Java Applet Client-Server Interface for a Complex Suite of Chemistry Applications

Edward S. Blurock
blurock@risc.uni-linz.ac.at
Research Institute for Symbolic Computation
Johannes Kepler University
A-4040 LINZ
Austria

July 14, 1998

Abstract

This paper describes a grapical interface for a complex suite of programs allowing users to access locally and remotely the underlying chemical software systems. The non-triviality of this task is rooted in the heterogeneous software (web-browsers and computers under different operating systems) and language (JAVA, C++ and FORTRAN) environments that have to communicate with each other. The system, however, is modularised in order to shield the user from this complexity and to provide ease of further development. This is accomplished with a set of portable, public domain resources. The system is used to interface a demonstration of the chemical modeling system REACTION.

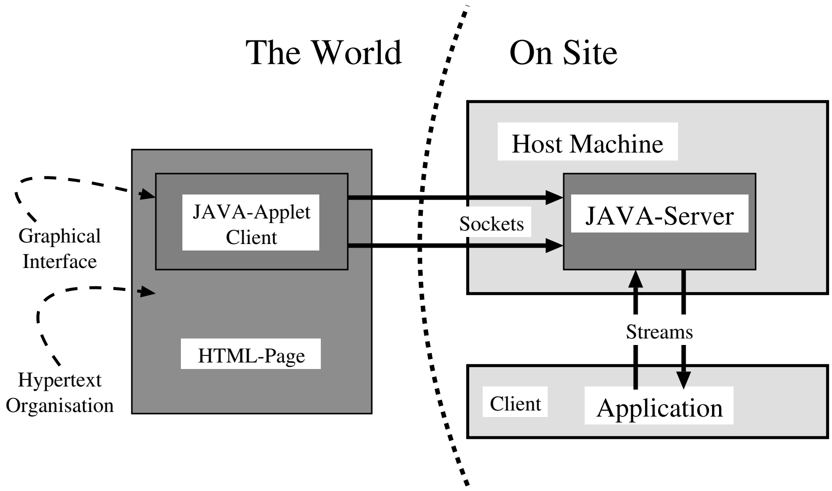

Figure 1: The communication between a graphical interface on the world wide web and an application on a host machine

1 Introduction

It is highly desirable to be able to have a graphical interface to a complex suite of programs that not only provides interaction for the local user of the application, but also can demostrate the application (still running on the local site) on any remote site on the internet. A complication to this task is that the application is usually written in a language chosen for historical or hardware technical reasons and that these diverse language environments have to communicate with each other.

This paper describes a modular approach providing an interface for the chemical modeling system REACTION which can be used by any standard world wide web browser and a viewer in the local environment. A schematic picture is shown in Figure 1. The graphical interface sets up the input for an application. This information is transmitted to a server which passes it on to the application. The output of the application passed back through the same route and is translated to the graphical output of the interface. The main modules of the system consist of a graphical interface (graphical classes written within JAVA applets called by an HTML-page), a program to manipulate and manage chemical information on a symbolic and numeric level (the REACTION program written in C++) and a program to numerically solve the differential equations associated with a complex mechanism (the SENKIN program written in FORTRAN). In addition, several other modules for the communication between the different languages and machines (UNIX shell scripts[8]) are involved.

The necessity for such a complex structure is based on pragmatic historical reasons. The SENKIN program is a numerical FORTRAN application that has been available for over twenty years. The REACTION system[1, 2, 3], representing 'symbolic' manipulation of the reaction data, uses an object oriented language for this purpose, namely C++[9]. With the expansion of the world wide web, the use of HTML pages[4] to disseminate information has become widespread. The relatively recent introduction of a programming language providing graphical interfaces using JAVA on top of the HTML pages opens up the possibility of not only providing static information about an application, but also providing 'on-line' demonstrations. This paper describes a graphical interface which has been built to guide the execution of the REACTION program, including its interaction with the SENKIN[7] program.

Using a JAVA-Applet[5, 6] interface within an HTML-site easily provides the necessary facilities for a 'self-explanatory' demo:

Graphical Input/Output A graphical interface providing menus, buttons, etc. can guide the user, especially one not familiar with the application, in giving the proper input to the application. The more 'graphical' the output is, the easier it is understand the results. These are provided by the JAVA AWT interface classes.

Information Transfer The network communication facilities defined in the JAVA language provide the means to communicate with the application and the browser in a client-server relationship.

Demo Usage Documentation The purpose of a demonstration is, of course, to provide enough interest in the application to warrant further exploration into its workings. This requires that the user must be able to use the demonstration effectively and this, in turn, requires that enough explanation of its use and output is present.

When dealing with such a complex suite of programs, binding them into a single (at least from the user standpoint) application is complicated by two fundamental problems:

Different Programming Languages No one language was able to meet the requirements of this system. Furthermore, as mentioned previously, for historical reasons, some applications are restricted to a specific languages.

Different Machines Specific applications can be bound to specific machines. The reasons can be as banal as the application has only been ported to one machine, or that within the developer's programming environment, certain facilities are restricted to certain machines.

These problems, though not insurmountable, necessitate the use of (now fairly standard) techniques that are common in network or distributed applications:

Data Communication Channels A suitable common communication form between the pairs of languages and the pairs of machines transfering data must be used. The communication between the JAVA clients and server is through sockets, an efficient network communication media between different machines.

Data Translation When transfering data, especially between applications in two different languages, the task normally taken over by the compiler within a single language has to be done by the user. The form of the data transfer is restricted not only by the structures available within the languages, but also the medium used to transfer the data. This has been accomplished in this system through ASCII streams.

Client-Server Relationship The 'server' acts as the control node passing information to and from 'clients' on the same or different machines. The interface client passes user commands to the server which then passes the information on to the actual application. The output of the application follows the same route back to the interface.

Databank of Information For an application with a broad base of fundamental data, it would be too restrictive and inefficient to load and/or transfer this data during each run. For this reason the use of a database, where the information is available in an efficient, organised and retrievable form, is required. The REACTION system has a data retrieval mechanism of all pertinent chemical information.

This complex system consists of a variety of functionalities. The philosophy behind its organisation is to separate them into distinct modules so as to make it easier to make different combinations specific to distinct applications. The modularity also shields the developer from unnecessary details of the other parts of the system. In addition, different users, machines and applications have different requirements and constraints. The system has design elements that provide class objects at a high enough level so that these differences are transparent. To enhance the portability of the system, the languages involved should not be restricted to the machine distributor's version of a language, but relies on languages (and uses within a language) common to a wide class of machines. The use of (stable) public domain software is one way to meet this requirement.

2 System Modules

The entire application, from interface to computation, consists of an interaction of five modules involving six different languages. (see figure 2):

HTML-Client This is the graphical interface called from the remote machine. The HTML page is called from a web-browser and has the text explanation of the demonstration (written in HTML derived from hyperlatex) and the call to JAVA-applet. The applet builds the graphical interface and provides the communication with the rest of the applications

Client-Server Interface This interface, written in JAVA and running on the application's web server (a security requirement of JAVA), communicates the commands built by the client and passes them to a UNIX process where the job is executed.

Shell Script This is the UNIX shell script, running under the application's account, which sets up and translates the command into a REACTION command line. This script also (if necessary) determines on which machine the REACTION application should run. The purpose of this interface is to shield the JAVA interface with changes and idiosyncracies on the UNIX side.

REACTION The REACTION system receives a command line and does the desired computation. The command line consists of keywords and parameters to guide the computation. The detailed chemical information needed is extracted from the REACTION database which is an integral part of the system. If the computation to be done is a SENKIN numerical run, then the REACTION system sets up the input files and, after the SENKIN job is finished, interprets the output files.

SENKIN The SENKIN system is a FORTRAN program which does the numerical computations of a chemical mechanism. The input and output proceeds over files. These files are created on the applications account by the REACTION system.

The differentiation between the modules is due to the different programming languages, machines and applications. The modules represent the stages of information flow and interpretation from the user interface to the chemical application.

3 Graphical Interface

One of the goals of the interface is to provide an easy implementation environment where the developer (writing a new interface) can concentrate on the aspects directly concerned with the application. The client-server communication aspects are kept invisible. The developer need only design the graphical interface with the objects provided by JAVA and to interpret the output from the application run. The system accomplishes this goal by providing the following:

Virtual Interface The developer need only implement a minimal number of JAVA virtual functions to build the interface, interpret the menus, buttons, actions etc. that arise in building the application command and to interpret the application output.

Graphical Classes In addition to the standard graphical interface classes provided by the JAVA interface, a few more classes have been implemented

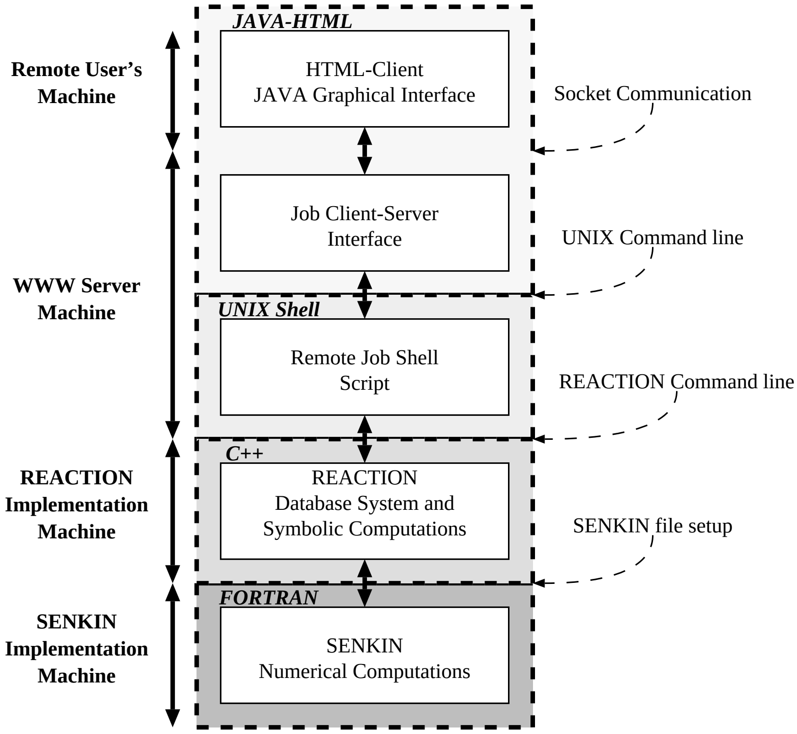

Figure 2: Application Modules

that are particularly useful for the REACTION interface. These include drawing a network of interconnected nodes and plotting X-Y numerical data.

3.1 Virtual Interface

The interface sets up the virtual functions that are needed by the system to build and run the graphical interface. The interface is defined with the following functions in a virtual JAVA interface class:

setUpDisplay This builds the graphical interface using the JAVA Abstract Window Toolkit (`AWT`) and the additional classes provided by this system. The objects used to build the interface are members of the container class.

extractCommand This command reads the current status of the buttons, menus, etc. to build a command line to run the application.

actionReaction This command is called each time an event, like pressing a button, occurs within the interface. These usually involve intermediary commands to setup and/or interpret the application.

displayTextOutput The output of the application is sent back line by line. It is the job of this routine to interpret each line as it is received. The results could be displayed into a window or interpreted into a graphical object.

3.2 Graphical Classes

There are basically two purposes of the graphical interface, to set up the input to an application and to display the output. The setting up of the input, especially when the input is a line command, as here, is the relatively simple task of putting together the status of the buttons, inputs and menus of the interface. The necessary REACTION input is reduced to keywords and parameters extracted from the menus and input fields of the interface. The chemical information contained within the databases in the REACTION system provide the detailed information needed to perform the chemical computation. The output, however can entail more complicated objects such as networks or plots.

One problem is that the communication passes through data structures within several different languages. The interface is written in one language, namely JAVA, and the application is written in another, namely C++ (not to mention the third language of FORTRAN). This means to transfer structures from one language to the other, an common interface, i.e. translation process, is needed.

In this implementation it was chosen to transfer ASCII text, line by line, from the C++ program back to the graphical interface. This means that the structures have to be converted to an ASCII form by the C++ program and then interpreted by the JAVA program. It should be noted that the use of sockets is not restricted to ASCII and a binary byte stream could also have been used. But since ASCII is easier to debug and the amount of information transfer is relatively small, the loss of efficiency is not very high. Thus, transfering ASCII instead of binary does not represent a large downgrade in performance.

Since the output cannot be transfered 'directly' (in the sense of a procedure call within a single language), the programs on both sides must do the translation. For the current system, three types of output display (and hence data transfer) are possible:

Text Output Here, the text output is just, line for line, displayed in a text box. The only manipulation of the output would be to ignore output text until a certain point. In the demonstrations this is used, for example, to display the text output of the SENKIN job.

X-Y Plot Two sets of information are passed, the triplets of first value, last value and increment and then the matrix of values. In the demonstrations this was used to plot the concentration values.

Network First the set of nodes are passed, with the title, type and size of the node and then the connection between the nodes and their type. In the demonstrations, this is used to plot the mechanism as a network.

4 Client-Server Socket Setup

All communication between the client applet and the server is through sockets. In this implementation, two communication sockets are used:

Listen This a socket set up initially by the server to 'listen' for clients. This socket will be used only to acknowledge the client and to send the necessary information to set up the client's own communication socket.

Client Each application client has its own communication socket. It is through this socket that commands and output will be transfered. This socket lasts for the lifetime of the client.

Within the JAVA language, to set up a remote socket connection, the remote machine IP address and the port used on this machine is needed. The determination of the machine address is relatively straightforward because the server is always on the same machine. Although JAVA provides a routine that can determine the machine, `InetAddress.getByName`, it can also be hardcoded at compile time (Within the implementation, a special class is provided which defines all the system dependent parameters). The other information that is needed to set up a socket is the port number. Unlike the machine name, this continually changes and must be determined at runtime.

The server sets up the 'listen' socket. This is done simply, within its constructor, with the routine, `ServerSocket.getLocalPort`. However, for the client to set up a socket connection to the server, this port number must be known. The port of the 'listen' socket is passed as a parameter in starting the JAVA applet in the HTML code:

```
     <applet code=rxn/DrawGraph/DrawGraph.class
             width=800
             height=800
             codebase=http://www.risc.uni-linz.ac.at/java/lib>
            <param name=port value="1331">
     </applet>
}
```

This `port value` is determined as soon as the server starts and must be edited into the HTML page. The port number is different every time the server is started and thus, if the server is restarted, another port has to be edited in. In order to automate the process (especially if the server machine is frequently rebooted), one of the tasks of the server, as it is started up, is to generate these pages with the necessary numbers inserted.

The server also sets up the 'client' socket, where the runtime data is communicated. Once again, the port number must be transfered (the machine name is already known). Once the client is acknowledged by the server, a new (independent) socket is set up for this client. The port number of this client is then sent over the 'listen' socket so the client can set up the other end. This is the only type of communication that passed through the 'listen' socket.

Thus, the server sets up an initial socket communication channel to listen for a client. This is used only for the initial acknowledgement of the client. When a client appears, the server sets up another socket and uses the initial socket to transfer the port number of this new socket. The client then uses this to set up its personal client-server communication socket. All further communications between this client and the server proceeds only through this socket. It should be noted that although the client applet can be run from any remote machine, for security reasons, the server jobs must be running on the same machine as the web-server.

This exchange of program and network information is implemented using three JAVA classes:

JobServer This sets up the server and the initial listening socket.

JobClientApplet This is the applet called by the HTML-page. It controls the communication for the client.

JobServerThread This is a server 'thread' set up for a specific client. It controls the server side of the client-server communication.

The general flow of operation and interaction between these classes is summarised in table 1. The server is started, the 'listen' communication channel is setup and it waits for a client to appear. The client sets up the interface then waits for input which will build up the command to be sent. When the command is to be sent, the 'client' communication channel is set up (if it is the first command sent) and the command is passed to the server which, in turn passes it to a UNIX process (to run the REACTION/SENKIN application). The server collects the output of the process. When the output is all collected, it is sent to the 'client' again which interprets it into the graphical objects of the interface.

5 REACTION

REACTION is a software system providing a multitude of tools and algorithms for the generation, manipulation and analysis of complex reaction mechanisms. As mechanisms become more complex, the ability to treat them manually (even with the support of additional algorithms) becomes increasingly difficult. REACTION is an attempt to generalise and automate existing manual techniques for studying complex reaction mechanisms. The specific goal in mind is to study hydrocarbon combustion[10], but the philosophy of the software system is to keep the algorithms general enough to be able to solve a wide variety of other systems.

One of the uses of REACTION is to characterise hydrocarbon combustion mechanisms. This is accomplished by expanding upon existing 'numerical' techniques of mechanism analysis and introducing comparative 'symbolic' (non-numerical, for example, graphical) techniques. A fundamental part of the 'symbolic' analysis is the use of algorithms dealing with graph theory. The most prominent algorithms are subgraph search, path and cycle recognition and permutation algorithms. Within these analyses, the mechanism is treated as a large graph where the nodes are the molecules and reactions. A reaction node is connected to all its corresponding the reaction and product molecule nodes. The characterisation of this mechanism graph with graph theoretical techniques is one one of the fundamental research areas of REACTION.

Symbolic analysis of large mechanisms can become overly complex, not only in the analyses themselves, but also in the interpretation of the results. Thus, they are used in combination with numeric analyses, used to reduce the analyses to only the 'significant' part. For example, significant reactions could be defined as those with rate constants, as determined by a numeric analysis, above a certain value. The symbolic analysis would then concentrate only on this reduced set.

The main source of the numeric results is the FORTRAN program SENKIN (combined with CHEMKIN). Within this numeric model, each reaction is represented as a differential equation involving changes in concentration and heat evolution (for more complicated models, spatial movement and even turbulence equations are added). The SENKIN package solves this set of equations. A combustion experiment with SENKIN yields the temperature and species concentration profiles versus time. Using these profiles, the REACTION system can limit its analysis to only the 'most significant' reactions or qualify its symbolic results.

The basis of any analysis technique is the availability of information about

JobClientApplet	**JobServer**	**JobServerThread**
	Set Up Listening Port	
	Set up HTML pages (with port number)	
	Listen for Client	
	Wait	
Applet Started by WWW		
Graphical Interface Set up		
Server Host Determined		
Wait For Command		
	Wait	
Extract Listening Port number from HTML page		
Set up Socket to Server		
	Acknowledges Socket	
	Sets up Client Data Port	
	Send Client the port number	
	Set Up ServerThread for Client	
		Wait For Command
Receive Port Number		
Set Up Communications		
Extract Command From Interface		
Send Command To Server		
Wait For Output		
		Receive Command
		Pass To JobServer
		Wait For Command
	Start UNIX process with command	
	Set up IO streams with process	
	Capture Output until job done	
	Wait	
	Send completed output (line by line)	
Transfer lines to Application		
Application interprets line		

Table 1: Flow of Operation

the mechanism and all its parts. This means that before any sort of analysis can take place, a means of data management must be present. This translates into having a convenient database management system in which the information can be efficiently stored and retrieved. The heart of the database system is a hierarchy of C++ classes built upon the gnu gdbm library. Having a software system with a database system behind it simplifies the interface to the REACTION system. The calculation commands need only refer to the names of the chemical objects involved. For example, to create a graph of the significant reactions of a given mechanism at a given reaction time, one needs only to give the mechanism name, the SENKIN experiment name, them reaction time and the 'significance' cutoff for the reaction rates.

The REACTION system basically has three databases, for molecules, reactions and mechanisms, respectively. Every calculation uses the database information as input and the result is stored with the particular object for later use. For example, to do a SENKIN calculation, first the mechanism definition and data is retrieved from the database and then after the calculation, the experimental conditions and the temperature and species concentration profiles are stored.

The REACTION system itself is made up of several modules (each an executable binary) each with specific tasks relating to database management and analysis. Some of the more significant modules are:

Molecule The molecule database manager

Reaction The reaction database manager

Mechanism The mechanism database manager

Senkin The interface of REACTION with the CHEMKIN/SENKIN Fortran programs

Thermo The thermodynamic information database manager

MechGraph Analysis of the mechanism as a network

LineGraph Plotting of molecular properties

The commands and parameters for each module are given on the command line. The JAVA interface is used to set up this command line.

6 Example: A Reaction Network

This is an example where the graphical reaction network representation of a mechanism within REACTION is displayed. How the input information is built up by the interface and passed to the REACTION system and how a complex structure like a graph is communicated back to the graphical interface is illustrated.

As part of the symbolic analysis within REACTION, the mechanism is represented as a directed graph. Each molecule and reaction is a node in the mechanism graph. The reactant molecules of a reaction are connected with a 'mechanism bond' going into the reaction node and the product molecules are connected with a mechanism bond coming out of the reaction. The purpose of this demonstration is the visualisation of this mechanism graph.

The mechanism used in this example models a combustion 'experiment' of a hydrogen and oxygen mixture in the presence of nitrogen and entails 11 molecules and 23 reactions. The numeric calculation done on this mechanism simulates the gas phase combustion process at constant pressure. More accurately, the goal is to calculate the autoignition point given a specific initial temperature. In this example the initial temperature is 950 degrees Kelvin and the pressure is one atmosphere. The SENKIN program computed the evolution of temperature and of the species concentration. This information is used in this example to calculate the reaction rates. However, this data was collected in a separate calculation (another interface) and was transfered back to the REACTION program which stored it with the mechanism in the database.

The network representation of this mechanism has thirty-four nodes (eleven molecule and twenty-three reaction nodes). Each reaction averages around three reactants and products and thus the reaction nodes have an average of three connections. However, each molecule is involved in several reactions, thus the connectivity is much higher for the molecule nodes. For example, of the eleven molecules, seven of them are involved as reactants or products in more than seven reaction, meaning that these molecule nodes have a connectivity of 7 or more. The hydroxyl radical is involved in seventeen reactions. As can be seen, the graphical representation of this relatively small mechanism, when graphed completely, is so interconnected that distinguishing features are hard to identify, let alone to display it.

In order to reduce the number of nodes in, and hence the complexity of, the representation, the symbolic data of the graph is supplemented with the numerical results of an actual combustion 'experiment'. This represents one approach in the relatively new philosophy of reducing the symbolic complexity with numeric results. If only the 'most important' reactions are examined, then the reaction graph is greatly simplified. The concentration data is combined with the reaction data (Arrhenius constants for each reaction) to compute the reaction rate at any given time. The rates of reaction can then be ordered and only those reactions above a given threshold are considered for displaying.

6.1 The Data Interface

The JAVA interface, shown in Figure 4, is used to set the 'reaction time point' at which to examine the reactions and the 'cutoff' (log scale) of the lowest reaction rate to allow (in addition, one can set a limit on the number of reactions shown). Upon pressing the 'Get Data' button, the REACTION program is called with the following system call in UNIX:

```
DrawGraph.run SenkinExample .00020 4.0 8
```

`DrawGraph.run` is a c-shell script which interprets the input parameters and makes call to the REACTION program (The REACTION call is also a single line command, but the script is used as an intermediary in case further processing is necessary on the UNIX level – for example, if a remote call to another machine must be made). The actual REACTION call is with still with keywords:

MechGraph DisplaySenkin SenkinExample Temp=950 .00020 4.0 8

where `DisplaySenkin` is the subcommand within the `MechGraph` module to create the mechanism graph to display, `SenkinExample` is the mechanism information, `Temp=950` is the SENKIN run job information and the three constants are the reaction time point, the cutoff and the maximum number of reactions, respectively. The necessary mechanism information is retrieved from the database. The database has all the molecule, reaction and mechanism data, plus the results of a previous SENKIN run. The corresponding information is retrieved and the necessary computations are done to give the reduced graph information back to the display. The REACTION process prints out (among other things) the graph representation for the JAVA job. The JAVA server captures this information and sends it (through the client socket) to the clients applet, which interprets it. It is interesting to note that the JAVA client (i.e. the interface) runs on the user's site, but REACTION runs on a remote site (with respect to the user).

The graph representation sent to the JAVA job is shown in Figure 3. The description has four parts:

Start Flag The line `----------` is the start flag of the ASCII graph representation. All input before this flag is ignored (or at least not relevant to the graph information).

Counts The number of nodes and connections that follow is given on one line.

Nodes The corresponding number of nodes is sent, one node per line. Basically a node label is given.

Bonds The corresponding number of bonds is sent. A bond is specified by the two node labels which are to be connected.

The node description is given as a set of two number-label pairs:

[100,hydrogen] # [20000,hydrogen]

The first is the internal numeric and label identifiers for the JAVA program. The number contains information about the type of node. For molecules, the type is `100`, meaning the node is a box with the label. For the reactions, the number is `200` in the lower three digits, meaning a round node with no label, and the two upper digits specify the radius of the node. The second number-label pair is the REACTION internal identifier.

After the corresponding number of nodes, the reactions are read in. These are given as a pair of JAVA identifiers (from the node description):

```
[100,peroxyl-radical] # [48200,For:SE-HO2+HO2=H2O2+O2]
```

For the mechanism graph, a molecule node, for example 'peroxyl-radical', is connected to a reaction node, for example, 'For:SE-HO2+HO2=H2O2+O2'.

The information is read into the corresponding JAVA graph class, consisting of a set of nodes and a set of bonds. In the implementation, this graph class is independent of the chemical meaning. As the nodes and bonds are put into the graph class, they are graphed on the canvas of the JAVA applet.

Note that no coordinate information about the graph display is transfered. The graph display tries to optimise the positioning of the nodes and bonds dynamically. This is done through a simple algorithm in which, for every cycle (about a tenth of a second), the bond lengths and overlap between the nodes are optimised. In addition, a simple algorithm to reduce crossovers is used. Within each cycle for each edge, a comparison is made with another (randomly selected) edge for a crossover. If there is, then the nodes are moved slightly in the direction of the crossover point. This optimisation approach was taken because elimination of all crossover points for mechanism graphs is usually impossible.

```
----------
16 29
[100,hydrogen] # [20000,hydrogen]
[100,hydrogen-peroxide] # [40200,hydrogen-peroxide]
[100,hydrogen-radical] # [10000,hydrogen-radical]
[100,hydroxyl-radical] # [20100,hydroxyl-radical]
[100,oxygen] # [20200,oxygen]
[100,oxygen-atom] # [10100,oxygen-atom]
[100,peroxyl-radical] # [30200,peroxyl-radical]
[100,water] # [30100,water]
[48200,For:SE-HO2+HO2=H2O2+O2] # [60400,For:SE-HO2+HO2=H2O2+O2]
[52200,For:SE-H+HO2=OH+OH] # [40201,For:SE-H+HO2=OH+OH]
[52200,For:SE-H+OH+M=H2O+M] # [30101,For:SE-H+OH+M=H2O+M]
[52200,For:SE-O+H2=OH+H] # [30100,For:SE-O+H2=OH+H]
[52200,Rev:SE-O+OH=O2+H] # [30200,Rev:SE-O+OH=O2+H]
[56200,For:SE-OH+H2=H2O+H] # [40100,For:SE-OH+H2=H2O+H]
[56200,Rev:SE-H+O2+M=HO2+M] # [30201,Rev:SE-H+O2+M=HO2+M]
[72200,For:SE-H+O2+M=HO2+M] # [30201,For:SE-H+O2+M=HO2+M]
[100,peroxyl-radical] # [48200,For:SE-HO2+HO2=H2O2+O2]
[100,peroxyl-radical] # [48200,For:SE-HO2+HO2=H2O2+O2]
[48200,For:SE-HO2+HO2=H2O2+O2] # [100,hydrogen-peroxide]
[48200,For:SE-HO2+HO2=H2O2+O2] # [100,oxygen]
[100,hydrogen-radical] # [52200,For:SE-H+HO2=OH+OH]
[100,peroxyl-radical] # [52200,For:SE-H+HO2=OH+OH]
[52200,For:SE-H+HO2=OH+OH] # [100,hydroxyl-radical]
[52200,For:SE-H+HO2=OH+OH] # [100,hydroxyl-radical]
[100,hydrogen-radical] # [52200,For:SE-H+OH+M=H2O+M]
[100,hydroxyl-radical] # [52200,For:SE-H+OH+M=H2O+M]
[52200,For:SE-H+OH+M=H2O+M] # [100,water]
[100,oxygen-atom] # [52200,For:SE-O+H2=OH+H]
[100,hydrogen] # [52200,For:SE-O+H2=OH+H]
[52200,For:SE-O+H2=OH+H] # [100,hydroxyl-radical]
[52200,For:SE-O+H2=OH+H] # [100,hydrogen-radical]
[52200,Rev:SE-O+OH=O2+H] # [100,oxygen-atom]
[52200,Rev:SE-O+OH=O2+H] # [100,hydroxyl-radical]
[100,oxygen] # [52200,Rev:SE-O+OH=O2+H]
[100,hydrogen-radical] # [52200,Rev:SE-O+OH=O2+H]
[100,hydroxyl-radical] # [56200,For:SE-OH+H2=H2O+H]
[100,hydrogen] # [56200,For:SE-OH+H2=H2O+H]
[56200,For:SE-OH+H2=H2O+H] # [100,water]
[56200,For:SE-OH+H2=H2O+H] # [100,hydrogen-radical]
[56200,Rev:SE-H+O2+M=HO2+M] # [100,hydrogen-radical]
[56200,Rev:SE-H+O2+M=HO2+M] # [100,oxygen]
[100,peroxyl-radical] # [56200,Rev:SE-H+O2+M=HO2+M]
[100,hydrogen-radical] # [72200,For:SE-H+O2+M=HO2+M]
[100,oxygen] # [72200,For:SE-H+O2+M=HO2+M]
[72200,For:SE-H+O2+M=HO2+M] # [100,peroxyl-radical]
```

Figure 3: The ASCII graph representation used to communication the C++ structure to the JAVA job

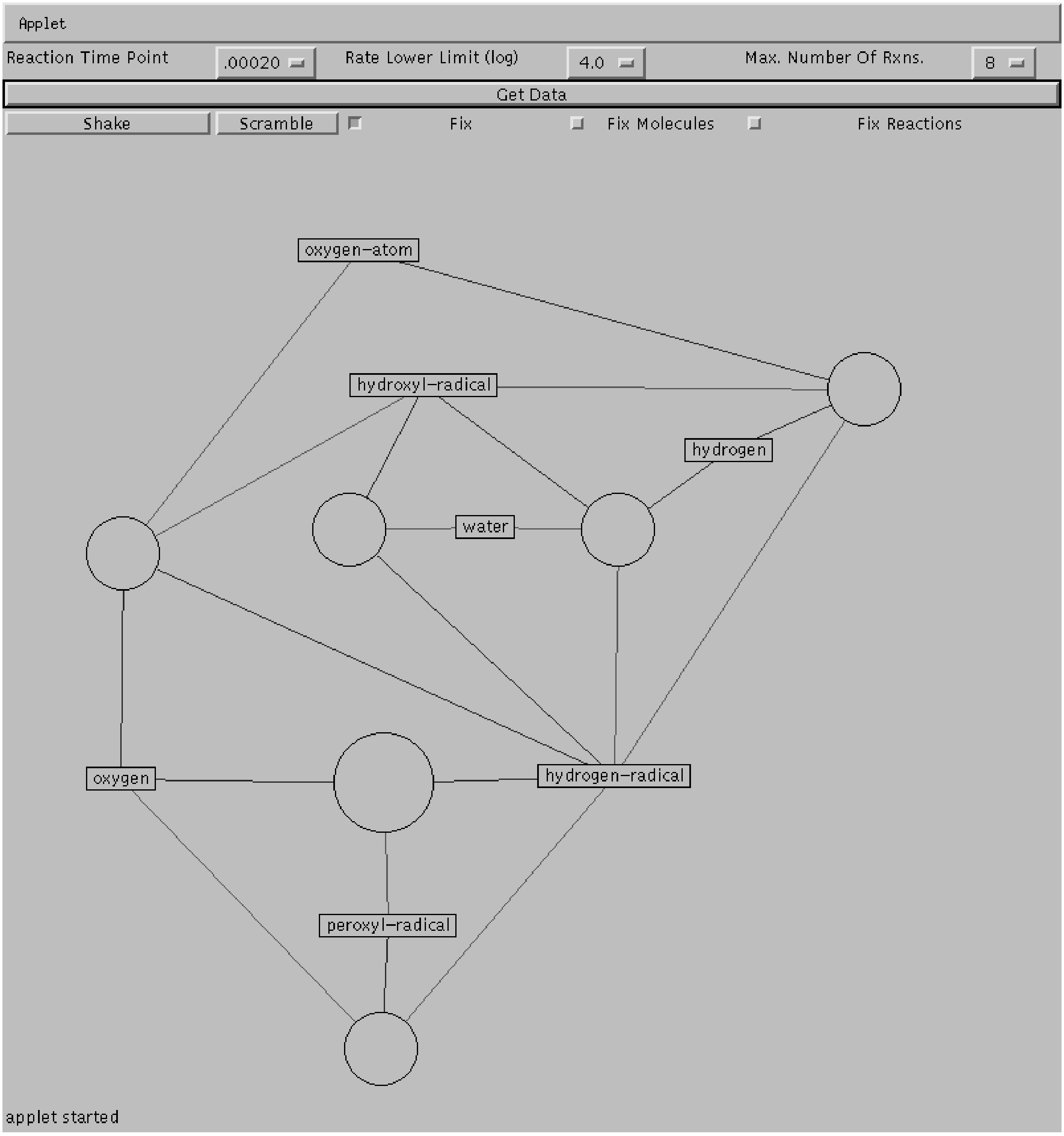

Figure 4: This is a screen dump of the JAVA interface for the viewing of the reaction network. The molecules in the reaction network are represented as boxes with the name of the molecule inside. The reactions are represented as nodes connected the reactants and products.

References

[1] Edward S. Blurock. Reaction: System for Modeling Chemical Reactions. (accepted for publication).

[2] Edward S. Blurock. Reaction: Symbolic methods in the generation and analysis of complex reaction systems. In Bertrand Braunschweig and Bernt Bremdal, editors, *Artificial Intelligence in the Petroleum Industry: Symbolic and Computational Applications II*, Paris, 1996. Editions Technip.

[3] Edward S. Blurock. Reaction: Symbolic modeling of complex reaction systems. In Johann Gasteiger, editor, *Software-Entwichlung in der Chemie, 10*, pages 33–46. Gesellshaft Deutscher Chemiker, 1996.

[4] John December and Mark Ginsburg. *HTML and CGI Unleashed, Professional Ref. Ed.* SAMS.NET, 1996.

[5] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[6] K.C. Hopson and Stephen E. Ingram. *Developing Professional Java Applets.* SAMS.net Publishing, 1996.

[7] R.J. Kee, J.A. Miller, and T.H. Jefferson. CHEMKIN: A General-Purpose, Problem-Independent, Transportable, FORTRAN Chemical Kinetics Code Package. Technical Report SAND80-8003, Sandia National Laboratories, 1980.

[8] Mark G. Sobell. *A Practical Guide to the UNIX System.* Benjamin/Cummings Publishing Company, 1989.

[9] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1991.

[10] J. Warnatz. *Combustion Chemistry*, chapter 5. Springer-Verlag, 1984.