Development of a Simulator for Universally Programmable Intelligent Matter:

Progress on Universally Programmable Intelligent Matter

UPIM Report 8

Technical Report UT-CS-04-519

Alex Andriopoulos[*]

Department of Computer Science
University of Tennessee, Knoxville

October 24, 2003

Abstract

This report describes the design and development of a prototype simulator for chemical reactions under consideration for "Universally Programmable Intelligent Matter." It also describes simulator requirements, input file formats, Java classes and methods, installation procedures, and usage.

---

[*] This research is supported by Nanoscale Exploratory Research grant CCR-0210094 from the National Science Foundation. It has been facilitated by a grant from the University of Tennessee, Knoxville, Center for Information Technology Research.

This report may be used for any non-profit purpose provided that the source is credited.

The background on Universally Programmable Intelligent Matter (UPIM) is virtually nonexistent. This research is exploratory in nature and has a limited number of objectives. The first objective is to develop a model of computation compatible within the constraints of molecular processes. The second is to identify at least two universal sets of building blocks for programmable intelligent matter. The third is to develop methods for interfacing with additional molecular building blocks for sensing conditions and causing effects in the external environment. The fourth is to develop a prototype simulator to investigate characteristics unique to molecular computation.

Intelligent matter is any material in which individual molecules or molecular clusters can work together to accomplish a purpose. Intelligent matter comes in different forms such as solid, liquid or gaseous. The most typical forms are liquids and membranes. Universally programmable matter consists of a small set of molecular building blocks. These individual blocks are universal because they can be arranged in any way or order. The programmable part is used because a computer program can describe what task is being accomplished by the rearrangement of individual blocks. A computer program simulates the behavior of the material at the molecular level.

Instead of engineering materials that have only one purpose, Universally Programmable Intelligent Matter is designed to be universal. Much like a computer is designed to meet the needs of its particular user, universally programmable material is designed to meet the needs of the programmer.

The research objective was the development of a prototype simulator, which can simulate the molecular building blocks of programmable matter. Also, keep in mind that the above stated objectives of developing a model of computation compatible to molecular processes and identifying two universal sets of building blocks was actually reached. These two objectives are being handled by SK calculus. I will not go into any great detail concerning the validity of SK calculus because that was not my focus.

We have accomplished our task of designing and implementing a prototype simulator. The simulator was designed in Java because Object Oriented programming was the best choice for simulating molecules in action. Object Oriented programming was seen also as the best choice for keeping the simulator very modular. The leading thought behind this argument was the fact that this will be the first simulator created.

Therefore, the potential for redesigning part or parts of the simulator were perceived to be highly likely. In order to make this possible redesigning easier to accomplish modularity was a seen as a main priority.

The simulator can read in a set of programming rules, which is also the description of the allowable rearrangement of building blocks based on SK calculus. Therefore, abiding by the rules of the model of computation compatible with the constraints of molecular processes the simulator can read in an initial contents file, which is the initial setup for the test to be run. One way to think of these initial contents is analogous to a laboratory experiment, in which chemicals/molecules are being added to a beaker. The reaction that is about to occur is based on the properties of the individual molecules. The simulator represents these properties by the rules' descriptions. Therefore, the only reactions that can occur are the ones described by the rules. It follows that, if a rule is not read in then it cannot be applied to a given set of molecules.

Once the Simulator has read in the initial contents file and the rules file, then it is ready to start simulation. The user will be prompted with a list of six choices, which allow the user to determine the next course of action. Examples include:

- Run through the simulation sequence five more times.
- Display a graph, which shows all the possible reactions and the number of times they occurred.

Also, you can add additional resources to continue looking for new possible reactions. Resources are simple molecules made up of two, three, or four molecular groups bonded together. They could potentially react with the contents of the Simulator if the conditions are correct. The user can also choose the choice of stopping the simulation.

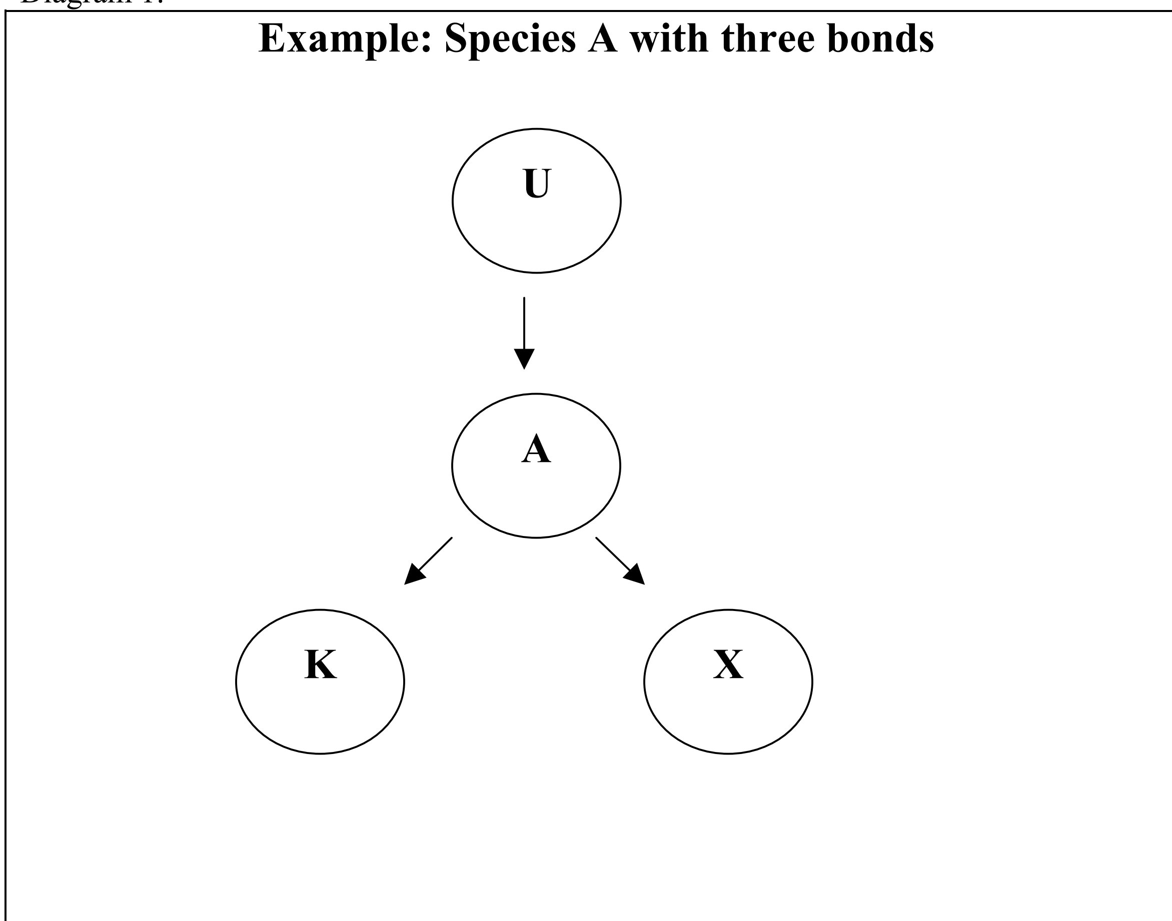
These initial test files are ones that include functional groups, or chemical reactions that are possible by programmable matter. These functional groups are represented by species, which have bonds and names that represent their abilities. These functional groups are often embedded within a cluster or arbitrary molecular network. These arbitrary networks are represented by trees, which make understanding how they are connected with each other easier to visualize.

Then by using SK calculus, which is our model of computation we can simulate two simple substitution rules. The first rule is the delete rule, which is used for breaking

down waste products formed by reactions. The initial molecular network you start with will react with resources found within the simulator, and this will produce waste products, which can be systematically broken down into smaller molecular networks by the deletion rule. The second rule is the replicate rule, which is used for replicating species found within molecular networks. By replicating species a molecular network can be replicated until a progressively larger network is created. Also, each time a functional group interacts within a larger molecular network it is accomplished by following the rewrite rules of combinatorial logic. The two basic operations are the S and K operations used in SK calculus, but what the simulator represents is, by using two simple operations of S and K, we are able to delete and replicate anything that can be computed on a computer.

Now we will discuss the initial simulator requirements. The general description was a program which will consist of an interface of which there will be the ability to read in a file. This initial file is composed of individual species, which represent different functional groups. These species also contain a number of bonds. The number of bonds cannot be changed after initially being set. Some species examples are A, R, and V, these particular species have three bonds each, and they must be connected to three other species. Below is Diagram 1 that contains an A species and three other species that are bonded to it. This illustration will give you a better visualization of what is being described.

Diagram 1:



As you can see from Diagram 1, an A Species has three other species connected to it, a U Species, a K Species, and an X Species. Also, note that the U, K, and X Species only have one bond. A bond as described in Diagram 1 is similar to a chemical bond that exists between two atoms. For instance, in a molecule of water there exist two hydrogen atoms and one oxygen atom, which together make up one water molecule. Within the research field of universally programmable intelligent matter the way the species are bonded together is similar.

The first requirement of the simulator is it must be able to read in the initial file. We will sometimes refer to this file as input, template, or rules. This file is needed in order for the simulator to establish internally what reactions are allowable. This file will also be necessary later on to pattern match between what reactions are allowable and what the actual internal molecular formation is composed of. The template or templates contained inside of the input file are the actual description of the reaction definitions to

be used. The template can be thought of as the laws of nature. These laws are constant and do not change. As well, once the simulator is running you cannot change the already described reaction definitions. If you want to do this you must quit the simulation and then restart it with the reaction definitions you want.

The second requirement for the simulator is to be able to produce output of the results. An important result is a display of the new chains that have occurred during the simulation process. The simulator can display the contents of the simulator any time after the initial input and contents files have been read in. Once the simulator is running there is a user menu that displays a list of potential options and one of the choices is to display the contents of the simulator. Another option is to dump the contents of the simulator into a file. This will give the user a chance to see what the new chains are as well as the opportunity to have the contents of the simulator in the same format as the contents or soup file. This is a handy feature because you can stop the current simulation after the contents have been dumped into a file and then resume simulation with a new input file, but with the remaining contents from the simulation previously ended.

The third requirement was to be able to display the complexes matching specified templates. This output can be reached from the menu list of options. The list of options is displayed to the user after they select a particular item from the menu list. Once that item has completed its assigned task the user will again be prompted to choose another option. This form of output is under the print graph choice. After the simulator begins executing you can request this option. The screen will then display a printed message with instructions to view the newly created graph. This graph will display the number of cycles run through by the simulator and the names of possible reactions and how many times they have occurred.

There are five other possible views for system output that were not deemed vitally important to the completion of the prototype. Therefore, they were not implemented but are left for the future. These views or structures that programs might create are to have nanotubes of specified diameter and length. Another structure is of membranes with pores of specified diameter and density. Another possible structure created is one that would show membranes with cilia that flex in a specified direction upon command. Also, a display of complexes at specified "addresses", or to classify them according to provided

templates. Since the simulator was programmed in an object oriented language with many built in methods that allow for the creation of easily made graphical user interfaces, these four structures can easily be created in the future.

The fourth requirement is to be able to fill in the simulator with molecular chains for testing. This specification can be reached through the menu list in which the user is asked if they would like to read in additional soup files or reaction resources. This means that any time a user would like to add more molecules or molecular chains they can do so, which is useful for testing purposes.

The molecular species names must be read into the simulator and their number of bonds or binding sites must be stored as well. If a reaction is attempted that calls for the number of bonds to be altered to a new specification this will cause an error. This error will be displayed on the screen and the user will be given a descriptive message stating this fact. The reaction definitions (or rules) must also be read in along with the species names and their number of bonds. The reaction definitions are described in a particular format that must be followed exactly or many errors will be generated. Before continuing, into greater detail of the reaction definitions we will digress for a moment with an example of a complete input file. This will help the reader to follow along in the ensuing discussion. The input format will be further explained by looking at Diagram 2 below. Diagram 2 shows an example of a K Reaction input or rules file.

Diagram 2:

Example: K Reaction Input File

```
Species: A 3
Species: D 1
Species: P 1
Species: Q 1
Species: Y 1
Species: X 1
Species: R 3
Species: U 1
Species: V 3
Species: K 1
Species: S 1
Species: N 1

Wildcard: * 4

MolecularTemp: temp1 species: * 0 3 5 species: A 1 2 species: K 4
    0 -> 1
    1 -> 2 -> 3
    2 -> 4 -> 5
END
MolecularTemp: temp2 species: D 0 species: Q 1
    0 -> 1
END
MolecularTemp: temp3 species: D 0 species: Q 1
    0 -> 1
END
Reaction: K_Reaction
    Probability: 0.20
    Requires: temp1 temp2 temp3
    Description:
    temp1.0-temp1.1 + temp1.1-temp1.2 + temp1.2-temp1.5 => temp1.0-temp1.5 +
    temp2.0-temp1.2 + temp3.0-temp1.1
END
```

Diagram 2 will be used to discuss the input format that is needed in order for the simulator to be operational. Looking at the first twelve lines of the input file the word Species: A 3 is found. The format of this first line is a description of the name of a particular species and its number of bonds. There are different species possible and in order to represent each individual one we will use a different name. The number of binding sites is also different and is represented by the number immediately following the species name. The line Wildcard: * 4 is next, this is our way of depicting that any species is a potential candidate for a reaction to occur if located in a wildcard position. As the name suggests the particular type of species is totally ignored and irrelevant when it is listed in a molecular template. The next line has this form, "MolecularTemp: temp1 species: * 0 3 5 species: A 1 2 species: K 4". The key word MolecularTemp is used to signify that a molecular template is about to be read in. Molecular template is the way we describe a rule. In this example, in order for a K Reaction to take place, this template must be loaded into the simulator. If it is not loaded, then this reaction can never occur because the simulator will not know how to complete the reaction. The word temp1 follows the key word MolecularTemp, and is also the name of the template being described. Each template must have a name associated with it so we can specify which template we are using. Then the words and numbers "species: * 0 3 5" are seen. The word "species:" is another key word used during the parsing algorithm to designate that what follows is the name of a species and its positions. The star "*" means that the K Reaction template is looking for wildcards at the 0, 3, and 5 positions, which can be any type of species. Then the key word "species: A 1 2" is seen and this means that 2 species of type A are located in positions 1 and 2. Lastly, "species: K 4" means that a K species is located in position 4.

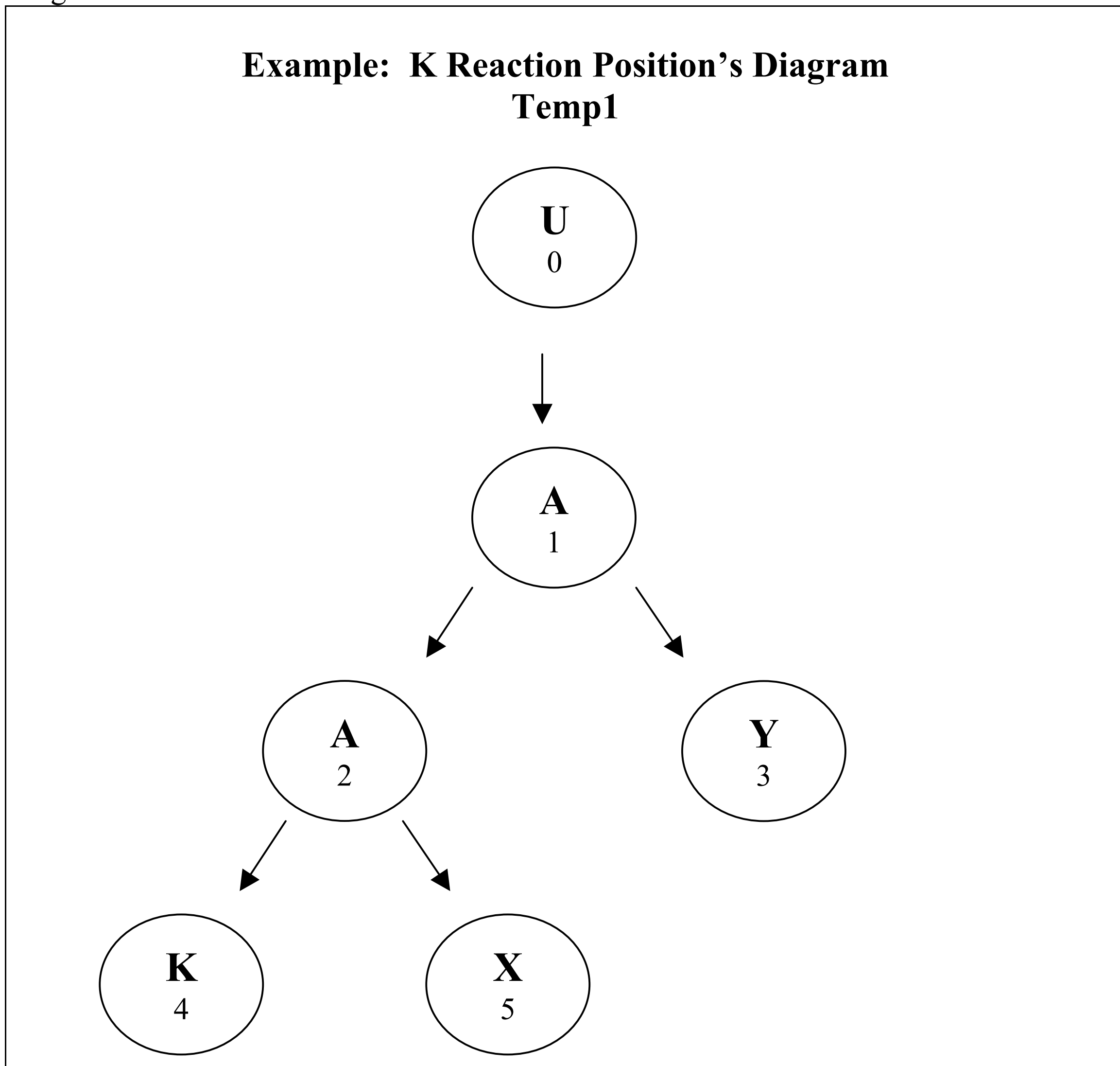
Next are three lines of the form:

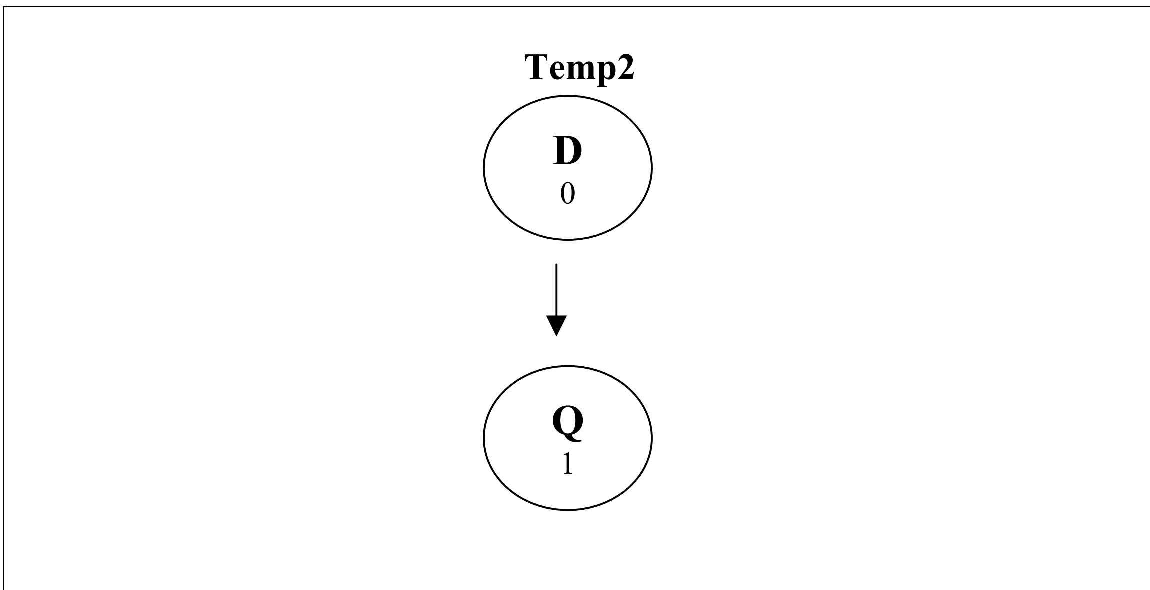
```
0 -> 1

1 -> 2 -> 3

2 -> 4 -> 5
```

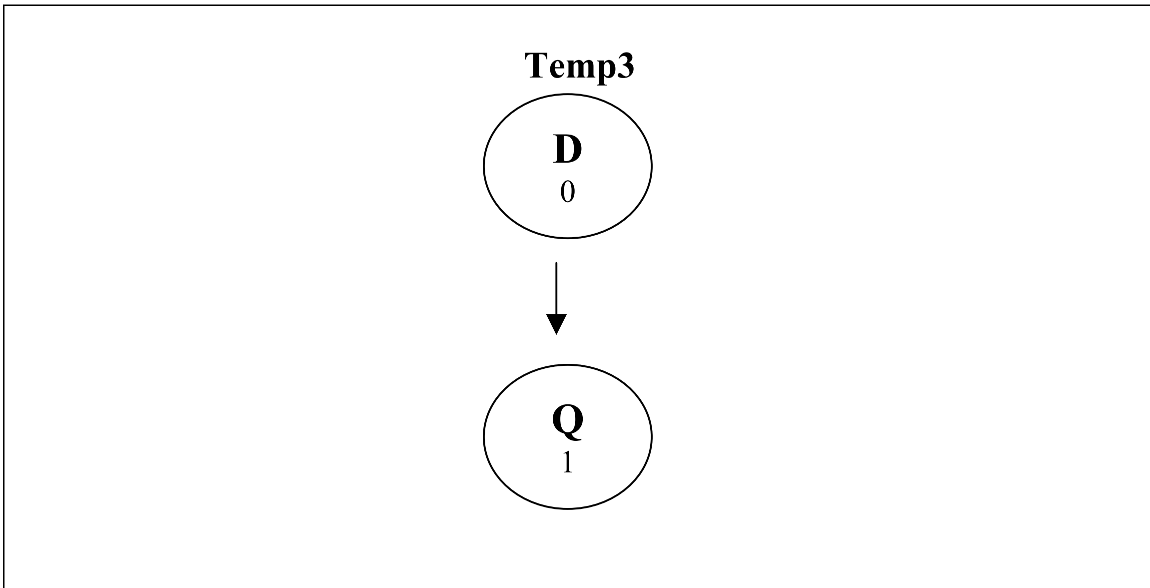
These are needed to show the actual connection of bonds between species. So "0 -> 1" describes that the species in the zero position is connected to the species in the one position. The "1 -> 2 -> 3" describes the connection between the species in the one

position and the two species below, in the two and three positions. The next line is describing a similar situation between the species at position two and the species at positions four and five. Then the key word “END” is seen, which tells the simulator that temp1 is now finished and the next template can be read in. The next six lines, which describe temp2 and temp3, respectively can be described similarly. Below, Diagram 3 is an example of the K Reaction’s positions diagram. There are three boxes depicting templates 1, 2, and 3, along with their species and positions.

Diagram 3:







The next line contains the key word “Reaction:” the name of the reaction. It is immediately followed by the key word “Probability:” and the probability of the reaction occurring. This number must be in decimal format or it will cause an error. Next is the line containing the key word “Requires:” and the template names. They are temp1, temp2, and temp3; this is so the simulator can keep track of what templates are needed in

order for the K Reaction to occur. Then the key word "Description:" is found, which describes the reaction. The description is contained in the lines of the form:

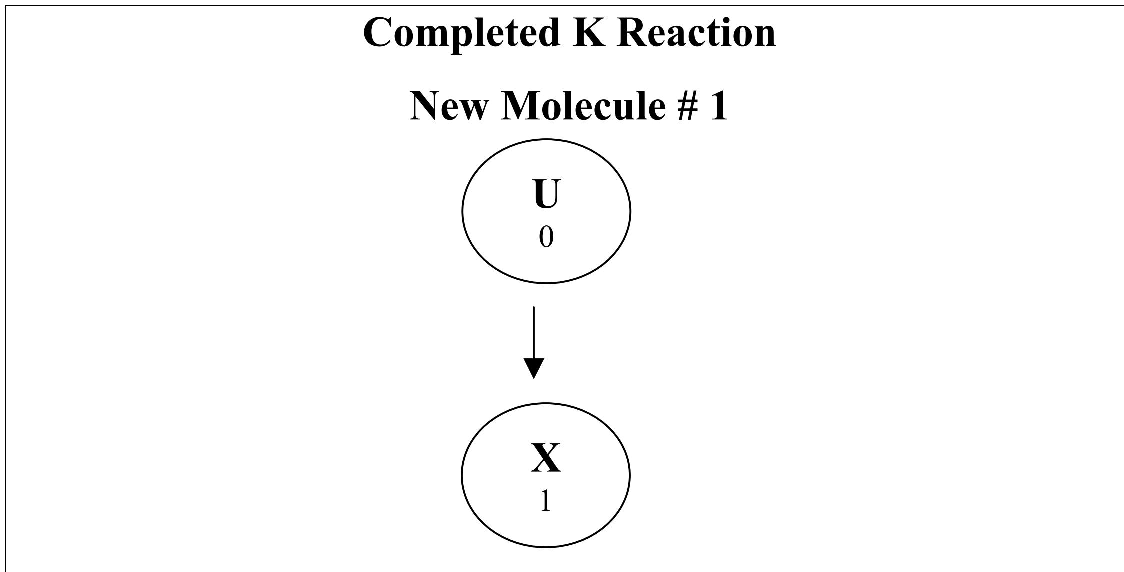
> temp1.0-temp1.1 + temp1.1-temp1.2 + temp1.2-temp1.5 + temp2.0-temp2.1 + temp3.0-temp3.1 => temp1.0-temp1.5 + temp2.0-temp1.2 + temp1.2-temp2.1 + temp3.0-temp1.1 + temp1.1-temp3.1

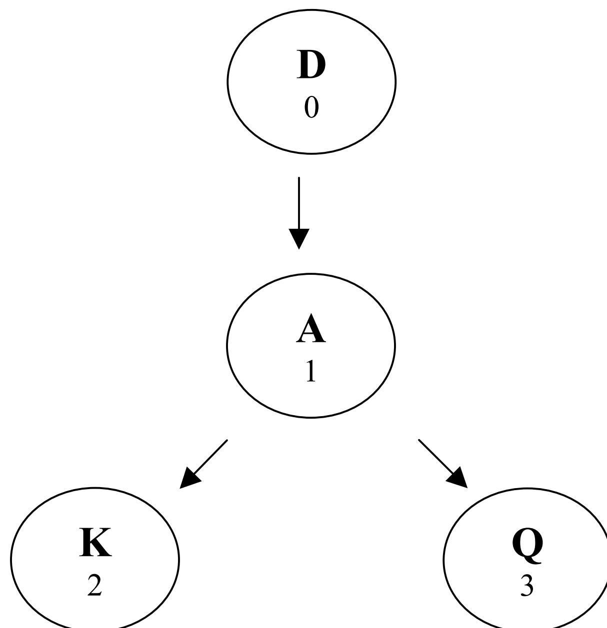
This tells the simulator that when a K Reaction has been picked, these are the bonds that need to be broken and then reattached to create a new molecule or molecules. Everything to the left of the "=>" symbol are the bonds that need to be broken or freed. Everything to the right of the "=>" symbol are the bonds that need to be reattached. Everything in between the "+" symbols are two species and where they are connected. Remember that species can have more than one bond, so we need to be able to identify which bond or two species we need. Since we have the correct two species, we then have to further determine where they are located. This is accomplished by the "-" symbol, which separates the positions we are looking for. Then everything to the left of the decimal point is the actual template name and everything to the right of it is the position of the species. So, "temp1.0-temp1.1", means that in "temp1", position "0", and "temp1", position "1", we have a "bond" which needs to broken because it is left of the "=>" symbol. Then you see a "+" which means that there is another bond which needs to be broken. As long as you keep coming across plus symbols you will continue to break bonds until you have reached the "=>" symbol, from then on we must reconnect the bonds. So, "temp1.0-temp1.5", means that in temp1 the species in position zero will need to be connected with temp1 the species in position five.

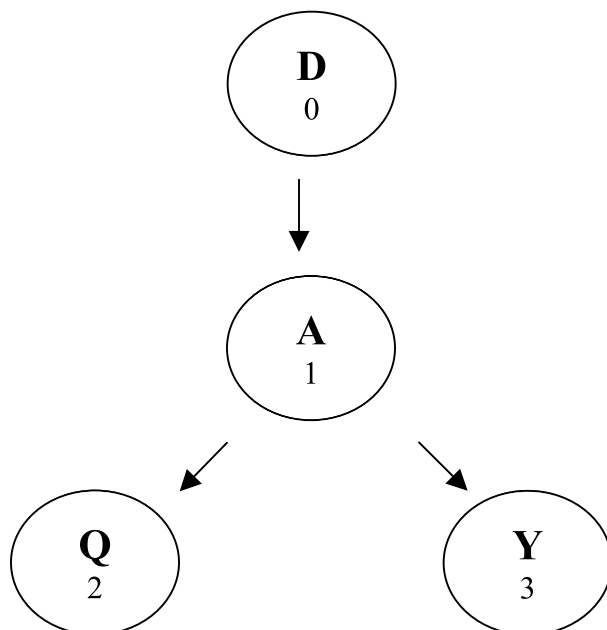
Now refer back to Diagram 2, which was an example of the K Reaction input file. Also, refer back to Diagram 3, which was an example of the K Reaction position's diagram. Starting with Diagram 2, look at the MolecularTemp: line and notice that temp1 has six species that make up one molecule. They are Species U, A, Y, K, and X, and their position are 0, 1 and 2, 3, 4, and 5. Notice also that in Diagram 2 there are two more molecular template lines and these refer to two additional molecules. Which are composed of two species each with just one bond connecting the two. They are in fact two copies of the same molecule. The reason they have two different template names is because when the reaction occurs, their bonds need to be broken or reattached. How will

it be possible for the simulator to know which instance of the molecule is being referenced at a particular time? There will be no way to know for certain if temp2 or temp3 is being changed. Therefore, we have adopted the convention that when using two copies of the same molecule they will be named differently. Once again the line "END" is reached and this means the end of this reaction description has been reached. Looking at Diagram 4, below you can see what the actual completed reaction will look like.

Diagram 4:




New Molecule # 2
D
0
A
1
K
2
Q
3


New Molecule # 3
D
0
A
1
Q
2
Y
3

The fifth requirement is the ability to set the initial contents of the simulator. We commonly refer to this file as the soup file because it is the initial complexes with their corresponding concentrations. Referring to Diagram 5 below is an example of a K Reaction soup file.

Diagram 5:

Example: K Reaction Soup File

```
Number: 1
Molecule: species: U 0 species: A 1 2 species: Y 3 species: K 4 species: X 5
        0 -> 1
        1 -> 2 -> 3
        2 -> 4 -> 5
END

Number: 2
Molecule: species: D 0 species: Q 1
        0 -> 1
END
```

The first line "Number: 1" means that one copy of a molecule is needed. The next line is identified by the key word "Molecule:" which means that all the species named after this word will be contained in this molecule. Which followed by "Species U 0" means that a U Species will be located at the zero position and "species A 1 2" means this molecule will contain two A Species in positions one and two. Again like previously discussed in the K Reaction input file the bond connections are depicted by the "0 -> 1" lines and the key word "END" means the molecule has no more information to gather. Then comes the line "Number: 2", which will give us two copies of this molecule. Just as the requirement demanded, the control of the quantity and description of complexes is at our disposal.

An additional requirement wanted the option of allowing a “tag” or “address” to be associated with some of the program complexes to simulate complexes attached to a fixed substrate. The simulator does not currently handle this option. This was not implemented because it was a feature that was not seen as critically vital to the completion of the prototype. The implementation of this feature is left for the future.

These previous paragraphs mark the end of the input and output requirements described for the simulator. Now we will discuss the actual work simulated and the initial requirements for them. First the ability to perform substitutions based on provided rewrite-rules the rewrite-rules are an analogous way of describing the reaction description or the breaking and reattaching of bonds. In the previous paragraphs where we explained the contents of the input file in great detail, we also described the rewrite rules. The only way for a reaction to make substitutions between molecules is when there is a template for each molecule and the templates have already been loaded into the simulator via the input file. If the reaction and its required templates are not loaded then this reaction will never happen.

The simulator was designed to be random in the way that it chooses reactions and the way that it chooses molecules to perform those reactions. This was also a required feature and has been completed, the details of which will be discussed later.

The probability of a particular reaction occurring is based on the concentration level of the resources. If the simulator is filled up with many copies of molecules that can perform a K Reaction then the probability of the K Reaction occurring is highly likely. On the other hand, if the concentration of these molecules is low then the probability is also very low.

Two other initial requirements deal with reactant depletion and replenishment. The general idea here is that molecules will be naturally depleted after a number of reactions have occurred. One analogy to depletion is firewood used by a fireplace, as the wood is burned or consumed by the fire there will naturally be less wood in the fireplace. Reactant replenishment is an option given to the user by the list of choices on the menu. The list of choices is given to the user after every previous command has completed. One option is to add additional soup files, or to add additional resources. So if the user would like to add one, five, or one hundred new copies of a particular molecule or

resource this can be accomplished. Adding additional resources or a new soup file are the same choice because the simulator does not distinguish between the two. From the simulator point of view they are two things that need to be added to its internal list of molecules.

Two additional requirements for the simulator were spatially fixed networks (embedded in gels) and free-floating (in fluids). These two were not implemented like other previous requirements because they were not of vital importance. The goal of completing a working prototype of the simulator took precedence.

The sixth requirement was the ability of the simulator to run for a specified number of cycles or under interactive control. This requirement was fulfilled by the creation of the execution method. This method is used to continually prompt the user for further instructions. The instructions are listed in numerical order and by typing into the keyboard the number representing the option requested. The simulator will complete the assigned task by calling all necessary classes and methods until the action requested is completed. One option is the request for additional cycles.

In one cycle of execution the simulator will randomly choose an integer by seeding a random number generator. Once the integer is chosen it will be used to randomly pick molecules contained within its list of possible molecules based on the number of molecules within its list. If the number of molecules is three then the simulator will randomly pick a number equal to or less than the number of molecules contained in its list. In this example, the number of molecules is three so the generator will randomly pick any number between zero and three. There is no reason to pick a number greater than three because we do not want to pick more than three molecules if there are no more than three molecules inside the simulator. However, if the number of molecules were larger than three say one hundred, the random generator would set the high bound to be one hundred.

After an integer is picked the simulator will randomly find that number of molecules from its list of molecules. Then from this list of choices the simulator will randomly pick templates to see if they match any of the molecules. If a match is found they are then placed on a new list and based on the probabilities of the individual reactions themselves a reaction is randomly picked. Then the reaction chosen will grab

the molecules that matched the templates from its templates list. Then use these molecules to complete the reaction and create the new resulting molecules. After the new molecules are created they will be returned back to the contents list and are available for any future reactions. Additionally, since the integers are chosen randomly it is possible that the simulator will go through a cycle or several cycles and not complete any reactions. It is also possible that the simulator can go through many cycles and pick a new reaction each time.

For the K Reaction to occur as described previously through Diagrams 3 and 4. The simulator must find three molecules that match the three molecular templates described in the K Reaction rules file. Once the simulator has found three molecules from within is contents list that match these templates a K Reaction will occur.

The seventh requirement was that at any stage of simulation, the simulator can write the internal contents to a file. This requirement was met and through the options menu, a user can at any time dump the contents of the simulator to a file. This file can then be used to continue the simulation or for analysis. When this option is chosen, the state of the simulation will be dumped to a file and this new file will have the same format as the soup or initial contents file. Every molecule within the simulator will be displayed in the exact format shown in Diagram 5.

Another initial requirement was to allow reactant concentrations to be changed under interactive control or according to a pre-specified schedule. This requirement was not completed or even attempted since it was not necessary to the completion of a working prototype.

An additional requirement was that at the end of a simulation stage or specified number of cycles, it should be possible to “flush out” all complexes meeting or not meeting specified templates. The flushing might also take place continuously while the simulator was executing. This additional feature was also not necessary because it was not of vital importance for the initial prototype.

One more initial requirement that was not implemented was the option of allowing instances of specified complexes to be introduced into the simulation that may be directed to a specific “address”. This newly introduced complex would be able to only

react with the complex at the specified address. This was also considered a feature that could be implemented in the future after a working prototype was in place.

The eighth requirement of keeping a running record of the number of times a reaction has occurred, and the ability to display them was incorporated into the simulator. This is another option that is available through the list of menu options. This option will display a graph that will show the number of cycles that are run by the simulator, and the different possible reactions that have occurred with their frequencies. This is another way the user can easily tell if the reactions being tested are the ones that are actually occurring.

Another requirement was the ability to keep track of the total change of free energy. This requirement was also purposely left out for future versions of the simulator. Since the reactions are all completed with the breaking and creating of bonds, the ability of keeping track of free energy, and the number of bonds that can possibly be broken in the future is quite trivial.

That is all of the initial requirements and explanations describing why they were or were not implemented. To reiterate the main reason to include a requirement was to actually design a working prototype simulator. The amount of time required to complete this prototype was limited and thus some tough decisions had to be made to hasten the process. This basic first attempt simulator was needed in order to determine if future exploratory research in the field of universally programmable matter was indeed worthwhile. Therefore, the requirements met were the most basic ones needed in order to reach this goal. The requirements that were left for future more robust versions were purposely left out since they were not deemed essential, although they are important features and will be incorporated into future versions of the simulator.

Next is a discussion of the actual Java classes and their methods along with the data structures used to implement the simulator. This discussion will be brief and in list form to make it easier to spot the individual topics of interest.

Design Implementation:

Simulator Classes Used and their Methods

I. *class Bond*

A. The class Bond is used to handle the placement of bonds. Bonds are used between two species, so they can be connected to each other. They are used within the templates and also within molecules.

B. The methods contained within class Bond are:

1. Bond( ), this is the default constructor.
2. addSpecies(Species sp1, Species sp2), this method adds a bond in between two species.
3. getNext(Species sp), returns the next species.
4. find(Species sp1, Species2), looks for a bond between the two species if found, returns it.
5. createBond(Species sp1, Species sp2), creates a new bond between two species.
6. breakBond(), nulls or breaks the bond between two species.

C. There are no data structures used in class Bond.

II. *class ChemicalSimulator*

A. The class ChemicalSimulator is where the main class method is contained and therefore is the heart of the Simulator. This class controls the Simulator options menu, which then calls all the other classes and methods contained within them to handle their specific tasks. The ChemicalSimulator class handles the reading in of the rules input file, and the initial specification soup files.

B. The methods contained within class ChemicalSimulator are:

   1. main( ), this method handles the command line error checking.
   2. ChemicalSimulator(String inputfile), this methods handles the parsing of the input file.
   3. addSoup(Soup sp), takes a soup file and makes it the current one in use.
   4. updateSoup(String file), takes a new soup file, and makes it the current one.
   5. execute( ), controls the execution of the Simulator.
   6. userControl(int numberOfCycles), controls the options menu of the simulator.
   7. endOfLife( ), removes any newly created reaction files, at the end of the simulation.

C. The data structures used and why:

   1. Hashtable, is used to keep track of species, templates, and reactions. The hashtables were used instead of other data structures because of their speed and efficiency.
   2. TreeMap, to keep a listing of positions. This was used for efficiency as well.
   3. Array, to print out the entire contents of the hashtables. This is the best way to enumerate the elements of the hashtables.

III. *class Molecule*

A. The Molecule class is used to read in the formation of the molecule. This gives the template the structure it needs to have for a reaction to take place. This class handles putting the species in their correct positions.

B. The methods contained in the class Molecule:

1. Molecule( ), default constructor.
2. Molecule(Object[] species), another constructor, takes an object array as input.
3. Molecule(Object[] position, Hashtable speciestable, BufferedReader in), another constructor method, takes as input the position array, the speciestable, and input file line being read in.
4. cloneMolecule( ), this method creates a duplicate of an existing molecule.

C. The data structure used and why:

1. Object array, to keep track of species and their positions. Which is very useful, when trying to make clones of molecules.

IV. *class ReactionTemplate*

A. The class ReactionTemplate is used to handle parsing of the input file, starting from the key word reaction, to dynamically create files based on reaction names, and to store their probabilities, and the templates required for a reaction to occur.

B. The methods contained in the class ReactionTemplate:

1. ReactionTemplate( ), default constructor method.
2. defineReaction(String rname, BufferedReader input, ChemicalSimulator ChemSim), handles the parsing of the input file, and the storing of the reaction information.
3. chooseReaction(Object[] availablereactions), seeds a random number generator, and then, based on the probabilities of the available reactions, randomly chooses a reaction. Then it does the reaction and updates the simulator contents.

C. The data structures used and why:

1. Hashtable, efficiency and speed of use.
2. ArrayList, speed of use, and storage of reaction information.
3. TreeMap, to keep track of probabilities, speed.
4. Object array, storage of new molecules, after reaction occurs.

V. *class Soup*

A. The class Soup is used to read in the soup file, which establishes the initial contents of the simulator, before reactions can take place.

B. The methods contained in the class Soup:

1. Soup(ChemicalSimulator cs), establishes the current soup file.
2. update(String filename), pulls in the soup file and reads its contents, then sets up and stores the number of molecules needed.
3. getSampling( ), used to randomly pick a molecule, so it can be used if needed.
4. dumpToFile(String filename), dumps the contents of the simulator to a newly created file.

C. The data structures used and why:

1. Hashtable, keeps track of simulator contents, species and bonds, used because of efficiency, speed, and storage.
2. TreeMap, keep a listing of valid positions, fast
3. Object array, to enumerate the elements contained in the hashtables.
4. PrintWriter, to write to newly created file.
5. StringBuffer, to hold information, before writing to file with printwriter, efficient and fast, also less time consuming than having multiple I/O calls.

VI. *class Species*

A. The class Species is used to read in the name and number of bonds allowed by a particular species.

B. The methods contained in the class Species:

1. cloneSpecies( ), takes a clone copy of a Species, and adds it to the soup file.
2. Species(String name, int bond), sets the name and number of bonds of a species, also handles some initializations.
3. addBondBit(Bond bit), places a bond between a species and its bit.
4. addBond(Bond b), adds a Bond to species bondlist.
5. numberOfBonds( ), counts the number of bonds in a species bondlist.
6. reset( ), resets a species visited field to be false, to initialize the newly created bonds for use.

C. The data structures used and why:

1. ArrayList, to keep track of the number of bonds, and whom they connect.

VII. *class Statistics*

A. The class Statistics is used to keep track of all simulator statistics when executing, the number of total cycles run, and the number of reactions occurring and which ones they are. It also produces a graph of reaction distributions.

B. The methods contained in the class Statistics:

1. Statistics(Object[] reactions), sets up the necessary hashtables that will be responsible for tracking reaction frequency and distribution.
2. update(String reactionName), keeps a running total of reaction occurrences.
3. incrementCycles( ), increments the number of cycles.

4. getCycles( ), returns the amount of cycles run.
5. printStatistics(String filename), would print out a histogram of reaction frequency, with *'s (not currently being used, was replaced with printJgraph( )).
6. printJgraph( ), this method creates a new file called newin.jgr, this file is used to put the simulator statistics into jgraph form. It is just a black and white bar graph, to view the number of cycles the simulator completed and its number of completed reactions. Note: for more complete information on jgraph please visit Dr. Plank's webpage.
7. convertToEps( ), creates a new file called out.eps, which is where newin.jgr is redirected through.

C. The data structures used and why:

1. Hashtable, keeps track of frequency and distribution of reactions, fast and efficient.
2. ArrayList, to keep track of occurrences of cycles.
3. PrintWriter, to write to newly created files.
4. StringBuffer, to write to before writing to PrintWriter, saves time, makes execution of the program faster.

VIII. *class Template*

A. The class Template is used to create the necessary template classes. To make it possible to pattern match between what's in the soup and what template is needed for a particular reaction to occur. It also extends the Molecule class, this allows for code reuse.

B. The methods contained in the class Template:

1. Template(Object[] position, Hashtable speciestable, String name, BufferedReader in, ChemicalSimulator ChemSim), used to create the necessary templates.
2. MaptoMolecule(Molecule M), goes through the soup contents and input file templates, trying to find matching patterns between the two. If a template is found that matches then it is returned, or else it keeps looking.
3. mapRecursive(TemplateMapping TM, Molecule M, int Mindex, int Tindex), to recursively search the current species, and the current template against each other and to see if their bonds, and positions match.
4. mapSampling(Molecule[] sampling), goes through the random sampling and maps templates to the samples found. Also makes a list of possible reactions that can happen and returns them.
5. class TemplateMapping, to run through the available templates from the input file, and the simulator contents trying to find possible matches.

C. The data structures used:

1. ArrayList, for mapping list, good for enumeration.
2. Object arrays, for enumeration of hashtables that contain template and reaction information.

IX. *class ConvertingLisp*

A. The class ConvertingLisp is a method that needs to be used in order to convert Lisp program output which, must be in the BNC format that describes a tree.

1. The class ConvertingLisp is not inside the Simulator, it is a stand-alone class, with its own main method declaration, and is self-contained. It was created to convert the BNC file (which is created by the Lisp programming language) into a format that the Simulator can read in and understand.

B. The methods contained within class ConvertingLisp are:

1. Node( ), to describe the position of the node within the tree.
2. setPosition(int p), takes an integer value as input and uses this value to set the position of the node within the tree.
3. addBond(Node n), takes a Node as input and adds it to the current nodes arraylist.
4. setName(String t), takes the species name as input and sets it to be the current species.
5. numberOfBonds( ), returns the number of bonds for the current species.
6. getPosition( ), returns the position of the current species in the tree.
7. getName( ), returns the name of the current species.
8. getBonds( ), makes an Object array out of all the bond list arrays in the tree.
9. main( ), checks the command line arguments for errors, and reads in the input file for parsing.
10. ConvertingLisp(String infile), is responsible for parsing the LISP file, and then storing its information into the necessary data structures.
11. print( ), is used to print out the new file newsoup.txt, which is in the correct soup format.

C. The data structures used and why:

1. Hashtable, to keep track of the species needed, and bond connections, fast and efficient.
2. ArrayList, to keep track of bonds, good for enumeration of lists.
3. PrintWriter, to print to newly created file, fast and reliable.
4. StringBuffer, to write information to the PrintWriter, fast saves from doing multiple I/O calls.

This is the end of the list of all of the classes and methods created and used by the simulator. We have tried to describe their purpose and the data structures used to implement them. Java is a robust and plentiful programming language that has many methods built in for easy of use. We have tried to incorporate as many of these features as possible. Also, in the future, if necessary to revise a class and its methods, it should be fairly easy to implement these changes without affecting the entire simulation process.

Clearly, one can see that we have just barely begun to delve into the many possibilities that exist in the future for universally programmable intelligent matter. We will now only briefly sum up the areas in which the simulator falls short. These areas are the previously listed requirements that we have not implemented. The main reason we have neglected to address some of the many initial objectives is that we came to the conclusion early on in this research, that if a specific objective was not vital to the creation of the initial prototype simulator, then we would hold off on the design an implementation of this objective. One example of an uncompleted requirement is the requirement for the development of a method for sensing conditions and causing effects in the external environment. This requirement, like many other requirements, where reasoned to be logical additional enhancements for the base model simulator. These goals and requirements are certainly attainable and will be reached in the future. They certainly will enhance the overall power of the simulation and give researchers more valuable insights.

APPENDIX:

A. Quick Java User's Guide

I. Compile all the Java files

A. This can be accomplished by first making sure that all of your ".java" files are located in the same directory. One side note, the reason I'm calling the files .java is because in the Java programming language once you create a file that contains your source code, you should end it with the file extension .java. This is how you can tell your source code from your compiled code.

B. Then, from the same window or a different window, try to compile all of your .java files. Use the following command line arguments (javac *java), and this will compile all the files simultaneously.

C. After you execute this command all the files in your directory will be converted into .class files. This is Java's way of telling you that your source code has just been compiled and is now in the correct format. The program is now ready to be executed from the command line.

D. Once you have successfully compiled all of your .java files you can then look inside your directory and all the .java files will be converted into .class files. This is Java's way of telling you that your source code file has just been compiled into byte code. Now the Java Virtual Machine (JVM), which is an interpreter, will be able to translate and run the byte code instructions whenever your program needs it. If this seems unclear to you, please visit www.java.sun.com. The java.sun website has a lot of excellent tutorials and explanations available to you. Of course you can also purchase a Java book,

which will also explain more of this to you.

E. If you only want to compile one source file instead of all of them you will have to type this command. (javac filename.java).

F. Remember that Java is case sensitive, so if your filename begins with a capitalized letter, you will have to name the file you are trying to compile as such or it will not work.

G. In the simulator if you try to compile a file separately it will not work because all the class files are accessed through one main file. This is the file that contains the main class within it. The main class for the simulator is contained in the file ChemicalSimulator.java.

H. If you are going to compile the .java files from a different directory then the one they are all currently held in type the following command line arguments. (javac directory name/*java).

I. This will work if the directory is below the one you are currently using, but if it is contained in the directory above the directory you are using, then you will have to type this command. (javac ../*java).

II. Some reasons why your java files will not compile successfully

A. The most common reason for your code not to compile is errors. Unfortunately, these will have to be corrected in order to progress with the rest of your coding.

B. The best way to fix these types of errors if you are not very familiar with Java, is to develop your source code files within a solution's package. A solution package is also commonly referred to as an Integrated Development

Environment (IDE), which is a collection of integrated programs that facilitates software development. Some common IDE examples are Metrowerks CodeWarrior, Enprise's JBuilder, Microsoft's Visual J++, Symantec's Visual Cafe, or IBM's VisualAge, just to name a few. I used Jbuilder, which is available for, free and can be downloaded from www.borland.com.

C. Another possible cause for error is that the version of Java you are trying to use is not installed on your system correctly. The version I used is Java 1.4 Standard Edition. Any version of Java starting from Java 1.4 and newer should work correctly.

D. Another possibility is that the version of Java you are using is installed correctly, but your system does not know where it is installed, then the system could be looking down the wrong class path, and is not able to determine the correct path. Then you will have to change the system PATH variable. If this is the case and you know how to change the PATH variable go ahead. However, if you are unsure how to change the PATH variable then you should try to get some help, because it could be more complicated then you think. Unfortunately, it is also beyond the scope and detail of this help section

B. Simulator User's Guide

I. List of items needed to run the simulator

A. First you will need all of the .java source files.

B. Second you will need the source files compiled into .class files.

C. Third you will need the initial specification files (input files). I have named them all ReactionName.txt files. The way to determine which type of reaction

you will be testing for is by the beginning part of the .txt file.

D. Fourth you will need the corresponding reaction specification files (soup files) for the reaction you are testing. I have named them all ReactionName_soup.txt. The way to determine which type of soup file you will be using is by the beginning part of the _soup.txt.

Some Input/Soup file Examples:

1. kr.txt stands for K_Reaction (input file), refer to Diagram 2.
2. kr_soup.txt stands for K_Reaction (soup file), refer to Diagram 5.
3. sr.txt stands for S_Reaction (input file), refer to Diagram 6 below.

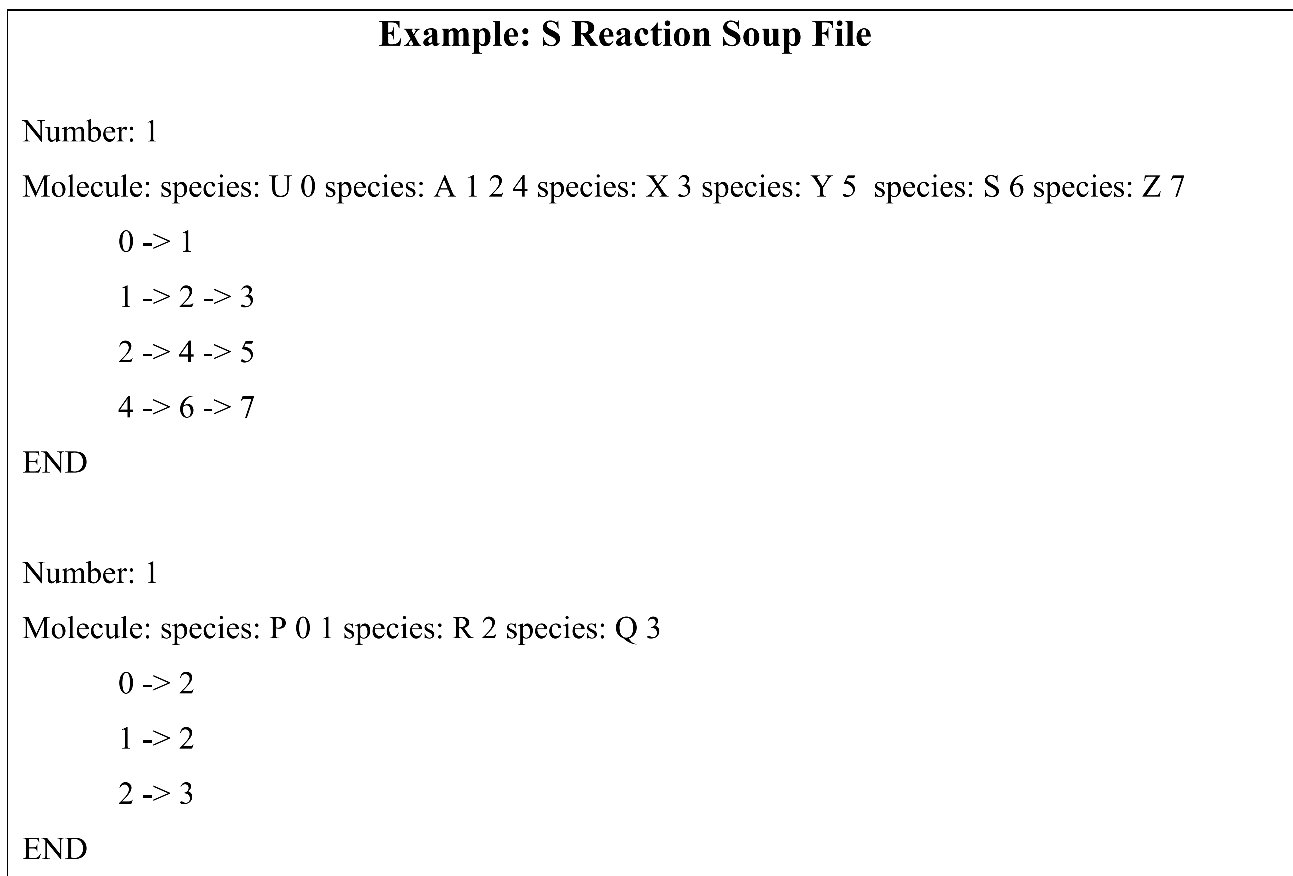
Diagram 6:

Example: S Reaction Input File

```
Species: A 3
Species: R 3
Species: Q 1
Species: P 1
Species: Y 1
Species: X 1
Species: Z 1
Species: U 1
Species: S 1
Wildcard: * 4
MolecularTemp: temp80 species: * 0 3 5 7 species: A 1 2 4 species: S 6
        0 -> 1
        1 -> 2 -> 3
        2 -> 4 -> 5
        4 -> 6 -> 7
END
MolecularTemp: temp45 species: P 0 1 species: R 2 species: Q 3
        0 -> 2
        1 -> 2
        2 -> 3
END
Reaction: S_Reaction
        Probability: 0.50
        Requires: temp45 temp80
        Description:
        temp45.1-temp45.2 + temp80.1-temp80.3 + temp80.2-temp80.5 + temp80.2-temp80.4 +
        temp80.4-temp80.6 + temp45.0-temp45.2 + temp45.2-temp45.3 + temp80.4-temp80.7
        temp45.2-temp80.3 + temp45.0-temp80.6 + temp45.1-temp45.3 + temp80.1-temp80.4 +
        temp80.4-temp80.5 + temp80.4-temp45.2 + temp80.2-temp80.7 + temp80.2-temp45.2
END
```

4. sr_soup.txt stands for S_Reaction (soup file), refer to Diagram 7 below.

Diagram 7:



Example: S Reaction Soup File

```
Number: 1
Molecule: species: U 0 species: A 1 2 4 species: X 3 species: Y 5  species: S 6 species: Z 7
        0 -> 1
        1 -> 2 -> 3
        2 -> 4 -> 5
        4 -> 6 -> 7
END

Number: 1
Molecule: species: P 0 1 species: R 2 species: Q 3
        0 -> 2
        1 -> 2
        2 -> 3
END
```

5. dfs1.txt stands for Delete_Final_Sharing1 (input file), refer to Diagram 8 below.

Diagram 8:

Example: Delete Final Sharing1 Input File

```
Species: D 1
Species: P 1
Species: Q 1
Species: X 1
Species: V 3

Wildcard: * 4

MolecularTemp: temp13 species: P 1 species: V 2 species: D 0 species: * 3
        0 -> 2
        1 -> 2
        2 -> 3
END

MolecularTemp: temp49 species: P 0 species: Q 1
        0 -> 1
END

Reaction: Delete_final_sharing1
        Probability: 0.05
        Requires: temp13 temp49
        Description:
         temp13.0-temp13.2 + temp13.2-temp13.3 + temp49.0-temp49.1 => temp49.0-temp13.2
        + temp49.1-temp13.2 + temp13.0-temp13.3
END
```

6. dfs1_soup.txt stands for Delete_Final_Sharing1 (soup file), refer to Diagram 9 below.

Diagram 9:

Example: Delete Final Sharing1 Soup File

```
Number: 1000
Molecule: species: D 0 species: P 1 species: V 2 species: X 3
        0 -> 2
        1 -> 2
        2 -> 3
END

Number: 1000
Molecule: species: P 0 species: Q 1
         0 -> 1
END
```

a. All the other reactions I have tested also have their own individual (ReactionName.txt) file, and corresponding (ReactionName_soup.txt) file. These can all be tested individually to make sure the Simulator is working correctly. All you would have to do is change the number of molecules you are copying into the simulator. For an example, just look at Diagrams 5 and 9 above.

    i. You will notice that in Diagram 5 there are three molecules, one molecule of six species and then two molecules with just two species apiece. That is all that is needed for one K_Reaction to occur.

ii. Looking at Diagram 9 you will see two thousand molecules total. One thousand molecules containing four species, and one thousand molecules containing just two species. This scenario would fill the simulator with enough molecules to produce one thousand Delete_Final_Sharing1 reactions.

b. If you would like to test some BNC_files (LISP version of Reactions). You will have to convert these into the correct soup format. Which can be accomplished by executing the ConvertLisp program. This program takes in a BNC_file and converts it into a new soup file, which is a form that the simulator can understand. There is an example of a BNC file in Diagram 10 before conversion. There is another example in Diagram 11 of the same file after it has been converted into the correct soup file format.

Diagram 10:

Example BNC File

```
a282
a282: A,
a281: A,
c280: S,
c279: S,
a278: A,
a277: A,
c276: K,
c275: S,
a274: A,
c273: K,
c272: S.
a282_1 a278,
a282_2 a281,
a281_1 c279,
a281_2 c280,
a278_1 a274,
a278_2 a277,
a277_1 c275,
a277_2 c276,
a274_1 c272,
a274_2 c273.
```

Diagram 11:

Newly Converted BNC Soup File

```
Number: 1
Molecule: species: Root 0 species: A 1 2 3 4 5 species: S 6 7 8 10 species: K 9 11
        0 -> 1
        1 -> 2 -> 3
        2 -> 4 -> 5
        3 -> 6 -> 7
        4 -> 8 -> 9
        5 -> 10 -> 11
END
```

E. The previous instructions should be sufficient to get the simulator up and running. Once the simulator is executing there is an options menu that you can follow which will also list other features available to you.

II. How to run the Simulator

A. You will need to have all the necessary files, and they will have to be in the correct format or they will not work.

1. The correct input file format will look like Diagram 2, which is an example K Reaction input file.

2. If you choose to create your own file everything in it will have to be in this format.

    a. The first line of the file will have to hold the key word Species followed by its name and number of Bonds. Once this input file is loaded into the Simulator you will not be able to change it! It is analogous to the laws of nature in that they are constant and are not allowed to change.

        i. This line can be followed by any number of additional Species lines that you plan on testing for a particular reaction scenario. Make sure that each species name is different. There cannot be multiple species with identical names, but having different bond numbers. Here is an example, ("Species: A 3, and Species: A 1", in the same input or soup file) this will not work because

the simulator will throw an exception error.

b. After the Species line or lines you will have the Wildcard line. This is necessary to have if you are doing a reaction, but unsure what type of species will be in that position the * indicates it can be any type of species. You can have one Wildcard line, but anything more will cause an exception error.

c. Next is the MolecularTemp line, which contains a specific name for the molecule you will be looking for. I have named them temp1, temp2, and temp3.

i. After the template name you will see "species: * 0 3 5 species: A 1 2 species K 4". What this means is:

a. species: * is located at positions 0, 3, and 5.
b. species: A is located at positions 1, and 2.
c. species: K is in position 4.

If you where to conceptualize this template using a diagram it would look something like Diagram 3, which was previously described. After the template line you will come across a series of lines that look like this.

0 -> 1
1 -> 2 -> 3
2 -> 4 -> 5

Which tells you is how the molecule is connected by positions. Since a Wildcard species is in position 0 you know that it is bonded to Species

A, which is in the 1 position (0 -> 1). The next line says (1 -> 2 -> 3), which means that Species A in position 1, is bonded to Species A in position 2, and is also bonded to a Wildcard species in position 3. The next line is similar, Species A in position 2, is bonded to Species K in position 4, and is also bonded to a Wildcard Species in position 5 (2 -> 4 -> 5).

e. The line directly after the last position line is the key word END. Which lets you know the Molecule is finished.

f. The next line is another MolecularTemp line, which has no position lines after it. So Species D is the only member of this molecule, and is not connected to any other species.

g. Then there is another MolecularTemp line which is identical to the previous one, except it is named temp3.

    i. This is done because for the K_Reaction to take place there needs to be two Delete molecules present. In order to distinguish which D Species you are referring to it is easier to simply give them different names.

h. The next line is the Reaction line which tells you the name of the reaction. All the previous

lines are just descriptions of what the molecule looks like. For the K_Reaction to take place there needs to be three molecules present within the simulator. The molecules are represented by the MolecularTemp lines and that is why there are three of them.

i. The next line is the Probability line which tells you the probability at which this reaction is expected to occur.

i. The following line is the Requires line which lists the required templates needed for the reaction to occur.

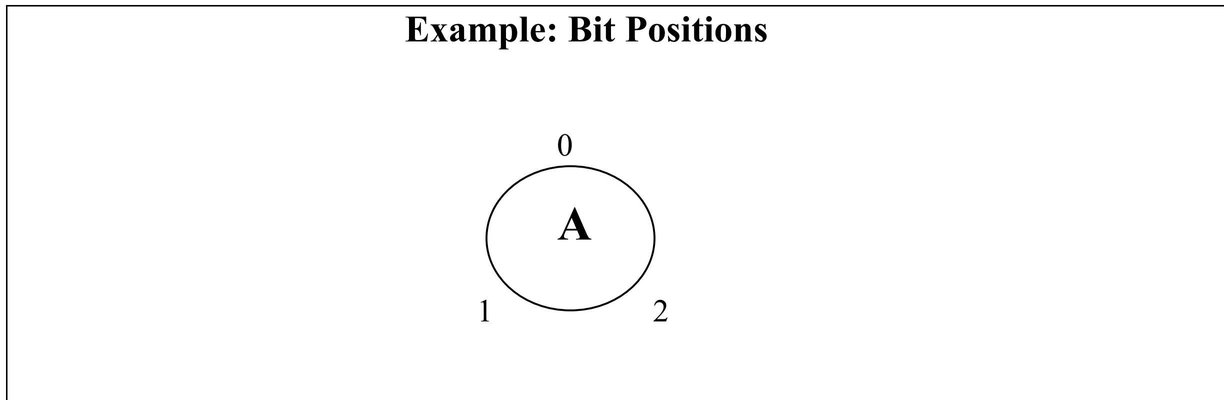
j. The Description line follows which describes the bond changes that are needed, in order for the K_Reaction to occur. First preexisting bonds needs to be broken. Second new bonds will need to be created. The line beginning with temp1.0-temp1.1 and ending with temp3.0-temp1.1 describes these bond details. Everything to the left of the "=>" symbol represents a bond that needs to be broken. All the bonds that need to be broken are separated by "+" symbol's. Everything to the right of the "=>" symbol is a bond which needs to be created. All of the bonds that need to be created are separated by a "+". Please refer to the K_Reaction (input file) as I explain the following.

1. temp1.0-temp1.1 is a bond that needs to be broken. The bond is in temp1, and is located between Species * (position 0) and Species A (position 1). How do I know this? Simple, everything to the left of the period is the template name and everything to the right of the period is the position. The minus sign between the two template names represents that they are bonded together. To reiterate, in temp1 positions 0 and 1, there are two species bonded to each other and this bond needs to be broken.

2. temp1.0-temp1.5 is a bond that needs to be created because it is to the right of the "=>" symbol. In order for this bond to be created you have to know what template or templates you need for this to occur. Then you also need to know the positions you are referring to. So just like the previous explanation, in temp1, positions 0 and 5, there are two species where a new bond needs to be created.

k. We will now discuss how we control the position by which a species gets connected to another species. First, the convention we are using is depicted below in Diagram 12.

Diagram 12:



We have created a method in class Species that is called addBondBit and whenever we create a bond between two species we immediately call this method. As seen in Diagram 12 above, Species A is allowed to have three bonds. We distinguish between the bonds by giving them bit positions 0, 1, and 2. Bit 0 is the species connected above Species A, which can also be called its argument. Bits 1 and 2 are called results, and they are the species that are connected below Species A.

Using this convention we can distinguish between an A species that is connected to a Species A in its bit 0 position, Species K in its bit 1 position, and a Species S in its bit 2 position. From an A species that is connected to a Species A in its bit 0 position, Species S in its bit 1 position, and a Species K in its bit 2 position.

B. Second copy all the of the corresponding soup files you plan to be using into the same directory. This way they can both be in the same area and easier for you to find.

1. The correct soup file format will look like Diagram 5, which was previously described.

2. Everything in this file will have to be in this format for the simulator to work correctly.

    a. The first line will be the Number line which tells you the number of molecules that will be made by the simulator. In this case, the number is 1, but it could just as easily be 100,000. This will initially load the simulator with 1 copy of the molecule.

    b. The second line reads "Molecule: species: U 0 species: A 1 2 species: Y 3 species: K 4 species: X 5". This tells you the shape of the molecule. Species U is in position 0 and Species A is in positions 1 and 2. Species Y, K, and X, are in positions 3, 4, and 5, respectively. Remember that Species U, X, and Y are not actual species they are wildcards.

    c. Then the position lines.

        0 -> 1

        1 -> 2 -> 3

        2 -> 4 -> 5

        This is visualized in the same way as step c above where I referred you to the position's diagram.

    d. Then the key word END signifies the end of the molecule.

e. The second molecule is started by line "Number: 2". This means that two copies of the Delete molecule will be in the simulator.

   i. If you run the Simulator with the K_Reaction (input file) and the K_Reaction (soup file), as I have described, then the simulator should only allow one K_Reaction to occur, as long as it is run for a sufficient number of cycles. Since the K_Reaction needs three molecules present for a reaction to occur, and there are only three copies being made, then it stands to reason that only one K_Reaction is possible.

   ii. If you decide you would like to see more K_Reactions occur then you will need to change the K_Reaction (soup file). The first Number line will need to be increased to ten, and the second Number line will need to be increased to twenty. Then run the simulator for a sufficiently large amount of cycles and you should have ten K_Reactions that occurred, but no more than ten.

      a. I say sufficiently large number of cycles because the simulator is probability based, so there is no guarantee that ten cycles run will be enough to produce ten K_Reactions. It could take twenty, fifty, or even one hundred cycles to get the number of K_Reactions you think you should get.