International Conference on Mathematics, Computational Methods & Reactor Physics (M&C 2009)
Saratoga Springs, New York, May 3-7, 2009, on CD-ROM, American Nuclear Society, LaGrange Park, IL (2009)

SOFTWARE ENGINEERING IN THE SCEPTRE CODE

Shawn Pautz,[*] Clif Drumm, Bill Bohnhoff, and Wesley Fan
Sandia National Laboratories[†]
Albuquerque, NM 87185-1179
sdpautz@sandia.gov; crdrumm@sandia.gov; wjbohnh@sandia.gov; wcfan@sandia.gov

ABSTRACT

The SCEPTRE radiation transport code suite is a relatively young development effort motivated by the need for coupled photon-electron transport analyses at Sandia National Laboratories. It contains multiple discretizations of both the first-and second-order forms of the Boltzmann transport equations. Although any of these capabilities may be used separately, they also may be used in a complementary manner in the same analysis for computational efficiency.

Reliable development and use of these interoperable capabilities is a complex task. Therefore the SCEPTRE project relies on various software engineering methods to manage this complexity. These methods include a modular, levelized design, generic programming, and object-oriented programming. We describe several specific examples of the application of these techniques, particularly those that enable hybrid capabilities. The result is a reliable and flexible code base that can be adapted to multiple needs.

Key Words: hybrid methods, levelized design, generic programming, software engineering

1. INTRODUCTION

The SCEPTRE radiation transport code suite is a new generation of the CEPTRE code. CEPTRE (Coupled Electron-Photon Transport for Radiation Effects) contains several discretizations of the second-order form of the Boltzmann transport equation and is implemented within the Nevada framework [1]. SCEPTRE (Sandia's Computational Engine for Particle Transport for Radiation Effects) contains discretizations of both the first- and second-order forms of the Boltzmann transport equation and is implemented within the "radlib" radiation transport code library. Both massively parallel code suites use the multigroup energy discretization, discrete ordinates angular discretization, and finite element spatial discretization on unstructured meshes.

The first- and second-order discretizations in SCEPTRE have complementary strengths and weaknesses. In a companion paper [2] we discuss the numerical properties of these discretizations and give performance results. The purpose of this paper is to discuss some of the software engineering techniques we have used to construct the SCEPTRE code suite. These

---

[*] To whom correspondence should be addressed

[†] Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94AL85000.

techniques yield benefits in terms of testability, adaptability, and maintainability. They have also enabled us to create hybrid solvers based on both discretization approaches.

The rest of the paper is organized as follows. A description of SCEPTRE's main design principles is given in Section 2. The decomposition of transport sweeps and other sweep-like processes into separate capabilities is given in Section 3. In Section 4 we show how we can combine separate solvers into a hybrid multigroup capability. We give some final discussion and conclusions in Section 5.

2. GENERAL DESIGN PRINCIPLES

SCEPTRE consists of several driver codes built upon numerous code packages in radlib. The structure of radlib is depicted in Figure 1. We use a modular design, in which closely related code is grouped together in physically and logically distinct packages; each code package in radlib is represented by a labeled box in Figure 1. We also use a levelized design [3], in which dependencies among packages are carefully analyzed and managed; the graph in Figure 1 depicts these dependencies. The goal is to keep low-level code components independent of higher-level "client" code, and in particular to avoid cyclic dependencies.

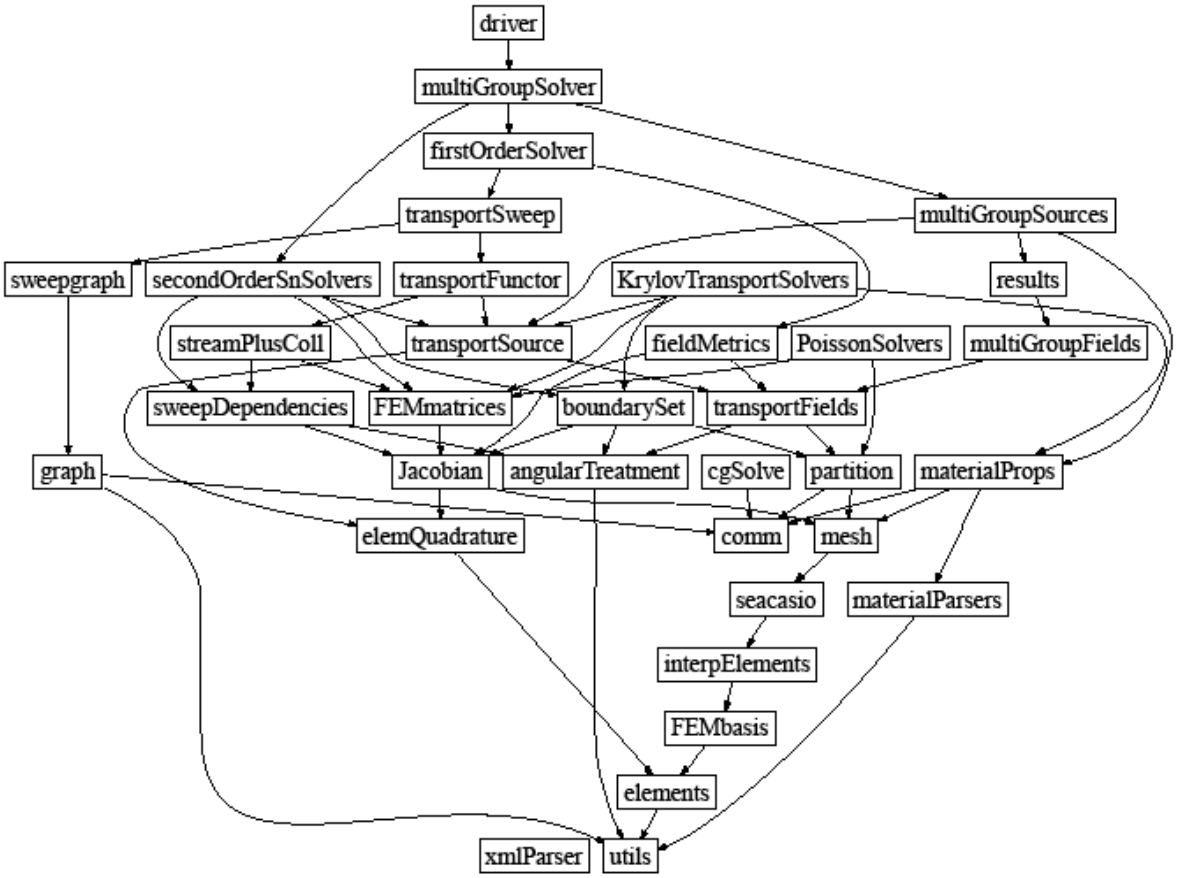

Figure 1: Levelization diagram of radlib.

The above approach yields multiple benefits. Deliberate limitation of the number of dependencies, especially cyclic dependencies, to the logically necessary ones greatly reduces the overall code complexity, even if the number of packages or lines of code remains unchanged. A levelized design aids development and analysis, since the developer may concentrate on the relevant package while ignoring the details of other packages. It promotes code reuse, since general low-level components may be used in different contexts by multiple higher-level packages. It also greatly facilitates testing and debugging, since each package contains unit tests of its functionality. Failures in a unit test of some package cannot possibly be attributed to higher-level packages, while successful testing of lower-level packages generally excludes them as the source of error; coding errors are thus usually rapidly isolated to the package associated with the unit test.

SCEPTRE is implemented in C++, which offers features that help us to decompose the code into packages with distinct responsibilities. The use of templates (a.k.a. static polymorphism or generic programming) allows us to write generic components that can be adapted to multiple uses in different contexts. The use of class inheritance (a.k.a. dynamic polymorphism or object-oriented programming) allows either the code or the user to switch among multiple approaches or algorithms at runtime. Both techniques convert many logical ("if") statements otherwise written and maintained by a developer into forms managed by the compiler, thus reducing or eliminating a notorious source of code errors.

3. TRANSPORT SWEEPS

The solution algorithm for the first-order discretizations in SCEPTRE makes use of transport sweeps. In a transport sweep the streaming-plus-collision operator is approximately inverted in each spatial element for each discrete angle, yielding an internal flux distribution and outgoing fluxes. Because of dependencies among these tasks it is necessary to execute them in a valid order and to communicate intermediate results to other processors as needed.

We have recognized that the above process involves distinct responsibilities and have decomposed them accordingly. One major division of responsibilities that we have identified is that the parallel scheduling of tasks with dependencies is a separate problem from the setup and execution of the tasks themselves. Consequently we have developed the distributed_sweep algorithm within the "sweepgraph" package; its interface is shown in Figure 2(a). The purpose of this algorithm is to traverse a distributed task graph in an order that respects task dependencies and to execute each task accordingly. Here we have generalized the notion of a parallel sweep to include any execution of distributed tasks that respects dependencies; it is not restricted to first-order transport sweeps.

```
template <class Graph, class SweepVisitor>
void distributed_sweep(const Graph& graph, SweepVisitor sweepVis, ...)
{
   // visit every vertex in graph in parallel while respecting dependencies,
   // calling sweepVis member functions at each step
}
```

Figure 2(a): Interface of distributed_sweep algorithm.

```
Vertex source(Edge e, const Graph& g);
Vertex target(Edge e, const Graph& g);
pair<EdgeIterator, EdgeIterator> out_edges(Vertex v, const Graph& g);
bool get(boost::edge_removed, const Graph& g, Edge e);
```

Figure 2(b): Partial listing of required Graph functionality.

```
void SweepVisitor::examine_local_vertex(Vertex v, const Graph& g);
void SweepVisitor::examine_local_edge(Edge e, const Graph& g);
void SweepVisitor::examine_outgoing_edge(Edge e, const Graph& g);
```

Figure 2(c): Partial listing of required SweepVisitor functionality.

The distributed_sweep algorithm encodes the separation of responsibilities described above through the use of two separate objects. The relationships among tasks are encoded in a Graph object, in which vertices represent individual tasks and directed edges represent dependencies. The tasks themselves are contained within a SweepVisitor object, whose member functions are called by distributed_sweep as the graph is traversed in order to execute the tasks and perform necessary communication. Neither of these classes have any direct knowledge of each other, so they may be developed and maintained independently.

Note that the algorithm is a template function. It does not depend on any particular Graph or SweepVisitor classes. One may use any classes that provide the necessary functionality and the algorithm will adapt itself to the particular situation. In Figure 2(b) we give a partial listing of the requirements on a valid Graph class (also known as the Graph "concept"); it includes obtaining the source and target vertex identifiers of an edge and inquiring whether an edge has been removed from the graph. Similarly Figure 2(c) gives a partial listing of the SweepVisitor concept; distributed_sweep will execute its member functions at the appropriate time to perform the associated tasks. The distributed_sweep algorithm does not need to know the implementation details of these functions and can instead restrict its attention to conducting a valid and efficient execution schedule.

```
class LevelSweepVisitor
{
   std::vector<unsigned>& vertexLevels;

  public:
   template <class Edge, class Graph>
   void examine_local_edge(Edge e, const Graph& g)
   {
      const Vertex u = source(e, g);
      const Vertex v = target(e, g);
      vertexLevels[v] = std::max(vertexLevels[v], vertexLevels[u] + 1);
   }
};
```

Figure 3(a): Partial definition of class LevelSweepVisitor.

```
GraphType graph;
// fill graph
.
.
std::vector<unsigned> vertexLevels(num_vertices(graph), 0);
LevelSweepVisitor levelSweepVisitor(vertexLevels);
distributed_sweep(graph, levelSweepVisitor, ...);
```

Figure 3(b): Use of distributed_sweep to compute graph levels.

As one concrete example of the use of the distributed_sweep algorithm we illustrate the calculation of vertex levels in a graph. The level of a vertex is the longest directed path to that vertex from the entry (top) vertices of the graph; this information is used by some scheduling heuristics to determine an efficient execution order. For this problem we define class LevelSweepVisitor; a partial listing is given in Figure 3(a). Its use with the distributed_sweep algorithm is shown in Figure 3(b). As distributed_sweep traverses the graph, it calls on various member functions of LevelSweepVisitor. In particular the examine_local_edge member function sets the level of a vertex to one more than the maximum of any of its parent vertices. The decomposition of responsibilities discussed earlier allows the definition of class LevelSweepVisitor to be rather short and it also allows us to reuse distributed_sweep for any sweep-like process.

```
template <class TransportFunctor, class SweepDirector>
void transportSweep(TransportFunctor transportFunctor,
                    const SweepDirector& sweepDirector, ...)
{
   TransportSweepVisitor<TransportFunctor, SweepDirector>
     transportSweepVisitor(transportFunctor, sweepDirector);

   distributed_sweep(sweepDirector.getSweepGraph(), transportSweepVisitor);
}
```

Figure 4(a): Complete transportSweep algorithm.

```
const Graph& SweepDirector::getSweepGraph();
```

Figure 4(b): Partial listing of required SweepDirector functionality.

```
void TransportFunctor::operator()(size_type element, size_type angle);
void TransportFunctor::send(int processor);
void TransportFunctor::recv(int processor);
```

Figure 4(c): Required TransportFunctor functionality.

In order to perform a transport sweep we do not directly call distributed_sweep; instead we use the transportSweep algorithm (defined in the "transportSweep" package) depicted in Figure 4(a). We do this in order to further decompose the responsibilities encapsulated in SweepVisitor. Recall that SweepVisitor classes are responsible for executing a task associated with a particular Graph vertex. We break this responsibility into two components. The encoding of elementary transport concepts such as elements and angles into a graph and the subsequent decoding back is handled by a SweepDirector object; some of its requirements are listed in Figure 4(b). The streaming-plus-collision inversion task associated with a particular element and angle is handled by a TransportFunctor object; the requirements on this object are listed in Figure 4(c). These two objects are conjoined by the transportSweep algorithm to form a TransportSweepVisitor object, which is a "model" of (fulfills the requirements of) a SweepVisitor class. Thus this short algorithm allows us to keep TransportFunctor objects completely ignorant of (and thus independent of) graph concepts and implementations, while keeping SweepDirector objects ignorant of any transport quantities other than meshes and angular quadrature sets.

In SCEPTRE we typically use a TransportFunctor object defined for finite element discretizations within the "transportFunctor" package. However, it is relatively easy to substitute other TransportFunctor classes as needed. For example, some of our tests of the transportSweep algorithm make use of a very simple TransportFunctor class in order to isolate them from the

complexities of finite element discretizations. The strength of this approach was also recently demonstrated as we performed research into a long characteristics discretization; a new "charTransportFunctor" package and the existing sweepgraph package were bound to the existing transportSweep package, thus enabling an entirely new first-order discretization with relatively low effort.

4. MULTIGROUP SOLVER

As a final illustration of the techniques used in SCEPTRE we discuss how we are able to perform hybrid calculations with both the first- and second-order discretizations. The primary application of SCEPTRE currently involves coupled photon-electron transport in electronic components. The physical properties of these problems are such that the first-order approach is very efficient for the photon groups but inefficient for the electron groups; the reverse is true for the second-order form. To obtain the greatest computational efficiency we desire to create a hybrid capability that applies one or the other discretization as desired to individual groups.

To accomplish the above goal we first create a common abstraction for the idea of a "within-group transport solver". We do this by defining the abstract "functor" class WithinGroupSolver, as shown in Figure 5(a). It specifies the interface for a single member function that accepts as input the angular fluxes, fixed sources, and cross sections that uniquely define a monoenergetic transport problem within a given code run; this function is assumed to modify the angular fluxes as part of its solution process. Note that the input data types are all abstract (i.e. template parameters), which allows us to adapt this class to multiple situations. We may, for example, use one implementation of fixed sources to obtain user-defined sources in one calculation and an entirely different implementation to obtain downscattering sources in another calculation, all without requiring any modifications to WithinGroupSolver.

The WithinGroupSolver abstraction is used in a concrete manner by both the first- and second-order forms through inheritance. For example, as shown in Figure 5(b), the FirstOrderSolver class inherits the WithinGroupSolver interface, translating the general function call into a call to the firstOrderSolve algorithm, which has a specialized interface. A similar technique is used by the second-order methods.

Finally, the first- and second-order forms are used interchangeably by the multiGroupSolve algorithm, listed in Figure 6(a). This algorithm takes as input multigroup data that defines a multigroup problem (we have suppressed some of the input parameters here for clarity). One of the input parameters is a vector of pointers to WithinGroupSolver objects, one for each group (we use the term "group" in a general manner to refer to both the particle energy and the particle species). Some of these objects may be FirstOrderSolver objects, some may be SecondOrderSolver objects, and others may be solver objects not discussed here; the precise distribution is determined by the caller of multiGroupSolve. The multiGroupSolve algorithm handles the details of setting up each within-group transport problem and then calls the group-specific solver, all without knowing the details of the solver. Thus we can arbitrarily determine the appropriate combination of solvers, and even introduce new solvers in the future, without needing to modify or specialize the multiGroupSolve algorithm.

```
template <class AngularFluxField, class FixedSourceFunctor, class XSFunctor>
class WithinGroupSolver
{
   virtual void operator()
      (AngularFluxField& psi, FixedSourceFunctor fixedSourceFunctor,
       XSFunctor xsFunctor) = 0;
};
```

Figure 5(a): Abstract class WithinGroupSolver.

```
template <class AngularFluxField, class FixedSourceFunctor, class XSFunctor>
class FirstOrderSolver : public WithinGroupSolver
{
   size_t maxIters;
   double tolerance;

   virtual void operator()
      (AngularFluxField& psi, FixedSourceFunctor fixedSourceFunctor,
       XSFunctor xsFunctor)
   {
      firstOrderSolve(psi, fixedSourceFunctor, xsFunctor, maxIters, tolerance);
   }
};
```

Figure 5(b): Concrete class FirstOrderSolver inherits from WithinGroupSolver.

As a concrete example of the use of multiGroupSolve we depict the hybrid application of the first- and second-order solvers in Figure 6(b). Here we assume that the first half of the energy groups are photon groups, for which the first-order solver is more efficient. The other half are electron groups, which we assume are better suited to the second-order solver. We therefore create one FirstOrderSolver object and one SecondOrderSolver object. We then create a vector of WithinGroupSolver pointers; the first half of the entries point to the FirstOrderSolver object and the other half point to the SecondOrderSolver object. Alternatively, we could have relied on user input or some more sophisticated algorithm to determine the appropriate mixture of solvers. We finally call the multiGroupSolve algorithm, which will apply the desired solvers for each group as necessary until the problem has either converged or the maximum number of outer iterations has been exceeded.

```
template <class MultiGroupAngularFluxField, class MultiGroupFixedSourceFunctor,
          class MultiGroupXSFunctor>
std::pair<bool, double>
multiGroupSolve(MultiGroupAngularFluxField& mgPsi,
                MultiGroupFixedSourceFunctor mgFixedSourceFunctor,
                MultiGroupXSFunctor mgXSFunctor,
                std::vector<WithinGroupSolver*> withinGroupSolvers,
                size_t maxIters, double tolerance)
{
   .
   .
   do
   {
      .
      .
      for (size_t group = 0; group < numGroups; ++group)
      {
         // form withinGroupFixedSourceFunctor
         (*withinGroupSolvers[group])(mgPsi(group), withinGroupFixedSourceFunctor,
                                      mgXSFunctor(group, group));
      }
     // determine errorEstimate and converged
   }
   while (iter < maxIters && !converged);
   return std::make_pair(converged, errorEstimate);
}
```

Figure 6(a): Partial listing of multiGroupSolve algorithm.

```
EnergyGroups energyGroups(...);
// form mgPsi, mgFixedSourceFunctor, and mgXSFunctor
.
.
FirstOrderSolver firstOrderSolver(...);
SecondOrderSolver secondOrderSolver(...);
std::vector<WithinGroupSolver*> withinGroupSolvers(energyGroups.numGroups());
for (size_t group = 0; group < energyGroups.numGroups()/2; ++group)
   withinGroupSolvers[group] = &firstOrderSolver;
for (size_t group = energyGroups.numGroups()/2;
     group < energyGroups.numGroups(); ++group)
   withinGroupSolvers[group] = &secondOrderSolver;

multiGroupSolve(mgPsi, mgFixedSourceFunctor, mgXSFunctor,
                withinGroupSolvers,10, 0.001);
```

Figure 6(b): Example use of multiGroupSolve.

5. CONCLUSIONS

In summary, a combination of general software engineering techniques and C++-specific constructs allows us to design and implement software components in SCEPTRE that are testable, maintainable, and adaptable. Careful packaging and attention to levelization techniques help manage the complexity of the code. The use of templates allows us to define components that can be reused in multiple contexts. The use of inheritance allows us to choose among multiple alternatives at runtime. In particular, these techniques have allowed us to create a hybrid solver capability that can freely mix different discretization approaches while maintaining a common multigroup implementation.

REFERENCES

1. C.R. Drumm, J. Liscum-Powell, W.J. Bohnhoff, W.C. Fan, S.D. Pautz, and L.J. Lorence, "Coupled Electron-Photon Transport with the CEPTRE Code," *Proceedings of Conference on Mathematics and Computation, Supercomputing, Reactor Physics and Nuclear and Biological Applications*, Avignon, France, September 12-15, 2005, on CD-ROM (2005).
2. S. Pautz, B. Bohnhoff, C. Drumm, and W. Fan, "Parallel Discrete Ordinates Methods in the SCEPTRE Project," *Proceedings of International Conference on Mathematics, Computational Methods and Reactor Physics*, Saratoga Springs, New York, May 3-7, 2009 (submitted).
3. J. Lakos, *Large-Scale C++ Software Design*, Addison-Wesley, Reading, Massachusetts (1996).