View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Elsevier - Publisher Connector

Procedia Computer Science

Volume 29, 2014, Pages 2034–2044

ICCS 2014. 14th International Conference on Computational Science

Web- and Cloud-based Software Infrastructure for Materials Design

Janos Sallai[1], Gergely Varga[1], Sara Toth[1], Christopher Iacovella[2], Christoph Klein[2], Clare McCabe[2], Akos Ledeczi[1], and Peter T. Cummings[2]

[1] Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, USA
janos.sallai@vanderbilt.edu

[2] Department of Chemical and Biomolecular Engineering, Vanderbilt University, Nashville, TN, USA

Abstract

Molecular dynamics (MD) simulations play an important role in materials design. However, the effective use of the most widely used MD simulators require significant expertise of the scientific domain and deep knowledge of the given software tool itself. In this paper, we present a tool that offers an intuitive, component-oriented approach to design complex molecular systems and set up initial conditions of the simulations. We integrate this tool into a web- and cloud-based software infrastructure, called MetaMDS, that lowers the barrier of entry into MD simulations for practitioners. The web interface makes it possible for experts to build a rich library of simulation components and for ordinary users to create full simulations by parameterizing and composing the components. A visual programming interface makes it possible to create optimization workflows where the simulators are invoked multiple times with various parameter configurations based on results of earlier simulation runs. Simulation configurations including the various parameters, the version of tools utilized and the results are stored in a database to support searching and browsing of existing simulation outputs and facilitating the reproducibility of scientific results.

Keywords: modeling, molecular dynamics simulation, code generation, scientific workflows

1 Introduction

Understanding and controlling the assembly of nanoparticles into arbitrarily sized and shaped functional structures underpins the evolving field of nanotechnology. An emerging philosophy is to use tethered nanoparticles (TNPs). If the tethers and the NPs dislike each other, these building blocks behave akin to surfactants, leading them to assemble into anisotropic structures. By changing the shape of the nanoparticle, even more complex structures are possible. The

The work presented in this paper was supported by the National Science Foundation grants NSF CBET-1028374 and NSF OCI-1047828.

 Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2014
© The Authors. Published by Elsevier B.V.

doi:10.1016/j.procs.2014.05.187

resulting assemblies potentially have broad utility across the physical sciences and biology. The ability to assemble metal particles into percolating structures allows us to dramatically increase the electrical conductivity of typically insulating polymers. Similarly, these TNP constructs might be particularly useful as membranes for water purification. The power to control particle self-assembly facilitates the synthesis of a range of biomimetic materials.

Typically, one starts with a given nanoparticle (or "product formulation") and then uses theory/simulation to examine the final assembled superstructure. The properties of the resulting nanoparticle assemblies ("Product Performance") are then determined. These predictions are validated against experiment as a means of calibrating the fidelity of the models and the simulations/theory tools used.

There are several different MD simulator packages that can be used within this domain. While these simulators conceptually do the same thing, there are differences between their feature sets and performance. Typically, all of them mandate using a tool-specific file format to represent the input system, and there are syntactic and semantic differences between the force-fields they support (e.g. different functional forms may be used to describe the behavior of a bond). The syntax and the expressiveness of the simulator scripts also vary significantly across simulators. For instance, HOOMD blue [1] uses Python as its scripting language, while LAMMPS [1] has a custom syntax with just rudimentary support for loops and conditionals. Often, there is no clear separation between the scripts and the data they operate on: while the particles, their positions and chemical properties, as well as their relations (bonds, angles, etc.) are defined in a data file, the force-field specific parameters (bond, angle values, particle-specific Lennard-Jones parameters, etc.) must often be specified in the script file. Because of all this, a common API for MD simulators has remained quite elusive.

Given this extreme heterogeneity, how does a typical research group manage their MD workflows? Clearly, setting up an input datafile, even if it contains several instances of the same molecule type, is not possible by hand, due to the large number of particles and interactions involved. As a result, research groups typically write their own codes to generate the systems of particles, enumerate bonds, angles, etc., and tag them with their appropriate parameters coming from published force-fields. If the research group uses multiple simulator platforms, custom code is written to convert between input and output file formats. Similarly, custom code is written to extract the quantities of interest from the trajectory dump files and the final output file generated by the simulators. Writing such code is very tedious, error prone, and last but not least, extremely time consuming. Unfortunately, these scripts are often one-shot solutions, that are subject to opportunistic reuse, with no programming and naming conventions or well-defined interfaces. They tend to lack proper documentation, and the knowledge is often lost if the codes author leaves the research group. Still, this "shared" code captures an immense amount of domain-specific and simulator-specific knowledge that is of great value to the research group.

To address these challenges, we have adopted Model-Integrated Computing (MIC) [6], an established methodology in systems engineering. The underlying idea behind MIC is to facilitate the design of Domain-Specific Modeling languages (DSMLs) for various engineering, science, and other applications. DSMLs are meant to capture the essential functionalities and features of the individual components of a given process they are modeling, at the level of abstraction that is appropriate for the end users (i.e., domain experts). The models, in turn, are used to analyze the system, provide input to simulators, create documentation, and synthesize (parts of) the system software. MIC has been successfully used in diverse scientific and engineering domains.

1.1 Overview

In this paper, we present a novel approach to improve the current state of authoring MD simulation workflows. First, we describe a hierarchical, component-based approach to build large systems of particles and enumerate the particles' relations in a structured way. This component-based approach promotes structuring the system into reusable units, employs composition operators that automatically carry out coordinate transformations to stitch components together, allows for parameterization of components through generative modeling (e.g. specifying a carbon chain of length n, n being a parameter), provides a simple interface to export the particle positions and interactions into simulator-specific file formats, as well as to parse back the trajectories into the same component structure to simplify the computation of derived quantities from simulator outputs.

We then put the component-based particle generation technique into context through a case study that uses MetaMDS [12], a web-based MIC environment to define a MD simulation workflow. MetaMDS aims at capturing simulator-specific knowledge by allowing simulator experts to capture common concepts as reusable basic operations, that can be subsequently used as the basic (parameterizable) building blocks of simulations in a simulator-agnostic manner. Domain experts work on this higher level of abstraction to build more complex, domain specific simulation steps. Simulation steps capture knowledge in a particular domain (e.g. tethered nanoparticles, or tribology), without requiring the domain experts to have expertise in a particular simulator tool. Simulations are built from these reusable (and parameterizable) simulation steps, and can be included in workflows specified in a visual orchestration language similar to Scratch. The language supports variables and control structures common to imperative languages (loops, branching, etc.) and treats metaprogrammed simulations as first-class language elements. Variables of the orchestration language can be used as parameters of the simulation statements, and conversely, the results of the simulation can be extracted into variables through simulator-specific evaluators (specified by the metaprogrammer). The web-based interface also allows for submitting the simulation workflows to remote HPC clusters, polling the job status, and downloading the outputs.

1.2 Related Work

MIC has been used in many engineering and scientific domains. The main novelty of its application to materials science is that instead of the standard desktop-based toolsuite called the Generic Modeling Environment [7], here we use a new web-based toolset specifically developed for this domain. Within the MD simulation domain, the approach most closely related to our work is the Nanohub [9]. The Nanohub provides a web-based interface for a variety of simulation softwares. However, the interface is somewhat unusual. Each simulator has its own front-end and Nanohub serves them up via a Java-based VNC (screen sharing) client. The complexity of the variety of simulators is addressed through simplified user interfaces: the user is only presented with a limited subset of options to help guide the simulations. Most of the modules have a consistent look and feel, so the learning curve is reasonable. Visualization and plotting tools are often built into the GUIs. Jobs are submitted to clusters and the results copied back to the Nanohub space. Unfortunately, Nanohub has its set of limitations. The VNC-based user interface is not very responsive. User-level customization is not supported. The user can only change the parameters that Nanohub includes in its simplified interface. There is no interaction supported between various tools: the output of one simulator cannot be trivially fed to the input of another. Similarly, the primary mode of operation is interactive, since most tools have been developed with education in mind, and thus submitting a large set of jobs is not easily

accomplished. The Atomic Simulation Environment (ASE) [2] is a Python-based tool that can connect to many different simulation codes as *calculators* you plug into the environment. It has thus far been primarily developed for quantum mechanical calculations and thus, in its current state, is not suited for most MD simulations. The power of ASE lies in its ability to bring in many different codes and tools that can be linked together in a common interface. The use of the Python programming language makes it potentially easy to expand and interface with other math toolkits, plotting and visualization libraries, etc. However, using ASE has a steep learning curve for those with limited or no programming experience. For example, since each *calculator* may in fact be very different, the functions required to use a given calculator are often unique, so tool integration with ASE is not seamless at all. MDAPI [8] is similar to ASE, however developed for biophysical simulation, where the interface and computational engines are separated. However, similar to ASE, a steep learning curve is required and it no longer appears to be actively developed. Etomica [4] is a molecular simulation code written in Java, enabling it to be easily used and distributed via the web. While it does not allow end users to directly create custom simulations via the web, nevertheless, the user can run a variety of prewritten modules with custom parameter settings, similar to Nanohub. Etomica has defined a molecular simulation API, enabling simulations to be constructed from generic pieces, however, the API contains many specifications that are related to Java and the interactive frontend development, rather than generic simulation elements. In contrast to these existing efforts, our approach provides an extensible, fully customizable, web-based environment where simulator experts can build a library of simulation components, define how these components are mapped to (potentially multiple) simulation platforms, create full simulation templates that can be customized and run by domain experts without the need to write computer programs. There are a number of reports on systems with similar objectives in the literature outside the MD simulation domain – e.g. the SAFE framework for automating network simulations [10], or WorMS [11], a workflow framework for modeling and simulation in general.

2 Building complex systems of particles

The inputs of an MD simulator can be divided into three conceptual groups: 1.) initial configuration of the system of particles, which includes particle types, their masses, charges and other intrinsic properties, the geometry of the initial configuration (3-dimensional positions), velocities and accelerations, as well as a bounding box, often with periodic boundaries; 2.) the interactions between the particles. This includes bonds, which is a binary relation, angles (a 3-ary relation), and dihedrals (4-ary relation), and also non-bonded interactions, e.g. the Lennard-Jones potential, that can be described by particle-specific parameters. Each of these relations have a functional form associated with them, and a set of parameters, specific to a particular bond, angle and dihedral type; and 3.) the simulation script, that describes a series of steps to be performed on the system (loading/saving the system state, changing the box dimensions, maintaining or changing pressure/temperature/volume, etc.)

Creating the input data files for MD simulators is a tedious and error prone task. While generating a desired particle geometry is relatively manageable programatically, properly enumerating the interactions is not trivial. One common approach is to identify a repetitive pattern in the desired system, which is created by hand and cloned iteratively to fill up a given 3-dimensional box. This works well if there repetitive pattern is a standalone molecule, but becomes immensely complicated when there are bonds (or angles or dihedrals) *between* atoms in different clones (e.g. for carbon chains attached to a crystalline surface). In such cases, these bonds (angles, and dihedrals) are external to the repeating cell, and have to be added to the

system in a second pass after repetitively cloning the cells in a first pass. Periodic boundaries (what flies out on one side flies in on the opposite side) can further complicate this: in an extreme case, there might be a dihedral involving all four of its atoms in four different corners of the box.

Researchers designing MD simulators face such and similar problems every day, and significant effort is put into solving them over and over again. Unfortunately, however, there is not much opportunity for *reuse* within this framework. Even a small change in a basic building block of the system (e.g. changing the chain length, or replacing hydrocarbons with fluorocarbons) could require a drastic rewrite of the code that clones and stitches them together.

To attack this problem, we propose mBuild, a hierarchical, component-based approach to building complex systems of particles. Following the composite pattern [5] commonly applied in MIC, a component can contain particles and other components, recursively, which generates a hierarchical structure with the particles being the leaves of the hierarchy. A component has its own reference coordinate system. When two components are composed to form a composite component, their respective particles are imported into the composite's coordinate system with their coordinates unchanged (a simple union operation). In most cases, this is not the desired behavior, therefore, we define the following composition operators that are applied to the child component before adding it to the composite: translation to a point, rotation around the x, y and z axes, reflection around a point or plane, and the equivalence operator (described later in detail). For convenience, we also allow for deleting and renaming particles at composition.

The equivalence operator allows for designating points in the composite's local coordinate system that are declared equivalent to points in the child's coordinate system. Using these point pairs, it is possible to compute a rigid transformation (an affine coordinate transformation conserving scaling and orientation) that, when applied to the child component, will transform the child's designated points to the composite's respective points. Specifying three or more pairs of non-collinear points is sufficient to compute an unambiguous transformation matrix.

The rigid transformation F that maps a point (vector) v to its image $F(v)$ in a different coordinate system can be expressed as a multiplication by a rotation matrix $R \in \mathbb{R}^{3\times3}$) and a translation with vector $t \in \mathbb{R}^{3\times1}$.

$$F(v) = Rv + t \tag{1}$$

Given three or more points $P_i(x_i, y_i, z_i)$ and their images $P_i^{'}(x_i^{'}, y_i^{'}, z_i^{'})$ in the target coordinate system, we can solve the following system for R and t using the singular value decomposition to get the pseudoinverse:

$$\begin{bmatrix} x_1^{'} & x_2^{'} & ... & x_n^{'} \\ y_1^{'} & y_2^{'} & ... & y_n^{'} \\ z_1^{'} & z_2^{'} & ... & z_n^{'} \\ 1 & 1 & ... & 1 \end{bmatrix} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 & x_2 & ... & x_n \\ y_1 & y_2 & ... & y_n \\ z_1 & z_2 & ... & z_n \\ 1 & 1 & ... & 1 \end{bmatrix} \tag{2}$$

The equivalence operator has proved tremendously useful for the domain scientists. Instead of having to manually compute how two fragments of a molecule need to be rotated and translated so that they attach properly to one another, now it is sufficient to extend one of the fragments with three particles from the other one, and then to specify the equivalence operator over those three particles and their images in the other fragment. Consider Figure 1 as an example, described in 2-dimensions with unit distances between the atoms for simplicity. Component C_1 represents an end of a carbon chain: a CH_3 group with the image of a CH_2 group (marked with dashed lines). Component C_2 represents a fragment of an alcohol molecule, a hydroxyl group and a CH_2 group. In order to compose the two to form an ethanol molecule,

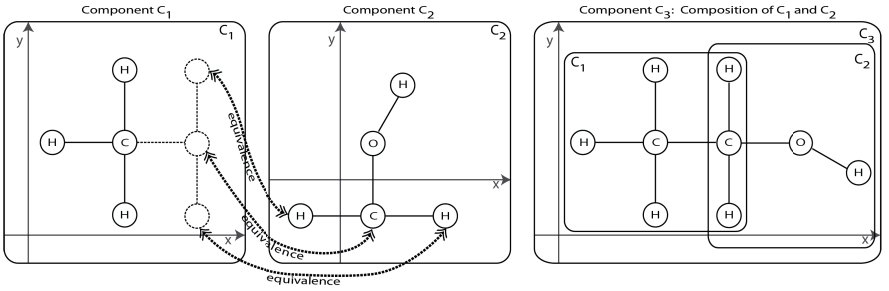

Figure 1: **Component composition.** C_3 is a composition of C_1 and C_2 with equivalence relations involving three atom pairs.

we express that the atoms of the CH_2 image in C_1 are piecewise equivalent to the atoms of the CH_2 group in C_2. The atom positions in the composite component C_3 are computed as follows. The atom coordinates of the first child component that is added (in this case: C_1) are imported without modification. However, subsequent child components that are part of an equivalence relation with already added parts will be subject to a coordinate transformation: the atoms of C_2 have to be rotated and translated (rigid transformation) such that the CH_2 group lines up with the CH_2 image in the already added subcomponent. The transformation matrix and translation vector are computed by solving Equation 2.

Taking this a step forward, we introduced the concept of *ports* to mBuild. Port (see Figure 2) is a component containing three ghost particles in a fixed, triangular arrangement. Ghost particles are used exclusively for equivalence-based compositions and are discarded thereafter. With the help of ports, attaching molecule fragments to one another becomes even simpler (see Figure 2). Each component representing a fragment can now include a port as a child component, rotated and translated to a desired position within the fragment's own coordinate system. Joining the two fragments is done by defining an equivalence relation between their respective ports – no adding of image particles are required. This ensures that components contain no information about their context, therefore they are easier to reuse across multiple simulation designs.

Figure 2 illustrates how to use ports to build an ethanol molecule from two fragments: C_1 containing a CH_3 group and C_2 containing a CH_2 group and an OH group, respectively. To attach C_1 and C_2 together, we add a port to C_1 translated halfway towards the direction where C_2 should be attached, i.e. to (0.5, 0, 0). Similarly, we need to add a port to C_2, rotated by 90 degrees and translated to (0, -0.5, 0). To construct the ethanol component (referred to as C_3), we compose instances of C_1 and C_2 with the equivalence relation expressing that $C_1.port \equiv C_2.port$.

With the help of ports, generative modeling of systems of particles with recurring patterns becomes feasible. For instance, consider the example generating alcohol molecules with alkyl chains of length n. Extending the previous example with component C_4 representing a CH_2 group, the repetitive part of the carbon chain, with two ports (one on each side of the carbon atom), we can describe an arbitrary long chain by inserting C_4 between C_1 and C_2 multiple times. Obviously, the equivalence operator must be used iteratively in this case, whenever a new subcomponent is created and added to the composite component.

Currently, mBuild components are specified as Python classes, derived from a base class that provides functionality for adding subcomponents or atoms to the component, as well as for the composition operators, including translation, rotation, reflection and equivalence. To create a new component, the user has to override the *create* method of the base class, implementing how

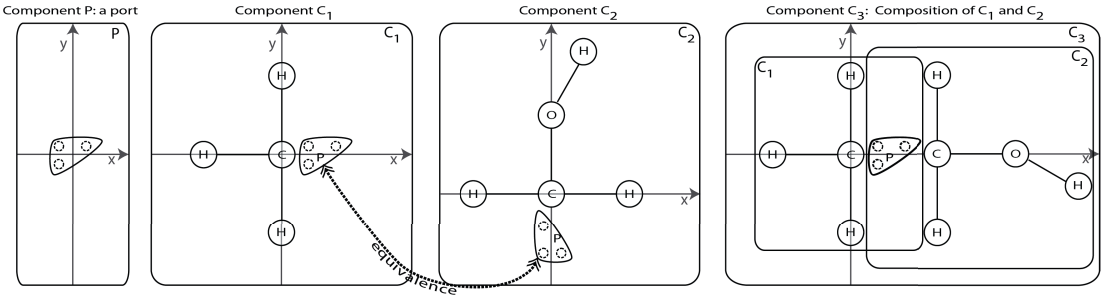

Figure 2: **Component composition using ports.** A port is a component with 3 ghost particles. C_1 and C_2 are composite components, both contain several atoms and a port as a subcomponent. C_3 is a composition of C_1 and C_2 with equivalence relations involving their ports.

port.xml:

```
<compound kind="port">
  <G pos=" 0.2,  0.1, 0" label="p1" />
  <G pos="-0.2,  0.1, 0" label="p2" />
  <G pos="-0.2, -0.1, 0" label="p3" />
</compound>
```

C_1.xml:

```
<compound>
  <C pos="  0,  0, 0" label="c" />
  <H pos=" -1,  0, 0" label="h1" />
  <H pos="  1,  0, 0" label="h2" />
  <H pos="  0,  1, 0" label="h3" />
  <compound src="port.xml" label="port">
    <translation pos="0.5, 0, 0"/>
  </compound>
</compound>
```

C_2.xml:

```
<compound>
  <C pos="  0, 0, 0" label="c" />
  <H pos=" -1, 0, 0" label="h1" />
  <H pos="  1, 0, 0" label="h2" />
  <O pos="  0, 1, 0" label="o" />
  <H pos="0.5, 2, 0" label="h3" />
  <compound src="port.xml" label="port">
    <rotation axis="z" angle="90" />
    <translation pos="0, -0.5, 0"/>
  </compound>
</compound>
```

C_3.xml:

```
<compound>
  <compound src="C_1.xml" label="C_1"/>
  <compound src="C_2.xml" label="C_2">
    <equiv my="C_2.port" target="C_1.port" />
  </compound>
</compound>
```

Figure 3: **XML definitions of the above components.** Carbon, hydrogen, and oxygen atoms are denoted with the C, H and O tags. G tags denote ghost particles. Subcomponents are added with the compound tag, with optional coordinate transformations defined on them in child tags.

child components are instantiated and transformed. For simple component definitions, where a fixed number of atoms or subcomponents are composed, we provide an XML-based language to describe the internals of the component (subcomponents, atoms, composition operators), avoiding the need to write Python code. Figure 3 shows the XML-based specification of the ethanol composition example shown in Figure 2. For complex cases, e.g. when the component includes repetitive structures, we support Python-based component specification only. For convenience, we support wrapping legacy chemical file formats (e.g. XYZ) into mBuild components, this way a large set of already existing molecules can be used as components in an mBuild based system.

Notice that mBuild components only describe atom positions. Relations between atoms, such as bonds, angles and dihedrals, are added to the system of particles in a postprocessing step after the system has been assembled recursively from its components and all atom positions become known. We use a rule-based approach to do this. Rules are simple Python methods that add a bond (or angle, or dihedral) to the system if a predicate over two (or three, or four) particles holds true. For instance, a rule may describe to add a bond of type "C-H" between a carbon and a hydrogen atom if their distance is in the interval [0.9,1.1]. A rule that adds an angle of type "C-C-H" over two carbons and a hydrogen can require that the hydrogen is bonded to a carbon, which, in turn, is bonded to the other carbon. Necessarily, the rules adding the angles to the system would need to run after the bond rules. After all the rules have completed, the system can be exported to file formats that can be read by the simulator tools.

To avoid the polynomial complexity of rule execution (e.g. enumerating all $O(n^2)$ pairs of particles for bond generation), mBuild constructs a kd-tree data structure to spatially index the particles. Kd-tree allows for querying the k nearest neighbors of a point in Euclidean space

in $O(logn)$ time, reducing the complexity of bond rule execution to $O(nlogn)$. As a result, currently, mBuild can handle systems with the number of particles in the 100,000 range, with the memory available to the Python interpreter being the limiting factor.

3 Case Study

3.1 Simulations and workflows

The MetaMDS environment was introduced in [12]. Here we present a brief overview of how simulations and optimization workflows are specified in MetaMDS.

Typically, simulations consist of the same conceptual building blocks, regardless of the MD simulator used, representing elementary operations such as reading or writing a data file, resizing the box, setting up integrators or evolving the system for a number of time steps. Their syntax is tool-specific, but semantically they are often similar or equivalent.

Expert users (metaprogrammers) are expected to create a rich library of these commonly used simulation steps. A simulation step has an identifier, a textual description, zero or more parameters with predefined default values, and a set of code templates for each supported simulator target (e.g. LAMMPS/2011.08, HOOMD-Blue 0.10.1). The textual description provides information for potential users of the given block on what functionality the step implements. Its semantics are captured in the code templates, which describe what code snippets will be generated from it and its parameters for a particular MD simulator target.

Simulation steps can have multiple parameters with name, type, default value, environment and visibility properties. Parameter values can be set or overridden at higher levels of the design hierarchy where the steps serve as building blocks. The visibility attribute may be used to mark a particular parameter as private, which means that its value cannot be altered outside of this definition.

The next level of the hierarchy groups simulations steps together in a well-defined order to specify an entire simulation. Parameter values that are public in the simulation steps can still be altered by the users prior to running the simulation. The simulator script for the selected MD simulator target is automatically generated from the simulation specification.

The workflow is the top level entity that connects simulation logic with particle data and server profiles. The user can load simulations into a workflow, define custom parameters, and set up a program flow that controls simulation execution. This is supported through a visual programming language built on top of Blockly [3]. An example workflow is shown in Figure 4. The next section will present a detailed explanation of this sample workflow.

To run a simulation, we need to set up specific simulators on local servers or use remote systems. To achieve this, MetaMDS maintains a list of server profiles where configuration settings (e.g. credentials to access the job manager (e.g. PBS), number of cores used, etc.) are stored.

Once the simulation has been designed, it is the workflow specification that aggregates all the required information: 1) the prototype of the particles used in the simulation and the description of how it will be replicated throughout the simulation box, 2) the specification of the simulation consisting of simulation steps, 3) the visual workflow specification describing how the simulation needs to be repeatedly run with all parameter values specified by the user including the selected simulation engine and finally, 4) the necessary information about the target server.

The orchestration engine is an extensible interpreter that executes the Blockly code. It can process a) control flow blocks (conditional branching, loops, function definitions and calls), b)

arithmetic and c) logic operator blocks, d) list and string handling blocks, as well as e) variable assignment and evaluation blocks, translating it into executable Python code.

Simulation blocks are handled by interpreter plugins, specific to the simulator platform on which the simulation is chosen to run. When the orchestration engine encounters a simulation block in the workflow specification, it locates the interpreter plugin registered for that particular block type, and invokes it with the following parameters: 1) the relevant part of the abstract syntax tree, including the extrinsic block itself and all of its descendants, and 2) the actual variable assignments that are used within the simulation block, as a key-value map. With this information at hand, the interpreter plugin invokes the simulator specific script generation and starts up the simulator.

Simulator blocks, along with their descendant simulation steps, are treated as opaque expressions in the visual language. The value of the expression is a handle, with which the orchestration engine can reference the outputs of the corresponding simulation run.

Introspection of simulation outputs is achieved through the concept of evaluators. Evaluators are Python functions, defined by the metaprogrammer for each supported simulator, that read the simulator outputs (identified by a simulation handle) to extract and return values of interest.

3.2 Isobar computation example

We demonstrate how MetaMDS is used to build MD simulation workflows through a simple example.

A large molecular system of tethered nanoparticles (called *Sample particle config*), specified in mBuild XML format, is added to the particle configurations database of the MetaMDS environment, and the property we need to compute is the isobar curve: the relationship between temperature and volume for a fixed number of particles.

The simplest way to solve this task is to execute an MD simulation that run an NPT integrator (conserving moles (N), pressure (P) and temperature (T)) iteratively at a series of temperatures within the temperature range of interest. The MD simulator script, after equilibrating the system, runs it in equilibrium state for a while to collect and log the system's volume at regular interval. The logged values are then averaged, and the pressure is computed, dividing the number of particles with the average volume.

In MetaMDS, the isobar workflow is expressed as follows. Variables are declared for the temperature range of interest (T_{low} and T_{high}), and the granularity of the computation ($n_{steps} = 10$). In a loop, we iteratively compute the 10 temperature values ($T_{current}$) at which we intend to identify the pressures through MD simulation. The NPT simulation (dark brown block) is executed within the loop's body. Notice that the input particle configuration is a property of the simulation block, which is set to the *Sample*

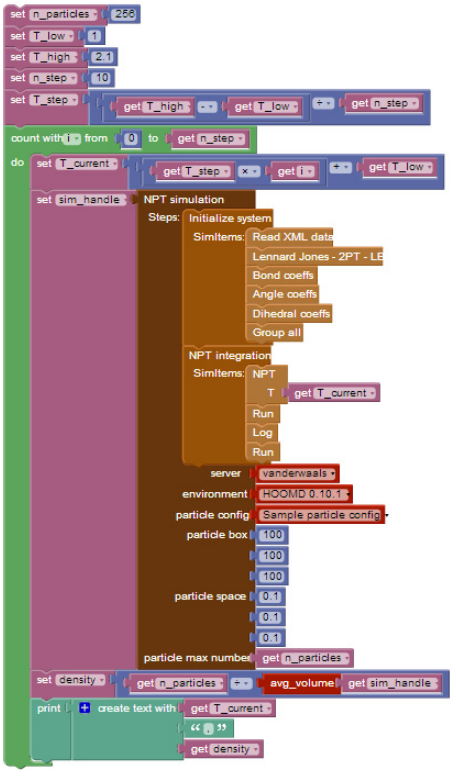

Figure 4: **Isobar computation workflow in MetaMDS.**

particle config database entry. The simulation is built up of two conceptual simulation steps: i.) System initialization step, that reads the particles from an XML file and sets the parameters for the bonded and non-bonded interactions defined therein, and ii.) NPT integration step, that sets the target temperature to $T_{current}$ for the NPT ensemble, runs the simulator to equilibrate the system, then turns on logging and evolves the system to collect the volume information.

Notice that the simulation block behaves like an expression in the workflow language. The value of this expression is a simulation handle, with which we can refer to the simulation later during program execution to access simulation results. One example of using the simulation handle is the *avg_volume* block: *avg_volume* is an evaluator that reads the logged volume values output by the simulator and returns their average.

Finally, the density is computed by dividing the number of particles by the average volume, and is printed to the standard output along with the current temperature. By the end of the last iteration, therefore, all 10 points of the isobar will be written out.

4 Conclusion

The key advantage of the presented approach to users is that the computational design of materials is decoupled from the design and implementation of simulation-based experiments. The MIC-based technique adopted here required multidisciplinary cooperation of various experts in putting together the modeling environment and the code generators on the back-end, as well as in building the component models. Users of the resulting environment interact with the web-based front end only, and hence, they only need to be experts in their own domain, e.g., nanoscale materials. They are able to design new structures from the provided building blocks and composition rules, adjust their attributes, specify the kind of simulation, property analyses and visualization they desire, pick from the available target hardware platforms - the tool environment then will automatically assemble the necessary simulation and other software components, execute them on the appropriate available platform(s), store the results in the database and display them on the front-end. Further, the availability of the system, software and hardware models creates the potential for the automatic tuning/configuration of key parallel code components for different designs/experiments. Since the results of these simulations are stored in a database, over time, this cyber-infrastructure will become a clearinghouse for NBB designs and their properties thus facilitating further scientific discovery.

References

[1] Joshua A. Anderson, Chris D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342 – 5359, 2008.

[2] Atomic Simulation Environment web page. `https://wiki.fysik.dtu.dk/ase/`.

[3] Blockly web page. `http://code.google.com/p/blockly/`.

[4] Etomica web page. `http://etomica.org/`.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[6] Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

[7] A. Lédeczi, A. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G Karsai. Composing domain-specific design environments. *IEEE Computer*, pages 44–51, 2001.

[8] MDAPI web page. `http://www.ks.uiuc.edu/Development/MDTools/mdapi`.

[9] NanoHUB web page. `http://www.nanohub.org/`.

[10] L Felipe Perrone, Christopher S Main, and Bryan C Ward. Safe: simulation automation framework for experiments. In *Proceedings of the Winter Simulation Conference*, page 249. Winter Simulation Conference, 2012.

[11] Stefan Rybacki, Jan Himmelspach, Fiete Haack, and Adelinde M Uhrmacher. Worms-a framework to support workflows in m&s. In *Proceedings of the Winter Simulation Conference*, pages 716–727. Winter Simulation Conference, 2011.

[12] Gergely Varga, Sara Toth, Christopher R. Iacovella, Janos Sallai, Peter Volgyesi, Akos Ledeczi, and Peter T. Cummings. Web-based metaprogrammable frontend for molecular dynamics simulations. In *3rd International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH)*, Reykjavik, Iceland, 07/2013 2013.