Theme Article

Developing a Computational Chemistry Framework for the Exascale Era

Ryan M. Richard
Ames Laboratory

Colleen Bertoni
Argonne National Laboratory

Jeffery S. Boschen
Ames Laboratory

Kristopher Keipert
Argonne National Laboratory

Benjamin Pritchard
Virginia Tech

Edward F. Valeev
Virginia Tech

Robert J. Harrison
Stony Brook University and Brookhaven National Laboratory

Wibe A. de Jong
Lawrence Berkeley National Laboratory

Theresa L. Windus
Ames Laboratory and Iowa State University

***Abstract*—Within computational chemistry, the NWChem package has arguably been the *de facto* standard for running high-accuracy numerical simulations on the most powerful supercomputers. In order to better address the challenges presented by emerging exascale architectures, the decision has been made to rewrite NWChem. Design of the resulting package, NWChemEx, has been driven by exascale computing; however, significant additional design considerations have arisen from the team's involvement with the Molecular Sciences Software Institute (MolSSI). MolSSI is a National Science Foundation initiative focused on establishing coding and data standards for the computational chemistry community. As a result, NWChemEx is built upon a general computational chemistry framework called the simulation development environment (SDE) that is designed with a focus on extensibility and interoperability. The present manuscript describes the modular approach of the SDE and how it has been used to implement the self-consistent field algorithm within NWChemEx.**

Digital Object Identifier 10.1109/MCSE.2018.2884921
Date of publication 6 December 2018; date of current version 8 March 2019.

Computational chemistry is a field aimed at modeling chemical phenomena in order to rationalize as well as predict experimental outcomes. The models within the field vary greatly in their computational and technical complexity. On the low-cost end one tends to find models suitable only for qualitative results; however, they are capable of modeling millions of atoms. At the other extreme lies the field of electronic structure theory (EST), which consists of methods capable of quantitatively replicating experimental measurements, but only for systems comprised of tens of atoms. An active area of research work in EST focuses on endeavors to extend the range of systems for which quantitative models are possible. Research work in this direction typically focuses on parallelization and/or developing new approximations for existing theories. Unfortunately, the code state of many of the current computational chemistry packages poses one of the largest barriers to such endeavors.

Most existing codes in EST originated as in-house codes for a single research group. Typically, these codes focused on one specific area of research and were developed with little to no formal software design practices (such as properly designed abstractions, data encapsulation, and standardization). Owing to the bootstrap nature of EST methods, as well as the evolving interests of the research groups, the scope of each of these packages grew and inevitably the packages' features began to overlap. Despite this overlap, most package developers have avoided leveraging existing implementations, instead choosing to reimplement the needed common functionality themselves. Historically this has been for a variety of reasons, although time constraints are arguably the leading factors. Making the situation worse, most EST package development relies solely on grant funding. The result is a perpetual cycle where established funding precedent means that in order for a developer's grant application to be competitive, novel methodologies must be implemented quickly and cheaply to maximize the production of scientific results. Thus developers often accumulate technical debt just to "get something working." This leaves little to no funding, or time, for repaying the already accumulated technical debt. Today the typical computational chemistry package has millions of lines of poorly structured, monolithic code, often with little to no reuse of the capabilities available in other packages or even within itself, and little to no developer documentation. This makes onboarding new developers, implementing novel methodologies, and porting existing methodologies to new computer architectures increasingly difficult. To make the situation worse, a typical package is not able to take full (or any) advantage of modern massively parallel platforms characterized by a great emphasis on data parallelism and vectorization, deep memory hierarchies, and heterogeneous and/or configurable hardware.

NWChem[1] has arguably been the *de facto* leader in high-performance computing (HPC) within the field of EST. Motivated largely by the challenges imposed by exascale computing we have decided to rewrite NWChem from the ground up. While the original NWChem did and does have an active software design process, the requirements of exascale computing and the current architectural diversity (many-core processors, graphical processing units, multilevel memory hierarchies, etc.) along with the additional requirements of advanced methods have outstripped the original design. The resulting new package, NWChemEx, is being designed foremost for performance on the upcoming exascale machines, but also for extensibility, workflow integration, and interoperability. Key to this effort is the simulation development environment (SDE), which provides a framework for writing an EST package in terms of callbacks to decoupled modules. As will be shown, this, in turn, leads to an ecosystem in which it is straightforward to introduce a new methodology, create an architecture-specific algorithm, or just call another package in a way that minimizes the amount of additional technical debt. It is our overarching goal that the SDE will foster a software community where researchers develop many SDE "apps," each for a specific task and architecture. Users of SDE based packages can then go to "app stores" and choose from among the myriad of available apps in order to customize their package.

BACKGROUND

The SDE can be thought of as a successor to the common component architecture (CCA), and the associated CCA Forum.[2] The goal of the CCA was to bring component-based software engineering practices to the scientific HPC communities. Within the CCA, software is assembled in terms of independent software units (called components). Each component had to maintain a specific interface and was in charge of a specific task. Coding language interoperability was achieved via the scientific interface definition language and the Babel tool.[3] The result was that the components could be assembled together into a larger application in a “plug and play” manner thereby promoting reusable, maintainable scientific software. The CCA also helped handle the architectural heterogeneity of supercomputers by allowing components optimized for different architectures to be swapped in and out.

Within the CCA chemistry community, efforts were focused on enabling the molecular geometry to be optimized from multiple quantum chemistry packages,[4] combining multiple theoretical methods to improve multilevel parallelism[5] and resource utilization.[6] In addition to these high-level efforts, there was also progress in standardizing interfaces to lower level functions, like integral calculations, which are an essential piece of most quantum chemistry codes.[7] Ultimately the CCA infrastructure was too heavyweight for most developers, requiring multiple levels of software to work together in a cumbersome manner with linking the software components together taking more development time than developing the components themselves. Although the CCA Forum was retired around 2010, the work done on standards and interfaces has continued to benefit the quantum chemistry community to this day.

Standardization efforts within the scientific computing community have recently been reinvigorated by the Molecular Sciences Software Institute (MolSSI), which is a National Science Foundation funded institute. In addition to establishing standards, the MolSSI is working with the community to ensure that these standards are adopted. While still early in the life of the project, a draft schema[8] has been developed for EST output data and NWChemEx will support this schema in order to capitalize on connections with other projects. Other elements of the MolSSI that are relevant to this work are the training of computational molecular sciences software developers in modern techniques for software engineering and the organization of leadership and community-lead workshops to engage the community in software and standards development.

With the growing popularity of languages designed around object-oriented (OO) paradigms, such as C++ and Python, developers in EST are gradually moving away from the procedural paradigms of the legacy codes; consequently, a number of EST packages embracing OO paradigms have sprung up. Unlike legacy codes, and with two notable exceptions,[9,10] generally these OO codes are unconcerned with performance beyond a single node, rather they see themselves as platforms for rapid development of new theories. For the most part, the ease of prototyping new methods comes from the fact that, compared to legacy codes, these packages tend to better embrace good software practices. For example, there has been a shift away from using domain-specific languages for package input, towards using established languages such as Python[11–14] or JSON[9,10,15] (JavaScript Object Notation) both of which integrate easily into typical workflows. Encapsulation also plays a key role in the design at both the algorithm level, by using classes, as well as at the package level via libraries. This, in turn, makes it easier to modify the code without side effects. Somewhat more recently many of these OO codes[9,12,13,16] have placed an emphasis on being able to easily extend the package by standardizing APIs and providing plugin mechanisms. Combined with MolSSI’s standardization efforts, this paves the way for interoperability among packages.

At a high level, the biggest innovation the SDE provides is automatic checkpointing of the calculation. EST calculations are notorious for the amount of compute time they require. Few EST packages (legacy or OO) have more than a rudimentary (if any) ability to save a calculation’s state. As architectures become more complex, hardware failures are going to become more common. Furthermore, in order to truly integrate into a dynamic workflow, it becomes essential to be able to refactor the workflow without needing extensive recomputation. The SDE also improves upon extensibility efforts pioneered by other OO EST packages. In particular, the SDE provides a plugin system that facilitates dynamic modifications to nearly the entirety of the call graph. By comparison, most other plugin systems[9,13] only allow users to add plugins relatively high on the call graph. This inhibits the amount of customization a user can do without actually modifying the package’s source code.

```
1  #include <SDE/SDE.hpp>
2  #include <LibChemist/LibChemist.hpp>
3  #include <SCF/SCF.hpp>
4
5  int main() {
6     //Define molecular system
7     auto mol = LibChemist::make_molecule(
8        "H 0.0 0.0 0.0
9         H 0.0 0.0 0.9"
10    );
11
12    //Declare modules
13    SDE::ModuleManager mm;
14    mm.insert("SCF", [](){return std::make_shared<scf::SCF>();});
15
16    //Change parameters
17    mm.at("SCF")->change_parameter("Energy Conv", 1.0E-6);
18
19    mm.at("SCF")->change_submodule("F Builder",
20                            [](){return std::make_shared<scf::DFF>();});
21
22    //Run calculation
23    auto E = mm.run_as<LibChemist::Energy>("SCF", mol);
24
25    return 0;
26 }
27
```

```
1  import NWChemEx as NWX
2  from MyFBuilder import myF
3
4  # Make system
5  mol = NWX.make_molecule("""
6  H 0.0 0.0 0.0
7  H 0.0 0.0 0.9
8  """)
9
10 # Get preloaded ModuleManager
11 mm = NWX.get_module_manager()
12
13 # Change parameter
14 my_scf = mm["SCF"]
15 my_scf.change_parameter("Energy Conv", 1.0E-6)
16
17 # Change submodule
18 my_scf.change_submodule("F Builder", MyFBuilder.myF)
19
20 # Run calculation
21 E = mm.run_as(NWX.Energy, mol)
```

Listing 1. Input to NWChemEx for specifying an SCF calculation. Left example uses C++, right uses Python.

SDE DESIGN

The goal of this section is to introduce and demonstrate the key aspects of the SDE by working through an implementation of the self-consistent field (SCF) method in NWChemEx. The SCF method serves as the first order approximation in EST. Almost all other methods in EST add improvements to a reference wavefunction that is computed with an SCF calculation. At a very high level, the SCF method can be viewed as a technique to compute the best initial guess for a molecular system's electron density **P**. **P** is found by minimizing the energy of the system with respect to **P**. Since the energy depends on **P**, this process must be done iteratively. The SCF procedure is summarized in Algorithm 1, where the inputs to the procedure are an initial guess at the electron density $\mathbf{P_0}$ and a core matrix **h** that does not change during the SCF procedure. Without going into too many details, an important part of the method is building **F**, the "Fock matrix," which can be done in multiple ways. We point this out because, from the perspective of the end-user, building the Fock matrix is encapsulated inside the SCF procedure. Thus throughout the following, we show how the end-user can influence the formation of **F** without modifying the SDE or NWChemEx source code.

Input Layer

Listing 1 shows an example NWChemEx input for running an SCF computation on the hydrogen molecule. For comparison purposes, we show both the C++ and Python versions. Lines 1 to 3 of the C++ example in Listing 1 illustrates a central tenet of NWChemEx, modularity. Three of the main components of the software stack-the SDE, LibChemist, and SCF-are implemented as independent, modular components, and are only brought together here at the highest level (these components are automatically imported when you import NWChemEx in Python). We have chosen to write these components in C++ because of the support for and efficiency of C++ on current HPC systems. The C++ base also leads to a higher level of API stability (C++ places more restrictions on an API than Python does, and so needing to convert a Python API to C++ could break existing implementations). Nevertheless, the Python bindings provide a convenient input layer and enable others to extend NWChemEx from Python if they so choose.

Algorithm 1. High-level overview of the SCF procedure

```
function SCF($\mathbf{P}_0$, $\mathbf{h}$)
    $E_{old} \leftarrow 0$
    while not converged do
        Form $\mathbf{F}(\mathbf{P}_0)$
        Use $\mathbf{F}$ to form $\mathbf{P}$
        $E_{SCF} = Tr[\mathrm{P}\,(\mathbf{h} + \mathbf{F})]$
        Check convergence
        $\mathbf{P_0} \leftarrow \mathbf{P}$
        $E_{old} \leftarrow E_{SCF}$
    end while
    return $E_{SCF}$
end function
```

After importing the necessary components, both examples create the molecular system. In keeping with typical OO practices, NWChemEx utilizes software abstractions for common EST concepts. The APIs and implementations for these abstractions form LibChemist. In contrast to most of the other OO EST packages, chemistry abstractions in NWChemEx are implemented on top of the framework and are not part of it. This decreases the coupling between the SDE and NWChemEx and facilitates the use of the SDE and LibChemist by other packages.

On lines 13 and 14 of Listing 1, the C++ code continues by declaring and filling an instance of the ModuleManager class. These lines begin to introduce the SDE and the module concept. Within the context of the SDE, every major aspect of an algorithm is encapsulated within entities termed modules. Modules are analogous to components in the context of the CCA and plugins in MPQC,[9] Pyscf,[12] and Psi4,[13] unlike MPQC's and Psi4's plugins, modules are much more fine-grained and permeate the entire call graph. Each module performs a specific task, e.g., computing an energy, calculating an initial density, or building **F**. The task a module performs is codified by a class called its "property type." For example, the Energy class is the property type for modules that compute energies. We have attempted to put minimal restrictions on each property type, only requiring that it derives from SDE::ModuleBase and that it defines a function run, which is the entry point into the module. From a design standpoint, this is similar to how Pyscf's plugins work, except that SDE modules also have the added benefit of type safety.

The ModuleManager is the software abstraction of the SDE framework. Loading a module into the ModuleManager means associating a module key with a callback function (an opaque function provided as input to another function) for creating the module. The module key is a unique string label (here "SCF" is used as the module key) that is associated with a particular instance of a module. The module key provides a mechanism for referring to a particular instance of a module. Generally speaking, each module contains a set of module keys for the "submodules" it will call. The resulting call graph is known to the ModuleManager and in turn, the ModuleManager is capable of manipulating the graph. Thus one can interject a particular module into the call graph simply by providing the ModuleManager with the module key for the parent module and the module key for the new submodule. This should be contrasted with Pyscf, which provides similar capabilities, but requires the plugin to be forwarded through the entire call graph. The decision to take a callback allows the SDE to remain agnostic to the details of the module's creation and facilitates dynamically loading modules (see line 18 of the Python example in Listing 1). The callback itself is stored in a type-erased manner so that the SDE remains decoupled from not only the exact type of the module but also from the module's property type. Admittedly, this design results in needing to register every possible module with the ModuleManager. While there is no way around this explicit registration, users are typically insulated from it by convenience functions; Line 11 of the Python example in Listing 1 shows the typical use of such a function.

Like plugins in MPQC and Psi4, modules within the SDE have state. In particular, we associate a set of parameters, submodules, and metadata with each module instance. Allowing modules to have state greatly facilitates manipulating deeply nested modules. For example, assume module A calls module B, which calls module C. If we want to change the parameters of module C with the SDE we simply ask the ModuleManager for C and modify the parameters of the resulting instance. For stateless plugins, like those In Pyscf, however, the parameters for module C need to be forwarded through modules A and B in order for C to be able to use them. In the input examples in Listing 1, lines 17–20 of the C++ example and lines 15–18 of the Python example show how to change the parameters. Lines 19 and 20 of the C++ example and line 18 of the Python showcase another way by which it is possible to change the submodule, i.e., by providing a new callback. After swapping the submodule both examples4 are configured and the last statement calls the SCF module to obtain the energy of the hydrogen molecule.

Module Design

Moving deeper into the details of the SCF example, Listing 2 shows how the Energy property

```cpp
1  #include <SDE/SDE.hpp>
2  #include <LibChemist/LibChemist.hpp>
3  #include <Integrals/Libint.hpp>
4  #include <SCF/Fock.hpp>
5
6  struct Energy : SDE::ModuleBase {
7
8  //Defines the Energy property type API
9  virtual Tensor run(const LibChemist::Molecule& mol,
10              std::size_t deriv = 0) = 0;
11 };
12
13 struct SCF : Energy {
14
15 SCF() {
16    add_submodule<scf::Fock>("F Builder");
17    //...register rest of submodules
18
19    add_parameter<double>("Energy Conv");
20    //...register rest of parameters
21
22    add_metadata<MetaProperty::version>("1.0.0");
23    //...set rest of metadata
24 }
25
26 Tensor run(const LibChemist::Molecule& mol,
27         std::size_t deriv=0) override {
28
29    //Get the atomic orbital basis set
30    auto basis_name = parameters().at<std::string>("basis");
31    auto bs       = LibChemist::apply_basis(basis_name, mol);
32
33    //Get h matrix and initial density
34    auto h  = run_submodule_as<CoreHamiltonian>("h Builder", mol, bs);
35    auto P0 = run_submodule_as<GuessDensity>("Initial Density", mol, bs);
36
37    Tensor Eold = 0;
38    bool converged = false;
39    do {
40       auto F = run_submodule_as<FockBuilder>("F Builder", mol, bs, h, P0);
41       auto P = run_submodule_as<UpdateDensity>("P Builder", mol, F);
42
43       auto E = P("mu", "nu")*(h("mu", "nu") + F("mu", "nu"));
44       converged = check_convergence(E, Eold, P, P0, F);
45
46       P0 = P;
47       Eold = E;
48    } while(!converged);
49    return Eold;
50 }
51 };
```

```python
1  import NWChemEx as NWX
2
3  class SCF(NWX.Energy):
4     def __init__(self):
5        self.add_submodule(NWX.Fock, "F Builder")
6        #...register rest of submodules
7
8        self.add_parameter("Energy Conv")
9        #...register rest of parameters
10
11       self.add_metadata(NWX.MetaProperty.version, "1.0.0")
12       #...add rest of metadata
13
14    def run(self, mol, deriv=0):
15
16        # Get atomic orbital basis set
17       basis_name = self.parameters().at("basis")
18       bs        = NWX.apply_basis(basis_name, mol)
19
20       # Get h matrix and initial density
21       h  = self.run_submodule_as(NWX.CoreHamiltonian, "h Builder", mol, bs)
22       P0 = self.run_submodule_as(NWX.GuessDensity, "Initial Density", mol, bs)
23
24       Eold      = NWX.Tensor()
25       converged = False
26
27       while not converged:
28          F = self.run_submodule_as(NWX.FockBuilder, "F Builder", mol, bs, h, P0)
29          P = self.run_submodule_as(NWX.UpdateDensity, "P Builder", mol, F)
30
31          E = P("mu", "nu")*(h("mu", "nu") + F("mu", "nu"))
32          converged = check_convergence(E, Eold, P, P0, F)
33
34          P0 = P
35          Eold = E
36       return Eold
```

Listing 2. Definition of the SCF module. Left example uses C++, right uses PYTHON.

type, and the SCF module, which is derived from it, can be defined in NWChemEx. If a user wanted to extend the SDE with a new property type or module it is done via an analogous pattern. The code begins in lines 6–11 of the C++ example by defining the Energy property type, which is the central property type for an SCF calculation. The Energy property type only takes two inputs: the molecular system (described by the LibChemist:: Molecule class) and the derivative order (i.e., the default, 0, simply computes the energy, a value of 1 would return the gradient with respect to displacements of the nuclear framework, etc.). One might have expected to see more than two inputs, since the SCF method requires input, such as the **h** matrix shown in Algorithm 1. However, the design philosophy for the SDE is to make the API applicable to as many algorithms as possible. To do this, we have distilled the set of possible input parameters down to those common to most property type implementations. Any additional input required by a module is obtained by calling a submodule. The careful design of the API for each property type is a consequence of C++ being a statically typed language. For the various EST packages with APIs defined in PYTHON, less consideration is required because of PYTHON'S use of duck typing. While duck typing has the benefit of flexibility, that flexibility means that at runtime the modules are responsible for checking whether the input parameters are acceptable or not. Even something as simple as passing the arguments in the wrong order is not detected until runtime for PYTHON-based APIs. In EST, this could mean losing hours of work if the EST package does not use checkpointing or save progress in some manner. On the other hand, in C++-based APIs, which are used in the SDE, the result would be a compile time error, which avoids the runtime error described above.

The remainder of the C++ example and the entirety of the PYTHON example in Listing 2 show how one implements a module. In both cases, the new module, SCF, derives from the Energy property type. Since all modules of property type Energy share a common base they can be used interchangeably. For example, if one writes a geometry optimizer in terms of the Energy property type, that optimizer would immediately be usable with all modules derived from Energy. While this is a simple OO idea, many EST packages have features that are unavailable for certain methods simply because the developer was unaware they needed to modify the package

```cpp
#include <SDE/SDE.hpp>
#include <LibChemist/LibChemist.hpp>
#include <Integrals/Libint.hpp>
#include <SCF/Fock.hpp>

struct DIIS : FockBuilder {

Tensor run(const LibChemist::Molecule& mol,
        const LibChemist::AOBasisSet& bs,
        const Tensor& h,
        const Tensor& P0) {
  auto     F = run_submodule_as<FockBuilder>("F Builder", mol, bs, h, P0);
  auto error = compute_error(F, P0);
  auto  newF = extrapolate(error);
  return newF;
}
};

```

```python
import NWChemEx as NWX

class DIIS(NWX.FockBuilder):
   def run(self, mol, bs, h, p0):
      F = self.run_submodule_as(NWX.FockBuilder, "F Builder",
                        mol, bs, h, P0)
      error = compute_error(F, P0)
      newF  = extrapolate(error)
      return newF
```

Listing 3. Illustrating DIIS, using nested modules. Left example uses C++, right uses Python.

in additional places to enable that functionality. Property types help mitigate such situations because there are no special modifications required once the module is written.

In Listing 2, after the Energy property type definition, the SCF module is defined beginning at line 13 in the C++ example and line 3 in the Python example. In the default constructor for the class, the first line (and the lines elided by the subsequent comment) declares the submodule callback points and the default submodule to use at that point. Next, the code examples declare the parameters associated with the algorithm. Within the SDE, parameters are algorithmic details, such as scale factors and thresholds. To our knowledge, this is unique to the SDE as all other EST packages use parameters to switch between algorithms. While this may seem like a trivial distinction, it makes extending the code easier. For example, assume that during the SCF there are two different ways to build the Fock matrix, such as by using core memory and using the disk. Other EST packages then define a parameter, which is then utilized by a control structure, to switch between Fock build algorithms. In most cases, if another Fock build is added, another parameter is needed, and the control structure needs to be altered. Consequently, the API of the SCF becomes coupled to the choices for the Fock builder. Depending on how the dispatch is handled, the resulting code may obscure the control flow and inhibit parallelization. Within the SDE, however, this coupling is avoided by simply switching the submodule that the SCF uses. The constructor declaration ends with the specification of metadata associated with the module. The metadata are items such as the version of the module, a description of what it does, and citations associated with it. Although not present in the SDE at the moment, one of our goals is to automate the generation of documentation for the module. This would be done by inspecting the module to obtain the parameters associated with it (and their default values) as well as the available callback points (and the default modules used). The other purpose of the metadata is to allow the caller of a module to figure out exactly what module they are calling.

The remainder of the code example in Listing 2 implements the run function of the SCF module. Lines 29 to 35 in the C++ example and lines 17 to 22 of the Python example obtain the remainder of the input, not provided by the Energy property type, but required for the SCF. In the iterative portion of the SCF procedure (line 39 in the C++ example and 27 in the Python example in Listing 2, and line 3 of Algorithm 1), submodules are invoked to calculate needed quantities. On line 40 in the C++ example and line 28 in the Python example, we compute the Fock matrix by calling a module of property type FockBuilder. Note that there is no control logic to select which module to call; the SDE will automatically make sure that the module resolves to the choice made in Listing 1 (the DFF implementation in C++ and the myF implementation in Python).

Readers familiar with the SCF procedure know that a production level SCF requires many additional considerations aside from those shown in Algorithm 1. For example, it is common to use numerical techniques to accelerate the convergence of the SCF process. The most popular, direct inversion of the iterative subspace (DIIS), uses the generated **F** matrix and extrapolates it towards convergence based on the errors of the previous iterations. Typically such a step would be coded explicitly between lines 40 and 41 of the C++ example in Listing 2. However, because of the module design in the SDE, we can add DIIS to our SCF without modifying Listing 2. To do this we nest FockBuilder instances. Listing 3 shows an implementation of DIIS, using nested modules, omitting the details of the actual DIIS algorithm for clarity. In Listing 3, on line 12 of the C++ implementation and line 5 of the PYTHON implementation, the DIIS module calls another FockBuilder to get the **F** matrix, computes the error in that **F** matrix, and uses the error to produce a better guess, which it then returns. To the SCF module, the use of DIIS is opaque. In fact by nesting many FockBuilder modules it is possible to create an arbitrarily complicated FockBuilder; for example, one could implement a FockBuilder so that one submodule is used when the error is above a certain threshold, and a different submodule is used for smaller errors.

As the DIIS example shows, writing a nested module is straightforward and users can do it from the input. This allows users to have fine-grained control over printing (for example write a FockBuilder that prints the **F** matrix and then returns it), introduce *ad hoc* algorithms that lack generality (e.g., on iteration 4 follow root 3), or to implement a completely new algorithm. Nesting modules is also useful for exploiting parallelism. For example, one can maintain a single process, multiple data model by having an outer FockBuilder, parallelize over calls to a series of smaller FockBuilder calls. Eventually, it is our goal to have modules that automate the selection of the algorithm given the characteristics of the computer system. Nesting modules achieves this by allowing the top-level module to contain the automation logic necessary to select the appropriate module to call, in an opaque manner.

SDE Layer

The reader may have noticed that in the Input Layer section, we did not invoke the SCF module directly, rather we used the run_as function (see Listing 1). Similarly, while writing the SCF example module we did not directly call the run member of the submodule, instead, we used the run_submodule_as function (see Listing 2). Algorithm 2 shows the internals of the run_as function. (The run_submodule_as function is a thin wrapper around run_as, so the behavior is the same as the run_as function.)

It is evident that the run_as function is not simply a thin wrapper around the call to run. Rather, run_as gives control to the SDE to enable automation of a number of common tasks. At the moment the SDE is capable of automating error checking, as well as saving/loading of the calculation's state. Efforts to automate performance optimizations (e.g., accelerator offload, parallelization, and intelligent algorithm selection) are ongoing.

Algorithm 2. High-level overview of the run_as function

```
function run_as(args...)
    if not_ready()
        throw "Not ready"
    end if
    lock()
    hv ← memoize(args...)
    if hv not in cache
        value ← run(args...)
        cache[hv] = value
    end if
    return cache.at(hv)
end function
```

Nearly every EST package has some sort of option/parameter system. Inevitably, some of the option values must be subject to constraints. For example, specifying a negative number of iterations is meaningless. Consequently, in most EST packages the first several lines of code in a routine are dedicated to error checking the options. The SDE automates error checking of options by allowing the module developer to provide callback functions for checking an option's value (although the use of such callbacks is not shown in Listing 2 for simplicity). Ensuring that these callbacks pass for each option is one of two things the not_ready function does. The other purpose of the not_ready function is to ensure that each required submodule callback is set and ready, i.e., not_ready is called recursively on each submodule as well. If the module or any of the submodules are not ready, then the call throws.

Once the SDE verifies that the module is ready to be run it locks the module and all submodules. A locked module can no longer have its options or submodules modified (the lock occurs after the check to provide the user a chance to rectify any of the errors preventing the module from being run). In designing for HPC environments one has to be mindful of data races. Locking is one safeguard for this. Locking is also essential for the next step of the run_as function, memoization. In computer science, memoization is a technique where the result of an expensive call is stored and the next time that call is performed, the cached value is simply returned. The cached values are uniquely indexed by the input parameters provided during the function execution. By locking the state of the module, we guarantee that the output of the module is solely determined by the input (assuming the module is deterministic).

Within Algorithm 2, the actual memoization occurs by first obtaining a hash representing the state of the module and the arguments. Next, we check to see if the hash is present in the cache (the cache is essentially a hash table). If the value is present we simply return it. If the value is not in the cache we call the module, add the value to the cache, and then return the cached value. Given that all of the important results computed in an EST program are the result of calling a module, the cache will contain all of the important results obtained so far. Hence, storage of the cache suffices as a record of the calculation. Furthermore, if the same calculation is run again, with the cache from the previous run, the entire calculation will be memoized. This provides a straightforward mechanism to checkpoint the calculation by periodically backing the cache up. As an added bonus, this mechanism is handled automatically by the SDE so that module developers do not have to manually implement restart logic.

Memoization is also important for avoiding global states. Global states, including files, are undesirable from a parallel perspective as they make control flow hard to reason about. In many legacy codes global states are used to get around the rigidity of the APIs. The SDE's reliance on property types poses the same problem. For example, in the SCF method the Fock matrix is a linear combination of three matrices: **h**, **J**, and **K**. The FockBuilder property type takes an instance of **h**, but does not take **J** or **K**. It turns out that the most computationally expensive step of the SCF is building **J** and **K**. Hence, once we have built **J** and **K**, we do not want to rebuild them unless absolutely necessary. This is where the rigidity of the FockBuilder property type is prohibitive. Since instances of FockBuilder do not return **J** and **K,** without memoization and without global states, **J** and **K** would need to be recomputed. With memoization, however, we can obtain **J** and **K** with minimal overhead, simply by recalling the submodule that built **J** and **K** within the FockBuilder.

SUMMARY AND OUTLOOK

The SDE provides a mechanism for interoperability and flexibility in package composition. The various components of the SDE allow for multiple modules and submodules that can provide new methodological approaches to calculating similar properties as well as allowing for optimization of performance based on hardware and software architectures. The transparency to both users and developers of the Parameters and Cache classes provides a spectrum of ways to interact with the modules ranging from beginner, where all of the defaults are set, to expert, where many of the options can be changed and used in unique manners to quickly build and prototype new functionality. While the SDE is flexible, the APIs for each property type are not, once they have been defined. However, this is a strength when coupled with community-driven standards development such as those of the MolSSI. Future development efforts will focus on expanding the property types, modules, and submodules that are available through the framework. Particular areas of interest include APIs for the calculation of atomic integrals, localized orbitals and higher level correlated methods with sparse representations.

ACKNOWLEDGMENTS

This work was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. Argonne staff support was provided by the ALCF Aurora Early Science Program. This work used resources of the Argonne Leadership Computing Facility, which is a DOE

Office of Science User Facility supported under Contract DE-AC02-06CH11357. The work of B. Pritchard was supported by the MolSSI under NSF grant ACI-1547580.

REFERENCES

1. M. Valiev *et al.*, "NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations," *Comp. Phys. Commun.*, vol. 181, no. 9, pp. 1477–1489, 2010.
2. D. E. Bernholdt *et al.*, "A component architecture for high-performance scientific computing," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 163–202, 2006.
3. A. Cleary, S. Kohn, S. G. Smith, and B. Smolinski, "Language interoperability mechanisms for high-performance scientific applications," *SIAM Workshop Object-Oriented Methods Interoperable Scientific Eng. Comput.*, Yorktown Heights, NY, vol. 9, Oct. 1998.
4. J. P. Kenny *et al.*, "Component-based integration of chemistry and optimization software," *J. Comp. Chem.*, vol. 25, no. 14, pp. 1717–1725, 2004.
5. T. P. Gulabani *et al.*, "Development of high performance scientific components for interoperability of computing packages," in *Proc. Spring Simul. Multiconf.*, 2009, pp. 111:1–111:6.
6. M. Krishnan, Y. Alexeev, T. L. Windus, and J. Nieplocha, "Multi-level parallelism in computational chemistry using common component architecture and global arrays," in *Proc. ACM/IEEE SC Conf. Supercomputing*, Nov. 2005, pp. 23–23.
7. J. P. Kenny, C. L. Janssen, E. F. Valeev, and T. L. Windus, "Components for integral evaluation in quantum chemistry," *J. Comp. Chem.*, vol. 29, no. 4, pp. 562–577, 2008.
8. C. L. Janssen *et al.*, "Enabling new capabilities and insights from quantum chemistry by using component architectures," in *Proc. J. Phys. Conf. Ser.*, 2006, vol. 46, no. 1, p. 220.
9. C. Janssen, E. Seidl, and M. Colvin, "Object-oriented implementation of parallel ab initio programs," in *Proc. ACS Symp. Series, Parallel Comput. Comput. Chem.*, 1995, vol. 592, pp. 47–61.
10. S. Toru, "BAGEL: Brilliantly advanced general electronic structure library," *WIRES: Comp. Mol. Sci.*, vol. 8, no. 1, p. e1331.
11. R. Muller, "PyQuante-Python quantum chemistry," (2009). [Online]. Available: http://pyquante.sourceforge.net/
12. Q. Sun *et al.*, "PySCF: The Python-based simulations of chemistry framework," *WIRES: Comp. Mol. Sci.*, vol. 8, no. 1, p. e1340, 2017.
13. R. M. Parrish *et al.*, "Psi4 1.1: An open-source electronic structure program emphasizing automation, advanced libraries, and interoperability," *J. Chem. Theory Comp.*, vol. 13, no. 7, pp. 3185–3197, 2017.
14. T. Verstraelen *et al.*, "Horton 2.1.0," 2017. [Online]. Available at: http://theochem.github.com/horton/
15. L. A. Burns *et al.*, "A schema for quantum chemistry," 2018, [Online]. Available at: https://github.com/MolSSI/QCSchema
16. V. W. zhe Yu *et al.*, "Elsi: A unified software interface for Kohn-Sham electronic structure solvers," *Comput. Phys. Commun.*, vol. 222, pp. 267–285, 2018.

Ryan M. Richard is an Associate Scientist with Ames Laboratory, Ames, IA, USA. His research interests include scientific software design, high-performance computing, and electronic structure theory method development.

Colleen Bertoni is an Assistant Computational Scientist with Argonne National Laboratory, Lemont, IL, USA. Her research interests include high-performance computing, electronic structure theory, and programming models.

Jeffery S. Boschen is a Postdoctoral Research Associate with Ames Laboratory, Ames, IA, USA. His research interests include high-performance computing, and electronic structure theory.

Kristopher Keipert is an Assistant Computational Scientist with Argonne National Laboratory, Lemont, IL, USA. His research interests include high-performance computing, lossy compression and reduced precision techniques for scientific computing, analytical performance modeling, and electronic structure theory.

Benjamin Pritchard is a MolSSI Software Scientist. His research interests include computational chemistry frameworks and development of high-performance electron repulsion integrals.

Edward F. Valeev is a Professor of Chemistry with Virginia Tech, Blacksburg, VA, USA. His research interests include the electronic structure of molecules and materials.

Robert J. Harrison is the Director with the Computational Science Center, Brookhaven

National Laboratory, Upton, NY, USA, and a Professor of applied mathematics and statistics with Stony Brook University, Stony Brook, NY, USA. His research interests include high-performance computing and nontraditional electronic structure theory methods.

Wibe A. de Jong is a Senior Scientist with Lawrence Berkeley National Laboratory, Berkeley, CA, USA. His research interests include scientific software development for parallel HPC platforms within the context of computational chemistry methods.

Theresa L. Windus is a Distinguished Professor of Chemistry with Iowa State University, Ames, IA, USA, and a joint employee with Ames Laboratory, Ames, IA, USA. Her research interests include high performance computing, software interoperability and engineering, ligand design, and reaction mechanisms.