Managing Application Complexity in the SAMRAI Object-Oriented Framework

Richard D. Hornung

Scott R. Kohn

Center for Applied Scientific Computing

Lawrence Livermore National Laboratory

Send correspondence to :
Richard D. Hornung
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
P.O. Box 808, L-561
Livermore, CA 94551
phone: (925) 422-5097
fax: (925) 423-2993
e-mail: hornung@llnl.gov

DISCLAIMER

Summary

A major challenge facing software libraries for scientific computing is the ability to provide adequate flexibility to meet sophisticated, diverse, and evolving application requirements. Object-oriented design techniques are valuable tools for capturing characteristics of complex applications in a software architecture. In this paper, we describe certain prominent object-oriented features of the SAMRAI software library that have proven to be useful in application development. SAMRAI is used in a variety of applications and has demonstrated a substantial amount of code and design re-use in those applications. This flexibility and extensibility is illustrated with three different application codes. We emphasize two important features of our design. First, we describe the composition of complex numerical algorithms from smaller components which are usable in different applications. Second, we discuss the extension of existing framework components to satisfy new application needs.

Keywords: object-oriented programming, design patterns, adaptive mesh refinement

Managing Application Complexity in the SAMRAI Object-Oriented Framework*

Richard D. Hornung† Scott R. Kohn†

Abstract

A major challenge facing software libraries for scientific computing is the ability to provide adequate flexibility to meet sophisticated, diverse, and evolving application requirements. Object-oriented design techniques are valuable tools for capturing characteristics of complex applications in a software architecture. In this paper, we describe certain prominent object-oriented features of the SAMRAI software library that have proven to be useful in application development. SAMRAI is used in a variety of applications and has demonstrated a substantial amount of code and design re-use in those applications. This flexibility and extensibility is illustrated with three different application codes. We emphasize two important features of our design. First, we describe the composition of complex numerical algorithms from smaller components

*This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract number W-7405-Eng-48. LLNL report UCRL-JC-141749.

†Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551, hornung@llnl.gov, skohn@llnl.gov.

which are usable in different applications. Second, we discuss the extension of existing framework components to satisfy new application needs.

1 Introduction

The design and implementation of a quality, high-performance numerical software framework must support the requirements of target problems. Providing algorithms and data structures that meet requirements that evolve during the lifetime of the software increases their usefulness and thus enhances the value of the software. In many software communities, libraries are commonly built to provide loosely-coupled components that are highly flexible, extensible and possess high re-use potential. However, scientific computing codes infrequently apply such practices. Reasons for this range from the emphasis on efficient performance to concerns about overburdening application scientists with abstract software engineering concepts. Large, monolithic, tightly-coupled codes that address a single application are common in scientific computing. Such codes tend to be inflexible. Reuse in general support libraries is usually restricted to basic structures such as vectors, arrays, and generic solvers. In our experience, directing object-oriented software decomposition toward higher-level algorithm and application development has proven to be very useful and productive.

In this paper, we present object-oriented design concepts applied in SAMRAI, a framework for parallel structured adaptive mesh refinement (SAMR) applications. Complicated numerical algorithms, such as those required for SAMR, can be decomposed into smaller parts that work together. Individual parts include time integration routines, mesh geometry descriptions, temporal and spatial data interpolation methods, and linear and nonlinear solvers. Years of experience in both SAMR library and application development are reflected in SAMRAI design. SAMRAI currently supports several diverse application development efforts. SAMRAI has been, and continues to be, developed in tandem with algorithm and application research efforts. It has shown to be flexible by evolving with the needs of computational scientists as they gain improved understanding of their applications and associated numerical methods.

Object-oriented techniques, such as design patterns [1], are ubiquitous in SAMRAI [2]. This approach has enabled SAMRAI to address new applications and exploit a significant amount of design and code re-use across applications. Our goal is to assemble applications from algorithmic "building blocks", or distinct functional components in the framework. We attempt to avoid imposing unnecessary restrictions on application development. However, we have observed that our software organization influences design decisions made by SAMRAI users during the development of application codes. For example, design patterns used in the framework to decompose elements of the software have been adopted by application developers using the library. This "design reuse", whereby application developers emulate organizational features and software abstractions found in the framework, facilitates new algorithm development by increasing the flexibility of application codes.

We begin the remainder of this paper with a brief introduction to the primary features of SAMR computations and an overview of the SAMRAI software library and related efforts. Then, we introduce the *Strategy* design pattern which is a central algorithm design concept in SAMRAI. This is followed by a description the implementation of three different applications using SAMRAI: a standard Euler gas dynamics application, a hybrid continuum-particle gas dynamics code, and a code for simulating laser-plasma instabilities. Next, we discuss the *Abstract Factory* design pattern which allows parts of the framework, such as communication routines, to be generic with respect to concrete data types, such as particles in the hybrid code. Finally, we summarize our presentation and compare our approach with other object-oriented scientific computing efforts.

2 SAMRAI Overview

A full description of structured adaptive mesh refinement (SAMR) algorithms and applications is well beyond the scope of this paper as is a complete discussion of the SAMRAI library. However, to fix ideas central to this paper, we provide brief descriptions of basic features of SAMR, the SAMRAI library, and related SAMR software efforts.

2.1 Structured Adaptive Mesh Refinement (SAMR)

In many important science and engineering problems, key features of the solution reside in localized regions of the computational domain. Adaptive mesh refinement helps to place spatial and temporal mesh resolution near these features where it is needed most. By focusing memory usage and computational effort, a highly resolved solution may be achieved more efficiently than if the entire mesh is refined uniformly.

SAMR is a particular approach to adaptive mesh refinement in which the computational grid is implemented as a collection of structured mesh components. The computational mesh consists of a hierarchy of levels of spatial and temporal mesh resolution. Each level in the hierarchy corresponds to a single uniform degree of mesh spacing. Also, the levels are nested; that is, the coarsest level covers the entire computational domain and each successively finer level covers a portion of the interior of the next coarser level. Computational cells on each level are clustered to form a set of logically-rectangular patch regions. Simulation data is stored on these patches in contiguous arrays that map directly to the mesh cells without indirection.

SAMR solution methods share certain characteristics with uniform, non-adaptive structured grid methods. In particular, the computation is organized as a collection of numerical routines that operate on data defined over logically-rectangular regions and communication operations that pass information between those regions, for example, to fill "ghost cells". However, SAMR methods can be substantially more complicated than those for uniform meshes since the solution is constructed on a composite mesh. That is, the solution algorithm must treat internal mesh boundaries between coarse and fine levels properly to maintain a consistent solution state.

2.2 Hydrodynamics With Structured Adaptive Mesh Refinement

Many SAMR computations are based on the work of Berger, Oliger, and Colella [3, 4], who developed SAMR techniques for integrating hyperbolic conservation laws on a locally-refined mesh in a conservative manner. The algorithm is central to the applications discussed in this paper and motivates much of the software organization described herein.

Systems of hyperbolic conservation laws encountered in shock hydrodynamics problems can be written in integral form as

$$\int_{\Omega} m|_{t_2} dx - \int_{\Omega} m|_{t_1} dx + \int_{t_1}^{t_2} \oint_{\partial\Omega} \mathbf{F} \cdot \hat{n} dS dt = \int_{t_1}^{t_2} \int_{\Omega} q dx dt \tag{1}$$

Here m is a vector of conserved quantities, $\mathbf{F}$ is a flux matrix, and q is a source term. Commonly-used shock capturing schemes define finite difference expressions for this equation where m is approximated as cell-centered quantity and $\mathbf{F}$ terms are located on the faces between cells. Time integration of m involves a conservative difference representing the divergence of these numerical flux terms.

Time stepping on an SAMR mesh hierarchy is a recursive algorithm where integration of individual levels is interleaved [3, 5]. The procedure involves two main steps: solution advance on a level and solution synchronization between levels. Since each level employs different mesh resolution, the definition of numerical flux terms at interfaces between mesh levels requires extra care so that the method remains conservative. Typically, the process of matching fluxes assumes that finer levels provide more accurate numerical results than coarser levels. That is, averages of fine flux data are used in the finite difference equations on coarser levels at fine mesh boundaries. Also, coarse values of the solution m are replaced by suitable averages of fine m data where levels overlap. Although the solution is computed differently in overlapping cells because of different mesh spacing on each level, the data synchronization procedure makes the scheme consistent with respect to conservation.

Figure 1 shows a typical timestep sequence for a problem with three mesh levels and a ratio of 4 between the mesh spacing on consecutive levels. Numbers indicate the order in which timesteps are performed on different levels. Recall that the time increment used in an explicit shock capturing scheme is subject to the Courant-Friedrichs-Lewy (CFL) constraint. Thus, larger timesteps are used on coarser levels than on finer levels. Since the levels are also nested, each coarser level is advanced before any finer levels. Space and time interpolated boundary values are provided for each level from coarser levels before the solution is advanced. Synchronization points, when fluxes and the solution are made consistent among levels, are indicated by horizontal lines in the figure. Re-meshing is performed at certain timestep intervals so that solution advance, synchronization, and re-meshing operations are coordinated. In the diagram, remeshing points are marked by open circles.

2.3 The SAMRAI Framework

SAMRAI is an object-oriented `C++` software library developed in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. SAMRAI provides a "toolbox" of classes that simplify the construction and management of SAMR applications. The aims of the SAMRAI effort are to extend SAMR technology to new problem domains, and to explore modern software design and implementation ideas in a numerical library that supports complex, multi-physics problems. Two central design goals of SAMRAI are: to allow specialization of patch-based data structures while reusing the exiting parallel communication infrastructure that automatically treats complex, dynamically changing data configurations in SAMR; and to provide a flexible algorithmic framework to explore solution methods for new applications.

The SAMRAI library is partitioned into a number of software packages. Each package holds a collection of logically-related classes that form some functional role in an SAMR application. To gain a sense of the functionality that SAMRAI provides, we list the packages here:

- **Toolbox** provides basic utility classes applicable throughout the framework, such as smart containers, input and restart file management, and performance monitoring.
- **Hierarchy** defines SAMR hierarchy structural abstractions, such as hierarchy, level, and patch, and provides basic index space and box calculus operations on which most SAMR routines depend.
- **Transfer** manages parallel data communication on an SAMR mesh.
- **PatchData** provides various patch data types for simulation data on an SAMR patch hierarchy, such as data arrays that are cell-centered, node-centered, face-centered, etc.
- **Math Operations** provides basic arithmetic and other operations, such as dot products and norms, needed for vectors defined on an SAMR mesh.
- **Mesh** contains classes used to construct and dynamically re-mesh levels in an SAMR patch hierarchy.
- **Geometry** supports specific coordinate systems on an SAMR mesh, such as Cartesian grids, etc.

- **Algorithms** contains time integration and other solution algorithms applicable to certain classes of partial differential equations solved an SAMR patch hierarchy. Solution algorithms, developed for a particle application, and which can be factored into elements that may apply to other problems are placed in this package.

- **Solvers** supports the development of linear and nonlinear solvers for SAMR problems by providing vector structures and interfaces to other libraries such as PETSc [6], PVODE [7], and *hypre* [8].

Computational scientists select items from these packages when constructing applications. Since many parts of the library can used without modification, SAMRAI users leverage a large code base that is shared across different applications. However, new algorithmic functionality is usually needed for new problems. Many classes in the "Algorithms" package, for example, are designed to be specialized or augmented for this reason. Also, new patch data types are occasionally needed to represent data, such as particles, on the mesh in a manner unique to a given problem. These issues are addressed in Sections 3 and 4, respectively.

2.4 Related SAMR Software Efforts

The CCSE Applications Suite and Chombo library at Lawrence Berkeley National Laboratory are off-shoots of the BoxLib package [9]. BoxLib and these newer packages are `C++` class libraries for developing SAMR and block-structured finite difference algorithms for systems of partial differential equations. These libraries have been the cornerstone of an impressive history of SAMR algorithm and CFD application development [10, 11, 12, 13, 14]. The Berkeley software libraries provide complete application codes that serve as useful guides for application development. The software representation of basic concepts in most SAMR libraries, including SAMRAI, is directly attributable to the efforts of the Berkeley researchers.

Basic ideas for designing flexible algorithm classes in SAMRAI have their origins in the SAMR software library developed by John Trangenstein at Duke University [5, 15, 16]. The software in that library is similar to BoxLib and its extensions. However, it provides greater freedom for specifying patch data types and provided flexibility for application developers to specialize algorithms through inheritance. These features greatly influenced SAMRAI goals and design.

The GrACE library and its predecessor DAGH are primarily data management infrastructures for parallel SAMR applications [17]. These libraries provide programming abstractions for managing parallel data distribution and data decomposition on adaptive mesh hierarchies. A unique feature of GrACE is its use of space-filling curves to manage parallel data distribution and maintain locality. These libraries have been used in a variety of applications, ranging from binary black hole computations to multi-resolution spatial databases.

AMR++ is a collection of C++ classes for implementing SAMR within Overture [18], which is a comprehensive framework for solving partial differential equations in complex geometrical regions described by overlapping regular meshes. AMR++ relies on the A++ and P++ array class libraries [19, 20]. These libraries support automatic data decomposition and fine-grain data parallel operations on distributed arrays.

The KeLP library is not an SAMR framework per se, but rather a support library for parallel block-structured applications [21]. KeLP provides powerful mechanisms for managing data decomposition and interprocessor communication for irregular, dynamic, block-structured parallel applications [22].

All SAMR libraries provide a similar set of basic abstractions for SAMR computations which include the notion of abstract index spaces, patches, levels, and hierarchies. Much of this support approximates very closely, if it does not directly mimic, the methods and data structures at the heart of BoxLib. Management of data on SAMR hierarchies, especially in parallel, differs among packages. Parallel data distribution in SAMRAI is similar to that employed in the Berkeley codes, KeLP, and the Trangenstein library. That is, each patch is assigned to a single processor and numerical routines on each patch are performed by serial Fortran, `C`, or `C++` kernels. In contrast, P++ automatically decomposes array data and requires that numerical operations be expressed using the P++ array syntax. The parallel data communication abstractions in SAMRAI are generalizations and extensions of ideas found in KeLP. SAMRAI differs from other software packages in several ways. The software is designed to address problems to which SAMR has not yet been applied. Our approach to object-orientation for composing larger, more complex algorithmic units from smaller, re-usable elements and the ability of SAMRAI to support new patch data types in parallel without modifying or recompiling the framework is unique.

3 Algorithm Composition

Recall that a fundamental goal of SAMRAI is to provide a flexible algorithmic framework that allows code reuse across different SAMR applications and supports extensions to new solution algorithms. To accomplish this, the behavior of individual algorithmic pieces is concisely defined and clean interfaces are developed to couple the different pieces. The result is a system in which individual parts may be replaced or enhanced without adversely influencing the behavior of other components. Although, we continue to grapple with these issues as new applications are attacked, the discussion in the following sections demonstrates significant progress toward our goal.

We begin by introducing the *Strategy* design pattern that we use in SAMRAI to decompose complex algorithms into flexible reusable parts. Next, we discuss three application codes that utilize the algorithm algorithm discussed in Section 2.2 and the *Strategy* pattern in various ways. The first code solves the Euler equations of gas dynamics. This is a standard SAMR application and thus is useful for introducing our approach to software organization. Then, we describe an extension of this code to a hybrid model that couples the continuum Euler model to a discrete particle representation at fine mesh scales. Finally, we discuss the ALPS laser-plasma simulation code which further illustrates the decomposition of a complex application code into components supplied by SAMRAI and those that are problem-specific. Each of the two latter efforts employs very different numerical algorithms within a single application code. The codes illustrate extensions of our basic software approach and show re-application of software components in new SAMR problems.

3.1 The *Strategy* Pattern

The *Strategy* pattern is the primary object-oriented design tool employed in SAMRAI to encapsulate algorithmic elements and define reusable interfaces between them. Generically, the purpose of the *Strategy* pattern is to define and encapsulate a family of algorithms by making their constituent parts interchangeable through common interfaces [1].

As an example of how we apply this pattern, recall the timestep sequence in Figure 1. This basic sequence applies to a potentially broad range of problems, beyond hyperbolic conservation laws. SAMRAI provides a class, called `TimeRefinementIntegrator`, that encapsulates the SAMR timestep sequence. This class orchestrates data initialization, time integration, data synchronization, and dynamic meshing operations. Other objects provide specific instances of those operations. Figure 2 illustrates the coupling between `TimeRefinementIntegrator` and classes that provide problem-specific level integration routines via the *Strategy* pattern.

Here, the `HyperbolicLevelIntegrator` class provides a particular set of level initialization, integration, and synchronization operations appropriate for hyperbolic conservation laws. Another set of these routines applicable to other types of problems may be provided similarly. Each such level integration class is derived from the `TimeRefinementLevelIntegrator` abstract base class. The base class defines the interface between the timestep sequence generator and problem-specific algorithmic routines for processing the levels. The `TimeRefinementIntegrator` object itself remains insulated from problem-specific procedures because it only recognizes the abstract `TimeRefinementLevelIntegrator` interface.

3.2 Euler Gas Dynamics

In this section, we make our approach to algorithm composition more concrete by describing the implementation of a standard hydrodynamics application based on SAMRAI. The code models the Euler equations of gas dynamics [23].

3.2.1 The Euler Application Code. The SAMR hydrodynamics algorithm in Section 2.1 was developed to solve problems such as the Euler equations. These equations form a system of hyperbolic conservation laws (Equation 1), where m is the vector of conserved quantities, density, momentum, and total energy, and $\mathbf{F}$ is the corresponding flux matrix.

To construct the Euler code, we compose objects found in the SAMRAI library and introduce numerical operations particular to the Euler model. Each object performs a certain set of algorithmic or numerical tasks, such as generating the timestep sequence that interleaves operations on the levels, integrating and synchronizing the solution, dynamic re-meshing, and numerical operations for the discrete equations defined on individual patches. The organization of the major objects comprising this composition are illustrated in Figure 3.

The coupling between `TimeRefinementIntegrator` and `HyperbolicLevelIntegrator` was introduced with the *Strategy* pattern in Section 3.1. The `TimeRefinementIntegrator` object also maintains a pointer to the `GriddingAlgorithm` object which constructs and dynamically re-meshes levels in a SAMR patch hierarchy. Re-meshing operations include tagging cells to refine (generally, a user-defined operation), clustering these cells into box regions, and load balancing new patches. The *Strategy* pattern is applied repeatedly to allow flexibility among these operations. The `StandardTagAndInitialize` object supports several options for tagging cells to refine. Application-specific routines provide the operations that are necessarily problem-dependent, such as tagging near sharp gradients or large errors in the solution. While SAMRAI provides default load balancing operations, for example, the decoupled organization that we have chosen allows application developers to easily incorporate load balancing routines specialized for a particular problem. To provide a new load balance strategy, one derives a class from the `LoadBalanceStrategy` base class that defines an interface for the needed operations to the mesh generation class. Similarly, alternative approaches for cell tagging and clustering can be introduced.

A configuration such as that shown in Figure 3 is designed to be assembled at run-time. In most cases, the organization is set in the main program when objects are created. That is, each object is initialized with objects it references when it is constructed. For example, `TimeRefinementIntegrator` depends on `TimeRefinementLevelStrategy`. Thus, `HyperbolicLevelIntegrator` is created and initialized, then this object is passed into the `TimeRefinementIntegrator` constructor. The entire configuration in Figure 3 is generated as follows:

```
Euler* euler = new Euler( ... );

HyperbolicLevelIntegrator* hyperbolic_integrator =

   new HyperbolicLevelIntegrator(euler, ... );

StandardTagAndInitialize* error_detector =

   new StandardTagAndInitialize(hyperbolic_integrator, ... );
```

```
BergerRigoutsos* box_generator = new BergerRigoutsos( ... );
UniformLoadBalance* load_balancer = new UniformLoadBalance( ... );
GriddingAlgorithm* gridding_algorithm =
   new GriddingAlgorithm(error_detector,
                         box_generator,
                         load_balancer,
                         ... );
TimeRefinementIntegrator time_integrator(hyperbolic_integrator,
                                         gridding_algorithm,
                                         ... );
```

Figure 3 only shows the static organization of objects employed in the Euler code. It is also important to understand the interaction among these objects. For example, during the timestep sequence, the `TimeRefinementIntegrator` calls the function `advanceLevel()` in the `TimeRefinementLevelIntegrator` object it references to advance the data on a single level. Operations specific to hyperbolic conservation laws, such as filling boundary data for the level, advancing the data, computing a new time step, and updating flux integrals, are performed by `HyperbolicLevelIntegrator`. These operations apply to an entire level. Numerical routines for individual patches, such as `advancePatch()`, are invoked by `HyperbolicLevelIntegrator` by accessing the `HyperbolicPatchStrategy` object it references. Level operations are separated from numerical routines for an individual patch via the abstract base class `HyperbolicPatchStrategy`. In other words, `HyperbolicLevelIntegrator` only sees the interface declared by the patch strategy, not the fact that the `Euler` object performs operations on the patches. Again, this is an application of the *Strategy* pattern.

`Euler` is the only object in the application that is specific to the Euler equations. `Euler` contributes two types of problem-specific information, the variables that define the problem state and the numerical routines for treating the Euler equations on each patch in the SAMR hierarchy. `Euler` registers variables, representing density, velocity, and pressure, for example, with `HyperbolicLevelIntegrator` during an initialization phase of the program. The integrator manages data storage for these quantities on the patch hierarchy according to the needs of the integration algorithm without knowing the specific number or type of variables involved. Data management operations include filling ghost cells for patches and moving data between levels during synchronization. Before each routine in `Euler` is invoked, the integrator sets up data on the patch appropriately.

3.3 Hybrid Continuum-Particle Methods

In Section 3.2.1, we describe the composition of a standard SAMR application from elements in the SAMRAI framework. Here, we discuss an extension of that code that combines the continuum Euler model with a discrete particle model. This hybrid application is a collaborative effort between the SAMRAI team and Alejandro Garcia of San Jose State University [24, 25]. This project extends previous continuum-DSMC (Direct Simulation Monte Carlo) hybrid work [26] by allowing multiple DSMC regions, and supporting full adaptive mesh capabilities for particles in parallel.

The aim of this effort is to develop hybrid continuum-particle methods combined with SAMR to model complex fluid interface dynamics, such as the Richtmyer-Meshkov instability which occurs as the interface between two fluids of different density is accelerated by a shock [27]. Problems of this sort involve different physical processes at different length scales. Hydrodynamic transport models the physics away from the interface while molecular diffusion is the primary mechanism at the interface. A mixing region grows from the interface and moves as the instability evolves. Particle methods are too expensive for large problem domains of interest while continuum-based turbulent mixing models are limited by the finest mesh resolution. Therefore, a hybrid approach combining continuum and discrete methods with adaptive meshing is desirable since it offers the potential to resolve flow features properly at different scales in an efficient manner.

3.3.1 Hybrid Model. The hybrid model couples the Euler approximation to a DSMC model. The two numerical approaches are vastly different. The continuum Euler model represents compressible fluid flow as a deterministic system of partial differential equations containing a few variables, such as density, velocity, and temperature. DSMC approximates the Boltzmann equation using a representative, stochastic sampling of a collection of particles. In DSMC, the system state is a collection of particles, each of which is defined by a position vector, a velocity vector, and a fluid type [28]. The DSMC integration process involves moving and colliding particles in a manner consistent with the kinetic theory of gases. The DSMC model is valid in flow regimes where the continuum approximation breaks down, such as where important length scales in the fluid dynamics are near the molecular mean free path.

In the hybrid application, DSMC is used on the finest mesh level in the SAMR hierarchy and the Euler model is employed on all other levels. Local mesh refinement concentrates the DSMC description near fluid interfaces and shocks. Major numerical and algorithm concerns in the hybrid approach involve the location of the continuum-DSMC interface and the coupling of the two approximations, as shown in Figure 4.

The hybrid solution algorithm uses the same local time stepping process in the SAMR integration scheme discussed in Section 2.1. The primary additional complexity in the hybrid application involves coupling the continuum and particle representations [26]. Auxiliary particles are generated in "sheath" regions around each DSMC patch before particles are advanced. This is analogous to filling ghost cell data in the continuum case. A probability distribution, such as Chapman-Enskog [29], determines the velocity of each particle in the sheath region from Euler solution values and gradients in nearby cells. During particle integration, flux information around each DSMC patch is accumulated to determine mass, momentum, and energy transfer across the continuum-particle interface. These fluxes are used in the Euler method along with proper averaging of the DSMC state to the continuum levels to make the two computations consistent in a manner similar to the level synchronization in the standard SAMR hyperbolic algorithm.

3.3.2 Hybrid Application Code. The organization of the hybrid Euler-DSMC application code is similar to that of the Euler code shown in Figure 3. A significant difference is that the hybrid code employs a new level integrator which coordinates DSMC and Euler operations. The new integrator is coupled to the `TimeRefinementIntegrator` instead of the `HyperbolicLevelIntegrator`. Figure 5 illustrates the class organization in the hybrid code. All classes appearing in the Euler application are reused without modification in the hybrid code. In the figure, we omit the mesh generation objects since their usage is identical to the Euler case.

The coupling between `TimeRefinementIntegrator` and `HybridIntegrator` is similar to the coupling between `TimeRefinementIntegrator` and `HyperbolicLevelIntegrator` in the Euler application. `HybridIntegrator` is derived from `TimeRefinementLevelIntegrator` and obeys that interface so that when `TimeRefinementIntegrator` invokes the `advanceLevel()` function, hybrid algorithm operations are performed. When advancing a level coarser than the particle level, `HybridIntegrator` defers to `HyperbolicLevelIntegrator` which applies the Euler model. Otherwise, the DSMC routines are used to treat DSMC data on the particle level.

It is interesting to note that bulk of the operations that couple the particles and the continuum solution in the data synchronization step are actually performed by `HyperbolicLevelIntegrator`, which knows nothing about particles. The continuum solution and the DSMC data are linked on each patch on the particle level via numerical routines in the `DsmcPatchModel` class which were developed to translate information between the two representations. We also remark that, in the hybrid code, the `StandardTagAndInitialize` object is coupled to the `HybridIntegrator` rather than the `HyperbolicLevelIntegrator`. Refinement operations in the `HybridIntegrator` must resolve features of the Euler solution as well as place fine mesh where particles are needed.

The merging of the Euler code and the DSMC particle data structures and numerical routines, which were developed independently of the hybrid application [30], was accomplished without modifications to either code. The `HybridIntegrator` class was developed to implement the algorithmic coupling between the two methods. The `DsmcPatchModel` class was developed to couple the two descriptions of the solution state. In Section 4, we will describe how the DSMC particle data uses the SAMRAI parallel communication framework without changing the particle routines or recompiling SAMRAI library code.

3.4 Adaptive Laser Plasma Simulator

The final application that we discuss is the ALPS (Adaptive Laser Plasma Simulator) project under development in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory [31, 32]. The goal of this effort is to investigate SAMR methods for modeling laser-plasma instabilities. The ability to predict and control the interaction between intense laser light and a plasma is critical in several important engineering applications. Filamentation instabilities, for example, appear as "speckles" as the light passing through the plasma refracts into regions of low plasma density. Increased light intensity in these regions causes further reduction in plasma density and, consequently, greater light refraction. Computational methods for this sort of problem require several cells to resolve important light wavelengths. Since real beams can be thousands of wavelengths across, SAMR offers the potential to efficiently resolve "speckles" that are small relative to the light beam. While still a research effort, the ALPS code has shown the ability to resolve certain instabilities and has demonstrated parallel adaptive speedup.

3.4.1 The ALPS Model. In ALPS, the plasma is represented by an Eulerian fluid model that describes the conservation of total mass, momentum and energy of ions. The ions are coupled to the electrons via a nonlinear potential equation. Light propagation is modeled using a reduced wave equation. The plasma fluid equations are approximated using a shock-capturing method similar to that described in Section 2.1. The ion-electron coupling involves the solution of a Poisson equation. Light propagation is solved either by employing a Fast Fourier Transform or a finite difference paraxial approximation. Integration of the full system of equations uses separate advance steps for the plasma fluid variables (density, pressure, velocity, and temperature), the light variables (amplitude and intensity), and the potential. When local mesh refinement is applied, consistency of the solution across the different levels in the hierarchy is maintained via hydrodynamic synchronization procedures similar to those discussed in Section 2.1 combined with composite mesh solves for the potential and the light.

3.4.2 The ALPS Application Code. The ALPS model and, thus the ALPS code, is substantially more complex than the Euler and hybrid applications discussed previously. The complexity arises from the need to coordinate three models with very different numerical characteristics and share simulation variables among these models. We note that SAMRAI provides mechanisms to simplify sharing variable data among different computational models without the models becoming dependent on each other. We do not discuss this feature here; instead we focus on high-level class organization.

The organization of the main integration objects used in the ALPS code appears in Figure 6. The ALPS application code shares fundamental organizational features and software components with the applications described earlier. The timestepping sequence used to process levels in the SAMR hierarchy is generated by `TimeRefinementIntegrator`. The level integration object `LaserPlasmaIntegrator` is analogous to the `Hybrid Integrator` in the hybrid application. It coordinates the integration of the plasma, light, and potential equations on each level. This integrator also orchestrates the synchronization process for variables shared by these models.

The `LaserPlasmaIntegrator` class is derived from the `TimeRefinementLevelIntegrator` abstract base class. Thus, it is coupled to `TimeRefinementIntegrator` using the *Strategy* pattern. The plasma is advanced by the same `HyperbolicLevelIntegrator` object used in both the Euler and hybrid codes. However, in ALPS, the `HydroPatchModel` object replaces `Euler` since the equations, and thus the variables and numerical kernels, applied to each patch are different. This flexibility is a direct result of our use of the *Strategy* pattern. That is, the hyperbolic level integration class manages can data for different numerical models without direct knowledge of those models or the variables involved.

The `LaserPlasmaIntegrator` object employs two other integration classes, `LightIntegrator` and `PotentialIntegrator`, to treat the laser light and potential equation, respectively. These classes were developed especially for the ALPS code since these procedures are specific to ALPS. The light routines and potential routines are decomposed into operations applicable to an entire hierarchy level and those which apply to individual patches. This organization mirrors the separation of the `HyperbolicLevelIntegrator` and a particular patch strategy. Organization like this is commonly found in SAMRAI so that data management operations, such as ghost cell filling for patches on a level at specific points in an integration algorithm, are insulated from numerical operations for patches that are more problem-specific. This decomposition encourages exploration of different numerical methods and makes it easier to manage such experimentation.

3.5 Analysis of The *Strategy* Pattern

The *Strategy* pattern is extremely useful for decomposing complicated algorithms and encapsulating their elements. The use of a common interface to characterize a family of related algorithmic components allows a system to be configured for a wide range of behavior. A result of liberal use of the *Strategy* pattern is a sizable collection of smaller classes instead of a few larger classes. While this may seem inconvenient and to over-complicated matters, we believe that decomposition of this sort provides certain advantages. First, smaller classes are more concise and easier to maintain in a library than large monolithic classes. Second, clean encapsulation and loose coupling encourages specialization of specific operations rather than wholesale rewriting of an entire class to achieve behavior that may be only slightly different than what already exists.

The encapsulation forced by the *Strategy* pattern provides a valuable alternative to large, overly-complex classes that often result from the abuse of inheritance. For instance, the design in Figure 2 could have been implemented by deriving new classes from `TimeRefinementIntegrator` directly. The result would be larger, more complicated classes that differ in level integration procedures, but have much timestepping code in common. In addition, the function call overhead that arises when indirection at the high algorithmic level in this way is negligible. We refrain from introducing abstractions below the level of the arrays of simulation data on patches. Once data is retrieved from the patch container, one can apply numerical operations at will. We note that many SAMRAI users write numerical operations in `FORTRAN` or `C` for simplicity and efficiency. It is also straightforward to employ pre-existing numerical kernels.

4 Flexible Data Structures

The hybrid application discussed in Section 3.3 was developed by combining two pre-existing codes, the Euler code described in Section 3.2.1 and a particle-based DSMC code [30]. The DSMC data structures and numerical routines are insulated from SAMRAI abstractions by a "wrapper " class, which follows the *Adapter* structural design pattern [1]. This class serves two important functions. First, it acts as a translator between SAMRAI patch data and the DSMC particle structures. Second, it allows the particles to be manipulated by the parallel communication infrastructure in SAMRAI. The serial particle data structures were incorporated into a parallel adaptive mesh refinement environment without modification to the DSMC routines or SAMRAI classes.

4.1 The *Abstract Factory* Pattern in SAMRAI

During a computation, the SAMRAI framework manipulates patch data by invoking operations such as allocation, deallocation, copying, and marshaling and un-marshaling data for parallel communication. An important design goal of SAMRAI is to support arbitrary user-defined data on an SAMR patch hierarchy. We want to be able to introduce new types without modifying or recompiling the framework. We also want to manage new and irregular data types, such as user-defined particles, similarly to standard types such as cell-centered and node-centered data arrays. Thus, the patch data and communication infrastructure in SAMRAI cannot know any specific data types.

The *Abstract Factory* creational pattern is used in SAMRAI to manage data which are unknown at compile-time. This pattern allows the creation of families of related objects without specifying concrete types directly [1]. The *Abstract Factory* pattern uses two related inheritance hierarchies. The first hierarchy includes an *abstract product* class that declares the interface for all objects created by the pattern. The second hierarchy provides *factory* objects that create those products. Figure 7 shows how the `DsmcPatchData` object that provides access to the DSMC data and numerical routines fits into these hierarchies like any standard data type.

In SAMRAI, a `Patch` is a container for any data whose storage can be mapped to a logically-rectangular region of index space. Each concrete patch data type is derived from the abstract `PatchData` class which defines a uniform set of member functions such as `copy()` and `packStream()` (used for interprocessor communication). Each `Patch` owns a pointer to a shared `PatchDescriptor` object that provides access to factory objects that create the concrete patch data objects. In particular, the `Patch` asks the `PatchDescriptor` to return the appropriate `PatchDataFactory` to allocate a concrete `PatchData` instance. Thus, the `allocatePatchData()` function in the factory created a new concrete `PatchData` object.

The *Abstract Factory* pattern separates concrete object creation and declaration by encapsulating the responsibility for creating product objects. Introducing a new patch data type, such as `DsmcPatchData`, requires two basic steps. First, the `DsmcPatchData` class is derived from the abstract `PatchData` base class and the required virtual functions are supplied:

```
class DsmcPatchData : public PatchData {
   void copy(const PatchData& source);
   void packStream(AbstractStream& stream, ...);
   int getDataStreamSize(const BoxOverlap& overlap);
   ...
};
```

Second, the `DsmcPatchDataFactory` is derived from the abstract `PatchDataFactory` base class.

```
class DsmcPatchDataFactory : public PatchDataFactory {
   Pointer<PatchData> allocate(const Box& box);
   ...
};
```

The `DsmcPatchDataFactory` allocates `DsmcPatchData` objects on the SAMR patch hierarchy. Each `DsmcPatchData` instance is manipulated by the framework like any other type via the `PatchData` interface.

4.2 Analysis of the *Abstract Factory* Pattern

The *Abstract Factory* pattern presents two potential disadvantages. First, the pattern requires that two classes—the product class and the factory class—be implemented for each new product class. Second, dynamic safe type casting is needed to obtain concrete class references. Although low-level data operations like copies are performed via the abstract `PatchData` interface, user-defined numerical routines must process data using the concrete class interface.

We view these concerns as minor inconveniences rather than significant drawbacks in our approach. The concrete factory class is small and is only used to call the constructor of the product class. The interface declared by an abstract product class, such as `PatchData`, concisely expresses the functionality required by any concrete product class to link into the framework. Also, to cast from abstract product to concrete product, we use the run-time type checking mechanism (*i.e.*, RTTI) provided by the C++ language [33]. The *Abstract Factory* pattern enhances software flexibility and extensibility since concrete product classes (*e.g.*, `NodeData` in Figure 7) are insulated from other parts of the framework code. Thus, new product classes are easily introduced to the framework after it has been compiled and archived into a library.

5 Summary and Conclusions

By using object-oriented concepts, such as design patterns, in the SAMRAI software framework, we have met two of our most important design goals: flexible and extensible algorithm support for diverse SAMR applications and general support for arbitrary data types. Design patterns such as *Strategy* and *Abstract Factory* provide SAMRAI with "plug-and-play" flexibility to rapidly explore new structured adaptive mesh refinement applications. We have found object-orientation to be very useful for programming "in the large" in scientific computing software, especially for managing complex, intricate algorithms. Plug-and-play interoperability of high-level algorithmic objects provides several advantages. First, it frees application programmers from unnecessary, redundant code implementation and reduces development time. Second, it promotes the exploration of different algorithmic choices within a single application. Third, it increases software reuse within the framework, which facilitates testing, maintenance, and extensibility of the architecture.

Another approach for managing complexity in scientific applications is to focus on hiding the low-level details associated with parallel data distribution and interprocessor communication. Both POOMA [34] and P++ [19, 20], to cite two examples, support whole array operations on distributed parallel arrays in C++; users write the same code for serial or parallel applications. The Overture [18] library is built on top of P++ and provides operator classes for overlapping grid calculations. The SIERRA [35] framework focuses on the management of data structures associated with adaptive finite element calculations found in engineering simulations. The primary difference between these approaches and the work presented here is that SAMRAI focuses on managing the complexity of the algorithm space, not the parallel data structures themselves. Although SAMRAI hides much of the complexity of parallel data management on a SAMR patch hierarchy, users tend to interact with this support directly, specifying when and where variable data is communicated and how it is may be distributed.

The implementation described in this paper emphasize software design using inheritance mechanisms. Others in the scientific computing community are exploring generic programming techniques for designing software libraries. Generic programming approaches exploit `C++` template support and are based on ideas first expressed in the Standard Template Library (STL) [36]. Generic programming design typically begins by identifying key algorithmic "concepts" and then separating the implementation from details of the data types through "container-free" approaches [37]. Perhaps the best example of this approach in the scientific computing community is the Matrix Template Library [38], which implements a high-performance linear algebra library using generic programming ideas.

Although generic programming implementations using `C++` templates may seem very different from the inheritance-based approaches described in this paper, they are, in fact, very similar. The generic programming analysis that identifies key algorithmic concepts is the same as that applied to identify the methods in the abstract base class of the *Strategy* pattern. Return values whose type is specified as a template argument can be implemented using the *Abstract Factory* pattern and explicit down-casting.

The primary difference between the generic programming and inheritance approaches lies in the time of binding between abstract concepts and concrete implementations. In the case of generic programming, this binding is done at compile-time during the instantiation of `C++` templates. For inheritance approaches, the binding is done at run-time. Because more information is known at compile-time for generic programming approaches, optimizing compilers typically generate more efficient code, especially for small, simple objects. However, inheritance-based approaches have the advantage of being more flexible, since connections between objects are made at run-time and may be changed during the course of a simulation. For example, this capability is useful for modifying load balancing strategies or when swapping in different preconditioning methods for a linear solver.

One problem for generic programming techniques is that the implementation of templates in C++ does not provide a convenient mechanism for specifying the interface for a complex object. Unlike inheritance approaches, there is no way—other than user documentation, of course—to document the abstract methods that must be provided by the template argument. Missing methods are caught by the compiler during template instantiation in the final compilation phase. Furthermore, complex algorithm arrangements such as that described in Section 3 would result in extremely complicated templated types with numerous, nested template arguments. In addition, ownership of shared objects would also need to be resolved. For these reasons, we believe that generic programming approaches are better suited to libraries with small objects that may be expressed simply and in portions of code where performance is critical whereas inheritance approaches are more useful for large, complex algorithms with many interacting objects.

One recent development in the scientific computing community is the use of component technologies. The Common Component Architecture (CCA) working group [39] is developing a scientific version of industry component approaches such as CORBA [40]. The design approaches used in SAMRAI are amenable to such component-based approaches, since the objects used to express SAMRAI algorithms can be directly translated into components. Using CCA component terminology, a *Strategy* abstract base class becomes a “uses port” because it uses the services of another algorithm object and each algorithm object becomes a “provides port” because it provides a concrete strategy service to another algorithm object.

In conclusion, we feel that object-oriented approaches offer the most benefit when applied at the higher levels of a numerical software architecture. Object-oriented techniques enable the composition of complex algorithms from smaller, more manageable parts that are suitable to a variety of applications. They promote code and algorithm reuse and also facilitate testing and management of software framework components. Most importantly, object-oriented patterns support more productive application construction by allowing rapid exploration of new algorithms that are built from both existing and new components.

6 Acknowledgments

We gratefully acknowledge the contributions of our collaborators whose work is discussed in this paper. Alejandro Garcia (San Jose State University) is working with us to develop the hybrid Euler-DSMC application. Milo Dorr, Xabier Garaizar, and Jeff Hittinger (CASC) are building the ALPS code. Without the efforts of these individuals, these projects would have never materialized.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of reusable Object-Oriented Software.* Addison-Wesley Publishing Co., Menlo Park, CA, 1995.

[2] Richard D. Hornung and Scott R. Kohn. The use of object-oriented design patterns in the SAMRAI structured amr framework. In *Proceedings of the First Workshop on*

Object Oriented Methods for Inter-operable Scientific and Engineering Computing, 1998. See `http://www.llnl.gov/CASC/SAMRAI`.

[3] Marsha J. Berger and Phillip Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82:64–84, 1989.

[4] Marsha J. Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.

[5] John A. Trangenstein. Adaptive mesh refinement for wave propagation in nonlinear solids. *SIAM J. Sci. Stat. Comput.*, 16(4):819–839, 1995.

[6] S. Balay, W. D. Gropp, L. Curfman-McInnes, and B. F. Smith. Efficient management of parallelism in object-oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997. See `http://www.mcs.anl.gov/petsc`.

[7] A. Hindmarsh. Parallel software for differential and algebraic systems. See `http://www.lnl.gov/CASC/PVODE/`.

[8] E. Chow, A. J. Cleary, and R. D. Falgout. Design of the *hypre* preconditioner library. In *Proceedings of the First Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, 1998. See `http://www.llnl.gov/CASC/linear_solvers/`.

[9] J.B. Bell and Phillip Colella. Amr software packages. See `http://seesar.lbl.gov/AMR/index.html`.

[10] Ann Almgren, John Bell, Phillip Colella, Louis Howell, and Micheal Welcome. A conservative adaptive projection method for the variable density incompressible Navier-Stokes equations. *Journal of Computational Physics*, 142:1–46, 1998.

[11] M.S. Day and J.B. Bell. Numerical simulation of laminar reacting flows with complex chemistry. *Combustion Theory Modeling*, 4(4):535–556, 2000.

[12] D.S. Nolan, A.S. Almgren, and J.B. Bell. High reynolds number simulations of axisym-

metric tornado-like vortices with adaptive mesh refinement. Technical Report LBNL-42860, Lawrence Berkeley National Laboratory, 1999.

[13] Rick Pember, John Bell, Phillip Colella, William Crutchfield, and Micheal Welcome. An adaptive cartesian grid method for unsteady compressible flow in irregular regions. *Journal of Computational Physics*, 120:278–304, 1995.

[14] Mark Sussman, Ann Almgren, John Bell, Phillip Colella, Louis Howell, and Micheal Welcome. An adaptive level set approach for incompressible two-phase flows. *Journal of Computational Physics*, 148:81–124, 1999.

[15] X. Garaizar and J. Trangenstein. Adaptive mesh refinement and front tracking for shear bands in granular flow. *SIAM Journal on Scientific Computing*, 20:750–779, 1999.

[16] Richard D. Hornung and John A. Trangenstein. Adaptive mesh refinement and multilevel iteration for flow in porous media. *Journal of Computational Physics*, 136:522–545, 1997.

[17] Manish Parashar. GrACE–Grid Adaptive Computational Engine. See `http://www.caip.rutgers.edu/ parashar/TASSL/`.

[18] D. Brown, W. Henshaw, and D. Quinlan. Overture: An object-oriented framework for solving partial differential equations on overlapping grids. In *Proceedings of the First Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, 1998. See `http://www.llnl.gov/CASC/Overture`.

[19] M. Lemke and Daniel Quinlan. P++: A C++ virtual shared grids based programming environment for architecture-independent development of structured grid applications. In *Lecture Notes in Computer Science*. Springer-Verlag, September 1992.

[20] Rebecca Parsons and Daniel Quinlan. Run-time recognition of task parallelism within the P++ parallel array class library. In *Scalable Libraries Conference*, 1993.

[21] Scott B. Baden. The KeLP programming system. See `http://www-cse.ucsd.edu/groups/hpcl/scg/kelp.html`.

[22] Stephen J. Fink, Scott B. Baden, and Scott R. Kohn. Flexible communication schedules for block structured applications. In *Third International Workshop on Parallel Algorithms for Irregularly Structured Problems*, Santa Barbara, California, August 1996.

[23] R. Courant and K.O. Friedrichs. *Supersonic Flow and Shock Waves.* Springer-Verlag, New York, NY, 1985. Vol. 21 in the Applied Mathematical Sciences Series.

[24] A. Garcia. `http://www.algarcia.org/Gallery/index.html`.

[25] R. Hornung. `http://www.llnl.gov/CASC/SAMRAI/tech_brochure01/AppDev_Collab.html`.

[26] Alejandro Garcia, John Bell, William Crutchfield, and Berni Alder. Adaptive mesh and algorithm refinement using direct simulation Monte Carlo. *Journal of Computational Physics*, 154:134–155, 1999.

[27] Y. Y. Meshkov. Instability of the interface of two gases accelerated by a shock wave. *Fluid Dynamics*, 4:101, 1969.

[28] G.A. Bird. *Molecular Dynamics and the Direct Simulation of Gas Flows.* Oxford University Press, Inc., New York, NY, 1998. Vol. 42 in the Oxford Engineering Science Series.

[29] Alejandro Garcia and Berni Alder. Generation of the Chapman-Enskog distribution. *Journal of Computational Physics*, 140:66, 1998.

[30] F. Alexander and A. Garcia. Direct simulation Monte Carlo. *Computers in Physics*, 11:588, 1997.

[31] M. Dorr and X. Garaizar. `http://www.llnl.gov/CASC/alps/`.

[32] M. R. Dorr, F. X. Garaizar, and J. A. F. Hittinger. Simulation of laser plasma filamentation using adaptive mesh refinement. Technical Report UCRL-JC-138330, Lawrence Livermore National Laboratory, Livermore, CA, 2001.

[33] Stanley B. Lippman and Josee Lajoie. *C++ Primer.* Addison-Wesley Publishing Co., Menlo Park, CA, 1998.

[34] J. Cummings, J. Crotinger, S. Haney, W. Humphrey, S. Karmesin, J. Reynders, S. Smith,

and T. Williams. Rapid application development and enhanced code interoperability using the POOMA framework. In *Proceedings of the First Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, 1998. See `http://www.acl.lanl.gov/pooma`.

[35] L. M. Taylor, H. C. Edwards, and J. R. Stewart. Functional requirements for SIERRA version 1.0 beta. Technical Report SAND99-2587, Sandia National Laboratory, 1999. See `http://www.cfd.sandia.gov/sierra.html`.

[36] Matthew H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1998.

[37] Geoffrey Furnish. Container-free numerical algorithms in C++. *Computers In Physics*, 12(3):258–266, 1998.

[38] Jeremy Siek and Andrew Lumsdaine. The matrix template library: Generic components for high-performance scientific computing. *Computing in Science and Engineering*, 1(6):70–78, 1999.

[39] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. Curfman-McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high performance scientific computing. In *Proceedings the Eighth International Symposium on High Performance Distributed Computing*, 1999. See `http://z.ca.sandia.gov/~cca-forum`.

[40] Object Management Group. *The Common Object Request Broker: Architecture and Specification*. Available at `http://www.omg.org/corba`.

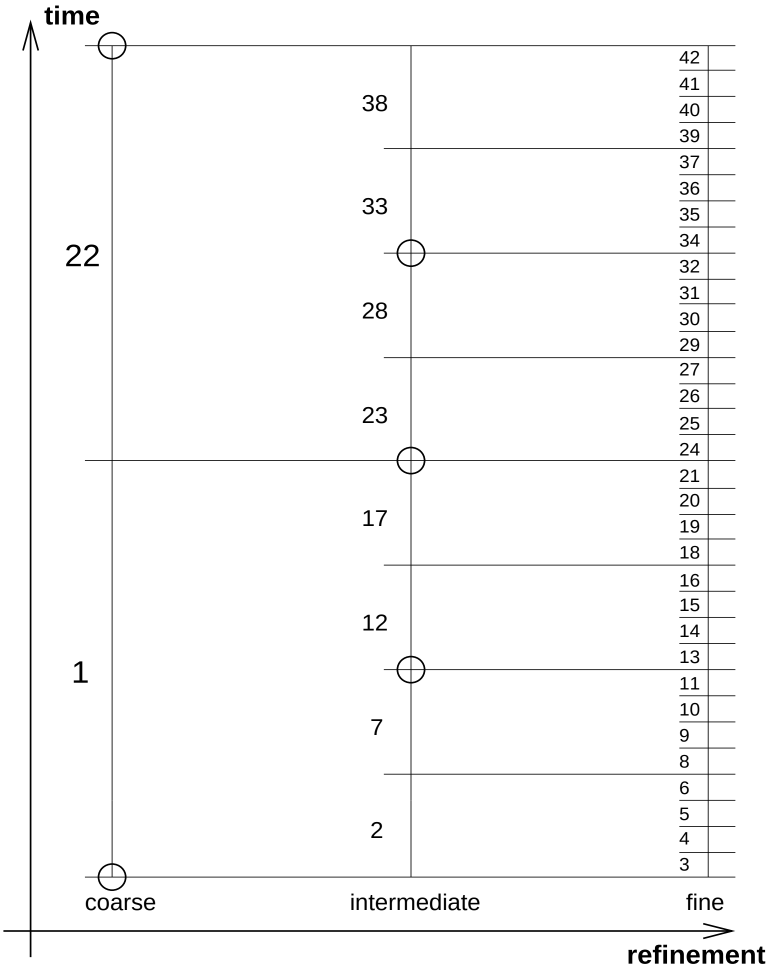

FIG. 1. *Example timestep sequence for three mesh levels (coarse, fine, and intermediate) and refinement ratio of 4. Horizontal lines indicate synchronization points between levels and circles indicate remeshing points.*

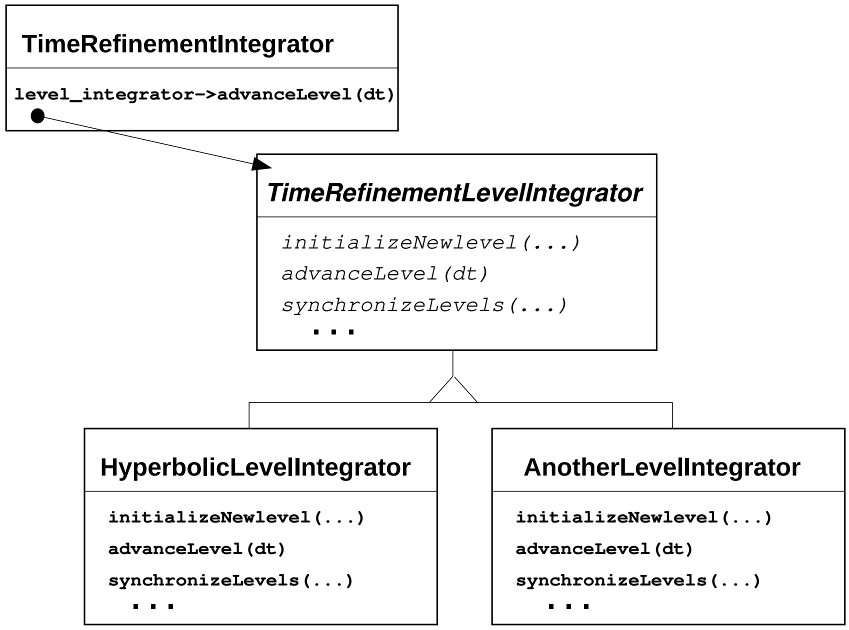

FIG. 2. *SAMRAI uses the* Strategy *pattern to define a family of time integration algorithms. The* `TimeRefinementLevelIntegrator` *abstract base class defines an interface between the* `TimeRefinementIntegrator` *object and problem-specific level integration objects. This diagram follows the OMT (Object Modeling Technique) notation [1]. Slanted type indicates abstract classes and methods, regular type indicates concrete objects and methods. Class inheritance is represented via a line segment with a triangle.*

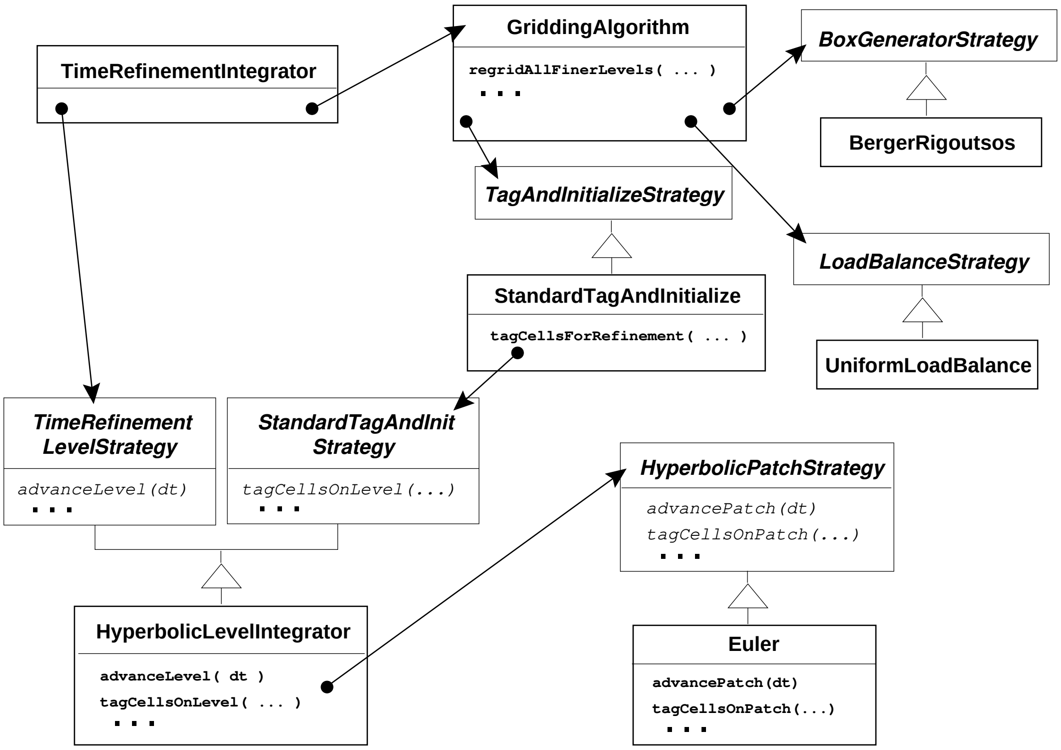

FIG. 3. *The complete Euler application is composed of various algorithmic and numerical components, each of which is further decomposed into other objects. Repeated use of the* Strategy *pattern captures this decomposition. Some key member functions are given in the figure to provide a glimpse into the functionality of some of the objects.*

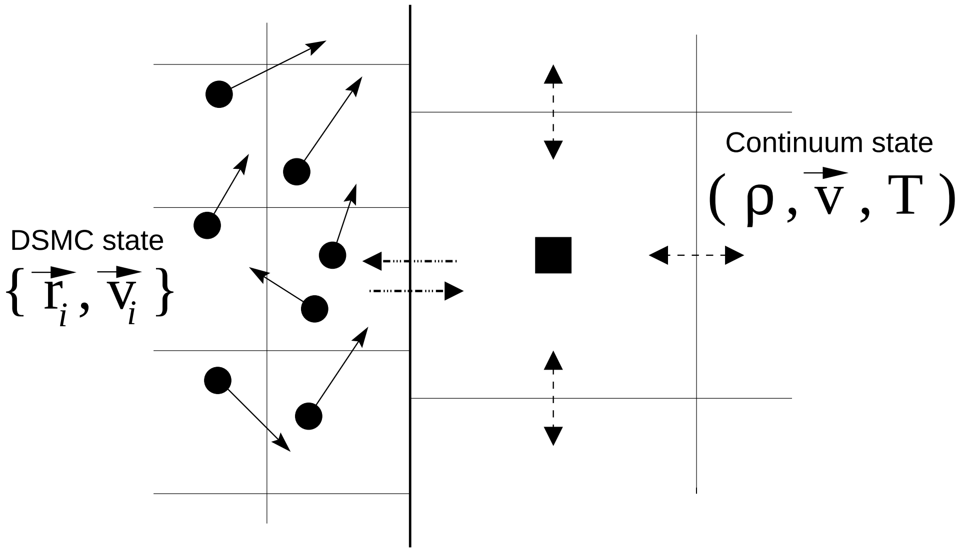

FIG. 4. *The Euler and DSMC models represent the state of the fluid in fundamentally different ways. Each DSMC particle is described by a position and velocity vector. The continuum model places density, velocity, and temperature values at the center of each cell. Proper physical coupling of the two descriptions requires that each method sees the same flux information at the interface.*

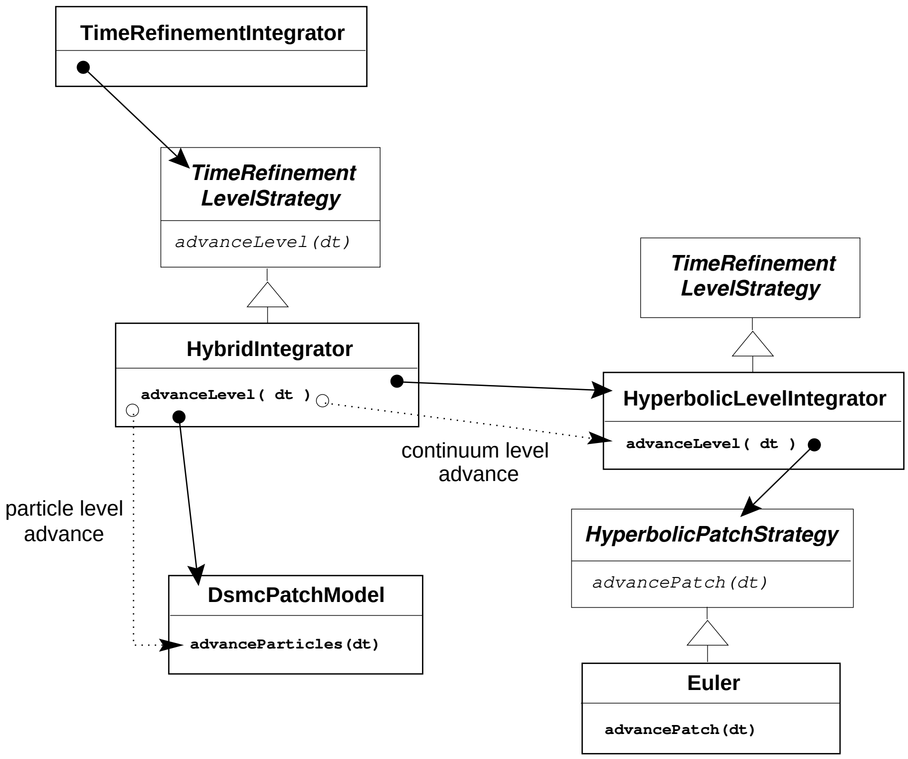

FIG. 5. *Many features of the class organization in the Euler code are reused in the Euler-DSMC hybrid code. The major difference is that the* `HybridIntegrator`*, which links the continuum and DSMC models, is now coupled to the* `TimeRefinementIntegrator`*. The meshing objects, which are identical to those in Figure 3 are omitted.*

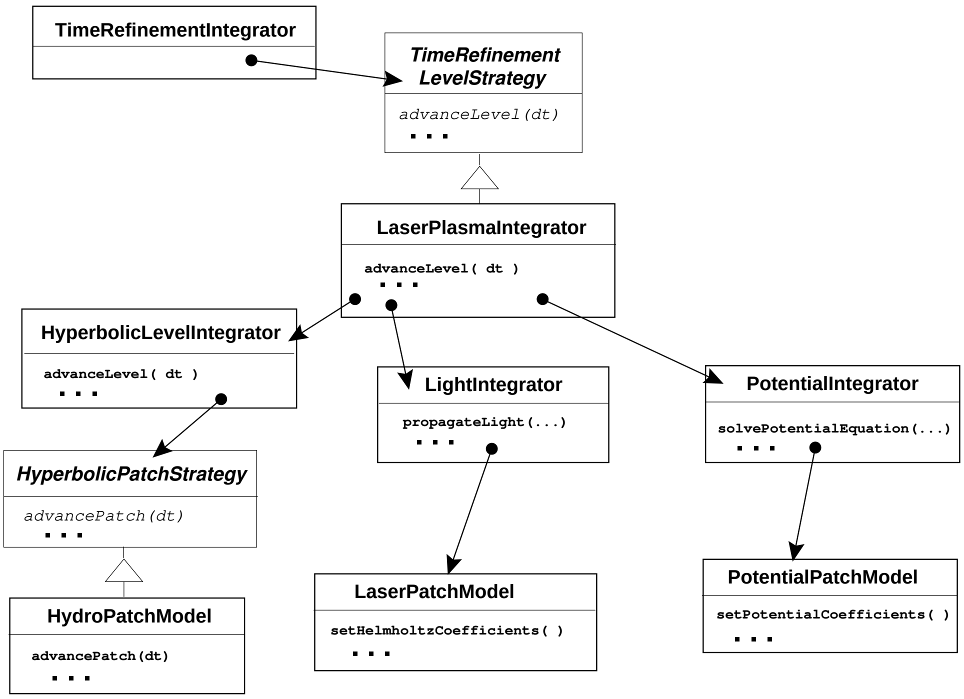

FIG. 6. *The ALPS code borrows time integration classes from SAMRAI and shares class organization concepts with other applications. The main integration object,* `LaserPlasmaIntegrator`*, coordinates plasma, laser light, and electrostatic potential integration on each mesh level. It is is driven by the* `TimeRefinementIntegrator` *class used in other applications. Hyperbolic algorithm classes in SAMRAI are also used for the plasma hydrodynamics. For brevity, we omit the meshing objects.*

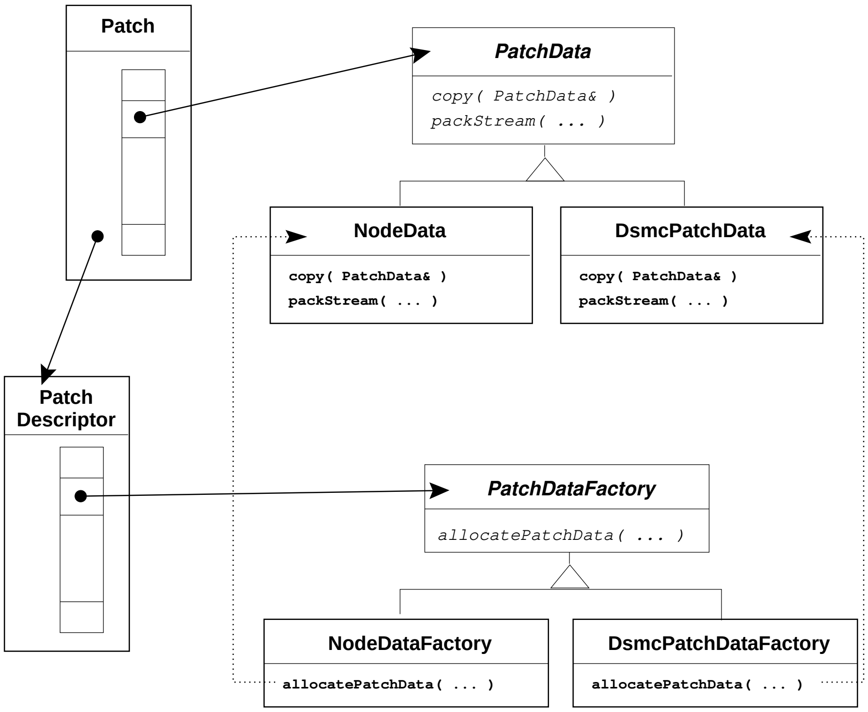

FIG. 7. *The* Abstract Factory *pattern manages the allocation of data for the SAMRAI patch hierarchy. Dotted lines indicate that subclasses of* `PatchData` *are created by associated subclasses of* `PatchDataFactory`*. In the Euler-DSMC application, the* `DsmcPatchData` *is introduced as a user-defined data type.*