A Front End Design For Simulation Engines

Author:
Trevor M. Cickovski
Graduate Student
Department of Computer Science and Engineering
University of Notre Dame
USA
tcickovs@nd.edu

Under supervision of:
Jesús A. Izaguirre
Department of Computer Science and Engineering
University of Notre Dame
USA
izaguirr@cse.nd.edu

`http://www.nd.edu/~lcls`
compucell@cse.nd.edu
protomol@cse.nd.edu

Other Contributors to Front End from University of Notre Dame:

Thierry Matthey
Thomas Slabach

August 1, 2002

Abstract

The front end design that is about to be described was originally used as a part of the framework of PROTOMOL , molecular dynamics software released by the Laboratory for Computational Life Sciences (LCLS) at the University of Notre Dame during August of 2001. This same overall design (with some modifications) would later be used by COMPUCELL , a framework for biocomplexity simulations, recently released by the LCLS in July of 2002.

This fact that the design was able to be used for two extremely different simulation engines illustrates an important aspect and possibly the most important goal of the design, which was to make the front end flexible enough that it could be used with other engines with small modifications. The hope is that this framework will be able to be used with future software developments under the topic of computational biology.

Acknowledgements

- Special thanks to the Center For Applied Mathematics at the University of Notre Dame for helping to fund researchers on the project.
- Parts of the development were supported by a National Science Foundation Biocomplexity grant (PHY-0083653) and a National Science Foundation ACI CAREER award (ACI 01-35195
- Other significant contributors to the front end design were made by Thierry Matthey and Thomas Slabach.

Contents

List of Figures

Chapter 1

Introduction

A typical simulation engine framework can be divided into a back end which controls all computations, and a front end. Sometimes a third layer, called a visualization layer if applicable, can be interfaced with these two. The responsibilities of a front end are to control output and user input throughout a given simulation. More specifically, it should accept any input from the user or from files and use it to initialize anything the back end needs to perform computations. Then it should be able to access back end information throughout a simulation and perform appropriate output, be it to the console or to files. In order to obey traditional front/back end interfacing which involves clear separation of the two, this front end was designed to not perform any computations whatsoever, and also to not force any back end that it is interfaced with to do I/O.

This front end has also been made to not allow changes to any back end that it is interfaced with to affect its operation, outside of calling the appropriate initialize and run methods that are unique for each particular back end. The reverse is also true, that changes to computational methods in the back end should not affect the operation of a front end. Thus numerous advantages in flexibility are offered by this design, and they have been proven in its recent reusability between PROTOMOL and COMPUCELL . In addition, there are many other advantages that this design holds. User-friendliness was an extremely important consideration when creating this framework, and its implementation was designed around file-based input which is preferable to any user because it means that he or she does not need to reenter all input parameters each time one changes.

Finally, this front end was designed to be easily modified. Modifications can occur on existing programs often by programmers or even users at times. Software maintenance can be very costly timewise if planning ahead is neglected, and efforts in implementing this design zeroed in on making this process as straightforward as possible. Code reuse and flexibility are invaluable assets to maintainable software. In addition, the front end has been coded in the high-level, object-oriented language C++. This was chosen for its advantages in combining abstraction at a high level with low-level optimizations. [2]

Chapter 2

Achieving User-Friendliness In The Front End

The first goal in the front end design was to make it as easy and friendly to use as possible. There were several techniques used in the effort to achieve this goal. They are described below.

2.1 Design Around File Input Over Console Input

Console input has its advantages for programs with small amounts of input needed. Typically however, simulations of the sort that biocomplexity demands depend on initial values of several different input parameters. It would be so difficult and time-consuming that it is infeasible to expect the user to input in the console all of the initial values that are required to run a program of that falls under this category. For this reason, the front end has been made to read files. It only takes one input from the console - and that is what is specified on the command line at a prompt.

2.1.1 The Configuration File

The front end has been built to read initial files to the simulation engine known as configuration files. As mentioned earlier, engines that run biocomplexity simulations typically require a number of different input parameters. The configuration file format that the front end is built to read conveniently has a simple format consisting of multiple lines of keyword-value pairs, with the keyword and its value on the same line, separated by whitespace. Chances are this format will be the most commonly used among these types of engines, since they are the most simple and most easy for the user to adjust appropriately for different types of simulations with the given engine.

2.1.2 Other Input Files

Readers for other input files can be interfaced as described later in chapter 3, which deals with the flexibility of this front end and how it can be modified to work with different simulation engines. However, as we shall see, the front end has clearly defined where input files outside of the configuration file would typically be read throughout a simulation and has a function designated and called in the appropriate location to do so. Thus adding other file readers, though usually the most time-consuming part of adjusting this front end to work with other back ends, is still about as straightforward as it could possibly be without knowing the required file formats in advance.

2.2 What Is Needed On The Command Line

2.2.1 Format Expected

As mentioned, the front end has been built around file input and only needs a command line with regard to console input. It has been designed to accept the executable name followed by the full path to the configuration file on the command line, followed by any other configuration file keyword-value pairs:

```
<executable> <configuration> <file> <path> -<keyword1> <value1> -<keyword2>
<value2> .....
```

Any configuration file keyword-value pair specified on the command line overrides the value of the same keyword in the configuration file. The advantage to this is if the user would like to vary one or two parameters slightly and experiment, it is easier to simply enter new values on the command line rather than return to the configuration file each time, find the keyword and change its value. Any number of these pairs can be specified on the command line. In addition, the user may specify what is known as an `inputfileprefix` on the command line in this way: `-inputfileprefix <pathname>`, and the front end will concatenate '.conf' at the end of this prefix for the configuration file name. Typically the `inputfileprefix` is used with other input files as well but that is solely up to the simulation engine designer. To help whatever engine is being used to be more catering to the user, the front end command line parser has also been built to handle several options:

2.2.2 Help Option

If `-h` or `-help` is specified on the command line, a help menu will be displayed.

2.2.3 Version Option

The actual version of the engine can be displayed with a `-v` or `-version`.

2.2.4 Keyword Option

Output of all actually supported keywords, their default values and their types in the configuration file can be displayed in the console with `-k` or `-keywords`. This could also be useful for the designer to keep track of what parameters and types are currently being used if they need changing, it saves the trouble of looking at any low-level code.

2.3 Output Files and Formats

There are four styles of output that are typically desired throughout a biocomplexity simulation, these are: console, files that are overwritten every certain number of steps (Restart files), files that are appended to every certain number steps (Trajectory files), and finally files that are written at the end of the simulation (Final output files). All are extremely useful in their own ways. This front end has been built to handle all four of these formats.

2.3.1 Console Output

Console output is straightforward - it involves displaying information to the console window in which the program is running. Typically an initial display, errors and warnings, and the most important information throughout the running of a simulation is displayed here.

2.3.2 Restart Files

A Restart file is overwritten every certain number of steps, given by the value of a certain variable that is usually specified by the user. PROTOMOL and COMPUCELL had a configuration file keyword called RESTARTFREQ that controlled this value. The idea behind restart files are that the program can be restarted completely from the point at which these files are written, using these files as inputs. This could prove very useful for several different types of simulation engines because it allows the user to save the current status of a simulation and enables it to continue from where it previously left off. Thus typically only files that can be input to the simulation engine are written here.

A `Restart` class has been implemented as a part of the front end design in order to handle these types of files. It is initialized with a value for this restart frequency so it knows how much time must pass between file writes.

2.3.3 Trajectory Files

Sometimes the user would like to see how data changes throughout a simulation, and perhaps even plot graphs and charts showing the results. This ability is given by trajectory files that are output every certain number of timesteps - and this is in general user-specified. PROTOMOL and COMPUCELL had a keyword specified in the configuration file, called OUTPUTFREQ, which held this value. Note the difference between trajectory files and restart files - though both are written throughout a simulation, the latter involves writing files that can be input to the program and that are completely overwritten each time, and trajectory files are simply appended to each time they are written, and usually are not input files. They are very useful for information purposes, especially when it comes to testing the engine for different types of simulations and assembling results for a potential presentation.

The front end has been built to handle this type of files writing, all implementation is included in the class `OutputData`. Once again, the decision to write any compatible trajectory files is generally left up to the user. Both PROTOMOL and COMPUCELL allowed the user to specify these filenames in the configuration file.

2.3.4 Final Output Types

In addition, the front end has been made to write certain files only at the end of the simulation. The user will typically specify the files he or she wants written in the configuration file. If all that the user cares about certain information are its final values, these files are useful.

Chapter 3

Improving Code Organization

Possessing organized and readable code as opposed to its antagonist, so called spaghetti code, aside from its importance in the ease of the initial design process, holds most of its advantages in the area of software maintenance. Recent studies have shown that typically 80% of the time in software development is devoted to maintenance, and much of that is due to the lack of planning ahead in the design process. [6] Thus the consequences are clear if maintenance is not kept in mind while implementing a software framework, and it was kept at the forefront when designing this front end.

3.1 Main Module

The `main()` function of the program should be almost the same everytime this front end is used, independent of what tasks the back end does. The front end was organized in a way that only four functions would need to be called in the main module. These sequentially are: `parseCommandLine()`, `Simulation::initialize()`, `Simulation::run()`, and `Simulation::finish()`. All that is needed for any main module that uses this front end is a simple structure like this.

3.2 `Simulation` Class

This class has been conveniently organized into three main functions, `initialize()`, `run()`, and `finish()`. All other methods of this class are called from one of these three functions. This is advantageous because it gives the program structure. If a front end task is to be added or even a back end task that needs to be called from the front end, one only needs to answer one question to know where in the code it should be placed. That question is, should it be done before the engine starts running, while the engine is running, or after the engine finishes running? At that point the designer knows which function in `Simulation` should contain the code that accomplishes the new task. The class `Simulation` also contains objects to hold all information given from the user at input time, and also objects to hold output information. Therefore a designer does not need to search through numerous different classes to find out where to access a piece of initial information or any information that would be output throughout a simulation - it is all contained in `Simulation`.

3.3 Clear, Concise Separation Of Input And Output Structures

More importantly, a designer not only knows to reference the `Simulation` class if he or she wishes to obtain input/output information, but also what objects of the class to reference. That is because all input

and output data are kept in separate structures. Data for input is kept in `SimulationConfiguration` and `SystemData` classes, and output data is stored in `OutputData`, `Restart` and if necessary an `ExtraInfo` structure.

3.4 Complete Independence Of Front And Back End

As mentioned earlier, this front end has been built to work independently of the back end. Besides offering numerous advantages with regard to flexibility as the next chapter will show, this fact greatly improves code organization as well. If a change to one of the front or back end needs to be made, the designer has a good idea of the classes that are affected and is saved time and effort in the process of maintenance.

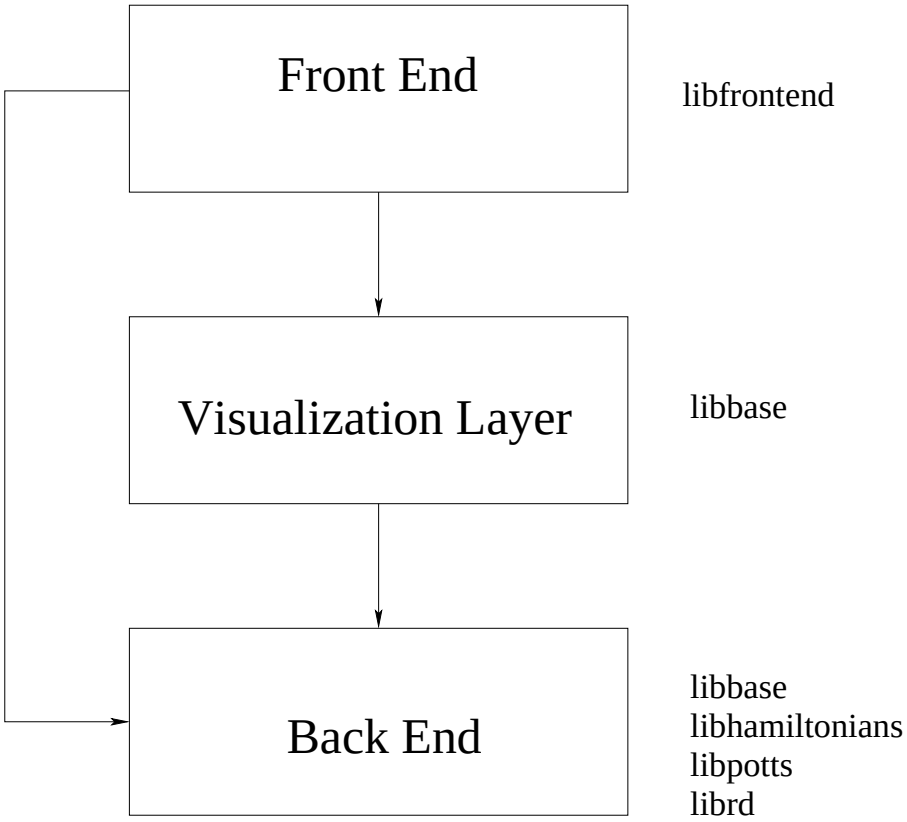

Figure 3.1: Information Flow In The Current COMPUCELL Framework. Note that the front end initializes structures in both the visualization layer and the back end, and the visualization layer passes information to the back end. The back end does not pass information anywhere but handles computations itself, nor does the front end receive information from anywhere outside of the user and input files

Chapter 4

Implementation: Addressing The Issue Of Flexibility

4.1 Adding New Keywords

The class `SimulationConfiguration` is where the reading of the configuration file takes place, as well as the coding of the keywords into a map called `myDataTypes`, which holds a mapping from the keyword (of type string) to its type which can easily be set with the user of a class `VarValType` currently used by the front end - it is very simple and examples are given. This is set in the constructor of the class. To add a new keyword to the program, all that is required is the addition of *one line* in the constructor - which adds the new keyword and its type to this mapping. After that the keyword and its value will be read and stored in the map `myData`. If the user desires a default value for the keyword, one additional line will be needed, adding the keyword and its default value to the `myDataDefault` map. Thus at most two lines would need to be added to allow the keyword and its value to be successfully read and stored with easy accessibility - the `myData` map can be indexed with the keyword and its value will be returned. And once this has been set up, all that is needed is a call to the appropriate back end `set` method that will initialize the back end accordingly. The amount of code needed here will vary but typically will not surpass a few lines.

4.2 Adding Other Input Files and Readers

In the process of tweaking this front end to work with different engines, this is typically the step that takes the most time. That is because the formats and quantities of input files desired by the makers of different engines cannot be predicted. PROTOMOL for example read PDB, PSF, XYZ and CHARMM parameter files, and COMPUCELL read PIF, CPF, and Viz files - and all of these were for the most part completely unrelated. Still, the front end framework caters to this need. There is conveniently a class called `SystemData` that is designed to hold information from all other types of input files, and a function located in the class `Simulation` that is called at the point in the simulation run at which typically input files outside of the configuration file would be read (and is called right after the reading of the configuration file, before any back end initialization). This function is named `readSystemData()`.

To add a file reader, a user would need to define a class to hold all data from the file and place an object of that class as a member of `SystemData`, and then build a reader for the file to store the information in this object, typically in the same class. Next, a call to the reader would need to be placed in `Simula-`

`tion::getSystemData`. And then appropriate back end `set` function would need to be called. Thus although unavoidably it is the case that the user would need to write a reader for each new input file from scratch since we cannot predict the formats desired, once that is complete it is straightforward to add the information to the simulation.

4.3 Adding Other Output Files and Writers

Next: Restart Files Up: Implementation: Addressing The Issue Previous: Adding Other Input Files Contents
Adding Other Output Files and Writers

This step which is a part of the process of adjusting the front end to work with different engines will take time proportionally to the complexity of the format of the output files that the designer desires to be written by the front end. This is because like the input files, these formats cannot be predicted, therefore the designer must write code for all of the file writers. Typically however, formats for output files are defined by the designer unless a format that is widely known is desired. For example, PROTOMOL and COMPUCELL had their own formats for energy files and several different trajectory files. Once the writers have been written, however, the interfacing is straightforward.

As mentioned in the first chapter on user-friendliness, the output files can be grouped into three categories: Restart, Trajectory, and Final Output. The first step for the designer who would like to add an output file writer to the framework would be to decide which type of file this is. Guidelines for each are given below.

4.3.1 Restart Files

A restart file as mentioned earlier is a file that when it is written to, is completely overwritten. Also it should be able to be used as an input file to the simulation engine, the usage of which would enable the engine to start the same simulation from the point that these files were written. The class `Restart` handles the entire process of writing restart files. A `Restart` object is currently initialized by passing the data map that contains all information from the configuration file. It is expected and overall likely that the designer would like the restart frequency and restart file prefix to be specified in the configuration file, since with its simple format it would be the easiest for the user. The current implementation of the `Restart` class contains a constructor in which member variables are initialized including the restart frequency, the number of steps for the simulation, and also if a configuration file is a desired restart file a structure to hold all configuration file data. Then it has only one method `run()` which takes as parameters any information that is needed to write restart files plus the current step of the simulation, has an opening check to see if restart files should be written at this step based on the restart frequency that the user specified, and then sequential calls to writers. Therefore to add another writer is straightforward: be sure that any information required for the file being written is passed to `Restart::run()`, and assuming the writer has been made a function add a call to it at the end of `Restart::run()`. Therefore to interface a file writer with the `Restart` class should not require more than a few additional lines added to its member method `run()`, and will not affect the constructor nor the class declaration at all.

4.3.2 Trajectory And Final Output Files

A trajectory file is appended to throughout a simulation, every given number of steps which is in general a user option called the output frequency. The writing of both trajectory files and final output files is handled in the class `OutputData`. Some default values for member variables are set in the constructor, and an

object is currently initialized with the data map from the configuration file. Once again the idea was for a designer who is using this front end to give the user the option to write any sort of output files in the configuration file, and once again the convenience of the end user was considered. Thus the assumption is that in `OutputData::initialize()`, enough information is known to decide which output files the user would like to write.

The value of an integer, `myWhichIOFiles`, determines which files will be written and is set in `OutputData::initialize()`. Then constant integer macros are defined as private data members of the class and are declared as the value 1 shifted by a different amount for each macro. One macro is declared for each different file that is written. For example, a chunk from `OutputData.cpp` in the current COMPUCELL framework, with macros declared for two different types of energy file formats:

```
const unsigned int OutputData::ALLENERGIESFILE = (10);
const unsigned int OutputData::SPLITENERGIESFILE = (11);
```

The next step is in `OutputData::initialize()`. The passed configuration file information is checked, and if conditions have been met that a given output file is written, the logical 'or' function is used with myWhichIOFiles and the corresponding macro and myWhichIOFiles gets reset to this new value. For example, in COMPUCELL , if the ALLENERGIESFILE should be written:

```
myWhichIOFiles |= ALLENERGIESFILE;
```

Then in `OutputData::run()`, the following logical 'and' condition is checked (once again the ALLENERGIESFILE example is used):

```
((myWhichIOFiles & ALLENERGIESFILE) == ALLENERGIESFILE)
```

And if it passes, call the appropriate file writer. Note that if the file being written is a final output file and not a trajectory, this check and writer call would be in `OutputData::finalize()`. Therefore, to add a new trajectory or final output file writer, only four simple additions need to be made:

1. Place new keyword(s) in the configuration file reader.
2. Declare a new macro for the new file.
3. Add the condition to check the configuration file data and perform the logical 'or' on myWhichIOFiles.
4. Add the logical 'and' condition to check the value of myWhichIOFiles and call the writer function. This would be added to `OutputData::run()` for a trajectory file and `OutputData::finalize()` for a final output file.

4.4 The Use of An API in COMPUCELL

An API, or Application Programming Interface [1], was a step taken by COMPUCELL in an effort to further increase flexibility by enabling the front end to work with multiple back ends that perform different tasks. The API has its effect when the back end is initialized by the front end with all of the initial information that it needs to run. In PROTOMOL , the front end contained several back end objects and simply called each respective object's initialization methods. While this succeeded in keeping front and back end independence, only one back end could be used, since the initialization methods called were specific to a back end class. In order to work with multiple back ends, conditions would need to be implemented to determine which initialization methods should be called. The API eliminates this problem.

The idea behind the API of COMPUCELL was to have one class `CompuCellAPI` to hold `set` and `get` methods for all potential parameters between all of the back ends, but made `virtual`. Thus any class that would inherit from `CompuCellAPI` would also inherit these `set` and `get` methods, and would need to define them. To employ multiple back ends, each back end would need a central class to hold all initial information that it needs to run, and that is referenced for information by the rest of the back end. In COMPUCELL , this class was conveniently named `CompuCell`. The class `CompuCell` inherited from `CompuCellAPI` and contained function definitions for all of the `set` and `get` methods declared in `CompuCellAPI`.

This structure is then initialized by a call to a function accepting a `CompuCellAPI` pointer and consisting calls to all of the `set` methods defined by `CompuCellAPI` sequentially. And any information that is needed by the front end can be obtained by the appropriate `get` method, but this would only be needed for output. Depending on the output file situation for the different back ends, some information may need to be known about the back end to determine which output files should be written, and once that has been established call the correct `get` methods. However, this can be taken care of with configuration file parameters. The key issue here is successful initialization for the running of any back end.

Now suppose a designer would like to add another back end to the program, with a central class named `CompuCell2`. Now, `CompuCell2` may consist of completely different parameters, in fact, it may even perform completely different tasks as `CompuCell`. But the sole job of the front end will be initialization. Both `CompuCell` and `CompuCell2` will now inherit from `CompuCellAPI`. Note that `CompuCellAPI` will at this point need to contain `set` methods for all of the variables for both back ends, and both `CompuCell` and `CompuCell2` will inherit all of the these methods and will have to define them. This is not a problem, however - though each back end needs to *define* all of the methods, the `set` methods of a back end central class for variables that are not used by that back end can just do nothing. The `set` methods that initialize values that are used by the back end will be written in its central class.

Now, the function that initializes a particular back end will accept a `CompuCellAPI` pointer as before - meaning that it can successfully accept a `CompuCell*` object or `CompuCell2*` object since both inherit from it. Therefore we now have one initialization function for multiple back end central classes. Sequential calls to *all* of the `set` methods of the API will be in this function (note whatever back end central class that was passed will have defined each one). But also note that each central class contains empty `set` functions for whatever parameters that it does not use. Thus the only function calls that will set anything will be for applicable parameters to that particular back end. And at this point, whatever back end central class object that was passed will have set all variables that it needs set in order to run, which was the goal.

Chapter 5

Future Plans

Work is currently underway to expand this front end design. This work involves three main challenges. The first is to allow the front end to possess a generic interface to parallelization. The second is to increase flexibility with the use of the factory method. And finally, research into implementing a more event-driven front end is in the process. The first two have achieved some success in PROTOMOL and are future goals of COMPUCELL , and are described in more detail below. Event-driven software typically requires a Graphical User Interface (GUI) [3] and so likely would be applicable to COMPUCELL , not PROTOMOL . Still, to have the implementation as a part of this front end would be a huge advancement with regard to catering to programs that use GUIs, and designers of programs that do not need it simply can choose not to use it.

5.1 Parallelization

The ability to integrate parallel communication into a biocomplexity simulation engine would offer numerous advantages in efficiency upon program execution. However, there are challenges in implementing this. The difficult part is that this is something that would need to be used by both the front and back end. Therefore, it is not an easy task to develop an interface to parallelization in a front end that is meant to be as generic as possible without requiring many back end modifications. Methods of pulling this off are being looked into however.

5.1.1 Success In PROTOMOL [5]

A Message Passing Interface (MPI) with one-sided communication features of MPI-2 is used in the PROTOMOL front end (and back end). "What PROTOMOL uses is an incremental parallelization scheme for clusters with a moderate number of nodes, e.g. up to 64 CPUs. The design is based on a force decomposition with a master-slave paradigm." [5] Ideas from this method of parallel communication are being researched.

5.1.2 Future In COMPUCELL

Parallel communication has not been implemented in COMPUCELL yet, but should be in the future. The hope is that the interface designed for COMPUCELL will be as generic to back ends as possible.

5.2 Factory Method

5.2.1 Definition [4]

The factory method is useful in object-oriented design when responsibility is distributed to objects of several different classes. This is exactly the case in PROTOMOL and COMPUCELL , both implemented in C++ and both having frameworks that consist of several different classes, each clearly designated its own tasks to perform. The factory method is used to handle several difficult situations that often arise in object-oriented design. Namely, the following:

1. When an application at runtime must instantiate a subclass of a given abstract class and knows the abstract class but does not know which subclass to instantiate.
2. When a class wants the decision of what objects to create to be made by its subclasses.
3. When a class delegates responsibility to one of several helper subclasses so that "knowledge can be localized to specific helper subclasses."

5.2.2 How It Would Be Useful

The usability of this method deals with flexibility with regard to input parameters. For example, in the current frameworks of both COMPUCELL and PROTOMOL , the parameters input from the configuration file or one of the input files always affect the type of simulation run. Many of the decisions towards the type of simulation to run and the types of objects to instantiate are made as hardcoded conditions directly referencing keywords and values in these input files. However, it would be a large improvement in flexibility and code simplicity, for that matter, if we just had one function call that was passed parameter values and based on those values, instantiate appropriately.

5.2.3 Accomplishments So Far With PROTOMOL [2]

A Force Factory has been implemented as a part of the PROTOMOL framework. Different types of forces are designed and implemented, and the desired force is specified in the configuration file, using a *force definition language*. Based on what is specified there, the appropriate force object is created, and due to the use of the factory method the use of a huge conditional statement with hardcoded parameters is avoided. It is particularly useful in this case because PROTOMOL handles many different types of forces. Code is shortened drastically, and is easily updated to handle more forces. Ways in which this design pattern can be modified for use in COMPUCELL and the front end being discussed are currently being looked into.

5.2.4 Future In COMPUCELL

Though not implemented yet in COMPUCELL , the interfacing of the factory method with the current COMPUCELL front end is being researched. Once again we would like to make the interfacing as generic as possible. Of course the method will depend on the parameters used and needed for each back end to run, but even if we could get an interface together it would be a huge advancement. This is a future goal.

5.3 Event-Driven Software [3]

The idea behind event-driven software is that all direct connections between a software unit in charge of executing an operation, and those in charge of deciding when to execute it are avoided. A GUI layer typically controls the decision to execute operations that are controlled by the application layer, and responds to commands given by the user. This is an area of future research. Perhaps the advantages in flexibility here can be used as a part of the front end. It will be looked into.

Work is already under way to further exapand COMPUCELL . In the next few years, various new algorithms will be placed in the back end, including new MOLLY integrators and semi-implicit integrators using iterative solvers and sparse matrices, rigid body dynamics, and table lookups. The goal is to acheive larger time steps and thus decrease the amount of time necessary to run a simulation.

Chapter 6

Overall Conclusions

Clearly this front end design has the potential to be very useful as it stands, and with the possible future additions planned could grow to be even more so. Interfacing it with two extraordinarily different back ends like PROTOMOL and COMPUCELL was a very important first step, and illustrates its potential. Future programs of this sort will likely be developed by the LCLS and surely be developed by other organizations. The hope is that this front end design will be a very useful tool towards the execution of these efforts, and we feel that it will.

Bibliography

[1] David J. Hardy. Theoretical biophysics group, university of illinois at urbana-champaign. http://www.ks.uiuc.edu/Development/MDTools/mdapi/files/mdapidoc.pdf, 2001.

[2] Thierry Matthey. Department Of Informatics, University of Bergen, 2002. Ph.D. Thesis: Aspects of Framework Design, Parallelization and Force Computation In Molecular Dynamics.

[3] Bertrand Meyer. http://www.inf.ethz.ch/personal/meyer/ongoing/events.pdf, 2002. The Power of Abstraction, Reuse And Simplicity: An Object-Oriented Library For Event Driven Design.

[4] Gopalan Suresh Raj. The factory method (creational) design pattern. http://gsraj.tripod.com/design/creational/factory/factory.html., 1999.

[5] Jesus A. Izaguirre with Thierry Matthey. http://www.nd.edu/~lcls/Protomol.html, 2001. User Guide 1.05: ProtoMol: An Object-Oriented Molecular Dynamics Framework.

[6] Bill Wolfer. Cse 432: Software engineering. Course Taught At The University Of Notre Dame, Fall 2001.