Framework-Based Design of a New All-Purpose Molecular Simulation Application: The Adun Simulator

MICHAEL A. JOHNSTON,[1] IGNACIO FDEZ. GALVÁN,[2] JORDI VILLÀ-FREIXA[1]
[1]*Computational Biochemistry and Biophysics Laboratory, Research Group on Biomedical Informatics (GRIB), Institut Municipal d'Investigació Mèdica and Universitat Pompeu Fabra, C/Doctor Aiguader, 80 08003 Barcelona, Catalunya, Spain*
[2]*Departamento de Química-Física, Universidad de Extremadura. Avda. de Elvas s/n, 06071 Badajoz, Spain*

Received 20 May 2005; Accepted 20 June 2005
DOI 10.1002/jcc.20312
Published online in Wiley InterScience (www.interscience.wiley.com).

Abstract: Here we present Adun, a new molecular simulator that represents a paradigm shift in the way scientific programs are developed. The traditional algorithm centric methods of scientific programming can lead to major maintainability and productivity problems when developing large complex programs. These problems have long been recognized by computer scientists; however, the ideas and techniques developed to deal with them have not achieved widespread adoption in the scientific community. Adun is the result of the application of these ideas, including pervasive polymorphism, evolutionary frameworks, and refactoring, to the molecular simulation domain. The simulator itself is underpinned by the Adun Framework, which separates the structure of the program from any underlying algorithms, thus giving a completely reusable design. The aims are twofold. The first is to provide a platform for rapid development and implementation of different simulation types and algorithms. The second is to decrease the learning barrier for new developers by providing a rigorous and well-defined structure. We present some examples on the use of Adun by performing simple free-energy simulations for the adiabatic charging of a single ion, using both free-energy perturbation and the Bennett's method. We also illustrate the power of the design by detailing the ease with which ASEP/MD, an elaborated mean field QM/MM method originally written in FORTRAN 90, was implemented into Adun.

© 2005 Wiley Periodicals, Inc. J Comput Chem 26: 1647–1659, 2005

Key words: molecular simulation application; Adun simulator

Introduction

It may be a truism to say that computational science is focused on the development of new algorithms and procedures to produce accurate scientific results. However, this statement logically leads to the conclusion that software development in computational science will tend to be "Algorithm Centric," that is, a program is a vehicle that enables the implementation of an algorithmic solution to some problem. In this paradigm issues of algorithm speed and efficiency take a front seat, and frequently productivity is measured in terms of program run-time with a certain parameter set. A procedural methodology dominates regardless of whether the underlying language is procedural or object-oriented. This is a valid approach for small well-focused projects where it is unlikely that the program will undergo much revision after a certain goal is reached.

However, as the use of computational methods expands across all disciplines the need for more powerful and advanced programs is becoming evident. Such programs are inevitably larger, display increased complexity and undergo continuous development, usually by more than one person. Speed is no longer the main metric of productivity as the competing factors of development time and maintainability begin to dominate. New algorithms and better implementations are published monthly, and it is desirable to implement them rapidly while they are still the "state of the art." The long life cycle of these programs means members regularly join and leave the development team, and so it is also desirable that new members can come to grips with the program in the minimum time possible.

Unfortunately, an algorithm-centric approach cripples the development of such programs, for example, Molecular Simulators, by raising the costs associated with maintainability and implementation time to exorbitant levels (Fig. 1). Focusing exclusively on algorithms leads to a rigid program structure with variable names,

Correspondence to: J. Villà-Freixa; e-mail: jvilla@imim.es

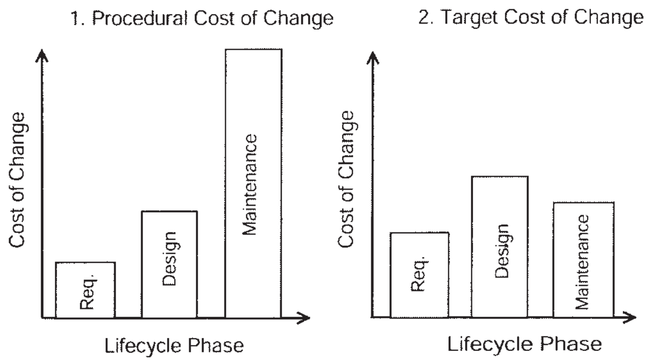


Figure 1. Procedural programming methods can lead to huge time costs in large programs. By taking a higher level view and concentrating more on the requirements (Req. in the figure) and design of the program we can dramatically reduce the cost of maintenance (addition of new code, modifying old, training new developers, etc.)

structure types, code groupings, data flow, etc., heavily tailored to currently implemented algorithms. The flow of control is often unclear, and hidden dependencies are common, causing bugs to proliferate. For a new developer understanding this structure presents an almost insurmountable burden, and even for experts coding new functionality into such a structure is usually a painstaking and time-consuming task. Finally, if an algorithm-centric approach is maintained code entropy accelerates because the desire to keep the original algorithms optimal structure intact and the competing desire to optimize the new algorithm leads to code whose separate parts use different structures for the same type of data.

Essentially, the traditional approach fails to take into account the fact that algorithms form only a low level part of a large, complex programs structure.

New Methods

To produce code that allows rapid development and remains easy to maintain we need to shift our view away from algorithms alone and take into account the structure of the program. This process is greatly aided by the proper application of object-oriented techniques (OOP) and concepts. The process of breaking code into objects that combine data and functionality intrinsically adds better structure to a program. These objects, however, must be encapsulated—the inner workings of an object should be irrelevant, only its methods and the data it provides matters. Encapsulation enables code reuse, one of the main aims of OOP and a key factor in decreasing development time.

The process of breaking code into objects, called factoring, is not always obvious, and may be done in many equally valid ways. Design patterns[1] assist developers in this task by providing effective solutions to common problems in OOP design. However, as programs grow requirements change and unseen problems develop. It is thus important to monitor the programs structure, making small frequent changes to constantly evolve the design. This practice is known as refactoring, and first achieved prominence with the development of extreme programming.[2,3]

Probably the most powerful concept enabled by object orientation is polymorphism, the idea that objects that provide the same services (i.e., perform the same functions) should provide the same interface (Fig. 2). This enables one to be switched to another without affecting the rest of the program. Polymorphism also requires that the return types and arguments to these methods should be the same (or at least themselves be polymorphic). Indeed, for a certain type of (nonobject) data the same data structure should be used all over the program. It is not much use if the objects in the program don't "talk" the same language, that is, they operate on and provide different data structures so they then require a translating layer to cooperate.

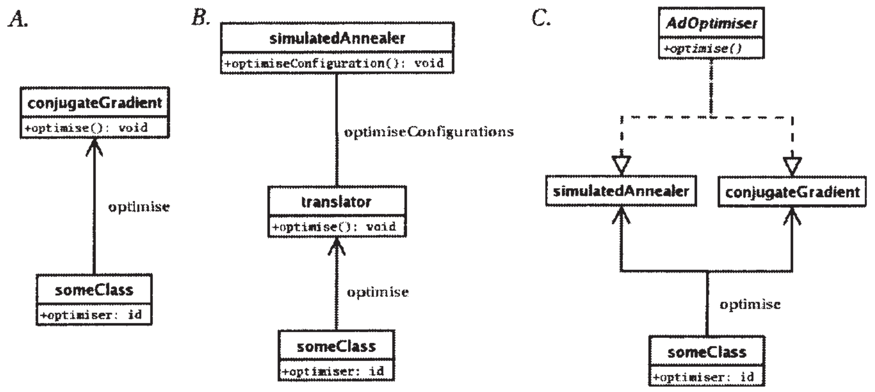


Figure 2. Once an object uses a certain name to send a message, only objects that have a method with that name can respond, that is, the object is coupled to a certain interface (A). When another object is available that performs the same service but with a different name it cannot be used without a translation layer (B). If the objects had the same name they would be polymorphic, and could be used interchangeably. We enable polymorphism by defining one uniform interface for objects that provide a certain service (C).

Frameworks

When different applications are developed in the same domain using the preceding methods, the same objects, interactions, and data types often occur over and over. The codifying of these standard objects, their interfaces, how they communicate, and the data types passed between them is the goal of a framework.[4]

An excellent definition of a framework can be found in the seminal program design book "Design Patterns."[1]

> [a framework] dictates the architecture of your application. It will define the overall structure, its portioning into classes and objects, the key responsibilities thereof, how the classes and objects collaborate, and the thread of control. A framework predefines these design parameters so that you . . . can concentrate on the specifics of your application. The framework captures the design decisions that are common to its application domain.

In essence, a framework identifies the building blocks of these programs and how those blocks interact.

When a framework exists for a certain domain developers can use it to create applications. Initially (when the framework is new) developers create all the necessary classes themselves based on the framework guidelines. Classes created in this manner are called components, and are usually collected into a component library. Once this library reaches a certain state, indicated by the number and diversity of the components it contains, it is used as the basis on which all the applications in the domain are built, thus enabling code reuse. Because the classes in the library have been designed according to the framework, they intrinsically work together. Making a new application is then as simple as combining the components in new ways or using a number of existing components and creating others for functionality the library does not provide.

By using a framework we gain a number of other advantages as well. It makes creating new components easier by providing specifications and templates for implementing them—when we want to add new functionality we simply identify the correct framework class, which tells us the interface we must provide and what information we have access to, and encapsulate our code. The problem of objects making different assumptions about their environment is removed because the framework defines standard ways to exchange data and access services.

Frameworks not only aid development but also maintenance. By their very nature they maintain code stability and reduce the process of code-entropy by orders of magnitude, and the rigorous guidelines they embody are invaluable for open source projects where large numbers of people can contribute to the code base.

Adun Overview

Adun has been designed, following the above considerations, as a platform to develop the multiple types of simulations that are central in computational (bio)chemistry and that in particular represent the main lines of research of our own group. These include all simulation levels, that is, micro (all atom molecular dynamics or Monte Carlo), meso (dissipative particle dynamics and related methods), and macro (whole cell simulations). Initial development of the program is concentrated on molecular dynamics (MD) applications while keeping in mind the final goal of an all purpose simulator. In the case of MD, Adun basically reads in topology information regarding a system and then, by evaluating the systems potential energy and the corresponding forces, outputs trajectories detailing the systems evolution through time. These trajectories can then be analyzed using the ideas of statistical mechanics to attain measurements of free-energy differences and other important quantities.

A simulator is an archetype of the complex applications now required by the scientific community. There are a host of different methods and protocols that can be used to generate MD trajectories, and the state of the art in simulation techniques changes rapidly and continuously. In addition, the simulation requirements can differ dramatically from lab to lab, and it is impossible for a single developer or lab to satisfy them all. Therefore, we require a platform that will enable rapid and easy implementation of yet to be determined functionality while simultaneously avoiding burdening developers with a convoluted and hard to understand program.

The basis of Adun derives from the observation that the different varieties of MD simulations are essentially variations of one fundamental MD design. This fact, coupled with our requirements, immediately suggests that a MD framework would provide the necessary foundation for our application. To this end we have developed the Adun Framework, which provides a coherent design for any MD simulator and confers all the advantages noted in section 1.

Implementation

The framework and Adun itself have been implemented in Objective-C, which provides Adun with a versatility that would be otherwise extremely hard to attain. Foremost, Objective-C is a dynamically typed language. This is a key factor, as it enables pervasive polymorphism, which is the cornerstone of the Framework, through dynamic-typing and binding—features not available with C++ or Fortran. This run-time dynamism also provides the basis for the controller class (see later) methodology as well as allowing other features such as distributed objects and run-time class extension through categories.

In addition, Objective-C is a pure superset of C. It consists of a thin layer over standard C, which not only makes it very easy to learn for anyone familiar with C, but also means we can use C to code the low-level algorithms where performance is the highest priority as well as use scientific C libraries such as GSL (Gnu Scientific Library).

Furthermore, we gain access to the powerful GNUStep (Cocoa) application development framework. These are a set of (GPL) object-oriented Objective-C libraries that provide many basic object types, advanced OO cabitilies (Key-Value coding, notifications, object serialization, etc.) and impressive GUI building facilities. Finally, the standard Objective-C compiler is the ubiquitous GCC enabling Adun to compiled on any UNIX platform as well as Mac OS-X (where Objective-C and Cocoa are the native language and libraries, respectively).

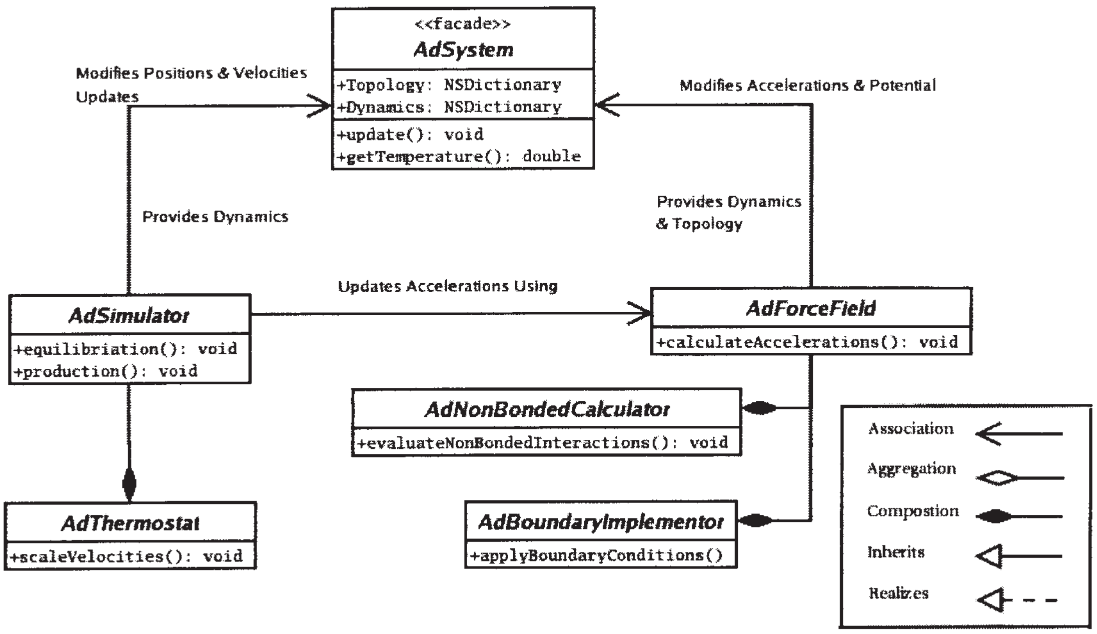


Figure 3. Fundamental structure of a molecular dynamics simulator in the Adun framework. The italicized names denote that the classes are abstract. The prefix Ad- helps avoid namespace conflicts.

Current Features

Using this framework-based development model we have easily implemented some typical and some specific features into Adun. Among them, the surface-constraint all-atom solvent model (SCAAS)[5] and the Langevin dipoles model[6] for representing solvent; the Enzymix force field;[7] velocity-verlet Newtonian and Langevin dynamics;[8] free-energy perturbation (FEP);[9] and Bennett's[10] methods for the evaluation of free energies as well as the ASEP/MD[11,12] method for QM/MM—while implementation of other algorithms continues rapidly. Below we detail how the framework was created using the previously discussed ideas, how it provides the basis for Adun itself, and how it enables new functionality to be implemented. As a precursor to the future work we will perform with Adun we present the results of some preliminary simulations.

Adun Framework Design

Although frameworks are powerful tools they cannot be designed from scratch without previous expertise in the problem domain (in this case MD). Without this experience it is impossible to identify the optimal factoring or to define how the resulting classes should communicate. Therefore, the development of a framework is an evolving process. It usually starts with the creation of a program in the given domain. As subsequent programs are developed certain trends become clear. These trends are coded in the first version of the framework and the initial programs are refactored to conform to it and become the basis of the component library. Newer programs are then based on this framework and library. As time passes, constant refactoring is applied as problems are encountered, thus improving the design and driving the framework towards a "meta-stable" mature state. This was the process followed to create the Adun Framework, and it was accelerated by the fact that molecular dynamics is a very well-known problem domain.

Framework Structure

An overview of the fundamental structure of the Adun Framework can be seen in Figures 3 and 4. It is defined by a set of abstract classes (also called base classes) and the relationships between them. An abstract class is a template for creating components. It defines the interface (method names and return types) and responsibilities each component based on it must conform to. A component associates itself with an abstract class through inheritance. Thus, abstract classes group-related components enabling a taxonomic view of the component library as well as ensuring that all their descendants are polymorphic.

The arrows in Figures 3 and 4 show which classes are related, and the type of arrow indicates the type of relationship. The relationships are grouped into three categories—association, aggregation, and composition. Composition is the strongest relationship—one class exclusively contains another and the presence of the contained class is usually required in order for the container to function, for example, AdForceField (container) and AdNonBondedCalculator. It also implies a shared life time, that is, if the container is destroyed then so are the contained objects.

Aggregation also implies a container/contained relationship and a shared life time. However, the presence of the contained object may not be necessary for the container to function, and the containment may not be exclusive. A standard array or list object in any object-oriented language has this relationship with the

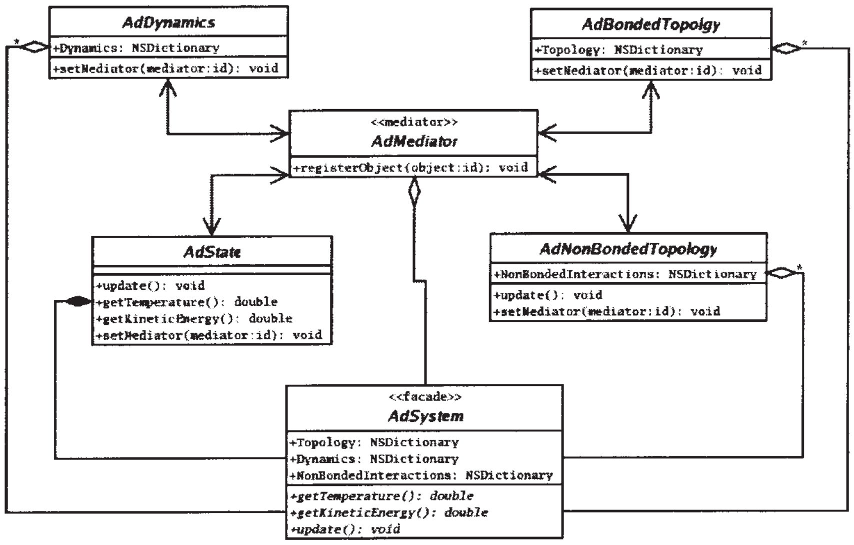

AdDynamics
+Dynamics: NSDictionary
+setMediator(mediator:id): void
AdBondedTopolgy
+Topology: NSDictionary
+setMediator(mediator:id): void
<<mediator>>
AdMediator
+registerObject(object:id): void
AdState
+update(): void
+getTemperature(): double
+getKineticEnergy(): double
+setMediator(mediator:id): void
AdNonBondedTopology
+NonBondedInteractions: NSDictionary
+update(): void
+setMediator(mediator:id): void
<<facade>>
AdSystem
+Topology: NSDictionary
+Dynamics: NSDictionary
+NonBondedInteractions: NSDictionary
+getTemperature(): double
+getKineticEnergy(): double
+update(): void
*
*
*

Figure 4. The System object acts as a facade and container for the system subsection of the framework.

elements it contains. Finally, in association there is no containment. There is one or two way communication, but if one is destroyed the other is not affected, for example, AdForceField, AdSystem, and AdSimulator.

Class Overview

The three core classes in the framework are listed in Table 1, along with their responsibilities. The stereotype ⟨⟨facade⟩⟩ on AdSystem indicates it is actually an interface for the System subsystem of Adun (Fig. 4).

A subsystem is a self-contained set of functionality and is typically made up of multiple components. However, because the subsystem components will have differing interfaces the overall interface can become very complex. To solve this problem we use the *facade* pattern. One class (the facade class) contains the subsystem classes and provides a unified interface to them, thus combining multiple sources of information and functionality into one.

In the case of Adun, AdSystem is the container for the subsystem classes (see Table 2), which it manipulates to provide data to AdForceField and AdSimulator. It not only provides a simpler interface to the subsystem but also eliminates all coupling of the other core classes to the exact subsystem composition making it very easy to switch one system type for another. Components based on AdSystem contain different varieties and amounts of the subsystem classes and manipulate them in different ways. Because there is no coupling between the exact composition of each component and the rest of the framework objects we are free to vary, combine, and manipulate the subsystem classes in any imaginable way, which leads to great developmental flexibility.

Evolving Framework Example: AdForceField

The overall structure presented in Figures 3 and 4 was reached by the application of the concepts discussed in the introduction, that is, polymorphism, refactoring, patterns, etc., and the evolution of the AdForceField class serves to illustrate and clarify this procedure.

Initially, AdForceField was a pure abstract class. It contained no functionality of any kind and simply acted as a template for force field components. A number of concrete implementations were made of this class as shown in Figure 5A. These corresponded to the same force field function but with different combinations of methods for computing the nonbonded interactions and applying boundary conditions.

Table 1. The Top Level Classes.

Class	Responsibility
AdSystem	The force field and otential evaluation algorithm
AdForceField	The force field and potential evaluation algorithm
AdSimulator	The integration algorithm

Table 2. The System Subsystem Classes.

Class	Responsibility
AdDynamics	coordinates, velocities, positions
AdBondedTopology	object to handle bonded interactions
AdNonBondedTopology	object to handle nonbonded interactions
AdState	monitors the state of the system
AdSolventBox	defines how the solvent is contained
AdMediator	controls communication between all parts of the system

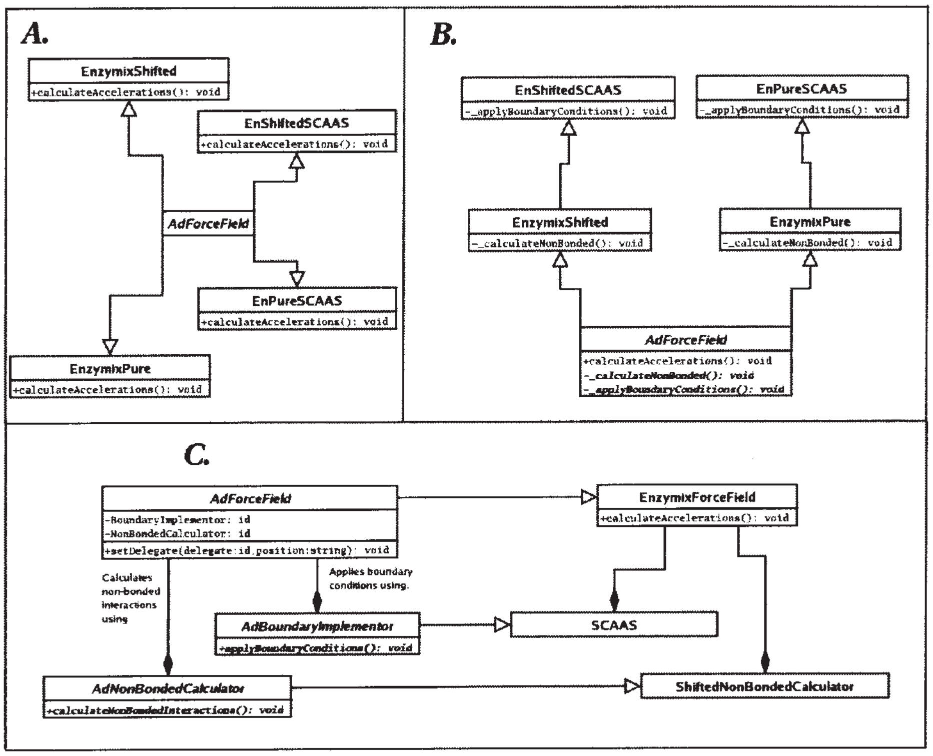

A.
EnzymixShifted
+calculateAccelerations(): void
EnShiftedSCAAS
+calculateAccelerations(): void
AdForceField
EnPureSCAAS
+calculateAccelerations(): void
EnzymixPure
+calculateAccelerations(): void
B.
EnShiftedSCAAS
-_applyBoundaryConditions(): void
EnPureSCAAS
-_applyBoundaryConditions(): void
EnzymixShifted
-_calculateNonBonded(): void
EnzymixPure
-_calculateNonBonded(): void
AdForceField
+calculateAccelerations(): void
-_calculateNonBonded(): void
-_applyBoundaryConditions(): void
C.
AdForceField
-BoundaryImplementor: id
-NonBondedCalculator: id
+setDelegate(delegate:id,position:string): void
EnzymixForceField
+calculateAccelerations(): void
Calculates non-bonded interactions using
Applies boundary conditions using.
AdBoundaryImplementor
+applyBoundaryConditions(): void
SCAAS
AdNonBondedCalculator
+calculateNonBondedInteractions(): void
ShiftedNonBondedCalculator

Figure 5. AdForceField evolving from basic, to Template Method, to Strategy. In (A), the abstract class contains no code, and hence all changes must be repeated many times. In (B), this problem is removed; however, now algorithm reuse is a problem; the large hierarchies created by inheritance are hard to maintain and class naming becomes a problem. In (C), we apply the "Strategy" pattern and achieve a very flexible polymorphic composition.

However, the algorithm encapsulated by each of the classes was generally the same, for example, the same bond, angle, and torsion operations. Therefore, if any of these areas was changed in one it had to be changed in them all. This not only makes changes take longer but also acts as a source of hard to find bugs due to "transcription" errors. This problem and its solution are addressed by the *template method* pattern. It proposes that the common code should be moved to an abstract class with empty "hook" methods replacing the varying code. Using inheritance, subclasses then just implement their own version of these hook methods (see Fig. 5B).

As development continued it became obvious that the hook methods were encapsulating generic categories of algorithms. Because they were defined as part of a class these algorithms could not be used outside of the class hierarchy. Each algorithm was also coupled to the force field class it was defined in. This meant that you could not change, for example, one nonbonded calculation technique for another without changing the entire class. Most importantly, every combination of an algorithm with another meant another class had to be created. For example, each nonbonded technique added means two classes have to be created, with and without SCAAS. If there are four boundary conditions and four nonbonded techniques 16 classes must be created. Adding another base force field type, that is, CHARMM would double this number.

These problems are addressed by the *Strategy pattern.* The solution is to encapsulate each algorithm by its own base class and then use delegation and dynamic binding to vary the algorithm (Fig. 5C). Thus, we enable these algorithms to be reused elsewhere, decouple different layers of functionality, make it much easier to create new functionality, and drastically reduce the number of classes needed. Now we only need 4 (nonbonded techniques) + 4 (boundary conditions) + 2 (force field) = 10 classes to achieve the same effect as the 32 classes with Template Method.

Template Method vs. Strategy

Template Method and Strategy are closely related. Strategy is more powerful but also more complex to implement. It requires the language used to support dynamic typing and binding. In addition, it is not always clear at the beginning how to encapsulate the objects. Template Method is quicker and easier but lacks flexibility. It is often the case that Template Method is used first and then it evolves to become a Strategy if necessary.

These two patterns are usually applied to the case of algorithms. However, the ideas they introduce are cornerstones of every framework. Template Method uses inheritance to extend functionality, while the method embodied by Strategy is called polymorphic composition. Another example of a polymorphic

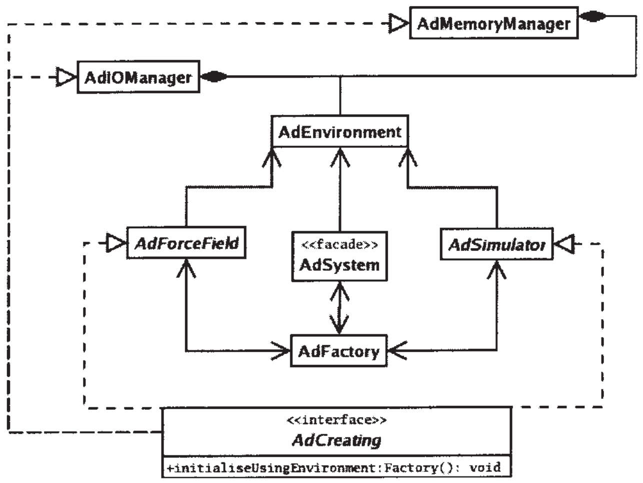


Figure 6. The support classes of the Adun Framework. All classes, except Environment and Factory, implement the AdCreating interface (The arrow to AdSystem was removed to enhance clarity. The core classes can also access AdIOManager and AdMemoryManager and utilize their services.)

composition are the core objects of Figure 3. These issues are discussed further in the development section.

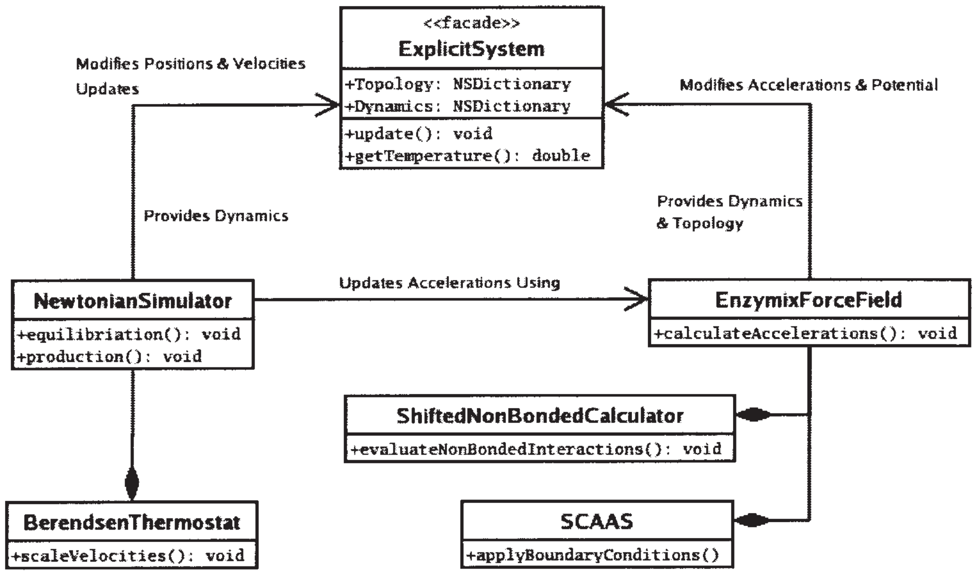
Completing the Framework: Support Objects

The framework as presented so far is incomplete. All MD programs need to input and output information, allocate and manage memory, get information on the processing environment, etc. In addition, each object will require a number of external parameters to function, for example, the time step, the number of steps, the long-range interactions cutoff, if any. These various functions are encapsulated by support objects, and their relationship with the core objects is shown in Figure 6. AdEnvironment is the main class. It contains all the information on the processing environment as well as the configuration values for the program. Every class in the framework (and, hence, all their descendants) contain a reference to AdEnvironment. AdEnvironment contains two other support classes, AdIOManager and AdMemoryManager, so they can be accessed by any object that needs their services.

Building The Program

Up to now we have detailed the Adun framework that describes any molecular dynamics application. By joining together components based on the framework we can create an MD program as in Figure 7. However, we do not want many separate applications but one root application that can create them all. Therefore, we require a way in which we can dynamically assemble the proper components into an application based on the value of certain options specified by the user. The AdFactory class is what allows us to do this. Every class contains a reference to AdFactory. When it wants to create an object based on the user values it calls the appropriate AdFactory method, for example, `createForceField`, `createSystem`, etc., which then instantiates and returns the correct object based on the current environment. This technology is enabled by polymorphism and dynamic binding and would be impossible otherwise.

Figure 8 shows the final structure of the Adun program. The AdCore class assembles and contains the top level objects that are created using AdFactory and AdEnvironment. The actual manipulation of the top-level objects is performed by an AdController class, which is dynamically loaded based on the Environment options. The controller classes are actually *bundles,* and are not compiled into the main part of the program. They can be viewed



<<facade>>
ExplicitSystem
+Topology: NSDictionary
+Dynamics: NSDictionary
+update(): void
+getTemperature(): double
Modifies Positions & Velocities Updates
Modifies Accelerations & Potential
Provides Dynamics
Provides Dynamics & Topology
Updates Accelerations Using
NewtonianSimulator
+equilibriation(): void
+production(): void
EnzymixForceField
+calculateAccelerations(): void
ShiftedNonBondedCalculator
+evaluateNonBondedInteractions(): void
BerendsenThermostat
+scaleVelocities(): void
SCAAS
+applyBoundaryConditions()

Figure 7. A concrete molecular dynamics application in the Adun Framework.

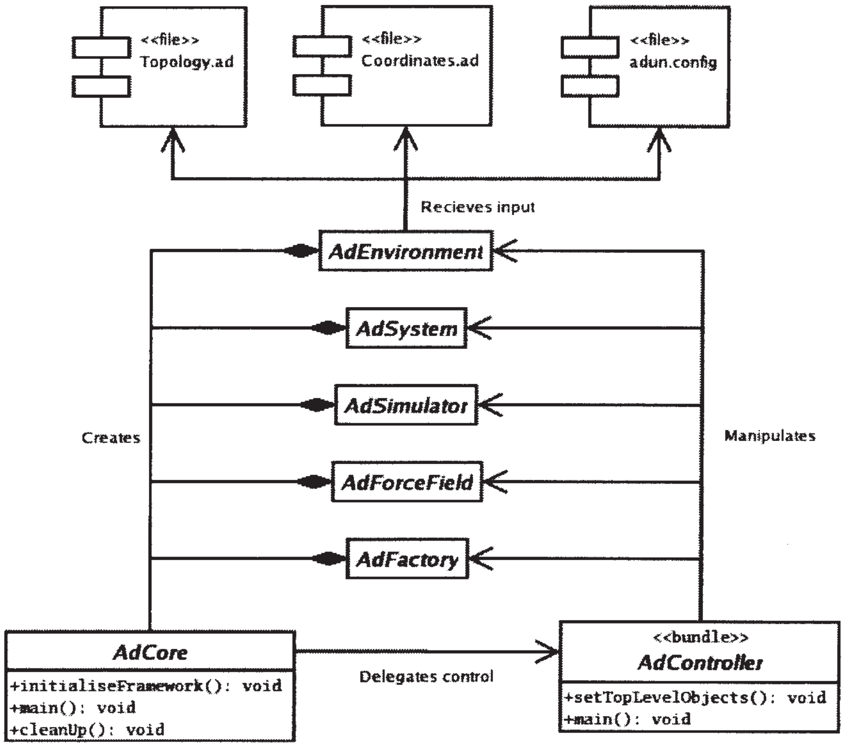


Figure 8. The structure of Adun. Currently, Adun reads in topology and coordinate information from files prepared from a script. In the near future this information will be sent directly to the Core from the Adun UserLand as data objects.

as plugins or scripts and provide one of the most powerful ways in which Adun can be developed.

Development

The techniques and methods of molecular simulation can be divided into two distinct types. On one hand, we have low-level algorithms associated with force calculation and application, numerical simulation, boundary effects, nonbonded interactions, etc. On the other, there are higher level simulation protocols such as free-energy perturbation, LIE, PDLD, and replica-exchange–based methods. Reflecting this division development of Adun takes place on two different levels: extending the component library to add functionality, and creating new controllers that utilize existing components in different ways.

Components

The parts of the framework where development is concentrated are called "hot spots," and the majority of components in the component library are associated with these areas. A hot spot can be one class or encompass a group of classes. In the latter case, the classes usually form a polymorphic composition. The Adun hot spots are detailed in Table 3.

Hooks

For each hotspot in Adun, the framework defines where we can add our code. Because we are "attaching" our code to these locations they are called hooks, and how we "attach" it depends on the type of hook, *method* or *class. Method hooks* are found in single-class hotspots, while *class hooks* are related to polymorphic compositions.

Method hooks are a generalization of the "Template Method" pattern discussed previously (see earlier) to areas other than algorithms. An abstract class leaves certain methods unimplemented (the method hooks) and subclasses then fill in these "holes" with different functionality. If a class has many method hooks an inheritance tree usually grows up (as with AdForceField).

Class hooks, on the other hand, are related to the Strategy pattern. Here we create a new subclass of a class in a polymorphic composition, for example, if we wished to add a new method for computing the nonbonded interactions we would create a new AdNonBondedCalculator subclass. Our algorithm or method can be used by simply placing an instance of the new class in the correct place in the composition.

Controllers

Developing controllers differs from developing components because the emphasis is on creating new simulation protocols rather

Table 3. Some Adun Hot Spots and the Types of Hooks Associated With Them.

HotSpot	Variability	Hook types
AdSimulator	Integration, temperature control methods	Method
AdNonBondedCalculator	Nonbonded calculation techniques	Class
AdBoundaryImplementor	Boundary conditions	Class
AdSystem	Systems	Class
AdNonBondedTopology	Nonbonded interactions calculation	Class/method
AdState	Controlling state	Class
AdSolventBox	Solvent geometry	Class

than new functionality. Controller classes manipulate components by changing parameters, polymorphic compositions, etc. By separating the functionality of the objects from how they are actually used we attain a flexible and easy way for developers to implement new behavior.

Controller classes are loaded from bundles, and as such are not a directly compiled part of the program. A controller bundle can just contain the controller class itself or can also define other classes, outside of the framework, which are loaded with it at run time. These classes can then be used to perform operations not available in the Adun framework without affecting it.

Thus, a controller can range from a few simple lines to a number of large complex objects. External objects defined in this way can then be added to the component library if they prove useful. For example, new AdForceField or AdSystem components can be defined in controller bundles for testing, and the controller can replace the initial component created at startup with the new one.

Controllers provide a much cleaner and clearer way to perform various protocols. They keep such code separate and prevent code entanglement. Because controllers are compiled separately from the core program it does not have to be recompiled when they are changed, and useful controllers can be distributed and used with any installed Adun.

Future Developments

Adun is an evolving application, and features are constantly being added and improved. These features can be roughly divided into two categories: (a) simulation algorithms and protocols, and (b) higher level interface and deployment functionality. On the simulation side we are currently implementing a local reaction field[13] treatment for the long-range electrostatic interactions, the LIE protocol,[14,15] configuration optimization methods, as well as new force field implementations. The later is aided greatly by our development of an XML-based template for describing force fields coupled with a flat file to XML conversion tool. This will allow rapid implementation of all current force fields.

On the other hand, we are improving our basic interface system for the program (Adun UserLand). This part of the program contains the topology generation code as well as a simple user interface, allowing creation, saving, and loading of Adun topology and option files and the ability to load statistical analysis plugins. By keeping the core simulation code and the UserLand separate and utilizing the built-in distributed objects features of the Cocoa/GNUStep libraries we can enable such novel technologies as dynamic and remote simulation management as well as intrinsic grid computing facilities. The possibility of ready-to-go grid provides an elegant solution to the many so-called *embarrassingly parallel* problems (problems that can be easily split into a bunch of several independent threads) that are common in the molecular dynamics domain.

Examples

As a short example of Adun and a precursor to future work we present results for the solvation energy of a sodium ion and compare the performance of the standard free energy perturbation[9] technique and the Bennett acceptance ratio[10,16] method in the equilibrium case.

System

The system studied involves a sodium ion surrounded by a sphere of flexible water molecules simulated by the SCAAS model and the ENZYMIX force field. The radius of the water sphere was 12 angstroms, and the electrostatic cutoff was 26, ensuring all interactions were treated explicitly. Two different values for the number of mapping steps (10 and 20) were used to see the effect of increasing the overlap between each perturbation surface. The duration of each mapping step was also varied so we could examine the effect of increased sampling. The durations were 100 fs, 1 ps, and 5 ps, and the corresponding number of samples taken was 10, 200, and 1000, respectively. Thus, the shortest total simulation was 1 ps and the longest 200 ps. The time step was 1 fs and the temperature of the system was kept at 300 K using a Berendsen thermostat.

Free Energy Perturbation

FEP estimates free energy differences by exponentially averaging potential energy differences between a reference state sampled at equilibrium and a target state.[9]

$$G_B - G_A = -RT \log\langle e^{-(\Delta H/RT)}\rangle_A \tag{1}$$

In the case where the potential energy surfaces of the initial and target state do not have significant overlap we can break the calculation into windows by defining a Hamiltonian $H(\lambda)$

$$H(\lambda) = \lambda H_B + (1 - \lambda) H_A \tag{2}$$

λ can vary from 0–1. When $\lambda = 0$, $H(\lambda) = H_A$ and when $\lambda = 1$, $H(\lambda) = H_B$. Equation (1) then becomes

$$G_B - G_A = \sum_{\lambda} - RT \log\langle e^{-(\Delta H'/RT)}\rangle_\lambda \tag{3}$$

where $H' = H_{\lambda+d\lambda} - H_\lambda$. The total free energy difference is just the sum of the free energy differences between the windows defined by λ.

Bennett Acceptance Ratio

Taking infinitely small values for $d\lambda$ ΔG should be the same if we go from 0–1 or from 1–0. However, in actual calculations this is not the case, and it is often found that it is not, that is, the FEP estimator exhibits hysteresis. One common way to deal with this is to take the average of the forward and backward perturbation; however, the validity of this procedure is doubtful. Also, the variance of the FEP estimator can be quite large, leading to poor precision.[17,18] An alternative method for obtaining ΔG from two states sampled at equilibrium was proposed by Bennett.[10,16] By utilizing the information contained in both the forward and reverse distributions of the potential energy difference he showed that a significantly better estimate could be obtained than by just using measurements in one direction. He first showed that, for an arbitrary function f,

$$e^{-\beta\Delta G} = \frac{\langle f(\Delta U)\rangle_F}{\langle f(\Delta U) > e^{-\beta\Delta U}\rangle_R} \tag{4}$$

where $\langle\ \ \rangle_F$ is the average over the forward direction and $\langle\ \ \rangle_R$ the average over the reverse. He then minimized the statistical variance with respect to $f(\Delta U)$ to find that

$$f(\Delta U) = \left[1 + \frac{n_f}{n_r} e^{(\Delta U - \Delta G)}\right]^{-1} \tag{5}$$

Here, n_f and n_r are the number of values obtained for the forward and reverse directions, respectively. Although this is an implicit function of ΔG the result can easily be found by iterative methods, for example Newton–Rhapson. With the recent discovery of the nonequilibrium work relation (Jarzynski equality[19])

$$e^{-\beta\Delta G} = \langle e^{-\beta W}\rangle \tag{6}$$

where W is the nonequilibrium work in bringing the system from one state to another, Crooks showed[20] that eq. (4) can be trivially generalized to the nonequilibrium case by replacing ΔU with W. Pande and Shirts[21] have recently shown that results obtained from the Bennett's Method can be seen as the most probable value of ΔG given the values of ΔU obtained. They also showed that the variance of this estimate is given by,

$$\frac{1}{\beta^2 n_{\text{tot}}}\left\{\left[\left\langle\frac{1}{2 + 2\cosh(\beta(M + W_i - \Delta F))}\right\rangle\right]^{-1} - \left(\frac{n_{\text{tot}}}{n_f} + \frac{n_{\text{tot}}}{n_r}\right)\right\} \tag{7}$$

Results and Discussion

Table 2 shows the results obtained for the adiabatic charging of a sodium ion using the standard FEP approach and Bennett's method. Both FEP and Bennett's method calculate the free energy from the same data, that is, from sets of potential differences between a reference state at equilibrium and a target state. Some confusion could arise here because this method of obtaining data is often synonymous with FEP. Here, FEP refers solely to the exponential averaging of such data from either the forward or reverse directions, while Bennett's method incorporates the data from both directions in eq. (4).

As expected, both the acceptance ratio method and the FEP average yield similar results for the free energy of the charging process, even for calculations with very limited sampling. Note, however, that although the (in)accuracy of the results for the different MD experiments is quite similar, the precision is significantly improved by the application of the Bennett method (varying from a two- to fivefold decrease in the standard deviation).

On the other hand, a high degree of hysteresis is observed for the FEP estimator. This is a well-known problem,[22] and it can be negated by increasing the number of mapping steps between the initial and final states. Table 4 reflects this—the hysteresis drops from around 13 ± 4 kcal · mol^{-1} for 10 steps to 7 ± 2 kcal · mol^{-1} for 20 steps. The decrease is a direct result of the greater overlap of adjacent potential energy surfaces caused by increasing the number of steps. As the amount of overlap is increased the difference between the configurations generated on the λ and $\lambda + d\lambda$ surfaces become smaller. Therefore, the difference between evaluating the potential on the configurations from the λ or $\lambda + d\lambda$ surfaces also becomes smaller and hysteresis decreases.

Because evaluating the actual value of the free energy for adiabatic charging of the sodium ion is not the main aim of the current article, we emphasize here that the time for implementation of these new algorithms represented an extremely small fraction of the time spent designing the Adun framework. A more striking example of how to implement a novel method into the Adun framework is given in the Appendix, where mean field QM/MM method ASEP/MD is implemented into Adun.

Conclusions

The explosion in the use of computational methods in science has brought into sharp focus the defects in the prevalent algorithm-centric development methodology, and the increasing need for advanced applications cannot be met by this programming paradigm. Computational scientists need to take advantage of and

Table 4. Results of Applying FEP and Bennett's Methods to the Analysis of Sampled Configurations during an Adiabatic Charging Process of a Sodium Ion in Water.

Mappings	Steps	Samples	FEP forward	FEP reverse	FEP average	Bennett
10	100	10	−87 ± 2	100 ± 2	−93 ± 4	−95 ± 2
10	1000	200	−85 ± 2	101 ± 2	−93 ± 4	−93.1 ± 0.8
10	5000	1000	−87 ± 2	100 ± 2	−93 ± 2	−93.8 ± 0.5
20	100	10	−89 ± 2	97 ± 1	−93 ± 2	−93 ± 1
20	1000	200	−90 ± 1	97 ± 1	−94 ± 2	−93.5 ± 0.6
20	5000	1000	−90 ± 1	97 ± 1	−94 ± 2	−93.8 ± 0.4

implement many of the ideas that have accompanied and driven the explosion in commercial computer software in the past decade. We have explained briefly these ideas and how they aid the development process and shown how we applied them to the design of a new Molecular Dynamics simulator, Adun.

Adun is the result of our ambition to provide a highly scalable, easy to develop open-source platform for computer simulations that also allows rapid implementation of new functionality and protocols. This stands in contrast to many current simulation packages that cannot keep pace with the rate of change in the state of the art, a situation that has lead to a proliferation of lab-centric MD programs that implement a few related protocols that subsequently remain unknown or unused outside of a small user core.

We have shown the use of the program in a simple free-energy perturbation example as an indication of the future work we will carry out with Adun. Other features are also currently available in the code, including the possibility of using the Langevin dipoles (LD) model for solvation or the recently implemented averaged solvent electrostatic potential from the molecular dynamics (ASEP/MD) method. The design philosophy embodied by Adun also allows us to explore novel ideas such as intrinsic grid computing, which will offer scientists new perspectives for computer simulations of biomolecules.

Appendix A: Implementation of ASEP/MD into Adun

This section includes the protocol followed to perform a complex new implementation into the Adun structure. The process will be exemplified with the implementation of the averaged solvent electrostatic potential (ASEP/MD) method,[12] a hybrid quantum mechanics molecular mechanics (QM/MM) methodology based on the use of the mean field approximation.[23]

The basic structure of the ASEP/MD method involves performing a molecular dynamics simulation where part of the system (the solute) is kept rigid, analyzing the obtained configurations to build an average solvent model, carrying out a quantum calculation with the solute surrounded by this solvent model, updating the force field parameters (the solute's atomic charges) using the results of this calculation, and performing a new molecular dynamics simulation.

The first step for the implementation of ASEP/MD into Adun was identifying the places where the new functionality was to be added. Shortly, these are the main elements to consider (see ref. 12 for additional details on the method):

- A main controller to manage the ASEP/MD protocol.
- A set of methods for performing the quantum calculations with a given external program.
- A grid of points over the space occupied by the solute, as well as methods to calculate the electrostatic potential on these points.
- The solvent model, represented by two sets of point charges: inner charges and outer charges.

Details of the Implementation

The controller is the core of the process, and little more would be needed if all the functionality was already included in the standard Adun program. However, some of these needs are rather specific to the ASEP/MD method and have to be provided here as an addition. Other features, like the interfacing with quantum calculation programs, may be useful in general, and could be included in the main Adun structure in further developments.

The following paragraphs give some more details about the different tasks and methods supplied by this implementation.

Controller

The new controller, called AsepMD, is in charge of organizing the molecular dynamics simulations and the quantum calculations as needed, and of the interface between them. Its main feature is a loop to launch successive MD simulations and quantum calculations until the convergence criteria are satisfied or a maximum number of iterations is reached. It also reads the additional input file and writes the output.

Quantum Handler

For the quantum calculations we rely on the use of external programs. The choice of the specific quantum program to use is, in principle, open to the user, but the needed methods were developed for Gaussian 98.[24] Thus, a class named GaussianHandler was written, containing in particular methods to write an input file (with solute and model solvent specification), launch the quantum program, and read the output files (energies, atomic charges, etc.).

If a different quantum program were to be used, another class providing these same methods (although with different implementations) would have to be written. This keeps the program structure and interface constant while the needed changes are confined to a small portion of code.

Solute Grid

To build the solvent model, it is needed to define a grid of points to sample the electrostatic potential on the solute. This grid can be considered as a "phantom image" of the solute, and should eventually move with it. A new class, named SoluteGrid, was created for this.

Additionally, the methods for calculating the nonbonded interactions had to be slightly modified to allow the obtention of the solvent electrostatic potential on each of the grid points individually. A new AdNonBondedCalculator subclass, named GridNonBondedCalculator, was thus created.

Solvent Model

For the quantum calculations, the solvent influence on the solute is represented by two sets of point charges, whose locations and values are calculated from the simulation results. To create and manage this solvent model, a new System class was written, called AsepSystem. It is a subclass of ExplicitSystem, so that it inherits all definitions and methods to deal with the MD solute–solvent system, but it has also added methods for building and modifying the solvent model.

In particular, this class contains methods to gather solvent configurations, calculate the average inner charges, and fit the outer charges' values. In this way, both the solute structure and the solvent model are easily available for the quantum handler (see above).

Using ASEP/MD in Adun

From the usage point of view, running an ASEP/MD calculation with Adun is just as easy as a standard simulation, but some details must be kept in mind. First, ASEP/MD is designed to work with an explicit solvent and Newtonian dynamics simulation, and with fixed (rigid) solute, so the appropriate keywords should be used in the main Adun configuration file. Second, AsepMD.ad should be specified as the controller of choice. Additionally, another specific input file (asepmd.config) is needed for setting the ASEP/MD parameters; an example is shown in Figure 9.

For the meaning of the different parameters, the reader is referred to the main ASEP/MD article[12] and to the provided `README` file.

Modifying ASEP/MD

The current ASEP/MD implementation into Adun is still under further development, although the main core of the implementation was performed in a very short time, including the time spent in familiarizing with the programming language used. A user might want to extend its capabilities, modify algorithms or change the controller's behavior. Some of the possible changes are outlined below.

```
{
        AsepmdDir = "AsepMD";
        OutputFile = "H2O.out";
        SamplingInterval = <*I10>;
        MaxIterations = <*I5>;
        EnergyConvergence = <*R0.001>;
        ChargeConvergence = <*R0.001>;
        GridSize = <*R0.2>;
        Factor = <*R0.7>;
        ShellRadius = <*R5.0>;
        MinDistance = <*R0.1>;
        QMProgram = "Gaussian";
        QMExecutable = "gaussian98";
        QMTemplate = "H2O.g";
        QMOutput = "H2O.ogau";
        ChargesType = "CHelpG&Dipole";
}
```

Figure 9. Sample input file for ASEP/MD with Adun.

- **Adding interfaces to other QM packages.** In principle, it is possible to use any QM program capable of performing calculations in the presence of external point charges. For each program a QMHandler class should be provided, and GaussianHandler can be used as a reference. The needed methods are those for writing the input, running the program, and reading the output.
- **Optimizing geometries.** At the moment, there is no geometry optimization in this ASEP/MD implementation, but some handles are provided for that and the link to other optimization methods already implemented into Adun is straightforward. To add geometry optimization, the NullOptimizer (which does a single point calculation) should be replaced by another class that applies the desired optimization algorithm to the solute geometry.
- **Defining a different solvent model.** The solvent model defined by the ASEP/MD method includes two sets of charges whose positions and values are calculated in a particular way. To explore other possibilities of representing the solvent as obtained from the molecular dynamics, the appropriate methods in AsepSystem could be modified or a new System class could be written. As long as the solvent is still represented by point charges, no additional changes are needed.

References

1. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. Design Patterns: Elements of Reusable Object Oriented Software; Addison–Wesley Professional: Reading, MA, 1995.
2. Beck, K. Extreme Programming Explained: Embrace Change; Addison–Wesley: Reading, MA, 1999.
3. Fowler, M. Refactoring: Improving the Design of Existing Code; Addison–Wesley: Reading, MA, 1999.
4. Roberts, D.; Johnson, R. In Pattern Languages For Program Design 3;

Riehle, D.; Buschmann, F.; Martin, R. C., Eds.; Addison–Wesley: Reading, MA, 1997.
5. King, G.; Warshel, A. J Chem Phys 1989, 91, 3647.
6. Warshel, A. Computer Modeling of Chemical Reactions in Enzymes and Solutions; John Wiley & Sons: New York, 1991.
7. Lee, F. S.; Chu, Z. T.; Warshel, A. J Comput Chem 1993, 14, 161.
8. Schlick, T. Molecular Modeling and Simulation; Springer–Verlag: New York, 2002.
9. Torrie, G. M.; Valleau, J. P. J Comp Phys 1977, 23, 187.
10. Bennett, C. H. J Comput Phys 1976, 22, 245.
11. Sánchez, M. L.; Aguilar, M. A.; Olivares del Valle, F. J. J Comput Chem 1997, 18, 313.
12. Fdez Galván, I.; Sánchez, M. L.; Martín, M. E.; Olivares del Valle, F. J.; Aguilar, M. A. Comp Phys Commun 2003, 155, 244.
13. Lee, F. S.; Warshel, A. J Chem Phys 1992, 97, 3100.
14. Qvist, J.; Medina, C.; Samuelsson, J.-E. Protein Eng 1994, 7, 385.
15. Sham, Y. Y.; Chu, Z. T.; Tao, H.; Warshel, A. Proteins: Struct Funct Genet 2000, 39, 393.
16. Bennett, C. H. In Algorithms for Chemical Computations; Christofferson, R. E., Ed.; American Chemical Society: Washington, DC, 1977, p. 63.
17. Lu, N.; Singh, J. K.; Kofte, D. A. J Chem Phys 2003, 118, 2977.
18. Shirts, M. R.; Pande, V. S. J Chem Phys 2005, 122.
19. Jarzynski, C. Phys Rev Lett 1997, 78, 2690.
20. Crooks, G. E. Phys Rev E 2000, 61, 2361.
21. Shirts, M. R.; Blair, E.; Hooker, G.; Pande, V. S. Phys Rev Lett 2003, 91.
22. Kollman, P. Chem Rev 1993, 93, 2395.
23. Tomasi, J.; Persico, M. Chem Rev 1994, 94, 2027.
24. Frisch, M. J.; Trucks, G. W.; Schlegel, H. B.; Scuseria, G. E.; Robb, M. A.; Cheeseman, J. R.; Zakrzewski, V. G.; Montgomery, J. A.; Stratmann, R. E.; Burant, J. C.; Dapprich, S.; Millam, J. M.; Daniels, A. D.; Kudin, K. N.; Strain, M. C.; Farkas, O.; Tomasi, J.; Barone, V.; Cossi, M.; Cammi, R.; Mennucci, B.; Pomelli, C.; Adamo, C.; Clifford, S.; Ochterski, J.; Petersson, G. A.; Ayala, P. Y.; Cui, Q.; Morokuma, K.; Malick, D. K.; Rabuck, A. D.; Raghavachari, K.; Foresman, J. B.; Cioslowski, J.; Ortiz, J. V.; Stefanov, B. B.; Liu, G.; Liashenko, A.; Piskorz, P.; Komaromi, I.; Gomperts, R.; Martin, R. L.; Fox, D. J.; Keith, T.; Al-Laham, M. A.; Peng, C. Y.; Nanayakkara, A.; Gonzalez, C.; Challacombe, M.; Gill, P. M. W.; Johnson, B. G.; Chen, W.; Wong, M. W.; Andres, J. L.; Head-Gordon, M.; Replogle E. S.; Pople, J. A. Gaussian 98, Revision A.7; Gaussian, Inc.: Pittsburgh, PA, 1998.