesh databases would lie in a domain design. The design of an abstract interface to mesh databases (e.g., ITAPS, Diachin et al. (2007)) remains a challenge, given the many competing and often contradictory demands of codes that use explicit or implicit algorithms, static and adaptive meshes, and C++ vs. C or FORTRAN. The interface has methods for the quantities needed directly in the finite element assembly, such as the Jacobian graph and coordinate information, in the data structures desired by the assembly. The offset between the mesh database and the Global Discretization interface is denoted as the Mesh Processing layer in Fig. 4. Functions in this layer satisfy the interface using calls and data structures specific to the underlying mesh database. We found the use of our Global Discretization abstraction in place of a generic mesh interface to be a tractable solution for our needs, but the scheme would only scale to a modest number of mesh databases with distinct interfaces.

Albany currently supports two independent implementations of the discretization interface; (1) the *Sierra ToolKit* (STK) package (Edwards et al., 2010) in Trilinos, and (2) the *Parallel Unstructured Mesh Interface* (PUMI) (Seol et al., 2012) being developed by the Scientific Computation Research Center (SCOREC) at Rensselaer Polytechnic Institute. The STK mesh database can be loaded in several ways: reading of a mesh file in the Exodus format (typically generated with the CUBIT meshing program), inline meshing (where the mesh is generated directly in source code) with the Pamgen package in Trilinos, and simple rectangular meshes directly created in the code base.

In a typical simulation, the interaction with the mesh library begins by Albany instantiating an object of the desired specialized class (which activates constructors in the appropriate places in the underlying mesh library), based on the type of input mesh and geometry files specified by the user. At construction, the mesh library reads the mesh information in serial or parallel depending on the simulation, and performs the degree of processing required to service requests from Albany for discretization data. As the simulation initializes, the Albany Glue Code invokes virtual member functions in the abstract discretization object to access coordinate data, connectivity, and to read (when restarting) and write solution data to the specialized class (and underlying library).

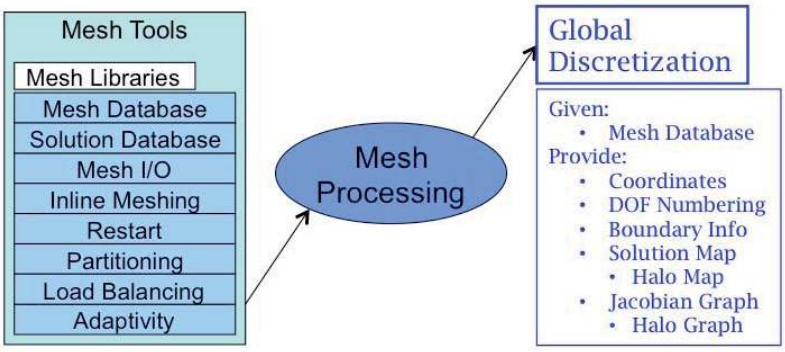

FIG. 4: Finite element mesh and related quantities are exposed to the Albany code through the abstract Global Discretization interface. Depending on the internal details of the mesh library in use, a specialization of the Global Discretization class will construct quantities in the layout needed by the rest of the code, such as coordinates, solution vectors, sparse-matrix graphs, and degree-of-freedom (DOF) numbering/connectivity information.

For adaptive simulations, there are two additional capability hierarchies that manage both the mesh adaptation process and the criteria used to determine the degree of adaptation needed, each Albany time, load, or displacement step. These interfaces are likewise abstract and specialized to suit the requirements of the mesh adaptation library specified for the simulation.

Other information that is processed on the mesh and accessed through the abstract discretization interface includes the multidimensional array that holds the list of elements on this processor, each with the array of local nodes, and pointers to the solution vector, coordinate vector, and any other field data stored at the nodes. By processing the element connectivity information, as well as some local discretization information (how many unknowns are on a mesh node), the sparse graph of the Jacobian matrix can be processed. For dealing with overlap (also known as halo or ghosted) information, several objects have both "owned" and "overlap" versions.

3.3 Problem Abstraction and Finite Element Assembly

Given a finite element mesh as supplied by the abstract discretization components, the purpose of the problem abstraction and finite element assembly components is to evaluate the discrete finite element residual, Eq. (2), for the problem at hand, as well as derived quantities such as Jacobian matrices and parameter derivatives needed for simulation and analysis. Our scalable approach for finite element assembly is described elsewhere (Pawlowski et al., 2012a,b). Here, we briefly summarize the salient features of the approach and its use within Albany.

Multiphysics simulation introduces a number of difficulties that must be addressed by the software framework including managing multiple physics models, adapting the simulation to different problem regimes, and ensuring consistency of the coupled residual evaluation with respect to the full system degrees-of-freedom. To manage this complexity, Albany employs the graph-based evaluation approach (Notz et al., 2012; Pawlowski et al., 2012a,b) as provided by the Trilinos Phalanx package (Pawlowski, 2015). Here, the residual evaluation for a given problem is decomposed into a set of terms (at a level of granularity chosen by the developer), each of which is encoded into a Phalanx `evaluator`. Each `evaluator` encodes the variables it depends upon (e.g., temperature evaluated quadrature points for a given set of basis functions), the variables it evaluates (e.g., a material property at those same quadrature points), and the code to actually compute the term. Phalanx then assembles all of the `evaluators` for a given problem into a directed acyclic graph representing the full residual evaluation for a given set of mesh cells stored in a data structure called the `field manager`. The roots of the graph are evaluator(s) that extract degree-of-freedom values from the global solution vector and the leaves are evaluator(s) that assemble residual values into the global residual vector. The full finite element assembly then consists of a loop over mesh cells with the body of the loop handled by the Phalanx evaluation [typically a preselected number (*work set*) of cells are processed by each evaluator, to improve performance by amortizing function call overhead over many mesh cells]. This approach improves code reuse by allowing common evaluators to be used by many problems, improves efficiency by ensuring each term is only evaluated as necessary, ensures correctness by requiring all evaluator dependencies are met, and allows a wide variety of multiphysics problems to be easily constructed. While not required, most terms within Albany employ the Intrepid package (Bochev et al., 2012) for local cell discretization services such as finite element basis functions and quadrature formulas. This graph-based evaluation approach is used by several frameworks for handling multiphysics complexity including the Aria application code in SIERRA (Stewart and Edwards, 2003), the Drekar code (Smith et al., 2011), the MOOSE framework (Gaston et al., 2009), and the Uintah framework (de St. Germain et al., 2000).

One of the design goals of Albany was to provide native support for a wide variety of embedded nonlinear analysis approaches such as derivative-based optimization and polynomial chaos-based uncertainty quantification. A significant challenge with these approaches is that they require calculation of a wide variety of mathematical objects such as Jacobians, Hessian-vector products, parameter derivatives, and polynomial chaos expansions, all of which require augmentation of the assembly process. This is a burden on the simulation code developer, which means these approaches are often not incorporated within the application. This not only limits the impact of these methods but also limits potential research in new analysis approaches for complex multiphysics applications. To address these issues, Albany leverages the template-based generic programming (TBGP) approach (Pawlowski et al., 2012a,b) to provide a framework for easily incorporating embedded analysis approaches. This technique employs C++ templates and operator overloading to automatically transform code for evaluating the residual into code for computing the quantities

described above, and is an extension of operator overloading-based automatic differentiation (AD) ideas to the general case of computing other nondifferential objects.

To leverage the TBGP approach, each evaluator in Albany is written as C++ template code, with a general `EvalT` template parameter. This parameter encodes the evaluation type, such as a residual, Jacobian, parameter derivative, or polynomial chaos expansion. Each evaluation type defines a scalar type, which is the data type used within the evaluation itself (e.g., `double` for the residual evaluation or an AD type for the Jacobian and parameter derivative). Each evaluator is then instantiated on all of the supported evaluation types relying on the Sacado (Phipps and Pawlowski, 2012; Phipps, 2015a) and Stokhos (Phipps, 2015b) libraries to provide overloaded operator implementations for all of the arithmetic operations required for each scalar type. This allows the vast majority of evaluators to be implemented in a manner agnostic to the scalar type and the corresponding mathematical object being computed. Furthermore, any evaluator can provide one or more template specializations for any evaluation type where custom evaluation is needed.

Albany leverages template specialization to implement the gather and scatter phases of the finite element assembly for each evaluation type (see Fig. 5). For example, the residual specialization of the gather operation extracts solution values out of the global solution vector and the scatter operation adds residual values into the global residual vector. Likewise, the Jacobian specialization of the gather phase both extracts the solution values and seeds the derivative components of the AD objects, while the scatter operation both extracts the dense element Jacobian matrix from the AD objects and sums their contributions into the global sparse Jacobian matrix. These gather/scatter evaluators are written once for each evaluation type, are the only place in the code where a significant amount of new code must be written each time a new derived quantity is desired by the analysis algorithms. These are written independently of the

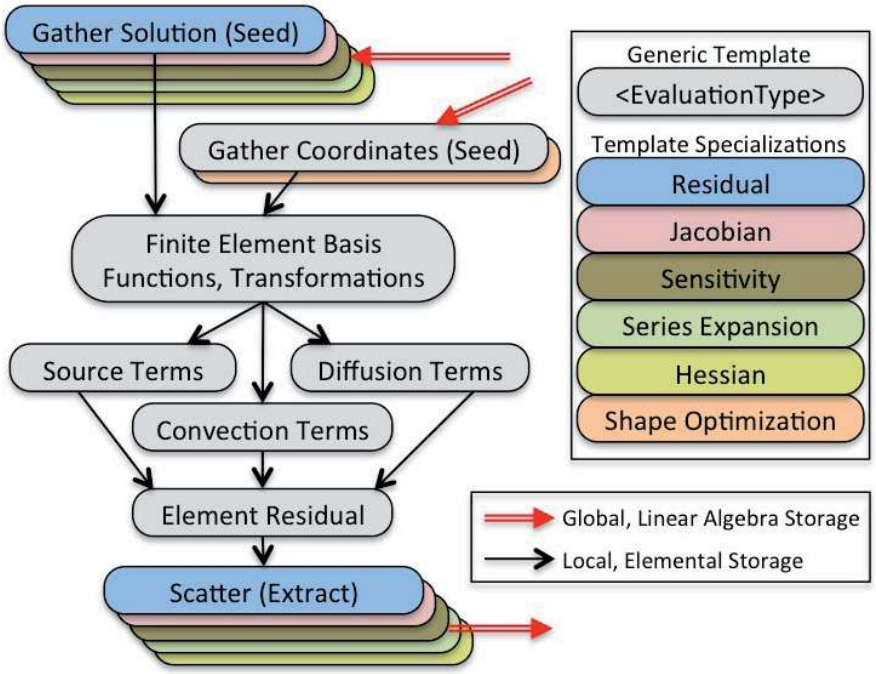

FIG. 5: Finite element assembly in Albany relies on the template-based generic programming (TBGP) approach and a graph-based assembly of individual 'evaluators' (Pawlowski et al., 2012b). With the TBGP approach, the developer programs the residual equations and identifies the design parameters. The TBGP infrastructure and automatic differentiation libraries in Trilinos will then automatically compute the Jacobian matrix and direct sensitivities. The graph-based approach simplifies implementation of new models and allows for broad reuse between applications.

equations being solved and are used for all problems. Thus, we have effectively orthogonalized the tasks of developing new multiphysics simulations from the tasks of incorporating new nonlinear analysis methodologies. A full description of the Template Based Generic Programming approach can be found in this pair of publications (Pawlowski et al., 2012a,b).

3.3.1 Boundary Conditions

Correctly and efficiently applying boundary conditions in multiphysics simulations over complex geometries/domains is also a significant source of software complexity. Currently, Albany supports simple Dirichlet conditions as well as a growing list of Neumann boundary condition types such as scalar flux conditions normal to boundaries, Robin conditions, and various traction and pressure boundary conditions. Dirichlet conditions are applied in the strong form directly to global linear algebra objects produced for each evaluation type after the volumetric finite element assembly by replacing the finite element residual equation for the corresponding nodes with a Dirichlet residual equation. For example, the Dirichlet condition $u(x) = a$ for $x \in \partial\Omega_D$ is implemented by replacing the finite element residual values corresponding to degrees-of-freedom associated with $\partial\Omega_D$ with $u - a$. Further, the Jacobian is modified to place the value of unity in the location corresponding to the degree of freedom and the remainder of the row is zeroed, and then enforcement of the boundary condition is left to the nonlinear solver. We have found that this approach is effective for nonlinear analysis problems such as sensitivity analysis, continuation/bifurcation analysis, optimization, and uncertainty quantification when the boundary condition must be a parameter in the problem as it allows for straightforward computation of derivatives with respect to the boundary condition, but with little additional cost in solver complexity.

The Neumann boundary condition implementation depends on a separate finite element assembly that performs the finite element surface integrals over the designated boundaries $\partial\Omega_N$. The form of these conditions can vary significantly, some examples supported by Albany include:

(i) Flux conditions for scalar equations, such as the heat equation. For this case, one typically wishes to specify a heat flux through a surface $\partial\Omega_N$,

$$\frac{\partial T}{\partial \mathbf{n}}(x) = q(x), \tag{6}$$

for $x \in \partial\Omega_N$, where $\mathbf{n}$ is the unit normal to the boundary $\partial\Omega_N$ and $q(x)$ is the specified flux.

(ii) Prescribed tractions on the boundary of mechanics problems,

$$\mathbf{t} = \sigma\mathbf{n} = \bar{\mathbf{t}}, \tag{7}$$

on $\partial\Omega_N$; σ is the Cauchy stress tensor, $\mathbf{t}$ is the traction vector on the boundary, and $\bar{\mathbf{t}}$ are the specified traction components. Pressure boundary conditions are a special case of traction $\bar{\mathbf{t}} = -p\mathbf{n}$, where p is the fluid pressure.

(iii) A Robin condition is a mixed condition taking the form of a weighted combination of both Dirichlet and Neumann conditions,

$$au(x) + b\frac{\partial u(x)}{\partial \mathbf{n}} = h(x), \tag{8}$$

on $\partial\Omega_R$, where $h(x)$ is the boundary function or constraint being applied, and a and b are weights. These types of conditions are often called *impedance boundary conditions* in electromagnetic problems and *insulating boundary conditions* in convection-diffusion problems where one specifies that the convective and diffusive fluxes sum to zero $h(x) = 0, \forall x \in \partial\Omega_R$.

In the case of Neumann conditions, the field manager accesses surface and boundary element information from the abstract discretization interface, and Albany performs a finite element integration and assembly process over each boundary $\partial\Omega_N$ defined. Similar to the element integration process employed elsewhere in Albany, the Intrepid package is used to integrate the weak form of one of the above expressions over the portion of each element (the *element side*) that lies on the boundary. The contribution of the Neumann integral term for all evaluation type (residual, Jacobian, etc.) is computed using the same TBGP infrastructure as the volumetric terms.

3.3.2 Responses (Quantities of Interest)

An implication of supporting embedded nonlinear analysis such as embedded optimization is that post-processing of simulation solution values must now be handled by the simulation code. Furthermore, not only must the quantities of interest be computed but also derived quantities such as response gradients. Albany supports a growing list of response functions that employ the TBGP framework to simplify the evaluation of these quantities. All response functions implement a simple interface that abstracts the evaluation of the response function and corresponding derivatives, and simple response functions such as the solution at a point implement this interface. Many response functions, however, can be written as an integral of a functional of the solution over some or all of the computational domain. These response functions employ the field manager described above and implement the functional as evaluators applied to the corresponding sequence of mesh cells. Generally, this approach is similar to the finite element assembly process described above except that response values and derivatives must be reduced across processors when run in parallel. To handle this complication, the response values for each evaluation type are reduced across processors before being extracted into their corresponding global linear algebra data structures using the template interface to MPI provided by the Teuchos (Thornquist et al., 2015) package in Trilinos.

3.4 Nonlinear Model Abstraction and Libraries

The "Nonlinear Model" abstraction in Fig. 3 is a Trilinos class called the `EpetraExt::ModelEvaluator`, which we will hereafter refer to as the ModelEvaluator. More complete documentation of this class and associated functionality is given in a technical report (Pawlowski et al., 2011). Albany satisfies the ModelEvaluator interface, making available all the embedded nonlinear analysis solution methods in Trilinos.

The purpose of the ModelEvaluator is to facilitate the use of general purpose solution and analysis algorithms such as those listed as "Solvers" in Fig. 6. For instance, a general purpose nonlinear solver needs an interface to ask the application code to compute a residual vector f as a function of a solution vector u in order to solve the nonlinear algebraic system,

$$f(u) = 0. \tag{9}$$

To complete the Newton solution process, the solver needs to query the application for other quantities, such as a Jacobian matrix or an approximation to the Jacobian for use in generating a preconditioner. By using a standard interface, sophisticated solution algorithms can be written that are agnostic to the physics, model, and data structures that are needed to support the above matrix and vector abstractions. This satisfies the component design philosophy

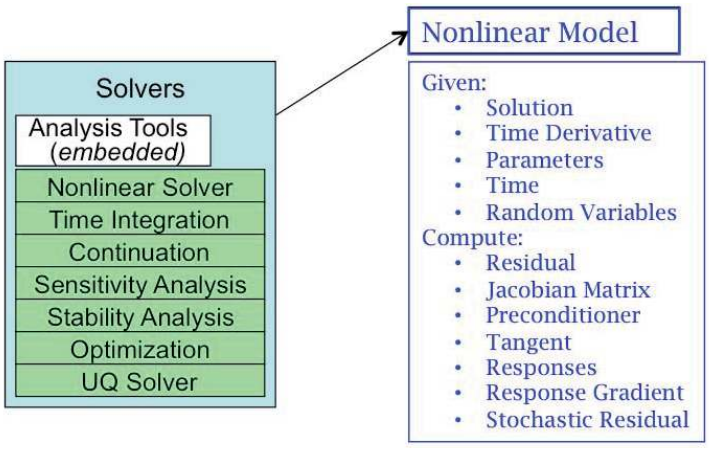

FIG. 6: Access to the embedded solvers in Trilinos requires that Albany satisfy the Nonlinear Model abstraction. In its simplest form, this abstraction is used to compute the nonlinear residual $f(u)$. The interface is general to accommodate the computation of Jacobian operators, user-defined preconditioners, and stochastic Galerkin expansions.

of making different parts of the development effort (in this example, the physics description and implementation of nonlinear solution algorithms) essentially orthogonal to each other.

Beyond this nonlinear solver example, the ModelEvaluator provides an extensible interface to the application code for the analysis algorithms. As time-dependent, continuation, sensitivity analysis, stability analysis, optimization, and uncertainty quantification capabilities are desired, the interface requirements grow to involve dependence on not just the solution vector u but also the time derivative $\dot{u}$, a set of parameters p, and the time t. Outputs of the interface includes sensitivities df/dp as well as responses (also known as quantities of interest) and response gradients with respect to u and p.

For Albany to remain useful as a research code, this interface definition needs to keep pace with the leading edge of algorithmic research and development. The design has been extended to support the ability to take polynomial expansions of stochastic random variables as inputs to return polynomial representations of the output quantities including the residual vector, Jacobian matrix, and responses.

In Albany, we have implemented a variety of these quantities to make use of the capabilities of the "Solvers" in Fig. 6. This single interface is all that is needed by the Trilinos Piro package. At run time, Piro will then select the desired solver method in the NOX, LOCA, Rythmos, or Stokhos package, and then subsequently performs the requested sensitivity analysis.

3.5 Linear Solver Abstraction and Libraries

The next capability described by an abstract interface is the linear solver, as shown in Fig. 7. The use of libraries and established interfaces for linear solvers is common in finite element codes. In Albany, however, there is no need to interface directly to the linear solver, as solves occur as inner iterations of the nonlinear, transient, continuation, optimization, and UQ solvers that were presented in the previous section.

For linear solves, there is a wide assortment of direct and iterative algorithms, and the iterative methods can make use of a variety of algebraic, multilevel, and block preconditioners. Furthermore, these algorithms can be called in a diversity of ways with different algorithms being used on various blocks of the matrix, on various levels of the multilevel method, and isolated to sub-domains of various sizes.

Much of this flexibility is configurable at run time through the use of the Trilinos Stratimikos package, the linear solver strategies interface. The Stratimikos package "wraps" the numerous linear algebra objects, solvers, and preconditioners found in Trilinos using a common abstraction (Thyra), and the approach supports the use of a factory pattern to create the desired linear solver object. The object is fully configurable at run time using parameters given in the Albany input file. In Trilinos, this involves the Ifpack, ML, Amesos, and Teko preconditioning packages and the AztecOO, Belos, and Amesos solver packages.

In Albany, the majority of the regression test problems employ a GMRES iterative solver that in turn uses either ILU or multilevel preconditioning. There are examples of how to use block methods and matrix-free solution approaches may be selected, also at run time from the input file.

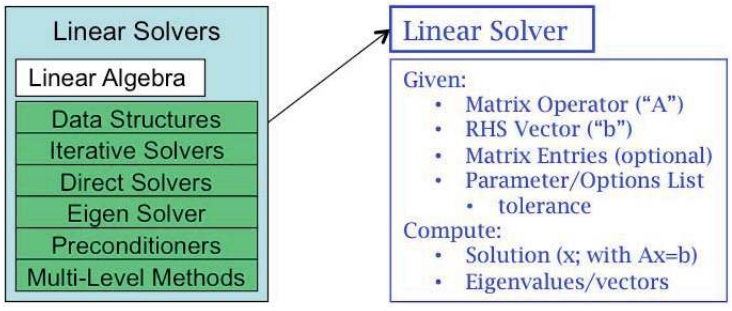

FIG. 7: Linear solver abstraction provides full access to all the linear solvers and preconditioners in Trilinos. A factory class supports run-time solution configuration through input file options.

3.6 Analysis Tools Abstraction and Libraries

Present at the top level of the software stack are the analysis tools, whose interface is shown in Fig. 8. These tools may be used to perform a single forward solve, sensitivity analysis, parameter studies, bifurcation analysis, optimization, and uncertainty quantification (UQ) runs. The analysis tools have a common abstract behavior in that they involve repeated calls to the solvers in Section 3.4 to determine how the solution changes as a function of the parameter.

The common abstraction layer that the analysis tools conform to (and that is implemented by the "Solvers") takes parameter values as input and returns responses and (optionally) response gradients with respect to the parameters as output. The analysis abstraction interface does not contain references to solution vectors or residuals, as analysis at this level operates along the manifold of the equations being solved.

The analysis abstraction layer is contained within the Trilinos Piro (Parameters In Responses Out) package. However, the majority of the specific analysis functionality actually resides within Dakota; a mature, widely used, and actively developed software framework that provides a common interface to a number of analysis, optimization, and UQ algorithms. Dakota optimization capabilities include gradient-based local algorithms, pattern searches, and genetic algorithms. Available UQ algorithms range from Latin hypercube stochastic sampling to stochastic collocation methods for polynomial chaos approaches. Dakota can be run as a separate executable that repeatedly launches a given application code, using scripts to modify parameters in input files. In Albany, Dakota is used in library mode through an abstract interface. A small Trilinos package called TriKota provides adapters between the Dakota and Trilinos analysis abstraction classes.

In Albany, a software stack is available to provide analytic gradients to the analysis tools. The parameters in the PDEs are exposed so that automatic differentiation can be employed to compute sensitivities of the residual with respect to the parameters. Likewise, the response functions use automatic differentiation to compute gradients with respect to the solution vector and parameters. The "Solvers" then use this information, along with the system Jacobian to compute the gradient of responses with respect to responses along the manifold of the PDEs being solved, *analytically*. Currently, Hessian information is not computed, although much of the infrastructure exists to do so.

4. ALBANY APPLICATIONS

Albany's general discretization interface together with the use of a templated physics residual abstraction makes it quite suitable to host a wide variety of applications. Furthermore, it is straightforward to rapidly implement new applications, which is best demonstrated by the number of different examples contained within the regression test suite and the number of analysis applications based on Albany. The regression test suite contains simple to moderately complex problems representing a broad spectrum of phenomena, including:

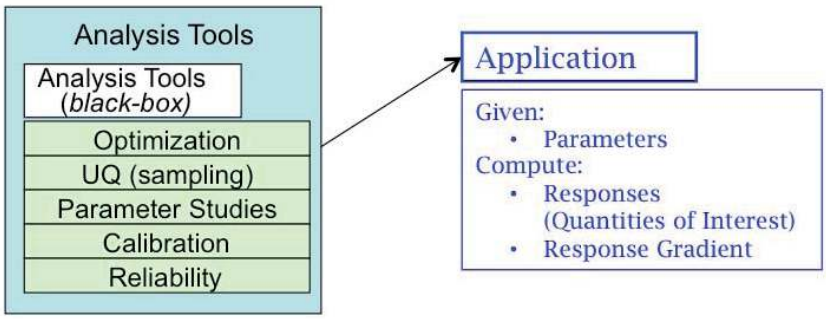

FIG. 8: At the top of the Albany computational hierarchy is the Analysis layer, where unconstrained optimization and UQ may be performed around the embedded nonlinear analysis solver layer. The interface accepts design parameters and returns responses (also known as quantities of interest or objective functions) and response gradients. The embedded solvers are wrapped to satisfy this interface and may be driven by the "Analysis Tools."

- Solid mechanics: elasticity, J2 plasticity, thermomechanics, unsaturated poroelasticity, thermo-poro-mechanics, diffusion-deformation, reactor fuel cladding hydride reorientation, gradient damage, rate-independent hardening minus recovery
- Fluid mechanics: compressible Navier–Stokes, ice sheet flows, prototype nuclear reactor model, vortex shedding, Rayleigh–Bernard
- Miscellaneous: heat equation, Poisson, Schrödinger, Cahn Hilliard/elasticity, Poisson–Nernst–Planck

In addition, the type of solution and analysis performed on these applications covers a broad spectrum:

- steady, transient, continuation/load stepping, embedded stochastic–Galerkin, sensitivity analysis, stability analysis, and uncertainty propagation.

There are several analysis projects and simulation activities that have adopted Albany. Albany is the code base for an Ice Sheet project based on a nonlinear Stokes equation (Tezaur et al., 2015). It is also being used to extend and mature mesh quality improvement techniques based on the Laplace Beltrami equations (Hansen et al., 2005) for ultimate use in arbitrary Lagrangian Eulerian (ALE) analysis codes and to model the behavior of hydrides of Zircaloy used in nuclear reactor fuel during transport and handling operations (Chen et al., 2014b).

Within the Abany configure process, each physics capability set can be turned on or off to support the needs of the user. All applications run from the same executable, where the physics set is selected at the top of the input file. There is, however, a separate executable for invoking stochastic–Galerkin analysis, distinct from deterministic solves.

In the remainder of this section, we highlight the two most mature applications hosted in Albany. The purpose of these examples is to illustrate to the reader the diversity of applications that can be supported within Albany, using the component-based development approach.

4.1 Laboratory for Computational Mechanics

The Laboratory for Computational Mechanics (LCM) project adopted Albany to serve as a research platform to study issues in fracture and failure of solid materials and multiphysics coupling. At present, LCM capabilities include quasi-static solution of the balance of linear momentum with various constitutive models in the small strain regime, as well as a total-Lagrange, finite deformation formulation. Since many problems of interest involve multiple physical phenomena, various coupled physics systems have been implemented in a monolithic fashion.

Abstractions in the code base permit virtual isolation for the application-specific physics developer. Implementation of a physical quantity, such as the strain tensor, requires virtually no knowledge of the underlying infrastructure or data structures. As a result, domain-specific expertise in writing constitutive models can be leveraged in an efficient way. To that end a number of constitutive models are available in Albany that span simple elastic behavior at small strain, through three-invariant models for geomaterials, and including finite deformation, temperature-dependent metal plasticity models.

Specific implementation of constitutive models is greatly aided by the use of automatic differentiation, available from the Trilinos Sacado package. Constitutive response often requires the solution of a set of nonlinear equations that govern the evolution of the internal state variables local to the integration point. The system of equations typically becomes more difficult to solve as the physical fidelity of the model increases. Efficient solution of the local set of equations is often achieved employing an implicit, backward Euler integration scheme, solved using a Newton–Raphson iterative scheme, and requiring formulation and construction of the local system Jacobian for optimal convergence. Implementation of the local system of equations using automatic differentiation types has two significant advantages. The first is that the computed local Jacobian provides analytic sensitivities for the Newton iteration, resulting in optimal local convergence. The second advantage is that model changes do not require the re-derivation and re-implemenation of the local Jacobian, saving substantial development time that can instead be spent on model verification and evaluation. An example calculation using a constitutive model that employs this strategy can be seen in Fig. 9, where a Gurson-type constitutive model with a set of four local independent variables is solved at each integration point.

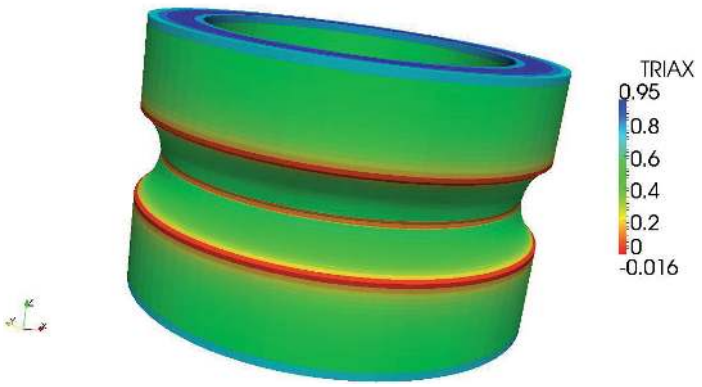

FIG. 9: Contours of stress triaxiality for a notched tube in a state of combined tension and torsional loading computed with a Gurson-type model

The existence of the load stepping capability, available through the continuation algorithms contained in the Trilinos Library of Continuation Algorithms (LOCA) package (Salinger et al., 2005), allows for the solution of boundary value problems with nonlinearities in both the material and geometric sense. In addition, the stepping parameter can be adaptively selected based on characteristics of the current solution. For example, this adaptive step refinement is essential for the robust solution of problems experiencing a plastic localization, where convergence is difficult to achieve and smaller continuation steps are required. Mechanics development can leverage this adaptive stepping capability without the need for domain expertise in its formulation and implementation, providing great value for the mechanics researcher.

From the perspective of the LCM application team, a strength of Albany is the ease in which coupled systems of PDEs can be implemented. This team has formulated, implemented, and demonstrated several coupled physics problems including thermo-mechanics, hydrogen diffusion-mechanics, and poro-mechanics. Each of these physics sets was implemented in a fully coupled sense and solved in a monolithic fashion with analytic Jacobian sensitivities provided by the automatic differentiation of the system residual. In particular, the graph-based assembly can explicitly show dependencies and can be a tremendous aid during model development and debugging. Example results from the thermo-mechanics problem can be seen in Fig. 10.

A demonstration of the poro-mechanics capabilities, outlined in Sun et al. (2013), and applied to a geomechanical footing problem can be seen in Fig. 11. As detailed in Sun et al. (2014), we chose a formulation that solves for the displacement vector and the pore pressure as primitive variables. The Galerkin weak form is stabilized via a polynomial projection method such that equal-order basis functions can be used. This model was verified against analytical

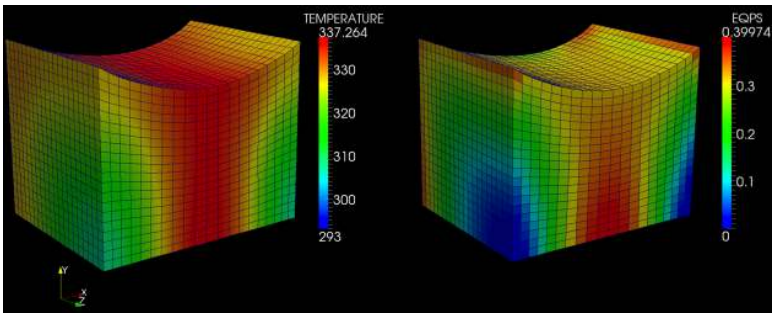

FIG. 10: Cubic domain fully clamped on x faces to eliminate contraction and given a prescribed displacement on top and insulated boundaries. Resultant temperature field stems solely from mechanical source terms.

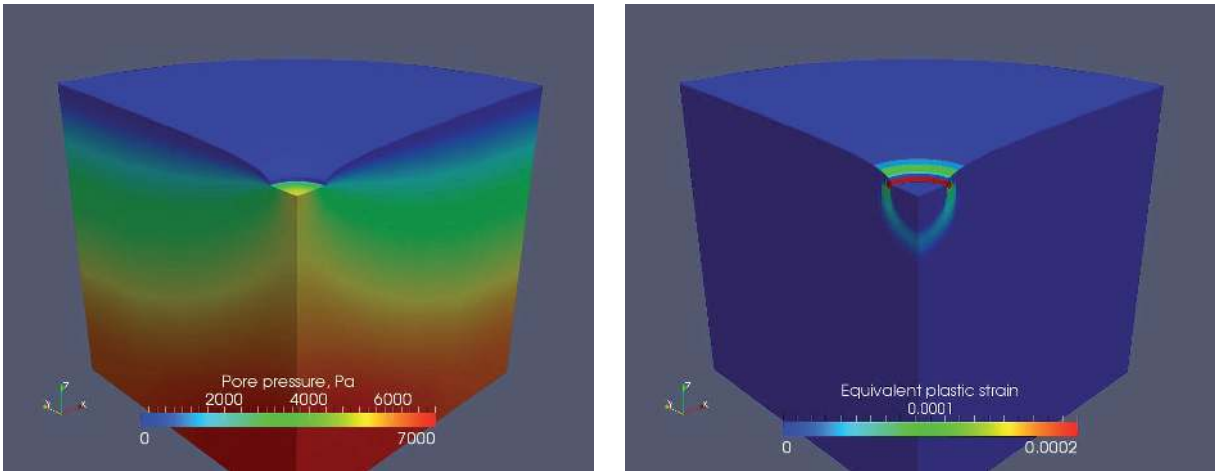

FIG. 11: Contours of pore pressure and equivalent plastic strain for a cylindrical footing

solution available in the literature and subsequently used to analyze how fluid diffusion changes the mechanical response of fluid infiltrating and how solid deformation induces fluid flow in elasto-plastic porous media (Chen et al., 2014c; Sun et al., 2014).

Another strength of the Albany system design becomes apparent when considering the scalability of solving the resulting linear systems. The ability to explore the use of massively parallel solvers and scalable multigrid preconditioners, such as that provided by the Trilinos ML package, makes Albany a desirable open source research environment. The general interface to the ML preconditioner involves obtaining mesh coordinate information from the abstract discretization interface that supplies information about the rigid body modes (the null space characteristics) of the system. Currently, Albany supports computing the number of rigid body modes both with and without the presence of other coupled solution fields, and the scalability of the preconditioner has been established up to many millions of degrees of freedom.

In summary, the design of Albany has allowed for the rapid implementation of the fundamental computational mechanics infrastructure, paving the way for research efforts into new methods and models. The open source nature of the code base serves as a foundation for academic collaboration. Successful research ideas are targeted for transition into Sandia's internal production analysis codes.

4.2 Quantum Computer-Aided Design

The quantum computer-aided design (QCAD) project uses Albany to develop a simulation and design capability for the electronic structure of laterally gated quantum dots, to determine their usefulness as qubits in quantum computing devices, and to help analyze experimental results on such devices (Gao et al., 2013). Such a task is a subset of semiconductor device simulation. In this case, we are targeting a regime not well covered by previous tools, specifically low-temperature operation close to absolute zero Kelvin, and few- or one-electron devices. Albany was chosen because it provided access to the many finite element, solver, and analysis libraries, and a programming model that enabled us to efficiently implement several physics sets that our application presents.

Quantum dots are regions in a semiconductor where the local electrostatics allows "puddles" of electrons to form, typically near a semiconductor–insulator interface. We often use a silicon metal–oxide–semiconductor (MOS) system, with an additional level of gates in the insulator to deplete the sheet into puddles that form quantum dots, as shown in Fig. 12. The depletion gates themselves in experimental quantum dots can have many different and complex three-dimensional (3D) geometries. Figure 13 shows three examples of typical depletion gate patterns in a top

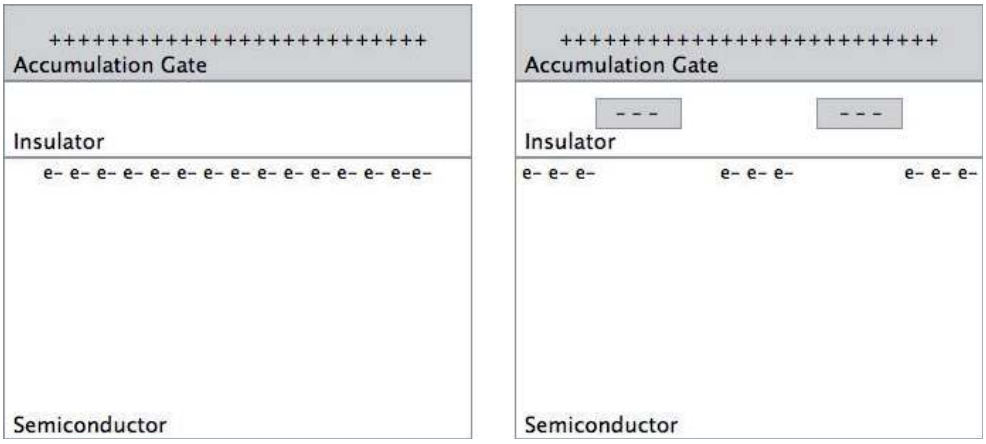

FIG. 12: Cross section view of a simplified quantum dot device to illustrate the concept. We can form sheets ("e-") of electrons at a MOS interface using an accumulation gate with a positive ("+") voltage (left figure). By introducing additional depletion gates with negative ("-") voltage, we can deplete most of this sheet, leaving puddles that form quantum dots (right figure).

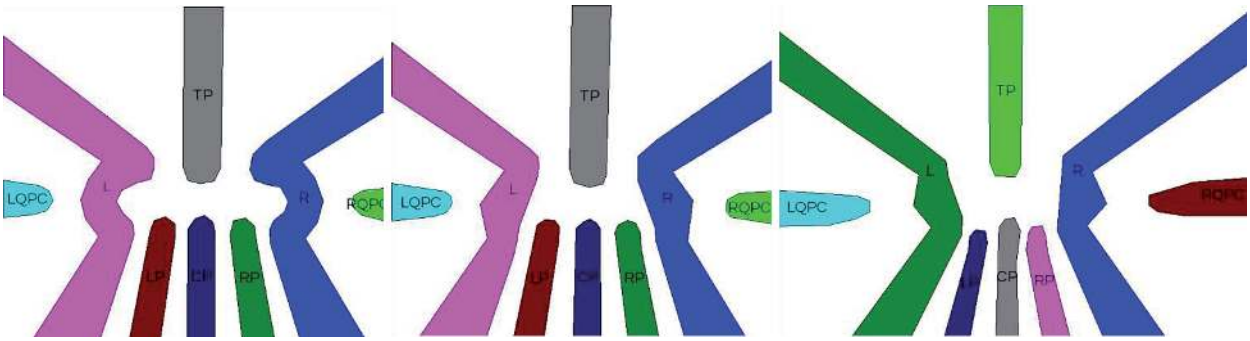

FIG. 13: Examples of typical depletion gate patterns in experimental quantum dot devices in a top view. Each color in the left, middle, and right figures indicates a metal or polysilicon gate that can be set to a different voltage to form a quantum dot. The labels on these figures are strings chosen during the meshing process, and used to associate a boundary condition specification in the Albany input file with each gate.

view. The quantum effects we wish to use to form qubits are most pronounced with few numbers of electrons, and a major challenge is to design robust enough structures that allow formation of few-electron dots. This often involves modifying the shapes of the gates and the spacings between different layers.

The gate voltages dictate Dirichlet boundary conditions along the surfaces of the regions that form the gates. We have developed and validated three major solvers of increasing computational complexity. The nonlinear Poisson solver determines the electrostatic potential profile that results from the gate voltages and other device parameters in a given device by treating electrons semi-classically, that is, as classical particles that obey quantum (Fermi–Dirac) statistics. The simplest formulation facilitates rapid simulations of many designs, which enables fast semi-classical understanding of device behavior and hence rapid feedback on device designs. The Schrödinger–Poisson (S-P) solver is a multiphysics model which couples the nonlinear Poisson solver and a Schrodinger solver in a self-consistent manner to capture quantum effects in our devices. Finally, the Configuration Interaction solver takes single-particle solutions from the S-P solver and determines multielectron solutions that include quantum interactions between electrons.

The Albany framework has made it straightforward and fast to implement these QCAD solvers. The general Poisson equation is written as

$$\nabla(\epsilon_s \nabla\phi) = \rho(\phi), \tag{10}$$

where ϕ is the electrostatic potential to be solved for and $\rho(\phi)$ can be a nonlinear function. The corresponding finite element weak form (leaving out the surface term for this presentation)

$$\int \epsilon_s \nabla\phi \cdot \nabla w d\Omega + \int \rho(\phi) w d\Omega = 0, \tag{11}$$

with w being the nodal basis function and the LHS being defined as residual. To solve the equation in the Albany framework, we created a concrete `QCAD::PoissonProblem` class derived from `Albany::AbstractProblem`, in which we constructed the residual by evaluating and assembling each term. The static permittivity ϵ_s and the source $\rho(\phi)$ are evaluated in separate QCAD-specific `evaluators`, while the integrations are done by general purpose Albany `evaluators`. The automatic differentiation (AD) capability, parallelization, and nonlinear and linear solvers were available without any development effort for the QCAD projects physics sets. Through parallelism, robustness, and automation enabled by analysis algorithms, the throughput of quantum dot simulations increased several orders of magnitude over the previous simulation process that was being employed.

The S-P solver self-consistently couples the nonlinear Poisson solver, above, with a Schrödinger eigensolver. The latter solves a single-particle effective mass Schrödinger equation

$$-\frac{\hbar^2}{2}\nabla\left(\frac{1}{m^*}\nabla\psi\right) + V(\phi)\psi = E\psi. \tag{12}$$

The weak form of this equation was implemented similar to the implementation of the nonlinear Poisson solver. The Trilinos eigensolver Anasazi is used to approximate the leading modes of the discretized eigenproblem after undergoing a spectral transformation, using infrastructure originally developed for stability analysis (Lehoucq and Salinger, 2001). The self-consistent loop is done in an aggregate ModelEvaluator, which splits the S-P problem into Schrodinger and Poisson sub-problems and calls the corresponding solve to solve each, as illustrated in Fig. 14. The iteration is continued until a pre-defined convergence criterion is satisfied.

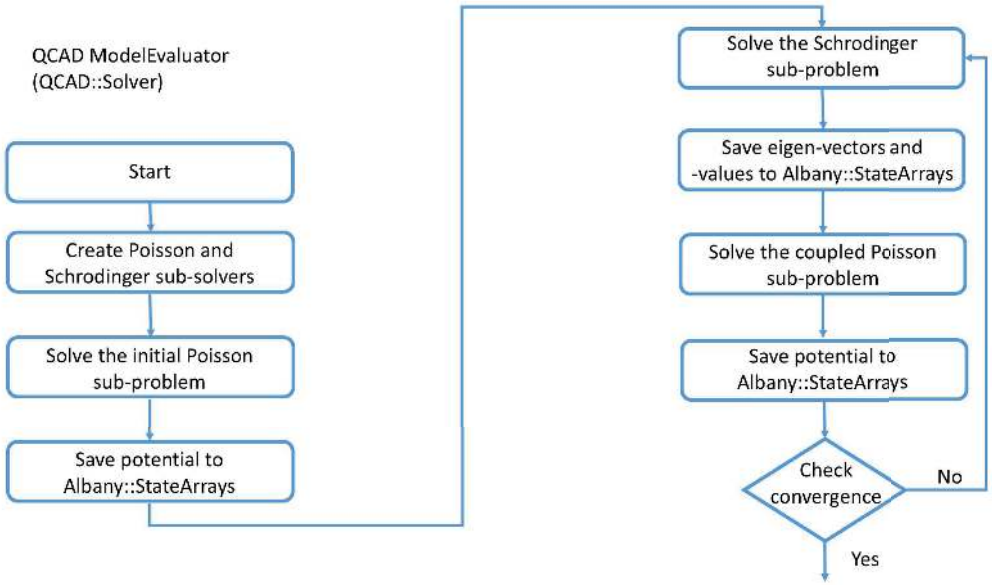

FIG. 14: Schematic diagram showing the Schrödinger–Poisson implementation in QCAD

A large part of the code development for the QCAD project is to compute application-specific response functions. We have coded several responses for our quantum devices, including average value and integral of a field in a given region. One particular response that has been crucial for our devices is finding the saddle path between two minima. The saddle path searching algorithm is fairly complicated and it was relatively easy to fit into the Albany response framework. The AD capability is critical for computation of gradients of responses.

Another key element to the process of developing the QCAD code was the additional packages integrated in the Albany workflow. In particular, the Cubit mesh generator (Hanks et al., 2013) and the Dakota optimization & UQ package (Adams et al., 2009). Albany supports a variety of finite element topologies such as quadrilateral and triangle in 2D, hexahedron and tetrahedron in 3D. The code is written for arbitrary nodal discretization order, though only linear and quadratic basis functions have been accessed. The code can import the meshed from the ExodusII (Sjaardema et al., 2013) format, which is generated by Cubit. This capability allows us to use Cubit to create highly nonuniform 3D test meshes, since our structures often have complex 3D shapes as shown in Fig. 13. The Dakota package available to Albany via the TriKota interface provides various optimization options that have been extremely useful in optimizing complicated targets for our devices.

An example of the type of optimization we performed is given in Fig. 15. We wished to optimize a quantum dot to contain exactly two electrons, with tunable tunnel barriers in and out of the dot region, between the left and right electrons of the dot, and with the channels on the sides also having tunable tunnel barriers. The voltages on all gates (shown from a top view on the left side of Fig. 15) are allowed to vary as design parameters, with the left/right symmetry in the gate voltages imposed as a constraint. The right side of Fig. 15 shows the resulting electron density after Dakota found the optimal voltages that satisfied all the targets. This was performed by repeatedly calling the nonlinear Poisson solver for the response and analytically computed gradients. The red region is the "sheet" of electrons, and the blue regions have few electrons and somewhat follow the shapes of the depletion gates. The quantum dot itself is the narrow curved region underneath the gate labeled TP in the left.

In summary, the numerous capabilities that Albany provides enable us to rapidly develop application-focused QCAD solvers. The resulting design tool has many more functionalities than we had proposed at the beginning of the project. As a result, QCAD simulations have become an integral part of the *experimental* effort in silicon qubit design.

5. CONCLUSIONS

In this paper, we have articulated a strategy for the construction of computational science applications that promotes the use of reusable software libraries, abstract interfaces, and modern software engineering tools. We believe that the component-based application development model proposed here has a strong potential for success, due to our experiences with Albany and other Trilinos capabilities cited. There are significant advantages to the degree of soft-

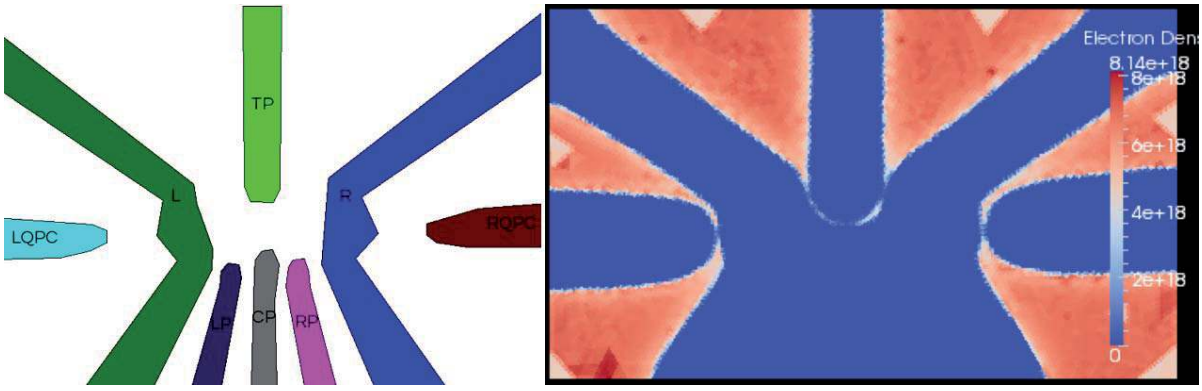

FIG. 15: Optimization of the Ottawa Flat 270 structure. The left figure shows a top view of the depletion gate configuration for the structure, and the right figure shows the resulting electron density after optimization was achieved, with a variety of constraints detailed in the text.

ware reuse that well-designed components can offer, along with the use of generic physics interfaces supported with template-based generic programming methods and the analysis capabilities that are enabled by their use. It remains common, however, for application codes to make limited use of external libraries. Some of the reasons include the learning curve of using (and debugging) someone else's code, difficulties in maintaining compatible versions and in porting, and the challenges with interfacing a collection of different libraries.

Many of these issues have been overcome in the Trilinos suite. The libraries built in Trilinos share a common build system and release schedule. Where possible, independent capabilities that should work together, like a nonlinear solver inside of an implicit time integrator, provide a general interface. Also, many capabilities that are typically used in a similar way, such as linear solvers and embedded nonlinear analysis tools, can be called with the same interface and selected at run time through a factory pattern.

We have built the Albany finite element code attempting to follow, and test the efficacy of, the component-based strategy, and making use of the broad set of computational capabilities in Trilinos. In Section 3 we provided an overview of the software design and abstractions important in the development of Albany, an extensible generic unstructured-grid, implicit, finite element application code. The design is modularized with abstract interfaces, where we have shown that we can independently replace physics sets, mesh databases, linear solvers, nonlinear solvers, and analysis tools, to achieve the ultimate application goals. Independently developed Trilinos libraries contribute to the code capabilities. The bulk of the Albany code base is devoted to the equations and response functions that describe the application.

The evidence presented on the success of this approach and our implementation comes from two applications that have been built in the Albany code base, which were presented in Section 4. The feedback from these development efforts is that it is straightforward to rapidly develop sophisticated parallel analysis codes employing advanced discretizations, high-performance linear solvers and preconditioners, a wide range of nonlinear and transient solvers, and sophisticated analysis algorithms, using the proposed methodology. The LCM code has been able to explore fully coupled solution algorithms for mechanics coupled with additional scalar equations. By writing tensor operations in an independent library that is templated to allow for automatic differentiation data types, the project demonstrated that one can quickly investigate new models. In less than 2 years of effort, the QCAD project was able to improve their throughput by several orders of magnitude, leading to a new workflow where a tentative design is thoroughly investigated by a running a suite of optimization runs on a high-fidelity model, instead of manually exploring a limited number of forward simulations. Using this capability, the project has been successful in incorporating computational analysis into the design cycle used by experimentalists.

ACKNOWLEDGMENTS

The Albany code builds on a wide variety of computational science capabilities; we would like to acknowledge the contributions of the authors of these libraries and tools. There are several who directly impacted the component-based code design strategy and the Albany code base, including Mike Heroux, Jim Willenbring, Brent Perschbacher, Pavel Bochev, Denis Ridzal, Carter Edwards, Greg Sjaardema, Eric Cyr, Julien Cortial, Brian Adams, and Mike Eldred. In addition, this effort has had significant management support, including that of David Womble, Scott Collis, Rob Hoekstra, Mike Parks, John Aidun, and Eliot Fang. This work was funded by the US Department of Energy through the NNSA Advanced Scientific Computing (ASC) and Office of Science Advanced Scientific Computing Research (ASCR) programs, and the Sandia Laboratory Directed Research and Development (LDRD) program. Sandia National Laboratories is a multiprogram laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

REFERENCES

Adams, B., Bohnhoff, W., Dalbey, K., Eddy, J., Eldred, M., Gay, D., Haskell, K., Hough, P., and Swiler, L., DAKOTA, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity

analysis: Version 5.0 User's Manual. Tech. Rep. SAND2010-2183, Sandia National Laboratories. Updated December 2010 (Version 5.1). Updated November 2011 (Version 5.2). Updated February 2013 (Version 5.3), 2009.

Balay, S., Brown, J., Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H., PETSc Users Manual. Tech. Rep. ANL-95/11 - Revision 3.4, Argonne National Laboratory, 2013.

Bangerth, W., Hartmann, R., and Kanschat, G., deal.ii: A general-purpose object-oriented finite element library, *ACM Trans. Math. Softw.*, vol. **33**, no. 4, pp. 24/1–24/27, http://doi.acm.org/10.1145/1268776.1268779, 2007.

Bartlett, R. A., Heroux, M. A., and Willenbring, J. M., Tribits lifecycle model, SAND Report SAND2012-0561, Sandia National Laboratories, 2012.

Bochev, P., Edwards, H., Kirby, R., Peterson, K., and Ridzal, D., Solving PDEs with Intrepid, *Sci. Program.*, vol. **20**, no. 2, pp. 151–180, 2012.

Booch, G., *Object Oriented Design with Applications*, Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991.

Brown, C. T. and Canino-Koning, R., Continuous integration, http://aosabook.org/en/integration.html, 2015.

Chen, Q., Ostien, J. T., and Hansen, G., Automatic differentiation for numerically exact computation of tangent operators in small- and large-deformation computational inelasticity, *TMS 2014 Supplemental Proceedings*, John Wiley & Sons, Inc., New York, pp. 289–296, http://dx.doi.org/10.1002/9781118889879.ch38, 2014a.

Chen, Q., Ostien, J. T., and Hansen, G., Development of a used fuel cladding damage model incorporating circumferential and radial hydride responses, *J. Nuclear Mater.*, vol. **447**, no. 1-3, pp. 292–303, 2014b.

Chen, Q., Sun, W., and Ostien, J. T., Finite element analysis of hydro-mechanical coupling effects on shear failures of fully saturated collapsible geomaterials, in American Society of Civil Engineers, GeoShanghai, Shanghai, China, pp. 688–698, 2014c.

de St. Germain, J. D., McCorquodale, J., Parker, S. G., and Johnson, C. R., Uintah: A massively parallel problem solving environment, in *Ninth IEEE Intl. Symp. on High Performance and Distributed Computing*, IEEE, pp. 33–41, 2000.

Diachin, L., Bauer, A., Fix, B., Kraftcheck, J., Jansen, K., Luo, X., Miller, M., Ollivier-Gooch, C., Shephard, M. S., Tautges, T., and Trease, H., Interoperable mesh and geometry tools for advanced petascale simulations, *J. Phys.: Conf. Ser.*, vol. **78**, no. 1, 012015, 2007.

Edwards, H. C., Williams, A. B., Sjaardema, G. D., Baur, D. G., and Cochran, W. K., SIERRA toolkit computational mesh conceptual model. Tech. Rep. SAND2010-1192, Sandia National Laboratories, 2010.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-oriented Software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

Gao, X., Nielsen, E., Muller, R. P., Young, R. W., Salinger, A. G., Bishop, N. C., Lilly, M. P., and Carroll, M. S., Quantum computer aided design simulation and optimization of semiconductor quantum dots, *J. Appl. Phys.*, vol. **114**, no. 16, 164302, 2013.

Gaston, D., Newman, C., Hansen, G., and Lebrun-Grandie, D., Moose: A parallel computational framework for coupled systems of nonlinear equations, *Nuclear Engineering and Design*, vol. **239**, no. 10, pp. 1768–1778, 2009.

Hanks, B., et al., Cubit web site, http://cubit.sandia.gov/, 2013.

Hansen, G., Zardecki, A., Greening, D., and Bos, R., A finite element method for three-dimensional unstructured grid smoothing, *J. Comput. Phys.*, vol. **202**, no. 1, pp. 281–297, 2005.

Hansen, G., et al., Albany CDash CI site, http://my.cdash.org/index.php?project=Albany, 2015.

Heroux, M. A., Bartlett, R. A., Howle, V. E., Hoekstra, R. J., Hu, J. J., Kolda, T. G., Lehoucq, R. B., Long, K. R., Pawlowski, R. P., Phipps, E. T., Salinger, A. G., Thornquist, H. K., Tuminaro, R. S., Willenbring, J. M., Williams, A. B., and Stanley, K. S., An overview of the Trilinos project, *ACM Trans. Math. Softw.*, vol. **31**, no. 3, pp. 397–423, 2005.

Kirk, B. S., Peterson, J. W., Stogner, R. H., and Carey, G. F., `libMesh`: A C++ library for parallel adaptive mesh refinement/coarsening simulations, *Eng. Comput.*, vol. **22**, no. 3-4, pp. 237–254, 2006.

Lehoucq, R. B. and Salinger, A. G., Large-scale eigenvalue calculations for stability analysis of steady flows on massively parallel computers, *Int. J. Numer. Methods Fluids*, vol. **36**, no. 1, pp. 309–327, 2001.

Logg, A., Mardal, K.-A., and Wells, G. N., *Automated Solution of Differential Equations by the Finite Element Method*, Springer, New York, 2012.

Long, K. R., Kirby, R. C., and van Bloemen Waanders, B. G., Unified embedded parallel finite element computations via software-based Fréchet differentiation, *SIAM J. Sci. Comput.*, vol. **32**, no. 6, pp. 3323–3351, 2010.

Michopoulos, J. G., Farhat, C., and Fish, J., Modeling and simulation of multiphysics systems, *J. Comput. Information Sci. Eng.*,

vol. **5**, no. 3, pp. 198–213, 2005.

Notz, P. K., Pawlowski, R. P., and Sutherland, J. C., Graph-based software design for managing complexity and enabling concurrency in multiphysics PDE software, *ACM Trans. Math. Softw.*, vol. **39**, no. 1, pp. 1–21, 2012.

Pawlowski, R. P., Phalanx web site, http://trilinos.sandia.gov/packages/phalanx/, 2015.

Pawlowski, R. P., Bartlett, R. A., Belcourt, N., Hooper, R. W., and Schmidt, R. C., A theory manual for multi-physics code coupling in LIME, Tech. Rep. SAND2011-2195, Sandia National Laboratories, 2011.

Pawlowski, R. P., Phipps, E., and Salinger, A. G., Automating embedded analysis capabilities and managing software complexity in multiphysics simulation, Part I: Template-based generic programming, *Sci. Program.*, vol. **20**, pp. 197–219, 2012a.

Pawlowski, R. P., Phipps, E. T., Salinger, A. G., Owen, S. J., Siefert, C. M., and Staten, M. L., Automating embedded analysis capabilities and managing software complexity in multiphysics simulation Part II: Application to partial differential equations, *Sci. Program.*, vol. **20**, no. 2, pp. 327–345, 2012b.

Phipps, E. and Pawlowski, R. P., Efficient expression templates for operator overloading-based automatic differentiation, in *Recent Advances in Algorithmic Differentiation*, Forth, S., Hovland, P., Phipps, E., Utke, J., and Walther, A., Eds., Lecture Notes in Computer Science, Springer, pp. 309–319, 2012.

Phipps, E. T., Sacado web site, http://trilinos.sandia.gov/packages/sacado/, 2015a.

Phipps, E. T., Stokhos web site, http://trilinos.sandia.gov/packages/stokhos/, 2015b.

Prud'homme, C., Life: Overview of a unified C++ implementation of the finite and spectral element methods in 1D, 2D and 3D, *Applied Parallel Computing. State of the Art in Scientific Computing*, Springer, Lecture Notes in Computer Science, vol. **4699**, pp. 712–721, 2007.

Salinger, A. G., Component-based scientific application development, Tech. Rep. SAND2012-9339, Sandia National Laboratories, 2012.

Salinger, A. G., Burroughs, E. A., Pawlowski, R. P., Phipps, E. T., and Romero, L. A., Bifurcation tracking algorithms and software for large scale applications, *Int. J. Bifurcat. Chaos*, vol. **15**, no. 3, pp. 1015–1032, 2005.

Seol, S., Smith, C., Ibanez, D., and Shephard, M., A parallel unstructured mesh infrastructure, *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion*, pp. 1124–1132, doi: 10.1109/SC.Companion.2012.135, 2012.

Sjaardema, G., et al., Exodus web site, http://sourceforge.net/projects/exodusii/, 2013.

Smith, T. M., Shadid, J. N., Pawlowski, R. P., Cyr, E. C., and Weber, P. D., Reactor core subassembly simulations using a stabilized finite element method, in *The 14th Intl. Topical Meeting on Nuclear Reactor Thermalhydraulics, NURETH-14*, Toronto, Ontario, Canada, 2011.

Stewart, J. R. and Edwards, H. C., The SIERRA framework for developing advanced parallel mechanics applications, *Large-Scale PDE-Constrained Optimization*, Biegler, L. T., Heinkenschloss, M., Ghattas, O., and van Bloemen Waanders, B., Eds., Lecture Notes in Computational Science and Engineering, vol. **30**, Springer Berlin, pp. 301–315, 2003.

Sun, W., Chen, Q., and Ostien, J. T., Modeling the hydro-mechanical responses of strip and circular punch loadings on water-saturated collapsible geomaterials, *Acta Geotech.*, vol. **9**, no. 5, pp. 903–934, 2014.

Sun, W., Ostien, J. T., and Salinger, A. G., A stabilized assumed deformation gradient finite element formulation for strongly coupled poromechanical simulations at finite strain, *Int. J. Numer. Anal. Methods Geomech.*, vol. **37**, no. 16, pp. 2755–2788, 2013.

Tezaur, I. K., Perego, M., Salinger, A. G., Tuminaro, R. S., and Price, S. F., Albany/FELIX: A parallel, scalable and robust finite element higher-order Stokes ice sheet solver built for advance analysis, *Geosci. Model Devel.*, vol. **8**, no. 4, pp. 1197–1220, 2015.

Thornquist, H., et al., Teuchos web site, http://trilinos.sandia.gov/packages/teuchos/, 2015.

Yu, Q. and Fish, J., Multiscale asymptotic homogenization for multiphysics problems with multiple spatial and temporal scales: A coupled thermo-viscoelastic example problem. *Int. J. Solids Struct.*, vol. **39**, no. 26, pp. 6429–6452, 2002.

Yuan, Z. and Fish, J., Nonlinear multiphysics finite element code architecture in object oriented Fortran environment, *Finite Elements Anal. Des.*, vol. **99**, pp. 1–15, 2015.