Portable Graphical Tools for Concurrent Plasma Simulation

October 18, 1996

Bradley D. Nelson

Faculty Sponsor:
Dr. Stephen Taylor

Abstract

Low pressure reactors (less than 1.5 Torr) are used for etching processes in the microelectronics industry. Hawk, a 3-D concurrent DSMC simulation tool, is used for large-scale simulations of these reactors. In order for Hawk to be convenient for the process engineers who use such tools, XHawk, a graphical interface to Hawk, and XFalcon, the graphical interface to a chemical database used by Hawk, were created. These applications were developed for the X-Windows platform [Nelson95]. In order to make this tool useful to a broader segment of users, and to maintain support for new simulation techniques incorporated into Hawk, a new version of XHawk/XFalcon has been developed using Tcl/Tk. This multi-platform, portable, interface development language has reduced the code size of XHawk/XFalcon, making the applications more manageable and expandable, and at the same time has allowed the incorporation of many new interface features. The XFalcon chemical database system has been expanded to incorporate an improved description of chemical reactions. These interface tools are being used in the microelectronics industry for the design and evaluation of plasma reactors.

1 Introduction

The microelectronics industry has made its most recent performance advances by way of improvements in manufacturing technology. Various processes and devices have played a role in these advances. A key piece of equipment in microelectronics manufacturing is the *plasma reactor* (See Figure 1.), used in 30 to 40 percent of the wafer processing steps. These reactors use plasmas, or energetic rarefied gases, to remove particles from, and deposit particles on, silicon wafers. Improving the design of these reactors, and the processes they are used for, will enable the microelectronics industry to make smaller, cheaper, and faster microprocessors.

The design and optimization of plasma reactors has so far been largely empirical. Experiments have been conducted to improve process configurations, but because of the high equipment costs of plasma reactors, detailed parametric studies have been economically impractical. Computer-based simulation of the plasma flow inside a reactor will allow manufacturers to evaluate reactor designs and study the effect of different operating conditions on reactor performance.

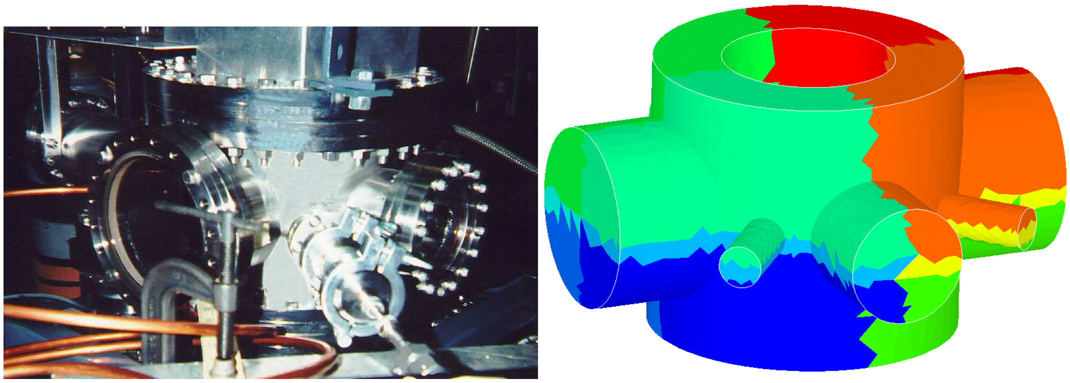

Figure 1
Gaseous Electronics Conference Plasma Reactor
A Standardized Reactor Design for Research Purposes
Experimental Model (left) and Partitioned Grid (right)

Hawk, a concurrent flow solver [Rieffel95], uses the Direct Simulation Monte-Carlo method [Bird70-94] to simulate the flow inside of a plasma reactor. The DSMC method simulates individual particles as they move through a computational grid, colliding with walls and other particles. It allows the calculation of such information as temperature, pressure, and flow velocity for discrete regions of the simulation.

Since the reactors are typically asymmetric, full three-dimensional simulations are necessary to obtain the proper results. This causes the need for the support of standard grid generation tools such as ICEM. Hawk uses a tetrahedral grid both to provide the geometry of a reactor and to divide the reactor's volume into grid cells in which macroscopic data such as temperature and pressure are calculated.

Running a Hawk simulation of a realistic reactor can be very computationally intensive. Large simulations may require more than 15 million particles and more than 1 million grid cells, with each timestep requiring millions of floating point operations. Generating useful results for such simulations may require over 100,000 timesteps, representing days of simulation time, and consuming gigabytes of memory. This makes large scale simulations impractical on single processor machines. Thus, Hawk takes advantage of parallel processor machines by mapping regions of the computational grid to each processing element.

Hawk has been designed to be portable to a wide variety parallel processing platform. It is based on the Scalable Concurrent Programming (SCP) Library [Taylor96], which uses an abstraction that allows Hawk to run on both message passing machines such as the Intel Paragon and shared memory machines such as the SGI Power Challenge.

The primary end users of this technology are process engineers who typically have little interest in the management of computer simulations. Instead, they desire easily-obtainable results. For this reason, a graphical user interface, XHawk (See Figure 2.), was created as a user environment that hides the details of configuring Hawk on a particular machine, and therefore allows the user to focus on the relevant information that a simulation can provide.

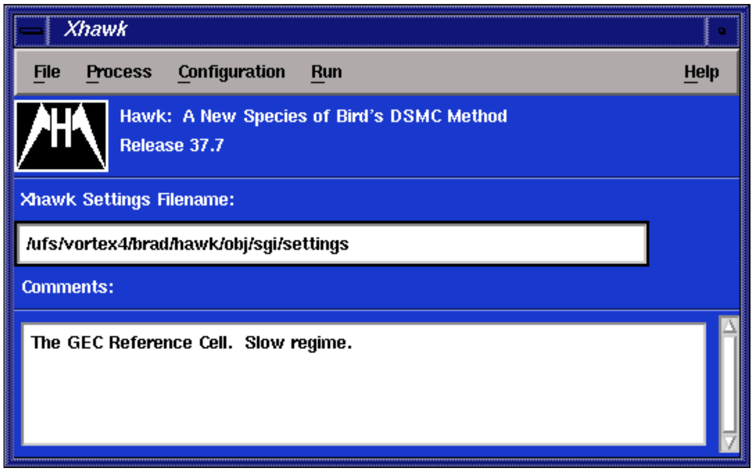

Figure 2
The Old XHawk Main Window

The simulations performed by Hawk have the potential to support complex chemical models as well as complex physics. In order to manage the wide variety of data that needs to be provided to Hawk for such chemistry, a chemical database system, Falcon, was created. This system allows the creation of a library of chemical information to be used in Hawk simulations. Falcon databases are maintained with a graphical interface called XFalcon (See Figure 3).

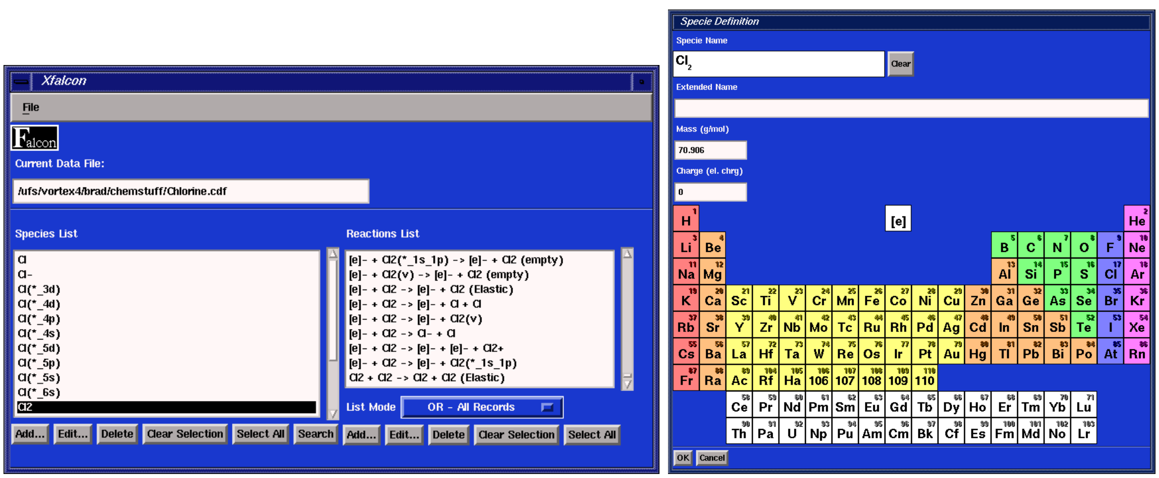

Figure 3
The Old XFalcon Interface

This application proved useful in making the Hawk simulation tool accessible process engineers. Since its first version, XHawk has gone through many changes. Though many different strategies were attempted to extend XHawk/XFalcon, they have proven difficult to expand. For this reason and the goal of Windows-UNIX cross-portability, a new version of the application was created in Tcl/Tk. This easily expandable user interface scripting language proved very practical in implementing a portable and expandable simulation interface.

In the last decade user interface software has become very complicated. Market pressures usually prevent commercial software from being completely redesigned, because of the high cost of starting a large project from scratch. Ideally, one should never have to do this; software should slowly evolve to take advantage of changes in goals and environment. In practice, large software usually cannot change beyond a certain degree simply because it was designed before the designer fully knew what the software would be used for. Sometimes software engineers have to live with their mistakes, if an application represents too much of an investment to reinvent it on a regular basis. Occasionally they get a chance to start over, however.

This report explores the some of the techniques learned in the process of recreating an application. It explore an attempt to take an application that used to require constant modification and recreate it in a form that requires minimal maintenance.

2 Language Choice

When recreating XHawk, much care was taken not to repeat old mistakes. The first major consideration was the selection of a language in which to create the new XHawk. The language would need to interface to C in order to share data structures with Hawk, but it would also need to be portable to both UNIX and Windows. The language would need to be able to deal with the changing input requirements of Hawk as it improves, while maintain compatibility with old simulation configuration files.

The C++ language was used in the previous implementation of XHawk. It proved somewhat awkward to interface it C precisely because of the subset/superset relationship between the two languages. Every time a uniquely C++ feature was used, large amounts of code had to be devoted to insulating C from this. Similar problems existed when interfacing to Motif, which was used to provide the user interface for the old version of XHawk. This toolkit, though slightly “object oriented”, proved to be very hard to “wrap up” in C++ style code. New types of interfacing approaches were considered, but ultimately the failing of C++ that made it undesirable for a second attempt was that it lacked any standard portable user interface libraries. Though it is a powerful and flexible language for handling user interfaces under a particular platform, the language is deeply in need of a broader set of standard libraries.

The Java language was seriously considered. It provides the main feature lacked by C++, a standard user interface library. The main problem with Java for this particular project was that although the language has a syntax similar to C, it provides no standard facility for linking with C libraries, and thus would be excluded from I/O routines written in C without re-coding. Also, the number of platforms with stand alone Java interpreters at the time of this project’s inception was small.

The Tcl/Tk language/library was chosen because it seems to represent the best of all possible options. It provides a high level language that can interface with C and also provides portable user interface primitives. Though the language lacks complicated hierarchical data structures, these can be added in a rather straightforward fashion within the language structure, or by linking to C code.

3 Data Management

General purpose simulation tools are inherently input and output intensive. In the extreme, the user might wish to input in the initial state of the system in as much detail as the simulation model allows. More likely, however, a user would want to configure the initial state of a simulation in terms of simplified global simulations parameters. If a simulation system is particularly general purpose, as the Hawk simulator is, it may also support the selection of different tradeoffs between precision and execution speed. Consequently large amounts of widely different types of data need to be presented to the user and stored in a file.

Data structures can be complicated both in the sense of large variety and in the sense of large quantity. In order to facilitate this, languages like C use hierarchical data structures that are defined in terms of atomic units such as individual floating point values. Usually such data structures are at least partially managed by the language itself in terms of memory allocation. Dynamic structures such as linked lists and dynamic tables often require the language user to produce code for allocating and managing them. Languages such as C++ even provide facilities for automating the allocation of such dynamic structures. What most languages do not provide, however, is an automatic mechanism to portably collapse an arbitrary data structure to permanent storage and latter retrieve it. Thus even though the compiler has a complete description of a data structure, in C for instance, in the form of a type declaration, this information is at most used to automatically allocate, copy, and free the data structure.

Since the compiler does not provide information about data structures to program code, their manipulation cannot be automated directly. Consider a system that needs to store and retrieve information contained within a data structure such as the following.

```
typedef struct {
   float x;
   float y;
} Vector;
```

This statement completely describes its data content in terms of how to allocate, free, and copy the data involved, but it also provides information about how the data might be stored portably.

Suppose a standard of text formatted numbers separated by carriage returns was settled upon as an interchange format. To implement this for the above, a C programmer would have to implement both reading and writing functions:

```
void VectorWrite(FILE *file,Vector v) {
   fprintf(file,"%f\n",v.x);
   fprintf(file,"%f\n",v.y);
}
```

```
void VectorRead(FILE *file,Vector &v) {
   fscanf(file,"%f\n",&v.x);
   fscanf(file,"%f\n",&v.y);
}
```

Both of these statements contain no more or less information than the Vector declaration above. They are all statements which repeat the structure of **`Vector`** and the perform some operation on each element in it.

Initialization, allocation, and storage become even more problematic when dynamic structures are used. In a language like C, allocating, freeing, and even copying must each be separately implemented. The C language does not provide a completely satisfactory construct for this but the next best thing can be implemented. Although C cannot be made to consolidate the information given to it in the declaration of **`Vector`**, it could use another construct to provide this information for use in the automatic implementation of the other operations that require knowledge of the data structure's content. XHawk/XFalcon uses such a technique in the form of what is called the **<u>Field Library</u>**. This library is simply a collection of functions and macros to handle managing data and data types. For example, in the Field Library, the above **`Vector`** declaration would be augmented with the following:

```
Field_Def(Vector) {
   Float_Field(x);
   Float_Field(y);
}
```

This can be implemented as a valid construct in C through the use of macros. It can be macro expanded into a function definition, which has general purpose arguments, to request allocation, duplication, and storage. This would allow the following to be performed in a function:

```
Vector *v;
Alloc_Field(Vector,v);
ReadText_Field(file1,Vector,v);
WriteText_Field(file2,Vector,v);
Free_Field(Vector,v);
```

These statements would allocate a vector field for use. Then read a value from a file into the field. Next, the value in the field would be written out to another file. The field would then be freed.

For a data structure such as a vector the benefits of sharing a code block for reading a writing to a file, may not be obvious, but this will become more relevant as the size and complexity of the data structure involved increases. Not only will this reduce by 4 functions the number of times the data structure must be described (code to read from disk and write to disk for simple structures, and code to allocate, free, and copy the data structure, for complex data structures), but it will also reduce the number of separate pieces of redundant code that must match in order to describe it.

This approach can greatly facilitate data structures that need to change over time yet be able to load data stored in previous versions of the data structure. For instance with a simulation, as the simulation becomes more complicated, new types of input information might be needed. Ideally, these new data elements would simply need to be added to the appropriate data structures without the concern of writing new code to handle loading the data etc. Also it might be desirable to store new data in an old file format or to automatically set new fields that were unspecified in some old format to a default value. If all of this is implemented as separated file reading, file writing, allocation, and copying functions, it will rapidly become impractical. Consistency will have to be maintained between all of the various parts.

The approach used in the Field Library provides a better solution to this problem. For example suppose the **Vector** class above was expanded to 3-dimensions. Note that it might be desirable to be able to load from disk vectors stored in the old 2D format and have a default z value. Large amounts of code would have to be changed with a traditional solution. Falcon instead would use the following.

```
#define Fields_Version 2
typedef struct {
   float x;
   float y;
   float z;
} Vector;

Field_Def(Vector) {
   Float_Field(x);
   Float_Field(y);
   if(Ver>=2) {
     Float_Field(z); INIT_VALUE { data.z=1; }
   }
}
```

This would allow the version to be specified for each operation. A version 1 file could be read even though a version 2 data structure is allocated and initialized to default values. Thus z would default to 1 if loaded from an old file.

This style of version aware automatic data structure management is used throughout XHawk/XFalcon to simplify the application as a whole. It actually supports two forms of automatically generated reading and writing to files. One is a human readable text format that shows the value of each field and its type name and nested structure. The other is a packed binary format implemented in SCPlib [Taylor96], which provides a compact structure that is still portable, but not human readable. A complete refrence of the field system's functions is provided in Appendix A.

This system of fields has been interfaced to Tcl. Tcl's expansion facility makes linking to C straightforward and allows it to use a similar but properly Tcl-like interface to data files generated by the Field Library.

The decision was made to implement support for fields in a form similar to widgets in Tcl/Tk. Each allocated field is a global procedure which though poorly suited to many application, is ideal for applications like XHawk/XFalcon which are centered around one main data type. A complete refrence of the Tcl interface to the field system is provided in Appendix B.

The data field management system used in XHawk/XFalcon is implemented as a reasonable compromise between ease of programming and speed. Each data type has associated with its type declaration a function that is declared and defined in the guise of being a second type declaration. This declaration is macro expanded into a function which contains a series of calls to atomic data type manager functions with parameters which are hidden in the declaration. This function is then called internally to each field in the data type to perform a particular operation.

This method incurs a small performance penalty when compared with creating a separate read/write/allocate functions. For each element in a data structure, one function call must be made and a number of parameters need to be passed. Since each data element handler needs to perform every supported kind of operation on the data element it is in charge of, there a several parameters that go unused in each call.

On balance, this performance penalty is mild. For I/O operations compared to the cost of file access it can be ignored. For allocation and copying, it essentially incurs an extra function call layer for each data type, as compared with the hand-made approach. This is likely to be a lighter speed drain than the processing that may need to be done for calls to memory allocation code. Additionally when allocating tables/arrays, this sort of performance sacrifice is likely to be small compared to the cost of allocating each member of a table (Note this is assuming that the table contain dynamic structures that cannot be block allocated). C does not provide facilities to initialize a complex data structure automatically deal, and so for these is thus no better than the Field Library. Optimizing C++ compilers can in principle handle this but it requires abandoning the atomic data types in favor of data aware versions of them, which can incur a drastic performance penalty, unless the compiler is capable of eliminating all of the overhead associated with classes that have a single atomic data element internally.

Another possible option which was considered and abandoned is using Tcl data types. Tcl provides only hash tables and strings as data types. Though in principle this is complete enough for any type of data, in practice, particularly when dealing with interfacing with C code, faster data structures would be desirable. Thus, it is wise to look into augmenting Tcl more conventional data types. The technique used in XHawk/XFalcon provides Tcl with something it lacks when dealing with large data structures, namely, it can now use compiled routines for manipulating specific data structures, rather then interpreted code using strings, which leads to a performance increase.

4 Application Framework

XHawk/XFalcon consist of two main types of interrelated information. First, it must have a description of the data that it must store and retrieve. Second, it must have a description of how this data can be logically and usefully presented to the user. Ideally, since this data is provided for the purpose of user selection, there will be a close correlation between the two.

One lesson learned in the development of XHawk is that although simulation configuration can involve complicated data structures and complicated user interface layouts to accommodate these data structures, application structure is very uniform. A simulation configuration simply consists of a means to allow a user to enter and edit simulation parameters and to store and retrieve them. It is a role that potentially could be handled by conventional databases, were it not for their lack of hierarchical data types.

This common set of features in a simulation configuration application led to the creation of a common **Application Framework** that is shared by both XHawk and XFalcon. All but one of the Tcl/Tk source files in each application is part of the framework and does not have any ties to the specific data handled by each application. This large body of common code describes a set of interface "widgets" that can be associated with data and it handles generic tasks like user input range checking. It allows a user interface composed of data aware widgets to be created.

The Application Framework allows the one un-shared Tcl/Tk source file of a simulation configuration application to be simply an abstract description of how each of the data elements maintained by the application should be associated with user interface elements. An example use of this framework is provided in Appendix C.

5 XFalcon

XFalcon is an application designed to maintain a database of chemical properties in the context of a Hawk simulation. These properties have a relatively simple but potentially flexible structure.

The chemical database consists of a collection of compounds. These compounds are described in terms of their atomic contents and in terms of charge and other parameters to uniquely distinguish them. Users can select elements on a periodic table to make up the content of a compound. They may also specify information describing the rotational and vibrational properties of multiatomic compounds.

Compounds are used to describe the inputs to a collision. Collisions are descriptions of how particles may be transformed when they come together. This description consists of the probability a collision occurring, and a description of its yield.

For example, consider the following table of chlorine reactions.

Table 1:
<u>Chlorine Reactions</u>

Electron Reactions

$Cl_2 + e \rightarrow 2Cl + e$ (dissociation)

$Cl_2 + e \rightarrow Cl_2^+ + 2e$ (ionization)

$Cl + e \rightarrow Cl^+ + 2e$ (ionization)

$Cl^- + e \rightarrow Cl + 2e$ (detachment)

$Cl_2 + e \rightarrow Cl^- + Cl$ (dissociation attachment)

$Cl + e \leftrightarrow Cl^+ + 2e$ (ionization)

Molecular Reactions

$Cl2^+ + Cl^- \rightarrow Cl_2 + Cl$ (mutual neutralization)

$Cl^+ + Cl^- \rightarrow Cl_2$ (mutual neutralization)

$Cl+ + Cl_2 \rightarrow Cl_2^+ + Cl$ (charge transfer)

$2Cl \rightarrow Cl_2$ (diffusion)

The chemistry model in XFalcon has been revised from the original XFalcon. The model now used describes collisions in terms of a "collision model", which describes the probability of a collision and a set of "reactions", which describe the probability of certain yields given that a collision does occur. This model is used in contrast with the old model which describes the probability that a given yield occurs, given two particles. For more information see [Gimelshein96].

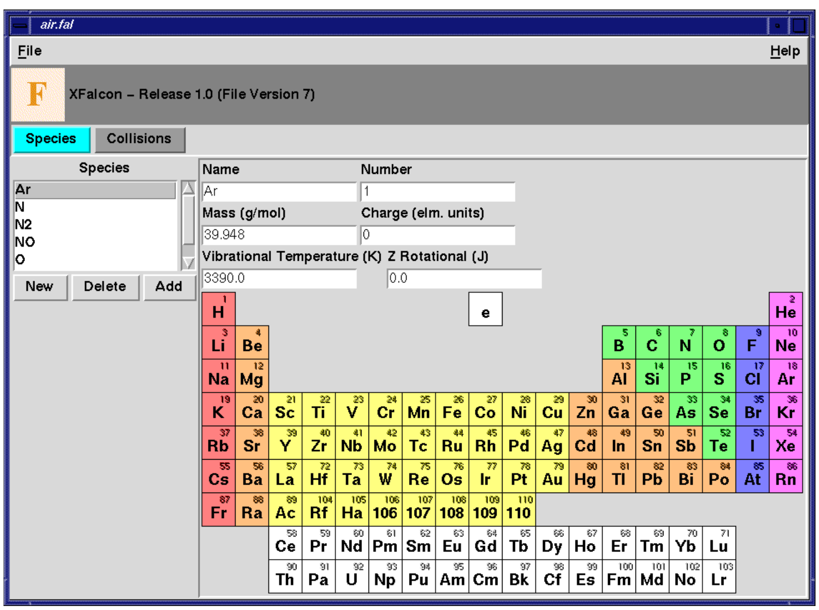

Figure 4
The New XFalcon (Species Panel)

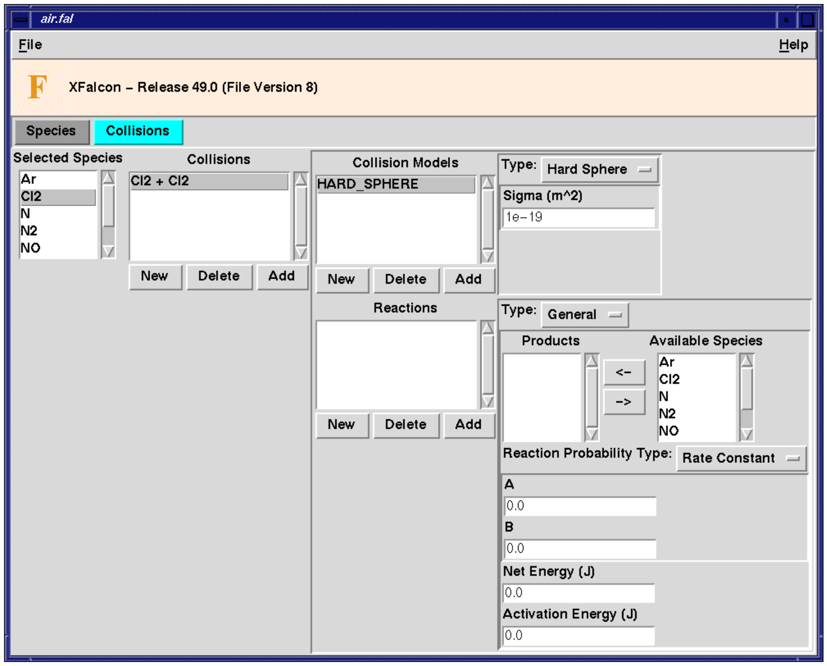

Figure 5
The New XFalcon (Collision Panel)

6 XHawk

XHawk is an application intended to configure a Hawk simulation. It manipulates a collection of data fields that describe how the simulation should be run on multiple processors, the source of geometry the simulation should use, the way in which information should be collected from the simulation, how gravity and electromagnetic fields should be simulated, and how surfaces and regions in the simulation geometry should behave. XHawk also handles importing subsets of chemistry databases created in XFalcon.

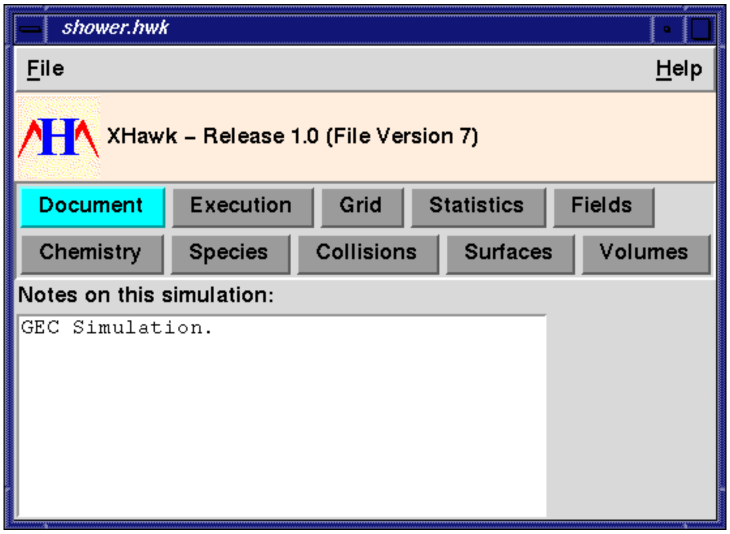

Figure 6
The New XHawk (Document Panel)

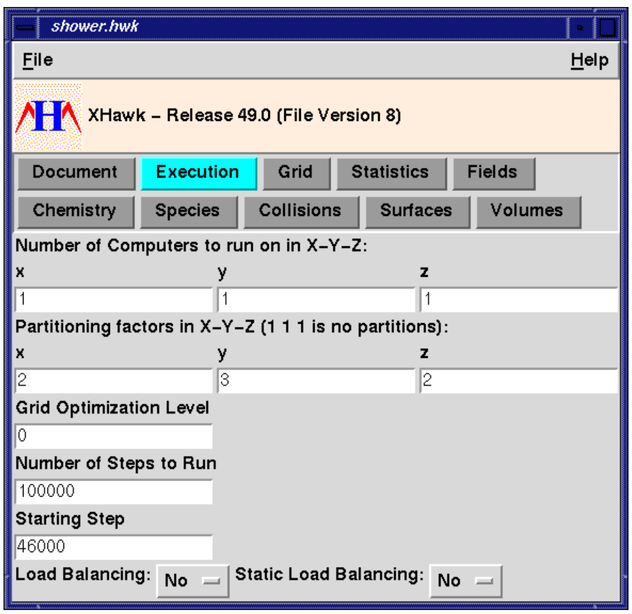

Figure 7
The New XHawk (Execution Panel)

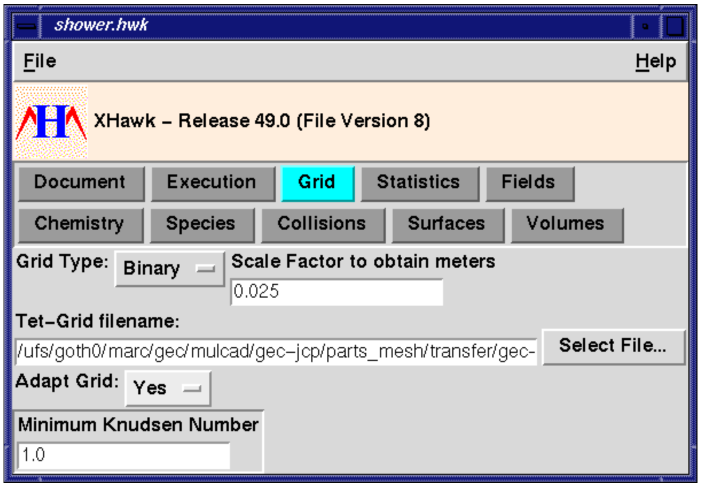

Figure 8
The New XHawk (Grid Panel)

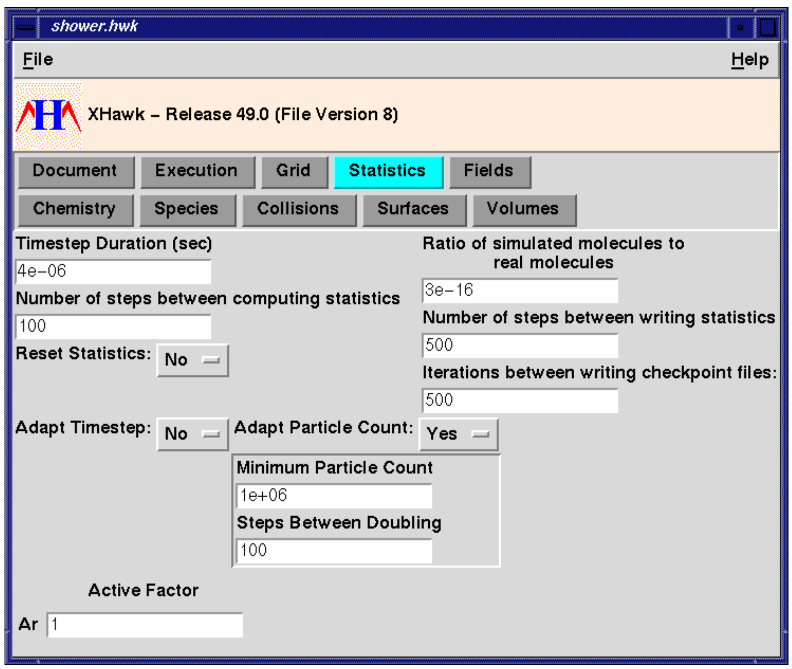

Figure 9
The XHawk (Statistics Panel)

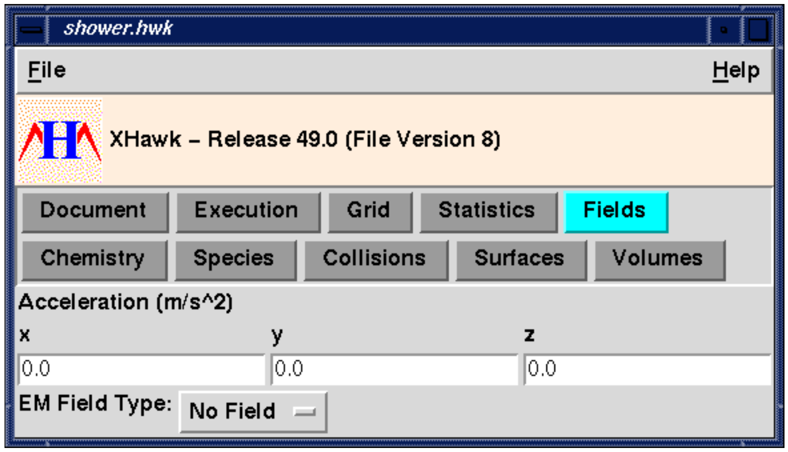

Figure 10
The New XHawk (Fields Panel)

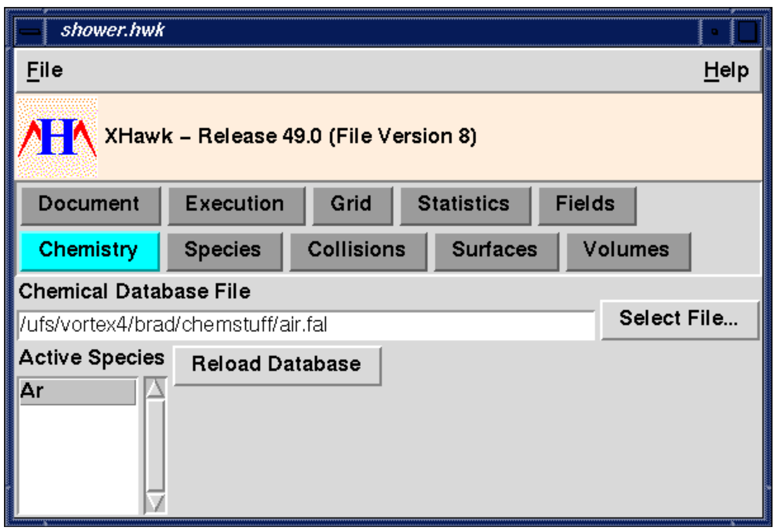

Figure 11
The New XHawk (Chemistry Panel)

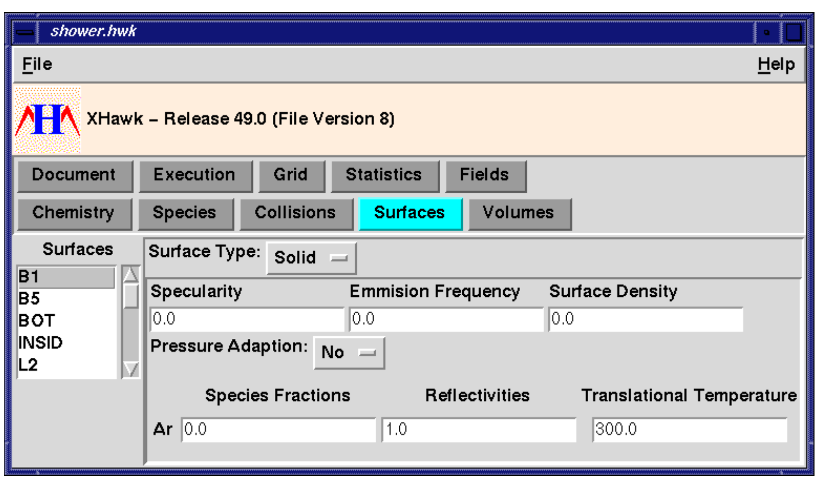

Figure 12
The New XHawk (Surfaces Panel)

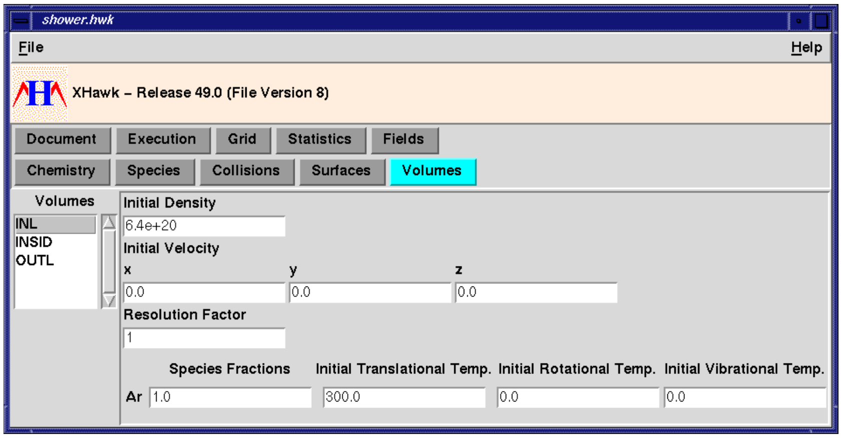

Figure 13
The New XHawk (Volumes Panel)

Conclusions

The use of the Field Library has made possible extensive simplification of XHawk/XFalcon's design in comparison to the previous version. It makes unnecessary what was once the single largest source file in XHawk, a set of reading and writing routines for the application's data structures. The Field Library illustrates that consistent, high level, handling of data can greatly reduce application complexity.

The Field Library also points out strong deficiencies in modern compilers. It suggests a methodology for managing data that illuminates rather than hides how data is internally stored. The Author strongly suggests that future languages and libraries take into account the need of applications to be able to be introspective with regard to data structures. It is pointed out that data structures should not be thought of as black boxes but rather as a category of data structure in themselves.

The Application Framework proved to be both the biggest help and biggest inconvenience in implementing XHawk/XFalcon. The framework, though general purpose, is also restrictive. Use of the framework meant that both applications look and function nearly identically, even in instances when this is not ideal. The basic premise of the framework was a success, however, since it lead to reduced total code size.

The use of Tcl/Tk proved successful in reducing the total code size of the application, while at the same time increasing functionality. The use of an interpreted language for the task of user interface management is reasonable and practical. This tool is highly recommended for use in future user interface application.

The final product has proven itself to be an easily expandable practical tool. The new XHawk and XFalcon are have replaced their predecessors, which have been used successfully by Intel, Corporation. These applications are now an active part of research to improve integrated circuit production technology.

Acknowledgments

The Author would like to acknowledge the support and guidance of Marc Rieffel in this project; with his help XHawk has hopefully become the kind of tool he envisioned over a year ago. The Author would also like to recognize the guidance and support provided in the project by advisor Dr. Stephen Taylor.

The research described in the report is sponsored primarily by the Advanced Research Projects Agency under contract number DABT63-96-C-0116. The information contained herein does not necessarily reflect the position or policy of the government of the United States, and no official endorsement should be inferred.

Appendix A - Field Library (C interface)

The field system contains a collection of types. Each is listed bellow with an example of its declaration syntax and its C declaration.

```
int x;

Int_Field(x);
```

The above declares an integer type field, which has a default value of 0.

```
float x;
Float_Field(x);
```

The above declares a floating point type field, which has a default value of 0.

```
double x;
Double_Field(x);
```

This above declares a double precision floating point type field, which has a default value of 0.

```
typedef enum {
ONE,
TWO,
THREE
} Counter;
Enum_Def(Counter)="ONE TWO THREE";
```

The above declares an enumerated type field. It is designed to be used for enumerated types that do not necessarily have a connection between their enumerated names and particular integer values. When a fixed integer value is needed, the Int_Field type should be used. When old names become outdated, they should be left in as place holders. It has a default value of the first name identifier.

Table(Vector,vects);
Table_Field(Vector,vects);
The above declares a table of another user defined type. This data type is designed to handle tables containing data elements that are all of the same type. These tables can be dynamically altered and should provide a reasonable substitute for most dynamic structures. They are implemented as an array of pointers to data elements. They can be reordered with only the cost of pointer array movements, which provides a reasonable balance between a completely dynamic structure that is fast when adding elements, but slow to traverse.

Vector v;
User_Field(Vector,v);
The above declares a user defined field. User defined fields are designed to allow the definition of nested field types. Note, although the outer most layer is handled as a pointer to the data structure, internally each structure should use direct references to allow the compiler to use its more efficient implementation of copying and allocation.

The following are functions for manipulating user defined fields. In each of the examples, assume the existence of a **Vector** pointer **v** in the current scope.

void Alloc_Field(Type, Type *x);
This function allocates **x** as a **Type** field and initializes its members to their default values.

void Free_Field(Type, Type *x);
This function frees **x** and all of its members.

void Write_Field(int portid, Type, Type *x);
void Write_Text_Field(FILE *file, Type, Type *x);
These functions write a binary representation and a text representation to an SCP_Port and a file respectively.

void Read_Field(int portid, Type, Type *x);
void Read_Text_Field(FILE *file, Type, Type *x);
These functions read a binary representation and a text representation from an SCP_Port and a file respectively.

Together, these constructs can provide the framework to describe most of the kinds of data structures that would be needed for input to a simulation. One possible addition would be a pointer type to allow for fully dynamic data structures. Though this is possible, it would be difficult to enforce in a language like C, which does not allow polymorphic functions and operator overloading.

Ideally a compiler should provide more information about the structure of a data type to code within the language, or should handle some of these issues more effectively itself. The separation between data structure definition and representation of its internal structure is useless in any practical implementation of a procedural language. Languages that do this ,such as C/C++, necessarily maintain the needed information in any reasonable implementation because they need to use it internally for implementation of copying/allocation. (Java attempts to address some of these issues by providing a string packing mechanism, but this is not automatic and does not provide the necessary enhancements.)

Appendix B - Field Library (Tcl Interface)

`scp open filename mode`
This command takes a filename and an access mode and returns a file handle of a special form.

Modes

w write to a text file
r read from a text file
wb write to an SCPlib binary packed file
rb read from an SCPlib binary packed file

`scp close portid`
This closes handles generated by scp open.

`scp writeint portid val`
This writes the integer **`val`** to the scp port **`portid.`**

`scp readint portid`
This returns an integer read from the scp port **`portid.`**

`scp writestr portid val`
This write the string **`val`** to the scp port **`portid.`**

`scp readstr portid`
This returns a string read from the scp port **`portid.`**

`field fieldtype fieldname`
This allocates a field of type **`fieldtype`** named **`fieldname.`**

After a field is allocated it can be used as follows:

`fieldname write portid`
This writes the field to a port.

fieldname read portid
This reads the field from a port.

fieldname = fieldname2
This assigns the value of **fieldname2** to **fieldname.**

fieldname = fieldname2 (subelement path dot delimited)
This does an assignment with a sub element of **fieldname2.**

fieldname subelement = fieldname2
This assigns a sub-element of fieldname the value of **fieldname2.**

fieldname subelement = val
This assigns a value to an atomic type in **fieldname.**

fieldname subelement
This returns the value of an atomic type in **fieldname.**

fieldname table-element count
This returns the count of items in a table.

fieldname table-element add item
This adds an element to the end of a table.

fieldname table-element ins pos item
This adds an element to a table at position **pos.**

fieldname table-element del pos
This deletes an element from a table at position **pos.**

fieldname table-element clear
This blanks a table.

Appendix C - Application Framework Example

```
#  Name of the application
set AppName particles

# Release number of the application
set AppRelease 1.0

# Extension used by the application
set AppExtension .part


# How to call the application
```

```
set AppFileType "Particles File"

# The filename of the application executable
set AppCommand particles

# The text used to identify the application's
# file type
set FileHeaderText "Pariticle Simulation files"

# Either binary for SCPlib packed binary files
# or text for human readable files
set FileMode binary

# The name of the field which is the entire
set MainDataType ParticleSettings

# The filename of an image to be used for display in the
# application all the time
set AppLogoSmall plogo.gif

# The name of an image to be used as a backdrop for the
# about box
set AppLogoBig pflash.gif

# The name of an image to be used as the application
# icon
set AppLogoIcon plogo.xbm

# The name of a Tcl action to bring up full help info
set HelpDocAction {exec netscape http://paticlesim.com / &}

# Main application procedure
proc AppMain {w} {
   ;# Sets up a window filled with tabs for different panels
   MakeTabs $w.1 5 {
       ;# Prepares two tabs, "Color Parameters" and
       ;# "Shape parameters" defined below
       "Color Parameters" {
           ColorsPanel $w.1
           pack $w.1 -side top -anchor w
       }
       "Shape Parameters" {
           ShapesPanel $w.1
           pack $w.1 -side top -anchor w
       }
   }
   pack $w.1 -side top -anchor w
}
```

```
# Defines the color parameters panel
proc ColorsPanel {w} {
    frame $w
    ;# Defines two prompts associated with data fields
    Prompt $w.1 .redones "Number of Red Particles"
    pack $w.1 -side top -anchor w
    Prompt $w.2 .greenones "Number of Green Particles"
    pack $w.2 -side top -anchor w
}

# Defines the shape parameters panel
proc ShapesPanel {w} {
    frame $w
    ;# Defines an option box with 3 choices, one of which
    ;# has an addition parameter for that state
    OptionMenu $w.1 .shapetype "Type of Shape:" {
        BOX_SHAPE “Box” {}
        BELL_SHAPE “Bell” {}
        CIRCLE_SHAPE “Circle” {
            Prompt $w.1 .circlesize "Radius"
            pack $w.1 -side top -anchor w
        }
    }
   pack $w.1 -side top -anchor w
}

# Text to appear in the about box
set AboutText "$AppName - Release $AppRelease

Collaborators
Manny
Moe
Jack

Contact us at:
feedback@particlesim.com"
```

This code produces the following application (the fields must be defined for a simulation application).

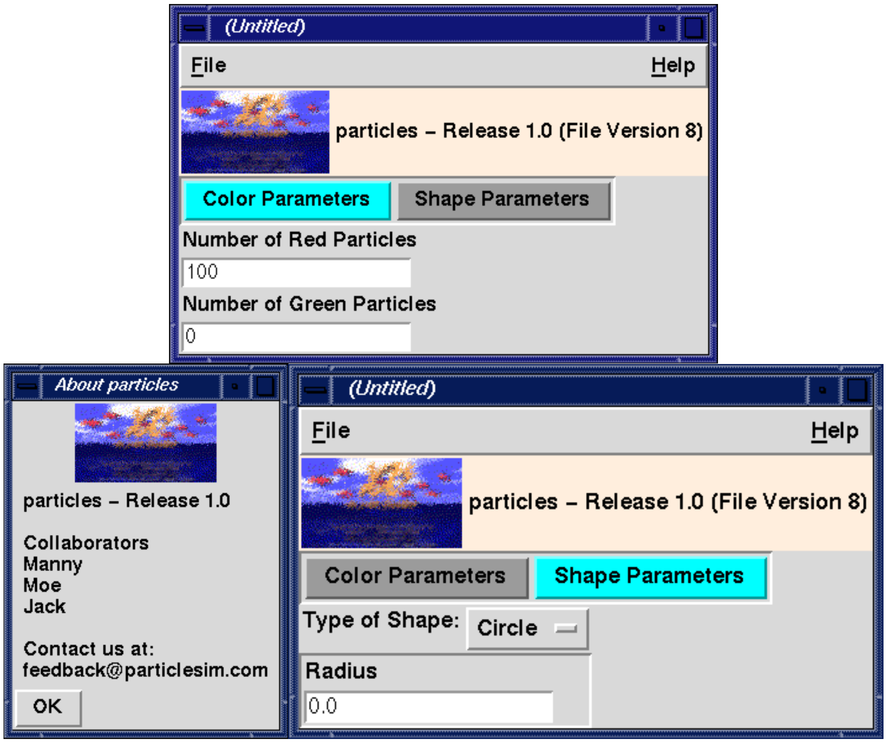

Figure 14
Sample Application

References

[Rieffel95] M. Rieffel. "Concurrent Simulations of Plasma Reactors for VLSI Manufacturing." California Institute of Technology Masters Thesis, 1995. CIT-CS-TR-95-012.

[Watts95] J. Watts. "The Extended I/O Library." California Institute of Technology Technical Report, 1995. CIT-CS-TR-???

[Watts96] J. Watts, M. Rieffel and S. Taylor. Practical Dynamic Load Balancing for Irregular Problems. Parallel Algorithms for Irregularly Structured Problems: IRREGULAR `96 Proceedings, pg. 299-306, Springer-Verlag LNCS, Vol. 1117, 1996. (NASA)

[Bird94] G. Bird. “Molecular Gas Dynamics and the Direct Simulation of Gas Flows.” Clarendon Press. Oxford, 1994.

[Taylor96] S. Taylor, J.Watts, M. Rieffel, M. Palmer. “The Concurrent Graph: Basic Technology for Irregular Problems”, IEEE Parallel and Distributed Technology, 4(2):15-25, 1996.

[Gimelshein96] S. Gimelshein, G. Markelov, and M. Rieffel. Collision Models in the Hawk DSMC Implementation, Caltech Technical Report CIT-TR-96-16, 1996.

[Nelson95] B. Nelson. “Graphical Tools for Concurrent Plasma Simulation.”, Caltech SURF ’95.