NOM Research Group

Digging into Digital Soil

REU: Proceedings from Summer 2002

Eric Chanowich
University of Notre Dame
Department of Computer Science

Section

1

Overview

Some Necessary Starting Information

Background

The NOM Research Group was formed to create a realistic stochastic simulator of the interactions of molecules in the top layer of the soil. The group, composed of biologists, geologists, chemists, and computer scientists from around the country, came together with a grant proposal that was accepted by the National Science Foundation.

The project, which started this summer, is expected to take up to 3 years for completion. This summer, we (the computer scientists) create a basic simulation, which the other scientists will then test against real-world experiments for validity. Feedback will be given back to the computer scientists as to what needs improvement in the simulation.

Unlike many scientific/computer collaborations where the computers would in support of some end scientific discovery, the NOM Research Group's goal is to create a realistic simulation that is supported by the work of scientists. It is speculated that the simulator created from this project may eventually be useful for evaluating the impact of such ecological problems as pollution and nuclear waste.

Personnel

Scientists from all over the country are collaborating as a part of the NOM Research Group. As previously mentioned, biologists, chemists, geologists, and computer scientists are all integral parts of this research.

I am computer scientist on the team, along with three other people at the University of Notre Dame. Dr. Gregory Madey (gmadey@cse.nd.edu) is a professor at the University and he serves as the computer scientists' team leader. Yingping Huang (yhuang3@nd.edu) and Xiaorong Xiang (xxiang1@nd.edu) are graduate students. I, Eric Chanowich, am in my senior year of undergraduate studies. Ironically, though I am the youngest member of team, I have been at the University of Notre Dame the longest.

Dr. Madey was essentially responsible for bringing together the team, as well supervising our daily progress and mediating interaction between scientists in other parts of the country.

Yingping Huang was our primary system administrator. He dealt with all hardware and software issues to keep our servers up and running. He was also our database administrator, creating tables in an Oracle database an employing data mining methods to obtain useful information from this data. Yingping also created the majority of the web user interface, which will be described later.

Xiaorong Xiang worked on parts of the core simulation and created a servlet to link the web interface with the core simulation. Xiaorong also served as a system administrator for one of the computers that she set up.

I, **Eric Chanowich**, was primarily responsible for the core simulation design and implementation. This included everything from the design of the class hierarchy to the implementation of movement and reactions among molecules to the graphical display of the molecules.

Frequent collaboration occurred between the four of us to brainstorm, debug, or simply provide insight into the recent actions of others. However, because of our limited chemical background, it was necessary to contact Dr. Cabaniss, one of the chemists in the NOM Research Group.

Dr. Cabaniss is chemist at the University of New Mexico. He provided us with useful insight into what we were trying to simulate, as well as how to accomplish this, including mathematical equations to calculate reaction probabilities based on molecular structure.

Section

2

The Chemistry

A Chemical Basis for Understanding the NOM Simulator

Molecular Attributes

Each molecule has attributes that identify it as a certain type of molecule. Several approaches are possible for representing molecules in a simulation. The molecules may be represented in an extremely detailed fashion with all of the bonds and atom position information included among other things. Molecules may also be represented in an more simplistic way, with each molecule represented simply by an empirical formula (simply the number of each atom type).

For our representation of molecules, a compromise was made somewhere in between the previously mentioned methods. Molecules, of course, include the empirical formula, which the number of each type of atom in the molecule. Molecules also contain a count of the number of functional groups including double-bond equivalents, total ring structures, phenyl groups, alcohols, phenols, ethers, esters, ketones, aldehydes, acids, aryl acids, amines, ring N, amides, thioethers, thiols, phosphoesters, H-phosphoesters, and phosphates.

Reactions

The molecular attributes that we chose to store about each molecule were chosen so that reactions, among other things, could be easily *and* accurately calculated. The basic reactions that we represented in our simulation included a second order reaction as well as first order reactions that involved both splitting and self-modification. Second order reactions occur when two molecules combine to create one molecule. First order reactions involve only one molecule, which may split into two or remain one molecule when it reacts.

Ester condensation is the lone second order reaction. Ester hydrolysis, amide hydrolysis, strong C=C oxidation are the first order reactions with splitting. Dehydration, mild C=C oxidation, alcohol oxidation, aldehyde C=O oxidation, and decarboxylation are the first order reactions without splitting that are included. Microbial uptake is also included. Microbial uptake is simply a micro-organism enveloping a molecule.

As per Dr. Cabaniss, these 10 reactions are assumed to be the only types of reactions that can occur. The probability of each type of reaction is calculated by formulas that Dr. Cabaniss provided us. These probabilities are determined solely by the individual molecular attributes and environmental variables (pH, temperature, etc.).

Flow

Flow describes the way molecules move. If water is present, such as when rain occurs, molecules may flow through a water column in the direction of the water's flow. Molecules always follow a general flow. In other words, the flow of a molecule is not based on its molecular attributes, but is rather based on environmental variables. Flow is measured in units distance divided by units time (ex. feet/second). Flow is not an absolute measure of transport, but rather an expected value. In other words, a flow of 5 mm/second does not mean that every molecule moves exactly 5 mm each second; it means that the average of all molecular movements is 5 mm each second.

Adsorption and Desorption

While moving, molecules may temporary become "stuck" to other materials. This phenomenon, where molecules become adsorped to adsorption sites and temporarily stop moving, is known as adsorption. When molecules become "unstuck" from adsorption sites, they are said to be desorped. Molecules are adsorped and desorped with probabilities based on the adsorption site's attributes, as well as the attributes of the molecule being adsorped or desorped.

.

Section

3

Software and Hardware

Learning the Basics of All the Resources Used

Java

About Java

Java is a high-level object-oriented programming language that was developed in the early 1990's by Sun Microsystems. Java is quite similar to other high-level object-oriented languages, such as C or C++. Structured programming is nearly identical to the C languages, though Java is based on classes (primarily through interfacing and inheritance) and the class syntax is quite different. Java provides programmers with a vast array of predefined classes through its API library. Classes outside of the java.lang library must, however, be manually included to be used by a program.

Unlike the compiled C languages, Java is an interpreted language. After writing a program, a Java programming must first compile the code into universal byte code, which is independent of the operating system. This byte code may then be read by native Java interpreters on nearly any operating system, making Java an incredibly universal programming language. For more information on Java, see http://java.sun.com.

Learning Java

When I began this job, I had never programmed in Java before. However, I found it quite easy to quickly learn the basics because of my extensive exposure to C++. I began the summer with Deitel & Deitel's How to Program Java, 2nd edition. I read the book from cover to cover before moving on to anything else, as I knew that a good knowledge of Java would form an excellent foundation for the programming that I needed to do for this research.

Swarm

About Swarm

Swarm is a Java package with a number of pre-defined classes and functions that are useful for agent-based simulations. A Java package is essentially an extension the basic Java library. Rather than adding each class or function in the Java code itself, a package can simply be included by adding a line the top of a Java file.

Any active element in a simulation is called an agent. Agents may represent anything from a particle of an atom to a human being. In agent-based simulations, agents act according to decisions they make for themselves, rather than instructions given to them by a central decision maker. For example, in human beings, body parts are sent instructions

from the brain, a central decision maker. However, when human beings interact with each other, they make decisions for themselves as to how to behave. Of course, one may argue that certain governments control their people's actions, but let's not get into that now. In this research project, the agents are individual molecules, which act according to their own probabilities or reaction and adsorption. Because Swarm provides an extensive array of classes and tools to aid programmers who are attempting to develop agent-based simulations, it was an obvious choice to use in building our simulation.

Learning Swarm

Swarm contains an extensive set of tools. Of course, learning *all* of the tools is often unnecessary. By doing several Swarm tutorials found at http://www.swarm.org, I was able to many of these tools demonstrated and sift through them to find those that were useful or necessary for this project. That website also includes a Swarm API, a listing of all Swarm classes and methods, which proved extremely useful as a resource.

Hardware

Throughout the summer, we used several computers. These machines acted as servers, both for storing versions of the programs, as well as assembling and running Java/Swarm code. The machines and their specs are listed below.

Computer Name	Operating System	Processor	RAM
joy.cse.nd.edu	Redhat 7.2	dual 800 MHz	1.5 GB
foyt.cse.nd.edu	Redhat 7.2	dual 400 MHz	1 GB
tenor.cse.nd.edu	Redhat 7.2	dual 800 MHz	0.5 GB
symphony.cselab.nd.edu	Windows 2000	dual 733 MHz	2 GB
bigband.cselab.nd.edu	Intel Solaris 8	dual 733 MHz	2 GB
gemini.cse.nd.edu	Windows 2000	dual 400 MHz	0.5 GB

Because of its relative processing speed and memory, joy.cse.nd.edu was the most frequently used machine.

.

Section
4

Design and Division

Program Design and Division of Labor

Program Design

The task of designing a stochastic simulation of the interactions of natural organic molecules in soil is initially very daunting. However, careful and adaptable program design can help make this much easier.

Yingping, Xiaorong, and I met several times to brainstorm a design for the program before we ever wrote a line of this program. With little experience using Swarm, our initial design was based loosely off of some of these tutorials. Of course, this design gradually mutated into its current state as we began to understand Swarm better and additional functionality was required of the program. However, our basic class hierarchy remained relatively static, with new classes usually added at the bottom of the pyramid. For more information on the class design, please see "*Section 5: Core Simulation* – Class Hierarchy".

Division of Labor

With three people doing the majority of the programming on a large project, division of labor is essential. When we first began writing the core simulation, Yingping, Xiaorong, and I decided to design on a general interface for each class and then split up the classes so that each of us were writing a few.

I was assigned the *ModelSwarm* class, when became the housing for actions of the core simulation. Xiaorong initially wrote several of the objects of the core simulation, including the *Molecule* and *ProbabilityTable*. Yingping worked primarily as our system administrator and database administrator in the early stages on the project.

As the project began to involve, we decided the best way of interfacing the project to scientists would be through "web wizard" that could customize and launch simulations on our local servers and provide users with detail analysis concurrently.

With many new tasks as a result, this meant a shift in the division of labor. I remained working full time on debugging and adding functionality to the core simulation. Xiaorong continued to work partially with me, coding and debugging. She also worked with Yingping on the front and back ends of the interface, writing servlets and corresponding web pages. In addition, Yingping worked on creating dynamic analysis web pages from the information placed in the database by the simulation.

Section

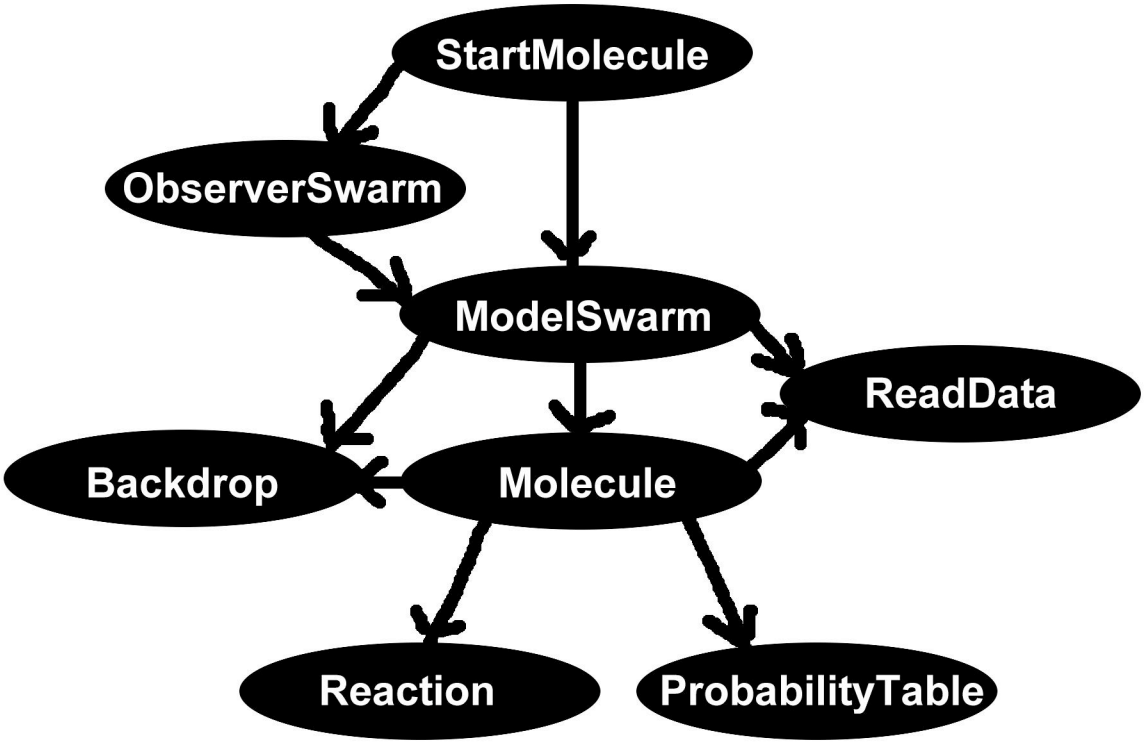
5

Core Simulation

Overview and Dynamics of the Central Engine

Class Hierarchy

The class hierarchy is illustrated in the diagram below. An arrow indicates that the class being pointed to is instantiated in or used by the class being pointed from.



Class Descriptions

Each of the classes has its own unique purpose(s) while enjoying close interactions with at least one other class. They are each briefly described below.

StartMolecule invokes the simulation. This class essentially encapsulates the entire simulation and provides necessary parameters for Swarm.

ObserverSwarm contains all of the display methods. This class is only included when a GUI is desired, which was useful for debugging.

ModelSwarm is the central simulation class, as the diagram may indicate. This class performs modification on molecules (movement, adsorption, reactions, etc.).

Molecule is the base object class in the simulation. Instances of this class represent molecules. It includes all necessary information about the molecule as well as methods to modify this information.

ReadData reads information from the database into the simulation. This information includes user specifications for both molecules and environmental parameters.

Backdrop contains information about adsorption sites including the type of adsorption site and its corresponding probabilities.

Reaction is instantiated inside of a molecule each time a reaction is to occur on that molecule. This class performs all of the necessary operations for a particular reaction when the molecule undergoes that reaction.

ProbabilityTable is an optimizing class that contains a table corresponding to type of molecules and its various reaction probabilities. This class allows the simulation to simply read from the table, rather than performing complex calculations, each time a molecule of know type is instantiated.

Section

6

ModelSwarm

Variables and Methods of the Central Simulation Class

Variables

ModelSwarm is the "brain" of the simulation. It links all of the other key simulation classes, both through instantiation and operation. Below are some of the important variables instantiated in ModelSwarm. They have been divided into logical groups.

Swarm Variable Types

These variables types are defined by the Swarm package:

Grid2dImpl grid – 2-D array of references to *desorped* molecules corresponding to their current position in the grid. References are to type Molecule. For all grids, (0,0) is the upper right corner

Grid2dImpl adsorpedGrid – 2-D array of references to *adsorped* molecules corresponding to their current position in the grid. References are to type Molecule.

ListImpl moleculeList – List of all molecules currently in the simulation. Each list position is of type Molecule. The list is managed so that its size is only equal to the maximum number of molecules in the simulation.

ScheduleImpl modelSchedule – Schedule of ModelSwarm methods to call at each time step. The schedule is executed by StartMolecule.

Created Variable Types

These variables types were created by the NOM Research Group:

Backdrop backdrop – 2-D array of adsorption sites. The adsorption site type and necessary information are contained in their relative locations in the grid.

ProbabilityTable p – Table where key (molecule type, reaction number) returns the reaction probability for that molecule.

User-Defined Variables

These are standard Java variable types that are read in from a database, based on the user's specifications for the simulation.

int xGridSize, yGridSize – Integer values representing the dimension, in grid squares of the Molecule grids. xGridSize refers to the left-right size. yGridSize refers to the up-down grid size.

int xFlow, yFlow – Integer values representing the general 2D flow of molecules, in the x-direction and y-direction. +x is right. –x is left. +y is down. +y is up.

int xFlovDev, yFlowDev – Integer values representing a linearly distributed deviation of flow movements in the x-direction and y-direction. If the flow deviation in a direction is 0, the molecules will move exactly according to the flow.

boolean printReaction – Flag to determine whether or not reaction details should be printed to the terminal. This is not used in the public release but it was particularly useful for debugging. True prints reaction details and false suppresses them.

boolean fillGridFirstTime – Flag to determine whether or not the grid should be filled with molecules before the simulation begins to run. True fills the grid while false does not.

boolean addMoleculesSubsequentTimes – Flag to determine whether or not molecules should be added into the simulation after it has started. True adds molecules at subsequent time steps while false does not. fillGridFirstTime must be set to true if this is set to false. This is checked for in ModelSwarm's initialization.

boolean bounceMoleculesOffWalls – Flag to determine whether or not molecules should "bounce" off of the walls of the simulation. "Bouncing" involves molecules being reflected back off of the bounds of the simulation. If the molecules are not "bouncing" off of the walls, the are wrapped around in the x-direction and they disappear through the bottom (+y-direction) of the simulation. Molecules always "bounce" off of the top (-y-direction) of the grid.

ModelSwarm Internal Variables

These are variable types that are used for ModelSwarm's internal purposes:

Stack openListPositions – Java Stack of integer values corresponding to open positions in the list of molecules. Values are "popped" on when list positions open up and "popped" off when list positions are filled. This allows the list to manage itself so that its size does not grow infinitely.

Stack futureListPositions, dyingListPositions – Java Stacks of integer values used in association with openListPositions so that molecules are not over written too early.

int simulatedTime – Integer counter representing the number of simulated time steps that have occurred. Each simulated time step represents a time period. The time period is currently set to 15 minutes, but can easily be changed.

Methods

As ModelSwarm performs the operations of the simulation at each time step, it contains many important methods. The methods of ModelSwarm can be viewed as initialization methods, repeating methods, or supporting methods. The methods and brief descriptions are listed below.

Initialization Methods

These methods are all automatically called when ModelSwarm is instantiated. They are not called again.

public ModelSwarm(Zone zone) – Class constructor that calls the constructor for ModelSwarm's super-class. This method begins the connection to the database. It also initializes a probe map for GUI interactions.

public buildObjects( ) – Swarm default construction method that initializes the objects used by ModelSwarm. User-defined values for objects are read in this method. All molecules are created and placed in the newly-created grid and list with desired distribution of molecule types. The grid may be filled with molecules if the fillGridFirstTime flag is set to true. The backdrop, a grid of adsorption sites, is created as well.

public buildActions( ) – A Swarm ScheduleImpl is created to be executed automatically at each time step. The ScheduleImpl is an ordered list of methods and parameters to give the methods each time they are called.

Repeating Methods

These methods are called at each time step, as defined by the schedule that was created in buildActions. They are listed in the order that they are called each time. The methods are all public and return "this" , as specified for methods to be added to a Swarm ScheduleImpl..

public Object cleanup( ) – Responsible for list clean-up. This method checks the Stack dyingListPositions. All Integer values in this stack are move to the Stack futureListPositions, which are later moved to the Stack openListPositions. This essentially creates a buffer time period for the molecule to finish its time step without being removed.

public Object addMolecules( ) – Adds molecules into the grid based on desired density. Molecules are only added into a rectangle that has dimensions xGridSize by yFlow and contains the two upper corners of the grid. This assumes that yFlow is positive and xFlow is 0. If the area of that rectangle is A, and the desired molecule density is D, then X/A = D where X is the expected number of molecules added at each time step.

public Object move( ) – Adsorps, desorps, and move molecules in the grid. Molecules that are in adsorpedGrid are checked, based on probabilities of desorption to see if they should desorp. The probability of desorption and adsorption is determined by the molecule's properties and the adsorption sites properties. Likewise, molecules in the grid (non-adsorped) are checked, according to their adsorption probability and the adsorption site below them, to see if they should adsorb and move to the adsorpedGrid. After this is complete, molecules are stochastically moved based on the desired xFlow and yFlow. If the bounceMoleculesOffWalls is set to false, the molecules are removed when they travel through the bottom of the grid (xPosition > xGridSize). Molecules that are in the adsorpedGrid do not move. Molecules are moved in a stochastic order by first choosing a random list position to start from and then a random direction to iterate through the list from, rapping around either "end" of the list.

public Object react( ) – Checks for and executes reactions for each molecule. When a molecule is created, the different probabilities of each type of reaction, in reference to the overall probability of reaction, are summed on a number line and normalized from 0 to 1. Thus, with 10 reactions, there are 11 segments on the number line (with bounds of 0, each

of the summed probabilities, and 1). Each time a molecule is checked for a reaction, a random number from 0 to 1 is generated. The number is measured against the bounds of the segments on the molecule's reaction number line. If the number falls in the last segment, which it does about 99% of the time, there is no reaction. If it falls in another segment, that type of reaction takes place. The reaction takes place by calling the molecules react method with a parameter of an integer representing the reaction type. The molecule mutates itself accordingly and keeps a history of this reaction.
All but one of the reactions are first order reactions so the molecules merely undergo some sort of mutation. However, for the lone second order reaction, another molecule is required to react with. This companion molecule is found using the spiral algorithm (see *Supporting Methods*), which finds the closest molecule in the grid. If it is determined that the other molecule would like to participate, the second order reaction takes place.
Molecules react in a stochastic order that is determined the same way as in the move() method.

public Object writeDB( ) – Writes information about molecules to the database. Because reactions are of particular interest, information is written for each molecule that reacted on the current time step. All relevant information about the molecule, including moleculeID, attributes, parentID, and reactionType, are written to the database for analysis.

public Object utilities( ) – Performs necessary utilities at the end of each time step. This method updates the simulatedTime and moves Integers from futureListPositions to openListPositions. Additionally, utilities suggests that the garbage collector collect every 10 time steps.

Supporting Methods

private Molecule findNearest(Molecule molecule, boolean findEmpty, boolean useGrid) – Employs self-developed spiral algorithm to return the nearest grid location with a desired value. A location is chosen to start "spiraling" from until a desired value is found. Psuedo-radii are spiraled where a radius consists of a square that incrementally radiates from the starting point. The first radius spiraled contains the 8 squares surrounding the starting point.
Because multiple correct values may be found in a given radius, the first correct value discovered is accepted. To avoid redundancy, a location is randomly picked for each radius to start spiraling from. Additionally, the direction of that spiral is randomly generated to be clockwise or counter-clockwise. Adjustments have been made to the algorithm to optimally "jump" around out of bounds locations, rather than exploring them.
findNearest accepts a molecule as one of its parameters. This molecule is the starting point to "spiral" from. It also accepts a findEmpty flag which find the first empty grid location if true or finds the first molecule if false. The boolean useGrid tells the method which grid to search through. True uses grid (non-adsorped) and false uses adsorpedGrid.
This method is called by the repeating move and react methods to find nearby locations with desired locations.

private removeMolecule(Molecule molecule) – Marks a molecule to be removed from the simulation. This method does not actually remove the molecule, but actually sets up the eventual removal by placing the molecule's list position on the Stack dyingListPositions, which is later modified by the utilities( ) method.

private String reactionName(int num) – Returns a string containg the name of the reaction represented by num.

private ListImpl getMoleculeList( ) – Get method to return the list of molecules.

private Grid2dImpl getGrid( ) – Get method to return the grid of molecules (non-adsorped).

private Grid2dImpl getAdsorpedGrid( ) – Get method to return the adsorpedGrid of adsorped molecules.

private Backdrop getBackdrop( ) – Get method to return the Backdrop grid of adsorption sites.

Section

7

Memory Management

Mitigating and Avoiding Memory Leaks

Mitigation

In a system with many dynamically created variables, memory leaks seem nearly inevitable. Java uses an automatic garbage collector that supposedly intermittently cleans up the memory used by a running program. However, we found that the garbage collector really only works in theory; it fails to clean up all of the memory leaks. In fact, memory leaks are so rampant in our program that they eventually lead to an unintentional termination of the simulation when the programs heap reaches its maximum size.

Xiaorong and I spent many hours attempting to deal with the memory problems that we encountered. With memory leaks, there are two solutions; you can attempt to remove the problem or lessen it. After several failed attempt at removing the problem, we decided that attempting to lessen the memory leaks was our best way to move forward.

Dynamic management of the list is employed in ModelSwarm. When molecules are removed from their list position, that list position is held in a stack to be "popped" off next time an open list position is needed. The list size starts at 0 and grows until the simulation reaches a state of equilibrium where the current number of molecules in the simulation does not exceed the maximum number of molecules that were simultaneously in the simulation. Thus, the list does not grow infinitely to an enormous size. Effectively managing dynamic data structures is one way to lessen memory leaks.

Another method we successfully employed for lessening the effects of memory leaks was to "null out" objects when they are done being used. The Molecule type and its various instantiations result in the majority of our memory leaks. Molecule contains a large amount of information and instantiates several other large data types as part of its internal information and methods. We wrote a destroyMolecule( ) function for the Molecule type to call whenever the Molecule was no longer needed. The destroyMolecule( ) method sets all of Molecule's attributes and instantiated data types to null. This simple addition doubled our run time from about 30,000 to 60,000 time steps.

Avoidance

While mitigation is certainly helpful for managing memory leaks, it is not an ideal solution. It is particularly unacceptable in a simulation, such as ours, that needs to run for long periods of time in order to be effective.

As a result are currently working on attempting to entirely wipe out memory leaks in our program. Though it is not yet implemented, my latest scheme for avoiding memory leaks involves creating a "pool" of molecules. Much like managing the list, these molecules will

be managed so that the maximum number of molecules created will never exceed the maximum number of molecules in the simulation at one time. As opposed to millions of molecules being created, only a few thousand will be initialized with this scheme.

Recycling isn't just useful for the environment. It's also very handy for the programmer. The molecules will be reused by calling an internal reset( ) method that will set all of the molecule's attributes back to original values or desired values. Rather than *constructing* a molecule with desired attributes, we will *reset* a used molecule to these values.

I anticipate that this "pool" of molecules will indeed be able to eliminate most, if not all, of our memory leaks in the program. Scientist may desire that the program run for upwards of 300,000 time steps and hopefully, these changes will allow the simulation to run for an infinite (or at least very long) time.

Section

8

Lessons Learned

Knowledge Obtained From a Summer of Digital Digging

Java and Swarm

As one of the most widely used and versatile languages, Java is a very important tool for the programmer. Java applications can range from small, web-based applets to large, computationally intensive applications, as demonstrated by our simulation.

Like any programming language, Java certainly has its advantages and disadvantages. As previously stated, Java is extremely versatile language. With its built in GUI objects, graphics can be easily displayed and manipulated. Likewise, Java has a very large API library that contains objects that can be used or easily derived into nearly anything a programmer could ever need, thus making it a very programmer-friendly language. For example, Java code is very easily integrated with a database by use of it JDBC library. Of course, Java is platform-independent so its "compiled" byte code can be run on any system with a Java interpreter.

However, Java has several critical downfalls. Because of its platform-independence, Java is slower than a language that is compiled for a specific platform, such as C++. Additionally, Java does not allow manual memory management. While the garbage collector is a nice idea in theory, it is not always effective, as we found, and it does not allow for objects to simply be deleted.

I could write for pages about the advantages and disadvantages of Java, but the majority of those relevant to this project and addressed above. Overall, Java is a quality language due largely to its portability, versatility, wide usage, and expansive libraries. Additionally the Swarm package is only available for Java and Objective C.

Swarm proved to be a very valuable tool for this simulation. It provided convenient objects and methods. Of particular use were scheduling, grids, display rasters, and robust random number functions. By writing native objects and functions rather than using Swarm, this simulation could probably be written more efficiently without Swarm. However, I doubt that such a speed-up would outweigh the convenience of Swarm.

Teamwork and Software Engineering

It is very easy to write software when only one person is working on it. The lone programmer always knows exactly what he or she has done to the program. However, using a team of programmers, rather than an individual, brings many advantages including a greater knowledge base and increased productivity. Though, having a team of programmers also introduces communication and design issues into the process. The

team members must be able to communicate very effectively and share a mutual vision and understanding of the overall programming design.

Despite varying backgrounds and levels of education, our team was able to work very well together. We initially began our group cohesion when we met to come up with an initial program design. We carefully examined the chemical basis for the simulation that we were to create. We held a number of discussions about potential structures for the program. Through careful research and open interaction, we were able to carefully convey our opinions and concerns for the program design, arriving at an excellent starting point.

After this major step had been taken, we began to code. Because we had established a rigid structure for the program and its class interactions, we were able to easily divide to write the classes. When we had completed our individual coding, putting the classes back together was a very simple task requiring little manipulation as a result of the extensive initial program design. Additionally, whenever adjustments needed to be made to the program, they were easily accomplished due the structure of our program design.

Teamwork, clearly defined objectives, and rigid, versatile program design are the heart of successful software engineering.

Section
9

The Future

What to Expect Next from the NOM Research Group

Around the Corner

The current pressing issues are memory leaks, full implementation of adsorption, and data analysis. I am personally concerned with memory leaks and adsorption, as they both directly relate to the core simulation. As we prepare to make the simulation widely available to scientists involved in the research, the memory management is crucial. The scientists will want the simulation for lengths of time that we can not currently achieve. Hopefully, the molecule "pool" discussed in the *Memory Management* section will solve this problem.

Additionally, adsorption has not been fully implemented. The framework exists but details need to be filled in. We plan to carefully study documentation on adsorption and discuss our implementation plans with Dr. Cabaniss. With little coding necessary, fully implemented adsorption should be available shortly.

I will personally be continuing to work with the NOM Research Group, particularly to take care of the issues discussed above. I look forward to completion of our prototype and the eventual launch of the simulation to the scientists involved in this project.

Looking Ahead

After the initial release of the prototype, the scientist will measure the accuracy of the simulation versus expected results or physical experimentation. They will then make suggestions on how to improve the simulation to optimize its accuracy. The simulation will then be re-evaluated for accuracy and suggestion will once again be made. This process will essentially repeat until a terminal accuracy has been reached for the simulation.

Upon completion of the research, the program will likely be used by those simulating environmental situations, including but not limited to pollution and nuclear waste. While optimistic, it is extremely exciting to imagine that the work we have done this summer may some day lay the foundation for research that saves the Earth.