Molecular Dynamics Language (MDL)
Chun Bong Benjamin Yau, Trevor Cickovski, and Jesús A. Izaguirre
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556

1 Introduction and Motivation

ProtoMol [11] is an object-oriented framework for running Molecular Dynamics (MD) simulations combining parallelization and flexibility, and is compatible with multiple time-stepping integration schemes yielding high performance and extensibility. CHARMM [5] data files formats are supported, along with long-range force calculation algorithms such as Ewald [8], [9], Particle-Mesh Ewald [6], and MultiGrid methods [3], [13]. ProtoMol is compatible with several operating systems such as Solaris, HP-UX, Linux, and Windows. Common I/O formats can be used, simulating a maximum number of particles of $10^6$.

ProtoMol is written in C++ and by default does not support interactive simulations. The closest feature is support for restart files, which are a set of generated input files representing the current simulation state, which can be later submitted as ProtoMol inputs to restart the simulation from the same spot. A desirable feature for ProtoMol would be the ability to change parameters and even extend the framework to encapsulate new integration and force calculation algorithms on the fly, to save user overhead of restarting an entirely new simulation to view effects. The Interactive Molecular Dynamics project [4] enabled interactive ProtoMol simulations through Python [12]. This project wrapped the entire ProtoMol back end using SWIG [1] for Python, enabling accessibility of classes, methods and variables through a Python scripting interface.

The challenge this imposes is a requirement for the user to understand Python syntax and to a certain degree, C++. However, this is not an assumption that can always be made for designers of new mathematical algorithms. A solution would be to enable interactive simulations through a domain-specific language (DSL) [7], containing syntax understandable to algorithm designers, and creating a high enough abstraction level for complicated functionality which requires multiple C++ method invocations. Our goal is to provide this DSL through the MDL (Molecular Dynamics Language). We present our design of MDL including our MDL-to-Python translator module, and some simple commands that can currently invoke ProtoMol back end and execute functionality interactively. Finally, we discuss implications and future work for the project.

There are two different MD DSLs which are comparable to MDL. Molecular Dynamics Markup Language (MoDL) [9] is one example. Through MoDL, users can define atom and molecule types and then use them to build a chemical compound. MoDL extends XML and MDL is characterized by a computationally oriented syntax. MoDL is tag-based which provides structure but for MDL, a computationally oriented language is more suitable because we want to use MDL to write mathematical equations interactively. Yanaconda is a macro language that interacts with the MD framework YASARA [7], creating animations. YASARA can also be extended through Python which is similar to our approach with ProtoMol in terms of interactivity, but through MDL we abstract to a higher level.

2 Algorithm

Figure 1 shows our high-level structure of MDL command processing. We allow a user to enter MDL commands through a shell. The shell can accept simple text commands like running an integrator for a number of steps line-by-line, but more complicated commands which define new integration scheme extensions on multiple lines.

Once a command is entered it is passed to an MDL Translator. The responsibility of the Translator is to accept MDL commands and convert them into ProtoMol back end functionality for interactivity. As ProtoMol has already been SWIG-wrapped for Python, back end functionality is available through a Python scripting interface. The Translator must generate this interface for individual MDL commands then subsequently execute the generated Python code, consisting of two main stages: parsing and generation. The parser takes the initial command in MDL and separates it into a list of keywords. Error checking will eventually be implemented at this stage. The generator accepts this list and generates Python code in a temporary Python
file. The Python commands in this file will invoke the ProtoMol back end, so that when this file is executed ProtoMol functionality will run. When this happens, any output is redirected back to the MDL shell and the temporary Python file is removed, recreated when the next MDL command is input.

Fig. 1. Block diagram of MDL command processing. An MDL command is input into the MDL shell and passed to a Translator, which contains Parser and Generator modules. The parser separates the command into keywords, which are passed through the generator which generates appropriate Python code that references the ProtoMol back end. This Python code is subsequently invoked, which calls back end functionality and redirects output back to the MDL shell.

3 Implementation

We encapsulate functionality for MDL command translation into a core Translator interface, which generates the file temp.py. temp.py is generated only temporarily, such that when an MDL command is submitted and translated into Python invocations of the ProtoMol back end, these invocations will be placed in temp.py and then actually invoked in the interpreter, and then temp.py can be removed. In this way, the user does not need to worry about extraneous files.

The Translator interface includes a translate() method which invokes two other methods: parse() and generate(). parse() traverses an MDL command and separates the command into a list of terms which is subsequently placed in an STL [10] vector. The generate() method detects the terms in this vector and performs translation to Python in temp.py. For example, detecting a RUN Integrator command generates code that will invoke the appropriate integrator in the ProtoMol back end, for some number of steps.

MDL abstracts the following keywords:
- \( V_i \): Velocity at index \( i \).
- \( F_i \): Force at index \( i \).
- \( A_i \): Atom at index \( i \).
- \( P_i \): Position at index \( i \).

The user can construct integrator equations using these terms and the MDL DEFINE Integrator command which we will illustrate in the next section.
4 Running MDL: Testing with Leapfrog

The first step to running Leapfrog integration with MDL is to initialize data, which can be accomplished through a Python script TestLeapfrog.py. When this is imported, it can read a ProtoMol configuration file and set up the molecular topology by reading PDB [2], and CHARMM [5] PSF and PAR input files, and builds initial velocities through random number generation.

The file MDL.py imports the Translator which has been SWIG-wrapped for Python, as a module. This file loads the MDL shell, allowing the user to input MDL commands into the shell invoking the Translator to generate temp.py. It then executes temp.py before prompting the user for another command. The user can repeatedly enter MDL commands before exiting the MDL shell with a QUIT command.

As Leapfrog is already a known integrator, it can be run for 5 steps for example through the command RUN Integrator Leapfrog 5. This statement translates to a series of Python invocations to the ProtoMol back end which invokes the LeapfrogIntegrator constructor, creating an integrator object. It then initializes the object with the initial configuration data before running it for five steps.

A user can also create methods for integration using the abstractions mentioned above, through a set of equations for positions and velocities, terminated by an END statement. For example, the user can enter in MDL:

```
MDL> DEFINE Integrator Temp
V_i = F_i * h / A_i.mass
P_i = V_i * h
END
MDL>
```

which currently can generate Python code to update velocity and position vectors of integrator objects. We are currently working on generating full integrators in Python with data member arrays for velocities and positions that can be updated in this manner.

5 Conclusions and Future Work

Currently, MDL allows users to enter commands to interactively run an existing integrator with ProtoMol or generate Python code for integration equations. Although far from complete, the main problem of requiring users to understand Python and the complicated C++ syntax of the ProtoMol back end is being addressed, and the current MDL is a solid beginning towards completely interactive ProtoMol simulations through the language. These simulations will enable the users to define full integrators, including initialization and execution methods, which is the immediate goal. Once this is accomplished we can test the language more fully by regenerating previously run integrators and comparing results of ProtoMol runs. After this, we want MDL to be able to define long-range force calculation algorithms which will involve similar syntax including local variables which must be initialized and a method to run the algorithm. Once this is accomplished we will need to extend the language to encapsulate a method for associating integrators and force calculators. Testing MDL with integration schemes more complicated than Leapfrog will help to determine the robustness of the language. We also hope to have parameter tweaking with ProtoMol through MDL operating as soon as possible.

MDL documentation will be very important. An MDL User Guide will be necessary for users to understand MDL, and an MDL Programmer’s Guide will be a solid tool to help programmers take advantage of language extensibility created by the object-oriented Translator, to hopefully extend the language themselves. A help section can be created in MDL to let users type “HELP” when they have questions about MDL’s usage.

Finally, MDL can potentially improve the usability and popularity of ProtoMol. The time MDL can save for users versus running ProtoMol can be significant, especially in terms of framework extension and parameter sweeps, and also will make the framework easier to use. We look forward to the future growth of both MDL and ProtoMol.