INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

University of Alberta

A Framework for Reactor-Scale PVD Simulations

by

Sean Leonard

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science.

Department of Electrical and Computer Engineering

Edmonton, Alberta

Spring 2002

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services

Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-69726-6

Canada

University of Alberta

Library Release Form

Name of Author: Sean Leonard

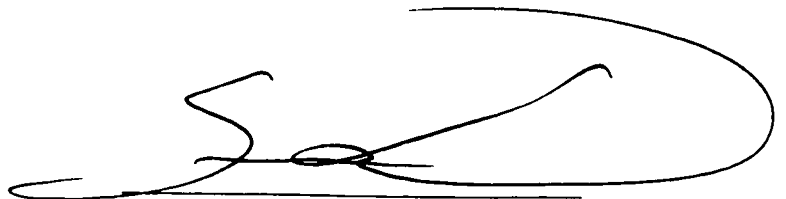
Title of Thesis: A Framework for Reactor-Scale PVD Simulations

Degree: Master of Science

Year this Degree Granted: 2002

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.



Sean Leonard
10980 74 Ave
Edmonton Alberta, Canada
T6G 0E6

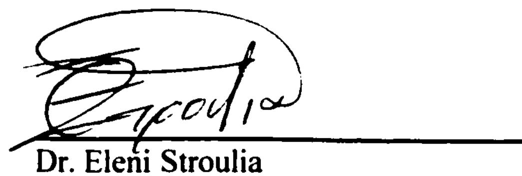
April 15, 2002

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled *A Framework for Reactor-Scale PVD Simulations* submitted by Sean Leonard in partial fulfillment of the requirements for the degree of Master of Science.

Dr. Steven K. Dew



Dr. Eleni Stroulia

Dr. Christopher Backhouse

april 15, 2002

Abstract

The microelectronic industry has been a driving force in thin film process evolution. The requirements of physical process optimization are becoming more stringent due to more demanding topographies and larger wafers. A common cost-effective solution to process optimization has been, and will continue to be, computer simulation.

One key process for the microelectronic and other coatings industries is physical vapor deposition. In an attempt to increase the flexibility of reactor scale physical vapor deposition simulators, a framework has been developed to address this problem domain. This thesis presents a framework that not only encompasses a more powerful version of the industry renowned product SIMSPUD, but also provides future developers with an extensible architecture for future enhancement.

Table of Contents

1. INTRODUCTION 1

1.1 BACKGROUND 1
1.2 MAGNETRON SPUTTER DEPOSITION 3
1.3 THE PROBLEM 6
1.4 SIMULATIONS, COMMERCIAL AND ACADEMIC 7
1.5 FRAMEWORKS 12
1.6 EXISTING FRAMEWORKS 14
1.7 SIMSPUDII FRAMEWORK FEATURES 16
1.8 THESIS ORGANIZATION 23

2. ARCHITECTURAL DOMAIN: FRAMEWORKS AND TECHNOLOGIES 24

2.1 FRAMEWORKS 24
2.2 KEY ENABLING TECHNOLOGIES 28

3. ARCHITECTURAL DOMAIN: FRAMEWORK ANALYSIS 36

3.1 THE SIMSPUDII FRAMEWORK 36
3.2 THE SIMPUDII PROCESS 48

4. PROBLEM DOMAIN: IMPLEMENTATION OF DESIGN 53

4.1 SIMULATION 54
4.2 OBSTRUCTIONPROCESSOR 62
4.3 THERMSPUD1 79
4.4 SOFTWARE SUPPORT SYSTEMS 93

5. PROBLEM DOMAIN: RESULTS 97

5.1 OBSTRUCTIONPROCESSOR 97
5.2 THERMSPUD1 99
5.3 GRIDS 109

6. CONCLUSIONS AND RECOMMENDATIONS 110

6.1 SUMMARY 110
6.2 FUTURE WORK 112

BIBLIOGRAPHY 116

Table of Figures

Figure 1-1: Sputter deposition process 4
Figure 1-2: PVD industry financial projections 5
Figure 2-1: Problem domain coverage 24
Figure 2-2: SIMSPUDII framework 26
Figure 2-3: Simulation parameter space 31
Figure 2-4: Example of XML format 32
Figure 2-5: Merlot, an open source XML editor 33
Figure 2-6: Module creation using XML 35
Figure 3-1: SIMSPUDII framework basics using the T-COM 39
Figure 3-2: Module initialization using XML 41
Figure 3-3: XML specified application 1 using the T-COM 44
Figure 3-4: XML specified application 2 using the T-COM 46
Figure 3-5: Basic application using the T-COM 47
Figure 4-1: Simulation XML tree 54
Figure 4-2: Message routing 57
Figure 4-3: Routed streams 58
Figure 4-4: Dependency structure of an Object Oriented Framework[25] 59
Figure 4-5: PPD XML Node 60
Figure 4-6: ObstructionProcessor XML node 64
Figure 4-7: CollectionRegions 66
Figure 4-8: LinearProfile XML node 66
Figure 4-9: ParticleGenerator XML node 71
Figure 4-10: Chamber wire-frame representation 73
Figure 4-11: Cylinder Obstruction wire-frame representation 75
Figure 4-12: PipeObstruction wire-frame representation 76
Figure 4-13: Box Obstruction wire-frame representation 78
Figure 4-14: Thermspud1 algorithm 83
Figure 4-15: Thermspud1 XML node 83
Figure 4-16: Particle path length calculations 86
Figure 4-17: ZBL sigma vs. energy 91
Figure 4-18 Magic formula fit with integrated universal potential[30] 92
Figure 4-19: Build results 95
Figure 5-1: Particle stopping of a pipe with a tilted cap (side view) 97
Figure 5-2: Particle stopping of a pipe with a tilted cap (oblique and top view) 98
Figure 5-3: Particle stopping of a box 98
Figure 5-4: Particle paths (oblique view) 99
Figure 5-5: Particle paths (side and top view) 100
Figure 5-6: Particle generation from a histogram 100
Figure 5-7: Thermal data slice 101
Figure 5-8: Thermal slice with obstruction overlay 101
Figure 5-9: Energy distribution 103
Figure 5-10: The effects of target-substrate distance on relative deposition rate 104
Figure 5-11: Erosion profile 105
Figure 5-12: Radial profiles 106

Figure 5-13: SIMSPUD radial profiles[28] ... 106
Figure 5-14: The effects of target-substrate distance on average deposited energy ... 107
Figure 5-15: Energy deposition profiles ... 108
Figure 5-16: Refined grid ... 109

---

1. Introduction

1.1 Background

At the end of June in 1948, an announcement was made that has irrevocably changed the world we live in. Bell Labs announced the invention of a device that they called a 'transistor'. Although the announcement did not create a unanimous 'Eureka!' from the world, this device touches the lives of every person in industrialized countries. In 1948 the pages of *Scientific American* showed a picture of a device just smaller than a paper clip that would some day replace all the tubes at the heart of the radios of the day[1].

In 1965 Gordon E. Moore (now chairman Emeritus of Intel) made an easily dismissible claim in the anniversary edition of *Electronics* magazine. Up to the time of the article, Moore recognized that a pattern was emerging in the manufacturing of integrated circuits; complexity was doubling every year (50 transistors per chip at the time). Moore predicted that this trend would probably continue and that some day integrated circuits would lead to such wonders as home computers, personal communications equipment, etc[2].

Moore's statement enunciated the driving force for the microelectronics industry and is now known as *Moore's Law*: transistor counts will double every 18 months. It has been argued that Moore's Law has produced a vision for the industry. Today Intel produces processors with 42,000,000 transistors: almost a million times more than seen in 1965. We now have the computers and personal communicators that Moore predicted. Today's

personal data assistants (PDA's) are more powerful than the desktop computers of five years ago. One could question if the current trend of the unprecedented miniaturization of integrated circuits is due to the world's insatiable thirst for electronic devices, or if the relationship is the other way around. With increases in the transistor densities, decreases in the cost per transistor have also followed. Incredible computing power at reasonable prices has been a tremendous benefit to society.

The success of Moore's Law can be attributed to the advances in process technology. Many barriers and stumbling blocks have been predicted for the industry, but each time process engineers have been able to develop a new approach to circumvent the challenge. However, the price of this success has been an inexorable increase in process complexity and ever more exacting equipment specifications. The result is that those process sequences, which formerly involved a dozen steps, now require hundreds; fabrication plants, which used to cost millions of dollars, now cost billions. The industry's ability to increase transistor densities is a direct result of research pushing the physical limits of the chip manufacturing process. In some cases, progress has emerged from engineering ingenuity that has been explained theoretically later, or progress has been a result of theoretical predictions that have enabled new processes. Not only are new materials and techniques regularly used in creating devices, but also new configurations of existing technologies often increase transistor densities.

In order to address the challenge of meeting very exacting requirements during process development without tying up expensive capital equipment, process engineers are

resorting to extensive use of simulators. However, there are many complex and interacting physical and chemical processes, which must be accounted for before these simulators can deliver on their full promise. In particular, the ability to account for the coupling of variables, both at the sub-micron feature scale and the multi-centimeter reactor scale, is essential to gain an adequate understanding of the key mechanisms involved and achieve even semi-quantitative simulation results.

1.2 Magnetron Sputter Deposition

One of the key steps in microelectronic device manufacturing is the process of depositing layers of thin films. Depending on the particular layer composition, various film growth processes are utilized. One film deposition technique that has been central to the growth of metallization layers is known as magnetron sputter deposition. Magnetron sputter deposition is categorized with evaporation deposition, both are known as Physical Vapor Deposition (PVD) processes. Although evaporation is classified as a PVD process, typically when one refers to PVD, one usually refers to magnetron sputter deposition. This convention will be followed for the rest of this thesis.

PVD films are grown via a process of collecting physically ejected (sputtered) particles from a solid target onto a substrate. The process is driven by a self-sustaining low-pressure (1-50 mTorr argon) plasma that forms at the target and is contained by magnets placed behind the target. A negative bias is applied to the target to attract argon ions from the plasma. Particles are physically ejected from the target as a result of momentum

transfer from the bombarding ions. These ejected particles make their way through a sequence of collisions with the background argon gas until they collect as a film layer upon the substrate (Figure 1-1). A variation on this process is achieved by introducing a reactive gas species into the vacuum chamber (reactive sputtering).

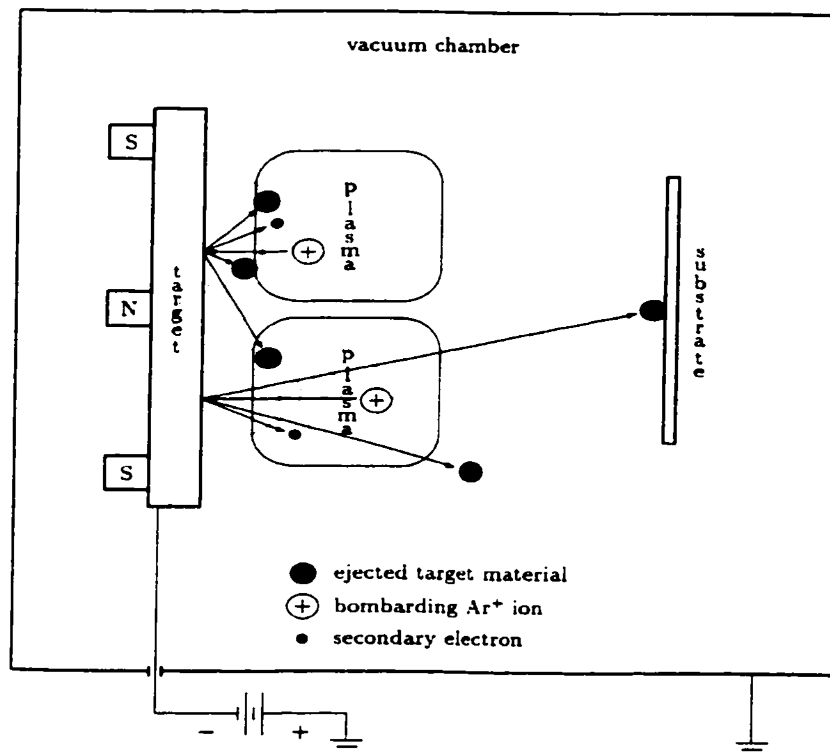


Figure 1-1: Sputter deposition process

This diagram outlines the basics of the magnetron sputtering process. The chamber is filled with a low-pressure argon gas. A glow discharge plasma is created from a potential on the metal target. Target material is ejected as a result of accelerated ion bombardment. Magnets confine the plasma near the target for greater efficiency[28].

The attraction to sputter deposition is the process's ability to deposit a variety of film materials onto a variety of receptor materials. This ability, combined with the multitude of control variables that can be 'tweaked' in the sputter process, greatly affect the deposited film properties. Sputtered films are known to have good uniformity and

acceptable coverage over non-planar features. Due to the recent interest in the use of copper metallization layers, the use of sputter deposition processes for seed layer deposition over very high aspect ratio topography has kept the interest in this PVD process technology high.

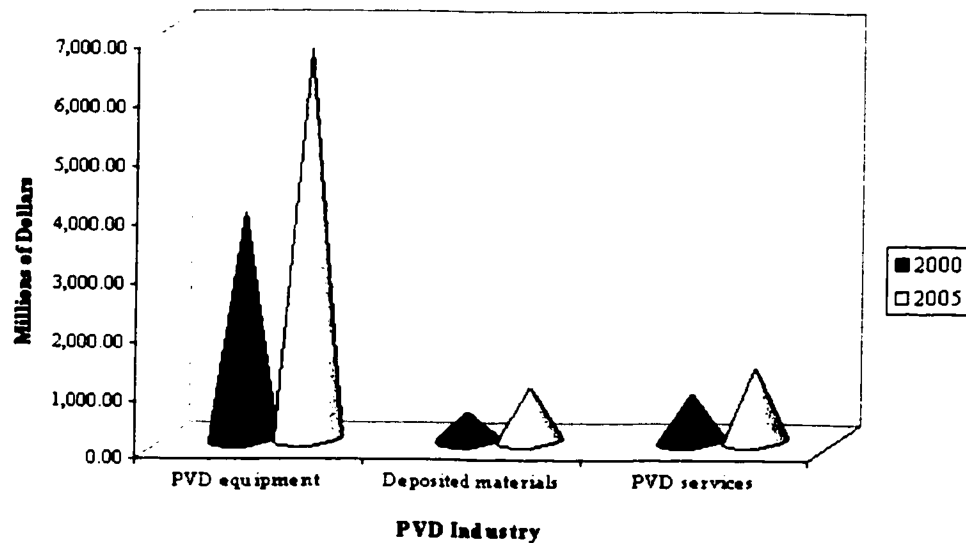


Figure 1-2: PVD industry financial projections

The PVD industry has been a good market performer in the past and is predicted to be growing[3].

PVD is used not only in the microelectronic fabrication industry, but also extensively in coatings for hard drives and other magnetic storage media, optical coatings, hardness coatings, textile treatments and packaging. According to a study by Business Communications Co., Inc., the PVD market is expanding. They argue that PVD equipment sales will see an average annual growth rate (AAGR) of 11.3% to reach $6.7 billion in sales by 2005, PVD deposited materials will see an AAGR of 11.9% to reach $943.9 million by 2005, and that PVD services will see an AAGR of 9.3% to reach $1.3

billion in sales by 2005[3] (see Figure 1-2). One can infer that an increase in the customer market will translate into an increase in the production market.

1.3 The Problem

This thesis focuses on the creation of a framework for developing modules that simulate reactor-scale aspects of magnetron sputtering processes. This framework will enable future engineers and scientists to develop independently and to integrate easily the process specific modules within this framework. Ultimately, the framework would enable rapid prototyping of physical process modeling simulations. This framework will be named ***SIMSPUDII***.

SIMSPUDII will address the academic and commercial need for a comprehensive PVD simulation package. This package will provide a framework for developing a reactor scale simulator, and eventually combine with a feature scale simulator, to create a comprehensive sputter deposition simulator. In particular, the industry lacks a simulator capable of simulating the plasma, the target, and transport through the gas with rarefaction and heating, as well as the film growth while considering the effects of all of these. SIMSPUDII will be designed to be flexible, extensible and sufficiently efficient to execute on a desktop workstation. The overall project will consist of several phases of development: each will focus on a different aspect of the process, but all aspects will couple together and ultimately give a self-consistent and comprehensive picture of sputter deposition. Each module will be designed to stand alone in certain appropriate regimes.

However, the integration of all current and future development will give the proposed model its unique and greatest capability.

1.4 Simulations, Commercial and Academic

In response to the need to better understand the sputtering process, a variety of simulators have been developed by both industry and academia. With the multitude of variables in the PVD process to tweak and the wide range of possible film variance that can occur given unique simulation parameters, the industry has become reliant on detailed computer modeling of PVD processes. Several products have entered the market to help service the demand for PVD process simulation. For example, SIMBAD (a feature scale simulator) and SIMSPUD (a reactor scale simulator) are two products developed and commercialized by research engineers at the University of Alberta. The commercialization and research successes of SIMSPUD/SIMBAD in the mid to late 90's are well-documented[4 5 6 et al]. The software reached over $2M in sales in fifteen countries. The software package's success could be attributed to the fact that SIMBAD integrated both a feature scale simulator and a reactor scale simulator, enabling the effects of reactor variables on feature-scale properties to be evaluated. Feature scale simulations are typically conducted using simplified models of reactor scale environments. SIMSPUD has the ability to create more realistic representations of the reactor scale variables. This enables SIMBAD to simulate more accurately the feature scale phenomenon. It could be argued that most feature scale simulations would benefit from a versatile reactor scale simulator.

The main customer of SIMSPUD/SIMBAD was the PVD industry. With customer applications becoming more demanding, there is a need for more sophisticated modeling capabilities.

SIMSPUD/SIMBAD were designed as research tools; commercialization happened after the fact. The software began to outgrow its research roots. The code, although valid, was originally designed to model specific processes. Once the software was commercialized, the need to extend the software to encompass various customers' unique apparatus and processes became overwhelming. The software rapidly became an unmanageable mess of complex interdependent code. By the late 90's the proprietors decided that it was no longer financially viable to continue to struggle with the software; the technology was thus licensed to a third party: Reaction Design Incorporated. In hindsight, the software packages could have been designed for extensibility and commercialization. One might argue that the life span for SIMSPUD/SIMBAD could have been extended considerably.

Reaction Design Incorporated has encompassed the SIMSPUD model in their PVD PRO suite. This suite is a replacement of the SIMSPUD/SIMBAD product set. It is graphical user interface (GUI) driven and provides reactor-to-feature scale simulations. Due to the commercial nature of the product, the proprietary enhancements to the reactor scale model are not clearly identified in their literature. One could argue that with the complexity of the reactor scale simulation space, their model is not all encompassing. Their feature scale product can only utilize the output of user-defined reactor scale output.

When Reaction Design licensed the SIMSPUD/SIMBAD products, they did not hire the support or research staff that created the product. Thus one could argue that the life span of their products will fall behind technologically without key research talent. It should be noted that the author of SIMSPUD leads the research group working on this project. The author of this thesis was one of the support and enhancement engineers on SIMSPUD/SIMBAD during its time at the University of Alberta.

EVOLVE, another feature scale simulator, developed in part by Thomas Cale and marketed by Process Evolution Ltd, has received some recent commercial interest. This simulator relies on the user to provide the reactor scale information in order for it to compute the feature scale results. This simulator has been used to simulate low-pressure deposition and etch processes, low-pressure chemical vapor deposition (LPCVD), plasma enhanced CVD (PECVD), and plasma etching.[7] This is another example of a product that would benefit from an encompassing reactor-scale simulator.

Many other research groups have also spent time simulating PVD processes. Typically these simulations, like the classic SIMSPUD model, involve some form of Monte Carlo process. A great deal of work has been invested at the feature-scale level compared to the reactor-scale. For example, O'Sulliven, Baumann and Gilmer have performed feature scale simulations[8]. They correctly identify the need for investigation into high aspect ratio trenches and vias. Although they also mention the need for more realistic and

encompassing models of reactor scale simulation, their work focuses on topographic evolution.

McCoy and Deng present a parallel simulation of sputter deposition.[9] Although McCoy and Deng present a novel approach to feature scale sputter simulation, they do not clearly indicate how they obtained or created their reactor scale derived angular distributions. McCoy and Deng mention that they use randomly generated angular and energy distributions, or distributions 'from a file'. Clearly, their work would benefit from a detailed reactor scale simulation model.

This was not the only academic instance of PVD simulation. Suzuki and Hoshi explore the effect that varying gas pressure has on the counts of various species traveling from the target to the substrate by using a reactor-scale simulator[10]. They assumed a homogeneous gas pressure for their calculations. Their work could benefit from a more detailed representation of the background gas. Lu, Wang and Wu explore the effects of reactive sputtering with constant voltage as opposed to constant current power supplies[11]. However, details of the extensibility or reusability of their simulation product are not evident in the paper. Pyka, Selberherr and Sukharev have studied the effects of reactor scale variables on feature scale simulations[12]. Arguments have been made for the need of advanced manufacturing processes that involve sputter deposited thin films[13].

As shown, there is considerable PVD research conducted on both the feature scale and some on the reactor scale. Simulators are available that model some aspects of the reactor

scale PVD process, yet none that exist (nor soon to exist) encompass the complete reactor scale picture. Feature scale simulations rely on reactor scale simulation results. Research into reactor scale effects on the PVD process will continue into the future. Due to the complexity of the sputtering process, one could argue that no simulator will provide the complete reactor scale PVD picture. There seems to be a need for a range of applications, or an application framework, to address the research and commercial needs of PVD process investigation.

For this reason, the first goal of this thesis is to develop and present an application framework for the development of the SIMSPUDII simulator. This framework will employ current software engineering methodologies to ensure its flexibility and extensibility to ensure the long-term viability of SIMSPUDII.

The second goal of this thesis is to apply this framework to develop two key modules within the SIMSPUDII framework, namely the module responsible for accounting for generic and user specifiable chamber configuration and a module for tracking sputtered particles and other energetic neutrals within that chamber. While the full SIMSPUDII functionality will not be complete until several other modules are added (an area which is beyond the scope of this thesis), this subset is sufficient to provide useful new information on the spatial energy deposition distribution by neutrals within the sputter gas and onto the film substrate during deposition.

One development requirement, however, is that the product must be highly extensible. This was a major limitation of the earlier SIMSPUD/SIMBAD, which became progressively less manageable as each new feature was added to help it address new technical challenges.

1.5 Frameworks

A prominent goal of software development is reuse, and by association, extensibility. Reuse can be observed on many levels of abstraction, from a single method implementation, to complete modules, all the way up to abstract system designs. Software is difficult to design and harder to maintain. The more code that can be reused, the less that needs to be re-designed and maintained. A brief introduction of the concept and utility of an application framework is necessary here.

Lajoie and Keller define three levels of reuse:[14]

- Reuse in the large
- Reuse in the medium
- Reuse in the small

This describes the range of software reuse that range from application reuse down to class/method reuse. Reuse in the large is the fundamental goal of the SIMSPUDII design. This type of reuse can be achieved through the development of an application framework. Froehlich, Hoover, Liu and Sorenson state that " [a]n application framework provides a

generic design with a given domain and a reusable implementation of that design"[15].
They also mention that the primary reason for building a framework, as opposed to an application, is reuse: reusing not just the implementation of the system, but the design as well[16].

Before building a framework, one must consider if an application framework is more appropriate than just creating an application. Froehlich et el[15] have mentioned that framework design is considerably more costly than that of an application. They also mention the possibility of the domain shifting, thus making the framework obsolete.

The domain for this proposed framework is not only clearly defined (reactor scale PVD simulators); it has been a relatively stable field of study for the past few decades. Although the apparatus used in production have been evolving, the fundamental physical processes have not changed. The motivation for a framework as opposed to an application for this domain is not the shifting of the problem domain; it is the domain's complexity. The number of variables in a PVD simulation makes the physical process extremely attractive to the microelectronics industry, and makes the number of unknowns worthy of continued research.

1.6 Existing Frameworks

Before setting out to design and implement a new domain specific framework, it is important to consider what frameworks are presently available. There are several frameworks in development for the use of simulating physical phenomenon. The one framework that is most relevant to the work discussed in this document is POOMA[18].

The Advanced Computing Laboratory at Los Alamos has designed a framework to address the need for high performance parallel computing and numerical simulation. This Framework is known as The Parallel Object Oriented Methods and Applications (POOMA) framework. With the halting of nuclear weapons testing, there has been a need to provide highly detailed particle simulations in order for the US to fulfill its mission of Science-based Stockpile Stewardship; POOMA was constructed to address this and related needs[17].

POOMA utilizes the strengths of the C++ language (generic programming, Object Oriented design) to provide a rapid development environment for scientific programming. It provides many of the commonly used scientific algorithms (Fourier transforms, interpolators, etc.), as well as some data structures common to scientific programming (Fields, Particles, Mesh types, etc).

The main goals of the POOMA project are clearly stated on their web site[18]:

- Code portability across serial, distributed, and parallel architectures with no change to source code

- Development of reusable, cross-problem- domain components to enable rapid application development
- Code efficiency for kernels and components relevant to scientific simulation
- Framework design and development driven by applications from a diverse set of scientific problem domains
- Shorter time from problem inception to working parallel simulations

One of the key strengths of the POOMA framework is its use of generic programming and data-parallel expressions using expression templates. The developers of POOMA were able to leverage the strengths of generic programming and C++'s template facility to create an intuitive abstraction of data parallelism.

If the algorithms used for scientific programming exhibit parallelizable characteristics, clearly POOMA would be worth investigating. If parallelism is not easily achieved, one may argue that the overhead and added complexity of a framework such as POOMA may be too high. Another issue to be addressed would be the licensing of POOMA. If the project developed within the POOMA framework were to be commercialized, one would need to know what the costs and limitations associated with utilizing the POOMA framework would be.

1.7 SIMSPUDII Framework Features

The ideal framework would require several key features that would enable future research scientists to develop and test physical processes. The ideal framework would allow a scientist to seamlessly integrate a module that models a specific, reactor-scale physical process related to magnetron sputtering.

Proprietary Technology

Commercialization of research results is increasingly an expectation at Canadian universities. Following the success of SIMSPUD/SIMBAD, one of the fundamental goals of this project is to produce a technology that can be commercialized with all intellectual rights maintained within the research group. If the framework can be developed using little to no externally owned intellectual property, the product can be commercialized with no external licensing constraints.

Grids

This thesis proposes to utilize adaptable grids to discretize the chamber volume. This discretization of the problem space approach is not new. The use of a grid encapsulates changing physical properties into cells that can be individually treated as homogeneous. Properties such as magnetic fields or gas density can be computed and represented by these grid structures. Higher resolution can be obtained by decreasing the volume of each grid cell. Thus increasing the number of grid cells can obtain a more accurate picture of a

grid-represented physical property. However, this increases the memory and execution time requirements of the application.

This approach works well for simulation properties where the property evolution throughout the simulation space changes gradually; however, for properties that have a few dramatic shifts in value in localized spaces, a coarse granularity grid cannot capture the details of the shift. A fine granularity grid would be an extreme waste of resources if the majority of the discretized property were well behaved throughout the rest of the simulation's space. We propose using an *adaptable* grid for simulation discretization. Thus, for example, a magnetic field in a portion of the simulation volume that has few field lines can be represented by large cells, and the portions of the simulation volume that have many field lines can be subdivided. Adaptable grids have been used in mathematical differential equation solvers[19], but use of this type of structure is not widespread in the PVD simulation field. The proposed simulation framework would benefit greatly if such a data structure could be incorporated.

Documentation

The amount of time for an engineer to learn to develop within this proposed framework must be considerably shorter than the time to re-develop this framework. This implies that the framework must provide a human interface to the framework that allows for rapid comprehension of its facilities. The framework must offer several levels of documentation that provide the engineer with the means to interact quickly with the framework and utilize the desired subset of the entire framework.

Extensibility

The framework must be extensible. If the current feature set provided by the tool is considered incomplete by some future standards, the framework must be easily modifiable to encapsulate the desired feature set. The time required to extend the feature set must be considerably shorter than the time required to rebuild the product. This implies the tool must be designed with loosely coupled, highly cohesive, replaceable modules. The ultimate design would provide engineers with not only a framework to build and extend, but also a set of reusable modules that encapsulate specific attributes of the magnetron sputter process.

By observing the software life cycle of SIMSPUD/SIMBAD, one can see that, initially, graduate students interested in exploring some aspects of the sputter deposition process will utilize the proposed framework. One could argue that most graduate students working in this field will not be trained in large-scale software development. Thus the framework must be *easily* extendable with little software development skills.

Later on in the life cycle, the framework could conceivably be commercialized. This prospect implies that the extensions made by graduate students must not restrict the end product. The framework should provide a 'sand-box' for the beginning developer and a fully extensible framework for the more advanced developer.

Optimized and Efficient Runtime Behavior

The framework will inevitably be used to develop numerically intensive algorithms that could potentially run for millions of iterations before converging on a solution space. This characteristic implies that the framework must provide all services with as little overhead as efficiently as possible.

Input/Output Facilities

The proposed framework would have to provide the research engineer with an abstraction of the many attributes of a complex software package. A system for abstracting some of the frequent tasks, such as file opening, default directory specification, information forwarding etc., would benefit a research developer. This system would standardize the message-routing facilities for the entire framework.

Obstruction Processor

For full flexibility in modeling physical processes, the proposed framework will need a dedicated module for handling the physical structures necessary for unique sputtering apparatus.

The minimum solution would require basic modeling of a cylindrical chamber and a physical representation of a target and substrate, similar to SIMSPUD's physical model. The ultimate model would have the capability of specifying complex 3-D structures, easily allowing the user to set up location specific data collectors. The ultimate model would also allow the user to easily specify surface specific runtime parameters, allowing

for specification of multiple targets, positioned within the chamber to provide the ultimate flexibility. The ultimate model would also provide the ability to visualize the physical layout of the configured structures.

User Interface (not necessarily graphical)

If the ultimate model were to be successfully implemented, a creative user interface would be required to enable the end user to configure intuitively the physical parameters of the simulation.

For physical structures (targets, substrates, clamps etc.) to be properly modeled in the simulation environment, the structures must be able to interact physically with moving particles. Due to the nature of a Monte Carlo simulation, millions of particle-surface interactions take place. The efficient handling of these interactions is imperative.

Extended SIMSPUD model (verification and validation)

One of the goals, as mentioned, is to provide an extended version of the SIMSPUD product's functionality. Completing this phase of the project will not only provide future research scientists with the first part of an extensible reactor scale PVD simulator, but will also provide an application development cycle that will test the viability and extensibility of the proposed framework.

SIMSPUD is a reactor scale PVD simulator that provides distribution data that could be used in feature scale modeling. The product incorporated several assumptions, as all

simulations must. Some assumptions that the original SIMSPUD model made are as follows:

1) The particle transport occurs through a homogeneous gas medium (significant inhomogeneities in gas temperature and pressure have been observed)[20]
2) There are limited physical shadowing effects due to the physical chamber apparatus
3) There is no gas heating by particle collisions
4) Particle scattering is based on a hard sphere collision model with an empirical (not physically based) energy dependence

These assumptions are just a few of the total assumption set that SIMSPUD used to create sputter distributions. Like any simulation, if any members of the assumption set can be replaced with more accurate or realistic models, the simulation comes closer to modeling real world phenomenon.

If the subset of assumptions that has been listed could be improved, the framework would have to complete the first phase of a prospective encompassing framework of PVD simulation models and applications.

The proposed model will thus require:

1) *Inhomogeneous gas population*: If a runtime configurable set of gas species could be introduced and redistribute as a result of the physical process, the overall

particle transport model would benefit from a more realistic representation of the physical phenomenon.

2) *More realistic physical shadowing effects due to chamber apparatus:* This requirement is fulfilled if the Obstruction Processor (as mentioned) can be successfully implemented.

3) *Gas species tracking*: The gas particles that interact with the target species during a collision event often arrive at the substrate. After a collision event, energy transfer effects should be considered. The heating effects on the background gas could affect the progression of the transport model. This extension to the SIMSPUD model would, as well, create a more realistic simulation.

4) *A more flexible and physically realistic particle interaction model.* Although the hard-sphere model provided reasonable results, the effects of long-range particle interactions would provide a more realistic picture.

This enhanced SIMSPUD model will be called Thermspud1.

1.8 Thesis Organization

The remainder of this work will be organized as follows: Chapters 2 and 3 deal with the software architectural domain of the work, a high-level view independent of the problem domain. Chapter 2 will provide a discussion of frameworks and the central enabling technologies used in the development of this work. It will provide the necessary background to explain the basis of the rest of the work. Chapter 3 examines the design and implementation of the developed framework. It provides a guide for the future framework developer to understand the fundamentals of the product as a software system. Chapters 4 and 5 deal with the ways that the work addresses the problem domain. Specifically, Chapter 4 addresses the low-level implementation details of the problem domain aspects of the software package. Chapter 5 presents tabulated results directly related to the problem domain using the software package. Chapter 6 summarizes the conclusions and provides some forward-looking statements regarding the overall project.

2. Architectural Domain: Frameworks and Technologies

2.1 Frameworks

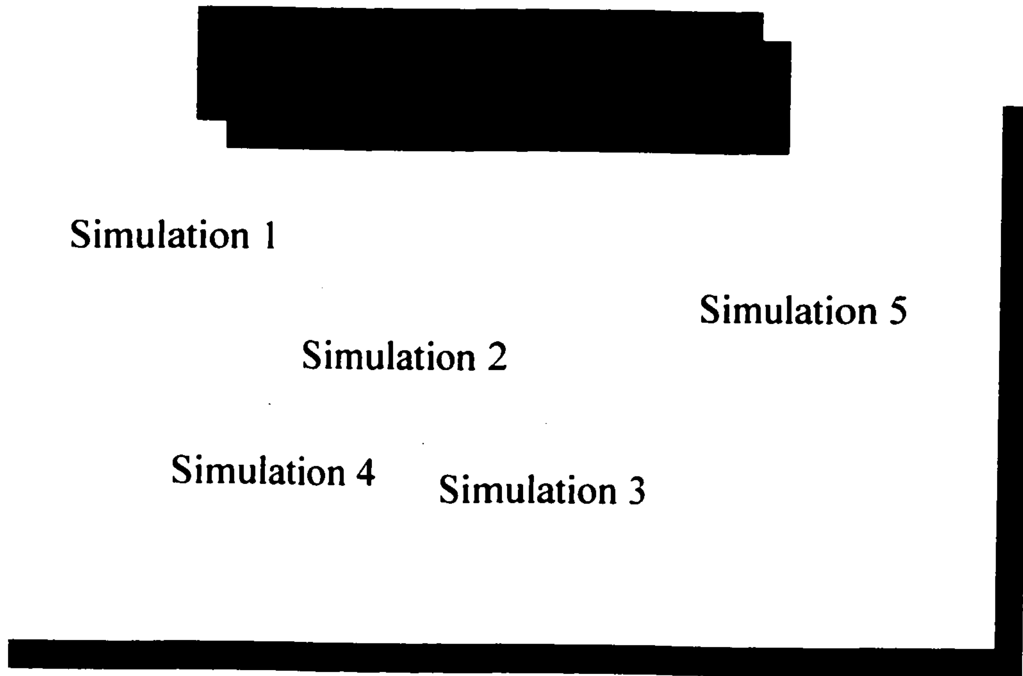


Figure 2-1: Problem domain coverage

Several independent simulations have been built. One could argue that there are common aspects and approaches within some applications built within the problem domain of "Reactor Scale PVD Simulations"

SIMSPUD/SIMBAD were initially built as research tools and later evolved to commercial applications. During their lifetime as commercial applications, several extensions were created either to increase the scope of the simulations or to create proprietary versions that depicted a specific customer's apparatus. A core of the applications remained untouched and common to all of the extensions. For many of the extensions, it was required, by design, to recompile the source code conditionally to 'enable' or build a specific version of the product. This was inconvenient, very difficult to maintain, and was almost impossible to provide a client with all functionality of the product.

It would be beneficial to learn from the mistakes or miscalculations in the design process of SIMSPUD/SIMBAD. Re-use and extensibility are key design pillars that must be carefully scrutinized in order to extend the application life cycle. Looking at SIMSPUD/SIMBAD, we see that several applications were created, all in an attempt to cover the problem domain.

The problem domain addressed in this project is the reactor scale portion of the PVD simulation spectrum. Looking at Figure 2-1 we can see that each of the SIMSPUD iterations, or even other academic or commercial applications, is an attempt to cover the entire problem domain. One could surmise that several applications have common aspects that have been re-used or could have been re-used across the spectrum to create an all-encompassing solution for the problem domain. Although the application would never fully cover the problem domain, with proper design the application would be extensible and the process increasing the scope of the application would be less than rebuilding each smaller version. Thus this paper argues for creating an application framework.

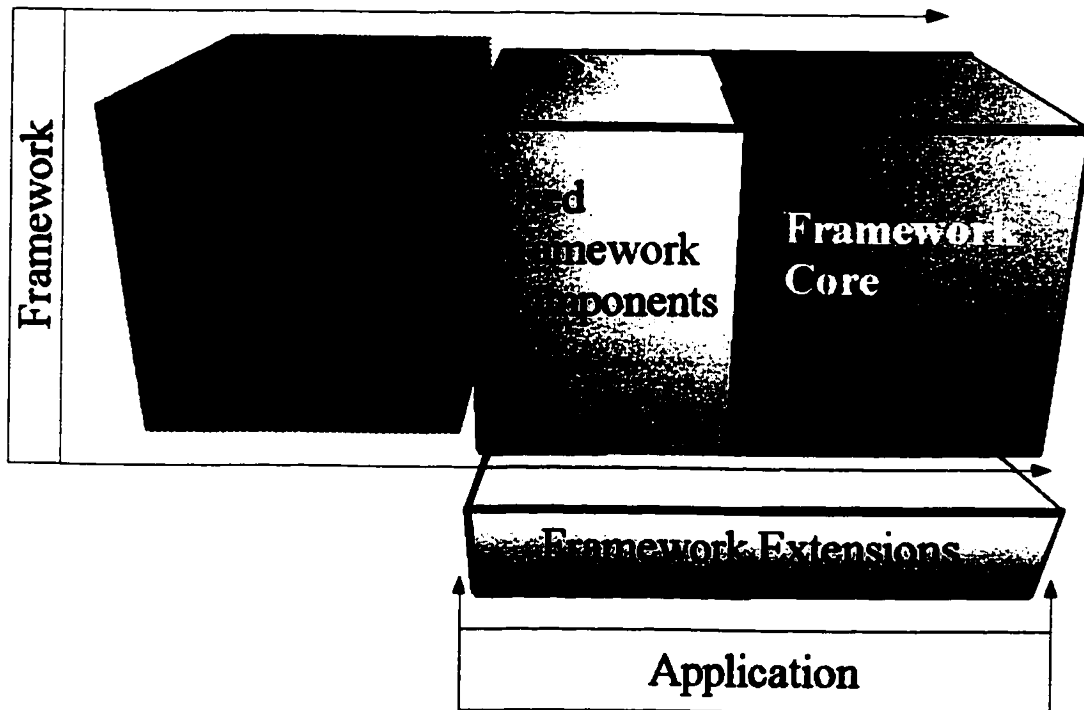


Figure 2-2: SIMSPUDII framework

Applications are created by utilizing a core architecture, a subset of the framework components, and possibly external framework extensions.

Froehlich et al[15] state that a framework provides a generic solution to a set of similar or related problems, but that an application provides a specific solution to one problem within the problem domain. This second distinction typifies the solution for the extensibility and re-use issue that this project seeks to provide for reactor-scale PVD simulations. It could be argued that the problem domain has an infinite solution set of applications. Conceivably, a robust, well-designed framework would cover a large subset of the solution set of applications in both the commercial and academic circles. The framework will never be a finished product, but always a product in development.

The first step of framework design is the clear definition of the problem domain and the set of applications that the end framework will encompass. This project, as mentioned,

has a clear problem domain specification: reactor scale PVD simulators. The domain is non-shifting, and thus a reliable candidate for a framework.

The ideal framework architecture for this problem domain would have a core reusable foundation that would bind together the problem specific application. This core would enable the end user of the framework to customize the application by choosing a set of pre-made framework specific components. The expert user may even be in a position to provide some framework extensions that may solve the user's specific simulation needs (see Figure 2-2). This type of architecture has been referred to as a "plug-in" or "hooked" architecture.

Froehlich et al have discussed some attributes of a hook framework[21]. Their discussion introduced the notion of a hook as a means of easing the understanding and use of a framework. By this they mean that a framework could provide a means of easily adding new functionality through a series of well defined and documented 'insertion locations'. These hooks would inform the end user how and where the design could be extended. There has been much discussion on design patterns and their relationship with the creation and description of frameworks. Martin provides an excellent discussion of framework design guidelines and patterns[25].

There are several promising attributes of the proposed SIMSPUDII framework. First there are many academic discussions promoting the desirable attributes of a successful framework. The study of framework development has been the focus of computing

scientists for over a decade with some of the research conducted here at the University of Alberta[15]. Second, the problem domain has been the focus of study for decades at the University of Alberta. A requirement of framework development is access to a domain expert. The University of Alberta is renowned for its internationally leading thin film research. Finally, the post-mortem analysis of SIMSPUD has provided invaluable information about the creation of applications for the stated problem domain.

2.2 Key Enabling Technologies

C++

Choosing a language for a software problem is a very important aspect of the design process. To meet software design goals, extensibility is an important aspect of the project. Most scientific simulation applications have been developed in Fortran or C. SIMBAD was originally written in FORTRAN (late 80's), eventually ported to C (90's), and subsequently ported back to FORTRAN (2000).

It has been argued that on large-scale projects, an object-oriented (OO) approach to software development can offer extensibility benefits[22]. The three tenets of OO design are encapsulation, inheritance, and polymorphism; all three are pillars of code re-use. Encapsulation is a "modeling and implementation technique that separates external aspects of the object from the internal, implementation details."[23] Encapsulation enables extensibility by enforcing locality of change to the scope of the changed objects. Inheritance allows existing objects to be extended without changing the original working

code or the context. Polymorphism (dynamic binding) allows tested algorithms, used with the object, to be reused with the child class implementations without retesting the algorithm. These features combine to provide programming constructs that are aligned with framework design and code extensibility.

OO techniques represent entities in the problem domain with objects in code. This abstraction can aid in program conceptualization and design communication between developers. One of the benefits of choosing OO is the ability to use computer-aided software engineering (CASE) in the design and extension process. Many books and papers have been written extolling the virtues of OO design methodologies. [24,25, et al.] SIMBAD/SIMSPUD had a reduced lifecycle due to extensibility issues that could be related, in part, by procedural based programming. For these reasons, an object-oriented approach was adopted for this SIMSPUDII development.

The only languages that offered the necessary speed to develop numerically intensive scientific applications and object technologies were C++ and Fortran 90. Fortran 90, although a language with OO features, lacked templates and the standard template library (STL), features that are predominant in generic programming. Fortran 90 has been used in supercomputing applications, but does not have the extensive user base of C++[26]. Choosing C++ had some major disadvantages. Not everyone developing on the product was a strong C++ programmer. It was argued early on (1999) that the overhead for using C++ would be a major hurdle. In the end, the research group chose C++. It was felt that

the trend in the scientific computing community was moving away from Fortran and towards C++.

XML

One of the big issues to be addressed in the initial design was the large body of simulation parameters required to run a product such as this. SIMSPUD/SIMBAD used text files to input the simulation parameters. In early drafts of the software, the author considered a regular expression engine such as FLEX. Text-based parameter input using FLEX had advantages. To begin with, the parsing could be done with minimal coding effort. FLEX has a very simple programming model and, for those familiar with regular expressions, is easy to use. However, some questions need to be answered for text file parameter input. What files would contain which variables? What is a good naming convention? How extensible is this form of parameter input?

One problem for SIMSPUD/SIMBAD was most developers' concerns that work on the two products to add new parameters to the list of control data were very cumbersome. Often, input variables were 're-used' to avoid the labor of adding new variables to the input system. This recycling led to instances where variables were inadvertently not set in the intended manner, which was extremely undesirable.

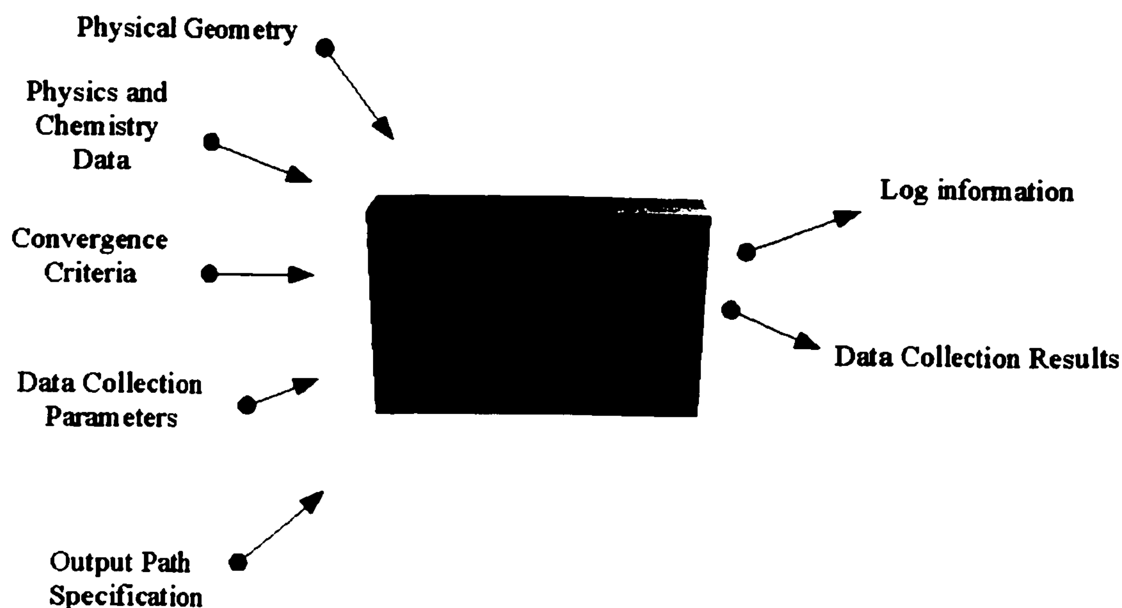


Figure 2-3: Simulation parameter space

The simulation requires the input of many parameters and the output of log and collected data.

One aspect to be addressed is how easy would it be to extend the parameter solution to a graphical user interface. If this tool's commercialization were to be considered, a GUI environment would be inevitable.

XML seemed to be the best choice. XML (extensible markup language) is a text-based data storage file standard. This standard is rapidly growing in popularity. Tools are being developed, as well as many libraries for handling XML documents. Extensible graphical user interfaces are available for handling the XML files.

XML allows for a very organized tree-like data storage within a text file. This permits module specific data parameters to be grouped intuitively. Particle species information is grouped in a ParticleSpecies node. Surface-Specific data can be attached directly to the surface node related to the data. The software system provides the ability for the user to

specify complex chamber geometries. Each surface in the chamber can have data collection regions attached and oriented. With XML, a very complex simulation parameter space can be defined and modified easily.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Simulation SYSTEM "dtd/Simulation.dtd">
<Simulation Version="0.3">
     <ExecutionQueue>
          <ExecutionTag IDREF="therm1"/>
     </ExecutionQueue>
     </SimulationRuntime>
     <SimulationPhysicalProperties initialTemperature="275">
          <GridDimensions k="2" j="2" i="2"/>
          <VolumeParticleManager ID="volumeParticleBucket"/>
          <VolumeParticleManager ID="hotChargedBucket"/>
          <SheathEntry ID="sheath"/>
     </SimulationPhysicalProperties>
</Simulation>
```

Figure 2-4: Example of XML format

This is a very simple example of the format in which XML is stored within a file.

There are open-source parser libraries for XML. The regular expression specification of text-file parsing is abstracted away from the problem using these libraries. Adding new parameters to the simulation is very simple. For the developer adding new parameters to the simulation space, a guide for working with XML was developed. There are also open source editors that provide for easily modifying the XML files.

The XML specification provides a mechanism for validation of XML files. Two files are required for the system parameters. One is a 'DTD' file that specifies a template for validation of the syntax of the actual XML simulation parameter file. The DTD file provides a rule set that the parameter file must follow to be considered 'valid'. The XML parsing library used with this project provides detailed feedback when an invalid XML file process is attempted. This is an excellent mechanism for ensuring valid input

parameters. It should be noted that the DTD only specifies the syntax of XML file; the parameter ranges and combinations of variables are left up to the developer to verify.

Various open source XML editing tools exist. These provide a GUI for editing the XML file that is very intuitive and easy to understand. The tool chosen for editing the XML files is named Merlot. Having a 'free' GUI with no extra development resources saved a great deal of time. The tool could be extended in the future to provide SIMSPUDII specific features.

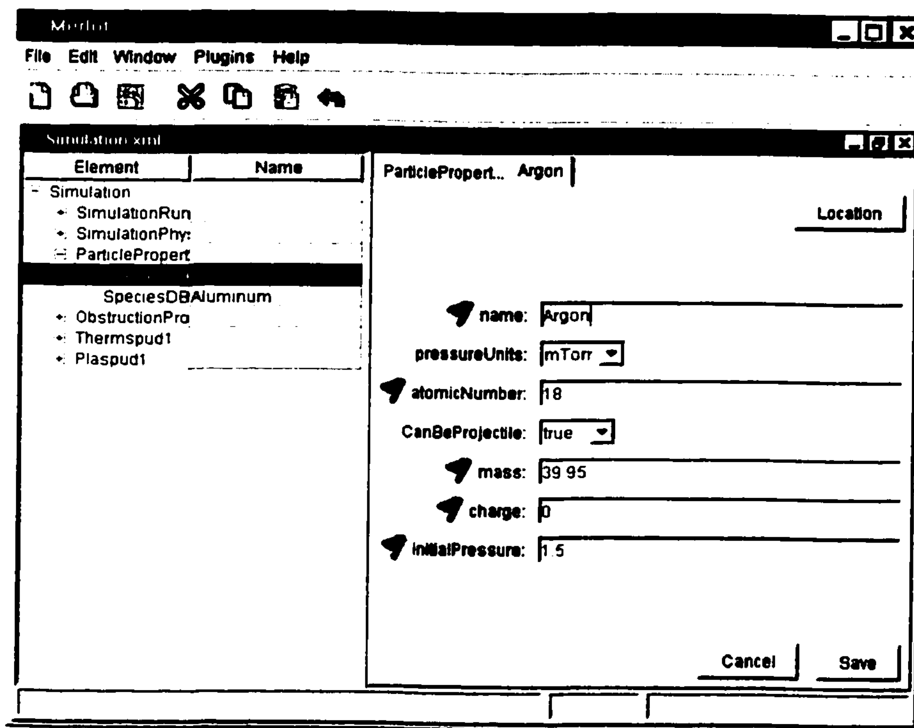


Figure 2-5: Merlot, an open source XML editor

The availability of open-source XML editors greatly simplifies the manipulation of XML files.

For the actual design, it was decided that a single developer or team of developers would develop each module. The XML library parses the XML file and creates a memory representation of the data called a DOM_Document. Instead of localizing the parameter parsing to one module, the individual modules are to be responsible for extracting the information from the DOM_Document. The process of 'building' a simulation module from an XML file would require an intimate knowledge of the details of the module under construction. Thus each module would have its own DOM_Document processor. During the initialization phase of the simulation (discussed later) the parsed DOM_Document is passed to each module. The module's DOM_Document processor would then extract the parameters and 'build' itself (see Figure 2-6 and Figure 3-2).

In practice this has proven very useful. Due to the ease of adding new simulation parameters in an organized fashion, the number of parameters that can be adjusted for a given simulation has grown rapidly. Instead of a lengthy process, a few lines of code and a modification to the DTD file is all that is required to add a new process parameter. This has sped up the development process a great deal. For a large software package, compile time is a big issue. If it was difficult to add a new simulation parameter, the developer would often just 'hard-code' a value. This requires a rebuild for each change in this hard-coded value. Not only is this time-consuming, it leads to unexplained 'magic-numbers' spread throughout the code.

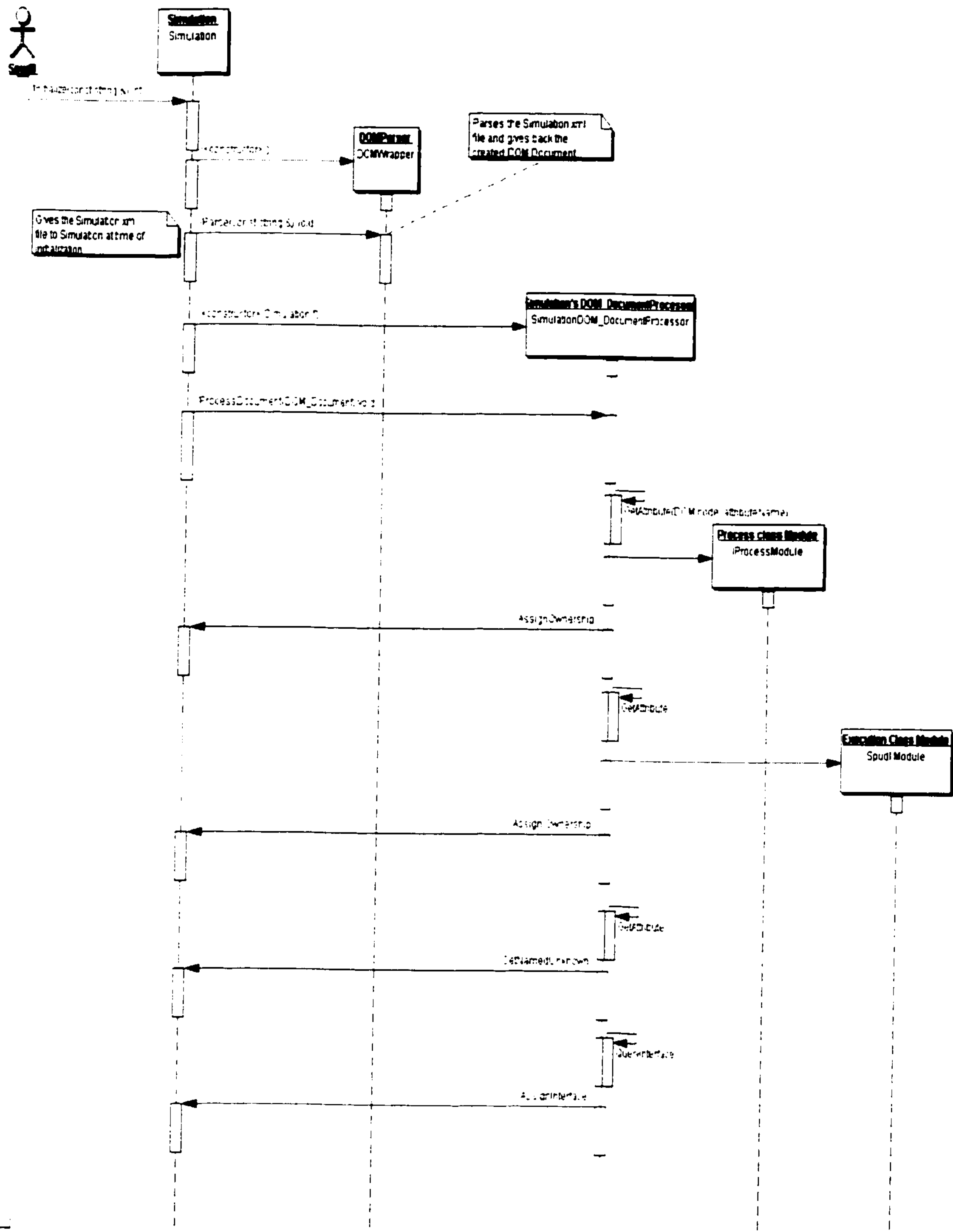


Figure 2-6: Module creation using XML

This depicts the sequence of events that occur as the simulation interacts with the XML file.

3. Architectural Domain: Framework Analysis

3.1 The SIMSPUDII Framework

The goal of this project was to create an extensible framework for future research. The services that such a framework must perform have been encapsulated within a Simulation object. The Simulation object creates an architecture within which all modules can operate. Basic services that this framework provide are as follows:

- Module Initialization
- Simulation object classification and management
- Runtime message logging and output
- Module Execution
- Data Output and Visualization
- Module Destruction

The project goals were to produce a set of loosely coupled modules tied together by this single Simulation object. This top-level Simulation object handles all actions of adding, removing and referencing simulation-created objects and simulation properties. The Simulation object provides the separate execution class modules with a snapshot of the current state of the system. as well as a database of system specific properties and objects.

The original design was to have individual modules having little to no interaction with each other. All interactions could be considered as acting on the system or extracting data from the current system status. This design would aid in an overall parallel development process, as well as provide a common extendable framework for future work. The definition of the simulation framework could be extended for future development with no side effects to the previously developed modules. Individual 'execution class' modules are developed as completely independent specialized processes that act upon the Simulation object. Any object that needs to be accessed by multiple modules would be built and destroyed by the Simulation object.

The author adhered to this initial design for the simulation for the most part. Some modules required specific communication with other modules. To maintain the original design premise, the author devised an object classification system. Three levels of object classification were employed:

- ExecutionModule
- ProcessModule
- Low level objects

ExecutionModule is the most significant object, next to the Simulation object. Any module that inherits SpudIIModule is considered an 'execution class' module. This module will have the ability to control an execution phase in the simulation. An ExecutionModule has some ownership responsibilities of non-ExecutionModule objects.

It is assumed that an ExecutionModule will need a DOM_Document for initialization, and have some form of output routines. The abstract base class SpudIIModule is extended to make a module an ExecutionModule. SpudIIModule is an abstract base class that forces children classes to implement Initialization, Execution, and Output methods.

The next level of object classification is the ***ProcessModule*** classification. These objects provide a service to ExecutionModules. ProcessModules do not have an execution phase. They provide the service of processing data upon request to any client object (usually an ExecutionModule). The Simulation, if more than one ExecutionModule requires access to one of the ProcessModule interfaces, creates these ProcessModules. If a designer deems that only one ExecutionModule will access a ProcessModule, the module would typically be created, initialized, and destroyed by that ExecutionModule. The object responsible for creating, initializing, and destroying a ProcessModule is responsible for extracting the XML details required for initializing the ProcessModule.

The ownership concept is mirrored in the XML structure. If an ExecutionModule has sole access to a ProcessModule, then that ExecutionModule is responsible for initialization of the ProcessModule. Thus the ExecutionModule node would appear above an owned ProcessModule node in the XML structure. If in the future other ExecutionModules need the encapsulated ProcessModule, either through an external reduced interface or as a full-featured object, the initialization phase will be moved from the ExecutionModule, where it currently resides, up to the Simulation initialization method (see Figure 3-2). This goal

is achieved by moving the relevant section of the XML DTD file into the appropriate place thus reorganizing the XML structure.

All ExecutionModules and ProcessModules have a *unique* identifier. They all must implement the IUnknown interface to enable this requirement. The IUnknown interface provides the services of ID tracking and interface management. The author has labeled this design the '*T-COM*' (Tagged Component Object Model). The XML DTD specification file guarantees the unique identifier property. This guarantee played a big role in the final design (as will be discussed).

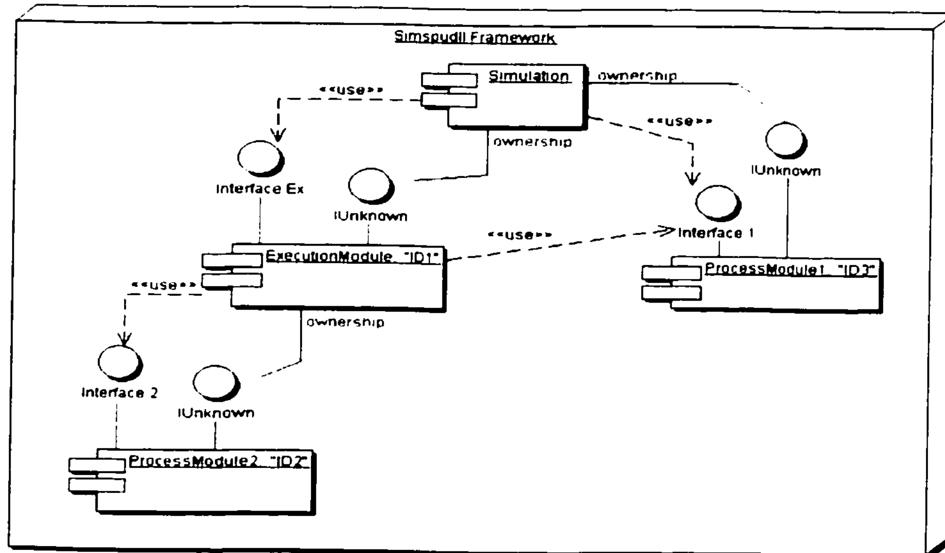


Figure 3-1: SIMSPUDII framework basics using the T-COM

SIMSPUDII is an assembly of execution modules and process modules. All interaction is done via collections of methods grouped by interface definitions. The IUnknown interface manages module ownership.

An example of this concept is the ObstructionProcessor, which is a ProcessModule. All ExecutionModules require its services. The Simulation creates, initializes, and destroys the ObstructionProcessor. Alternatively, the ZBLCalculator is a ProcessModule that is

only needed by Thermspud1. Thus Thermspud1 creates, initializes, and destroys the ZBLCalculator. All ProcessModules can be referenced through their unique identifier.

Looking at Figure 3-1, we see the basics of the two module classifications discussed so far. Note in the Figure that all objects, as mentioned, implement the IUnknown interface. The Simulation object collects a mapping of all IUnknown objects that it has ownership responsibility for. This mapping is used for memory management and interface referencing. The Simulation 'owns' all ExecutionModules and any 'global' ProcessModules. The ExecutionModules own 'private' ProcessModules, which, similar to the Simulation, are bound to the ExecutionModule through the IUnknown interface and used with a series of one or more behavioral interfaces (Interface 2 in the diagram).

Note in the diagram that the Simulation requires an interface (Interface 1 in the diagram) in order to function properly. This interface is attached during initialization. All dashed links in the diagram are set up via the XML file and the unique id references. All solid links refer to 'ownership' of the IUnknown interface.

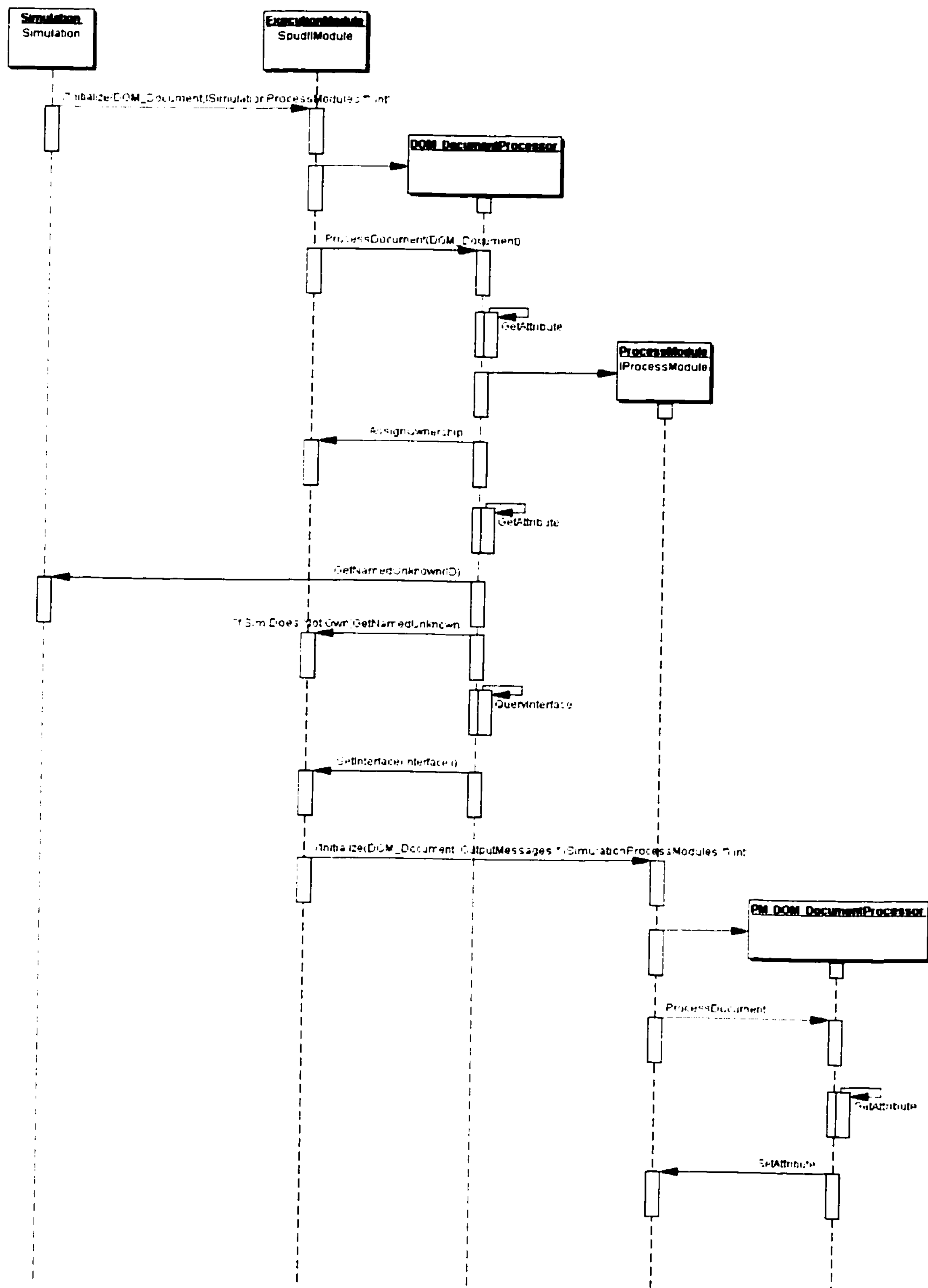


Figure 3-2: Module initialization using XML

This figure maps a detailed view of the sequence of events that occur during initialization of an ExecutionModule. The ExecutionModule owned ProcessModules are created using the XML specified parameter lists (see XML discussion on page 34)

The true power of the implemented design lies in the combination of object oriented programming and XML. As mentioned, a framework is an embodiment of a problem domain set of applications. By shifting the scope of a problem-specific framework, the end user can create different **applications**. Typically, one assembles a set of objects and libraries together to create an application that covers a subset of the problem domain. Using XML and the aforementioned T-COM, one can fully specify an application within the framework developed for this project **at run time.** One builds an application by specifying in the XML file the modules to be created and the interaction needs between the modules. The Simulation application is built **at run time** without the need for recompilation. Only those modules that are specified in the XML file are loaded into memory. Thus a user can build a problem-specific application just by customizing the XML file. Module interactions/dependencies are specified through the unique identifier assigned to each ExecutionModule and ProcessModule.

For example, Thermspud1 currently needs particle sources to generate particles. A typical source is a Simulation-owned particle container. In the XML file, each Simulation-owned particle container would have a unique id. In the XML file, particle source nodes are added to the Thermspud1 node. Each added particle source node would contain an IDREF (a reference in XML to a declared unique ID) that will indicate to Thermspud1's DOM_Document_Processor that the Simulation object is the location of the specified particle container.

The next two examples use generic names for demonstration purposes only. Assume an interface is a collection of related methods that the underlying object guarantees to have implemented.

Figure 3-3 depicts a more detailed example of the framework design principles. This framework example has 2 ExecutionModules and several ProcessModules. This example depicts a typical runtime configured application using the SIMSPUDII framework. Note that the Simulation requires Interfaces 5 and 6. One can assume that these interfaces provide some specific services to the Simulation object. These interfaces are implemented uniquely (different implementations of the same services) by two independent versions of ProcessModule1 (1A and 1B). The application builder must choose between the two interfaces for the current run of the application. The Simulation can have multiple versions of all ProcessModules; the only requirement is that each version has its own unique runtime identifier (ID); this is guaranteed by the DTD.

No ExecutionModule can access a ProcessModule 'owned' by another ExecutionModule. ExectionModule1 cannot access any interface on ExecutionModule2 owned ProcessModule4 (see Figure 3-3). Thus an ExecutionModule has only two sources of interface implementations: the ExecutionModule's ProcessModules or the Simulation's ProcessModules. This design was adhered to in an attempt to keep the true tree-like structure of the XML file mirrored in the application.

Note also in Figure 3-3 that there is no aggregation (association through instantiation). All objects are connected via interfaces. This keeps coupling very loose and relationships can be determined at runtime. Due to this feature, ID tracking is important. The module (ExecutionModule or Simulation) that has the IUnknown interface of a ProcessModule is responsible for removing the ProcessModule from memory when the application is complete.

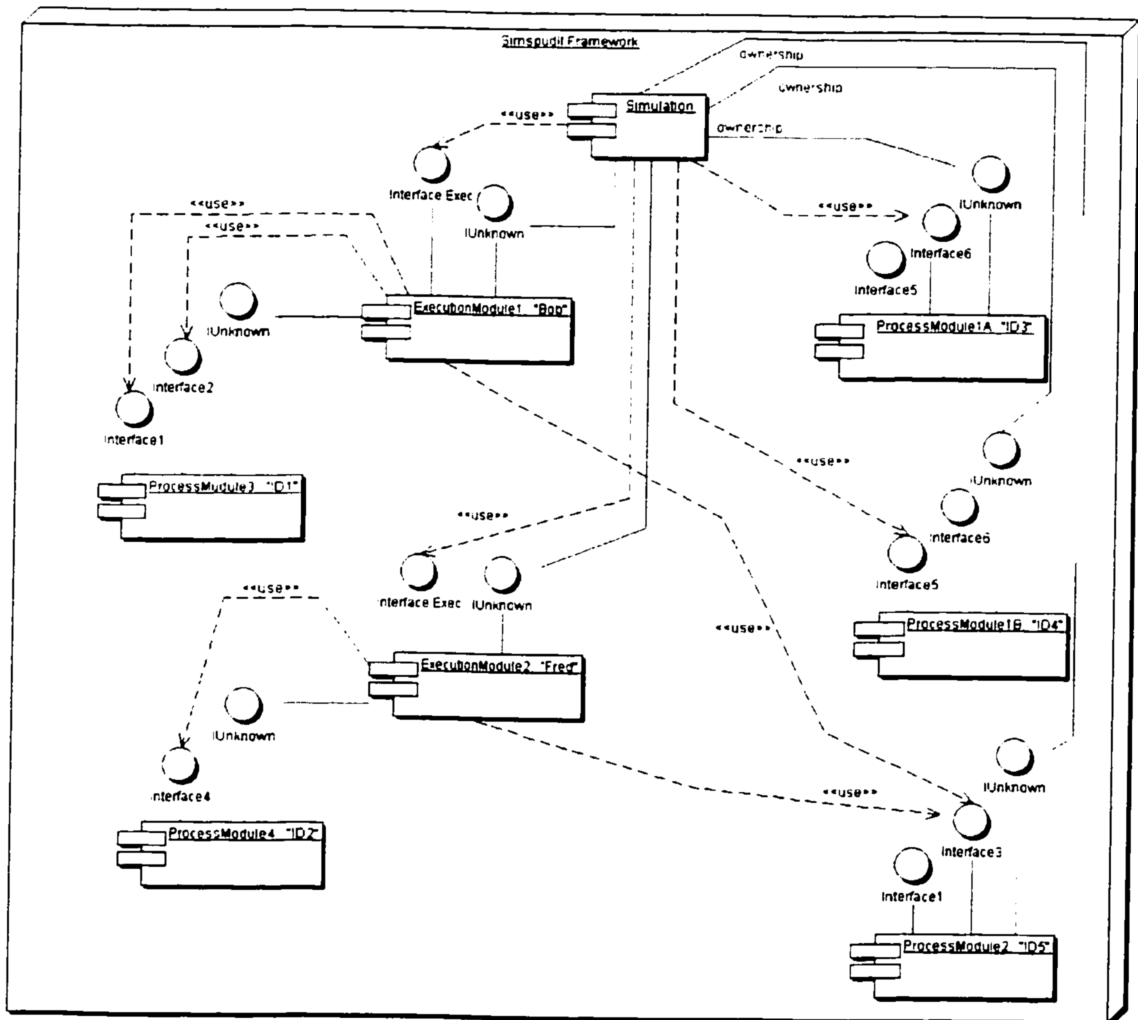


Figure 3-3: XML specified application 1 using the T-COM

In this example, two ExecutionModules have their interface requirements met by internally owned ProcessModules, as well as Simulation owned ProcessModules. The Simulation also requires specific ProcessModule provided interfaces.

Look in Figure 3-4; we see a new application constructed. This time the Simulation object is satisfying its interface needs with ProcessModule1A. Thus ProcessModule1B was not created. With the XML file, the ProcessModules created are those that are specified in the XML file. If a module is not needed, it is not created. Again, all dashed links in Figure 3-3 and Figure 3-4 are created using IDREFs in the XML file. The application is thus dynamically created each time it is run. The framework embodies all developed applications, but it only creates those specified.

Finally, any objects that are shared anonymously as non-singleton (more than one occurrence in memory) objects by the ExecutionModules are classified as ***low level*** objects or utility objects. This category consists of Particles, Vectors, Points, etc. All ExecutionModules, ProcessModules, and the Simulation use these objects anonymously (no id). Many of these objects are created, destroyed, passed from module to module, etc. Quite often the role of these objects is to contain messages that are passed between ExecutionModules.

This design has kept circular dependencies (file 'a' depends on file 'b', and file 'b' depends on file 'a') to a minimum and helped keep the project file organization under control.

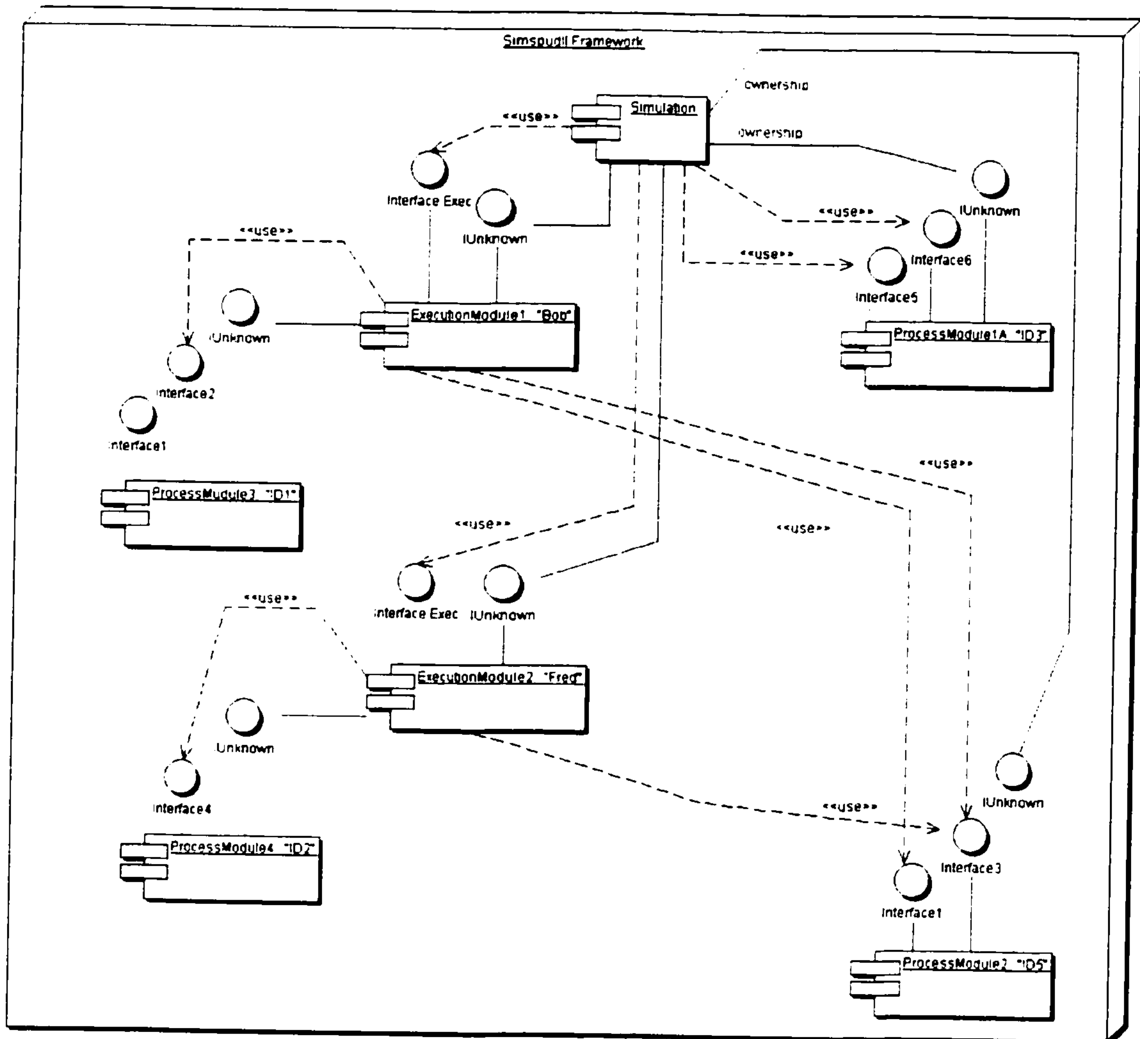


Figure 3-4: XML specified application 2 using the T-COM

This example demonstrates that the two ExecutionModules and the Simulation module have all of their interface needs satisfied without the need of ProcessModule1B of Figure 3-3. *Thus ProcessModule1B was not created for this application.*

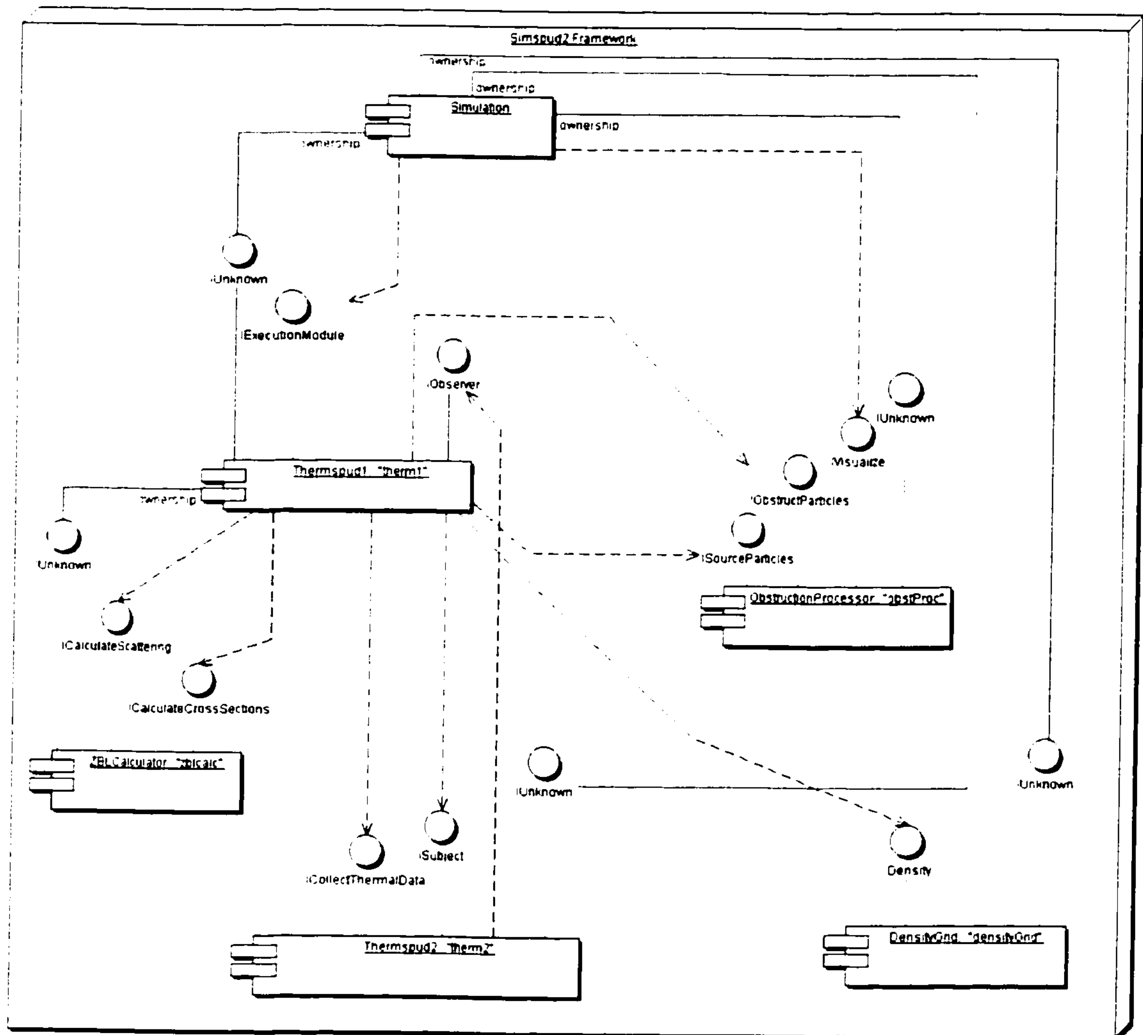


Figure 3-5: Basic application using the T-COM

This diagram depicts a simple application using the T-COM. Thermspud1 is the only ExecutionModule.

3.2 The SIMPUDII Process

The Simulation object provides the service of IO (input-output) for all ExecutionModules. The Simulation only requires a valid XML file and a MessageRouter object (see page 55). The initial design has no user interaction during runtime. It is assumed that the user will specify the runtime parameters, the simulation will run, and the results will be output. Extending the simulation from this batch-type environment to one that has more interaction would not require a great deal of work.

From a very high level view, the typical Simulation phases of execution are the following:

1 Simulation Construction

2 Simulation Initialization

3 Simulation Execution

4 Simulation Output

5 (*optional*) Simulation Visualization

6 Simulation Destruction

Initialization Phases

The initialization phase is important. The order and manner in which all ExecutionModules and ProcessModules are constructed and initialized need to be addressed. The initialization had to be designed in a manner, which, if followed, would enable future extensibility and reduce confusion over module responsibilities.

The initialization phases appear below:

1.*Simulation Object Initialization*: In this phase the Simulation object creates the DOM_Document from the XML file. The parameters for the Simulation and the Simulation owned ProcessModules are extracted from the DOM_Document. Typically, ProcessModules require specific parameters before they can be created. These parameters are stored in the Simulation object.

Contract: DOM_Document exists. Simulation framework and message routing facilities are loaded into memory.

2.*Simulation-owned ProcessModule and ExecutionModule Creation*: In this phase the Simulation creates all of its ProcessModules and the ExecutionModules that are specified in the XML file. This phase is completely dynamic. Only those ProcessModules and ExecutionModules specified in the XML file are created and added to the framework. Each ProcessModule and ExecutionModule has a *unique* identifier. No initialization occurs at this phase; only module creation is undertaken (Figure 2-6).

Contract: All Simulation-owned ProcessModules are created; all ExecutionModules are created. All Simulation-owned interface classification is complete. All created modules have a unique identifier.

3.*Simulation ProcessModule Initialization*: In this phase, all simulation owned ProcessModules are initialized. Either the setting of the stored parameters or the

passing of the DOM_Document to the ProcessModule's initialization method completes this task (Figure 3-2).

Contract: All ProcessModules are ready to be used. The modules' services can be accessed after this phase.

4.*ExecutionModule Initialization*: In this phase, all ExecutionModules are initialized. Calling the ExecutionModule's initialize method completes this phase. Each ExecutionModule extracts the pertinent parameters from the DOM_Document and creates the ProcessModules that are subordinate to the ExecutionModule. In this phase each ExecutionModule also initializes its own ProcessModules by passing the DOM_Document. During this phase the ExecutionModules connect with the specified services provided by the Simulation. All interactions are specified in the XML file via the module *unique* identifiers. No connections between modules are established unless the connection is specified in the XML file. All relationships and links are created at run time.

Contract: All ExecutionModules are ready to have *Execute()* called. No further initialization is undertaken by any module after this point. All module interconnects are created.

5.*Input/Output (IO) subsystem Initialization*: In this final initialization phase, the ExecutionModule IO subsystem is initialized. This is engaged last in an attempt to simplify the responsibilities of the ExecutionModules. This removes the requirement that the ExecutionModule must have IO systems initialized.

Contract: Entire product is ready to run. All initialization is complete.

If during any of the phases an exception is thrown, the initialization stops, and the program exits. If the contract has been fulfilled at the end of an initialization phase, the next phase is attempted.

Execution Phase

Every ExecutionModule has an execution phase. The setting of the appropriate XML flag can disable this phase on a per module basis. During execution a queue of ExecutionModule identifiers is built. This queue defines the order and number of times each module is executed. Each module is expected to execute until some statistically significant event occurs. This will signal the end of a module's execution phase at which time the next module in the queue will be executed.

The Output Phase

This phase calls each ExecutionModule's output routine. The Simulation will also call Simulation owned ProcessModule's output routines. Typically, any ProcessModule that needs to output data will implement the ICollectData interface. All Simulation-owned ProcessModules that collect data are appended to an ICollectData data structure that is processed during the output phase.

The Visualization Phase

Any module that has the means to visualize collected data will be called upon to process this visual data. These modules implement the IVisualize interface. Currently, only systems with gnuplot[27] will use this feature.

4. Problem Domain: Implementation of Design

This section will detail the implementation of the software package and the software support systems. SIMSPUDII is essentially a software package that was designed to simulate the sputter deposition process. This chapter will outline the problem domain specifics of the framework presented thus far and some of the design decisions during implementation. Only ExecutionModule objects, ProcessModules, and low level objects that have been designed and implemented by the author will be discussed in this section. The reader should have a good understanding not only of what the SIMSPUDII development framework provides for the problem domain, but also of how to use the facilities after reading this section. A discussion of the Simulation object and its facilities will be followed by a discussion of the ObstructionProcessor. Finally, the design and implementation of Thermspud1 will be explained.

Throughout these discussions, certain naming conventions will be used. Capitalized names refer to C++ classes. A commonly used naming convention in C++ is that each word in a name is capitalized, with no spaces or underscores. For example, the ObstructionProcessor is a class that provides the services of obstruction (physical chamber objects) processing (containing, maintaining, visualizing, etc).

4.1 Simulation

The Simulation object binds all aspects of the framework together. This object is the container that provides global interaction subsystems between ExecutionModules, as well as contains all globally accessible ProcessModules.

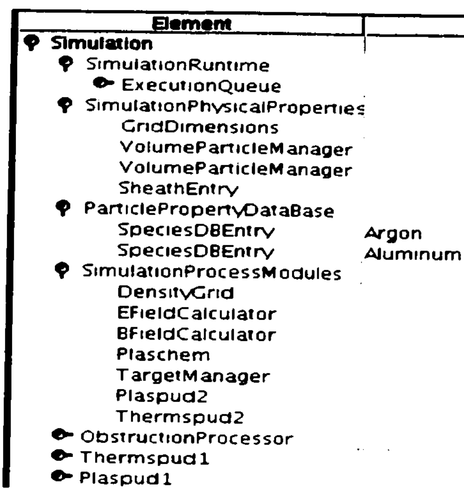


Figure 4-1: Simulation XML tree

This is a view of the simulation nodes as they appear in an XML editor.

The first issue addressed in the framework design was the concept of data IO. When the author examined the architecture of SIMBAD/SIMSPUD these things were noted:

1. Many files are required as input (histograms, physical structure specifications etc.)
2. Many parameters are required for simulation initialization
3. The product will need to provide progress information, during runtime, to the user
4. A large amount of data is generated by the program

It was imperative to devise a system that allowed the user to create the specifications of both the simulation being run and the output data storage location. The specifications of all parameters are handled by the previously discussed XML solution.

MessageRouter

The study of SIMSPUD/SIMBAD revealed some abstraction for output messaging and file movement. A more robust system would be beneficial if the framework being designed were to have a longer lifecycle.

Some requirements of the message routing module were defined:

- *Must have the ability to direct formatted input to specific, runtime-defined locations.*

Initially the framework would be a console application that used 'standard out' to communicate with the end user. This, of course, would eventually evolve into some type of graphical interface. The design would have to be such that no clients of the module would need to be modified in order to move the interface to a graphical environment or from a batch to a runtime framework.

- *Must have the ability to open, close, and verify all directional lines of output*

Instead of forcing the clients to open files, check for success, etc., this module would need to encapsulate all file input-output (IO) completely. This would reduce the need for future developers to be concerned with the platform the simulation ran on (file permissions is an issue in Linux, but not Win9x). This encapsulation also would ensure

that the framework would not be susceptible to 'lazy' programmers who extend the framework at a later time.

- *Must have the ability to direct single data streams to multiple, pre-runtime defined, locations*

The location of the input and output files is a pre-runtime defined parameter, thus hard coding file was not a solution. It was necessary to abstract all file IO and have the routing defined by the user (information going to standard out, or to a unique output file).

- *Must be able to provide a mechanism that was useable and transparent so future developers would not feel hindered by the use of the system*

No matter how clever the module's design or its features, if the task of utilizing this feature set is not completely intuitive, the system will be ignored. The solution must abstract all message passing so that the encapsulation provides natural and easy to use interfaces for the developer.

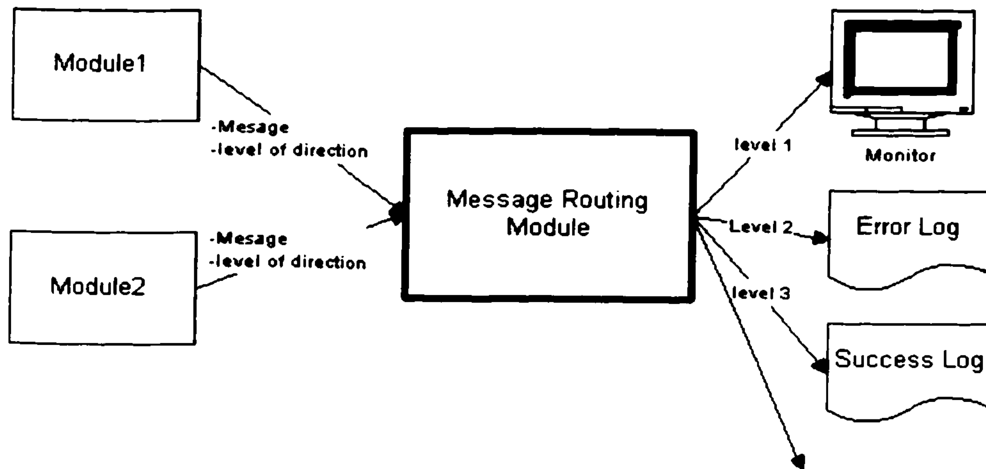


Figure 4-2: Message routing

The MessageRouter manages communications from all modules. The routing is configured at runtime.

The solution devised addressed the issues mentioned. The MessageRouter module is one of the most ambitious module designs in the project. It has the ability to route single data streams into multiple file streams, all runtime defined. It allows the user to create files by only passing a file name, and to be guaranteed that the file will be created in the pre-runtime (or runtime if the system evolves to an interactive system) default directory. If files cannot be opened, streams are routed to standard error. With a single flag, all output can be echoed to standard out (a feature effective for debugging).

The MessageRouter is essentially a Standard Template Library (STL) map of string-keyed ostream pointers, combined with a custom designed ostream object that routes messages (STL string) to multiple output streams. The MessageRouter associates streams to identifiers ('stream tags'). Sets of identifiers can be created and associated with a set identifier ('route tag') so that a single message can be sent to a group of previously created streams.

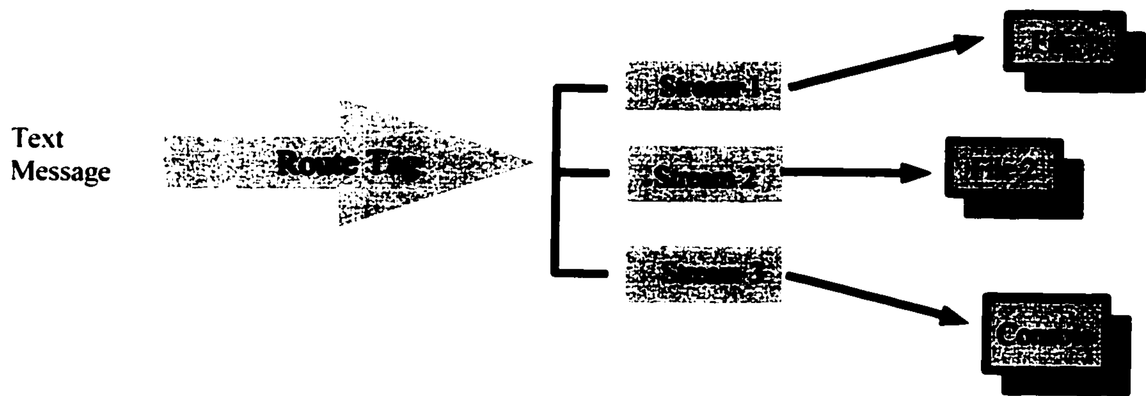


Figure 4-3: Routed streams

A text message is directed, simultaneously, to several run-time configured streams.

The MessageRouter has two preset stream tags that represent standard out and standard error:

MessageRouter::COUT

MessageRouter::CERR

Any other created streams will have the default path appended to the stream name (file name) before opening the stream. This feature ensures that any client (developer) needs only to create a file by using a single path independent name, and thus the default (and runtime modifiable) path is abstracted. This solves the post-runtime output data organization issue as well. All output is directed to the specific location defined by the user (usually as an XML parameter). Log data can be collected in a specific file **and** output to the screen to provide run time user feedback.

By default, all created stream identifiers (stream tags/filenames) also become set identifiers. Thus a set with the one stream is always created. Other stream tags can be appended to this set to create multiple route streams.

Interfaces

As mentioned earlier. Interfaces were used to reduce the software complexity and strengthen the notion of object ownership and access privileges. Aggregation was avoided wherever possible. All inter-module communication is achieved through thin (concise) interfaces. Martin refers to this design concept as the Dependency Inversion Principle[25]. He states that every dependency should target an interface rather than an object. There should be no dependencies on concrete classes, only on interfaces or abstract base classes. This provides what he calls 'hinge points' that create locations where the architecture can 'bend' or be extended. As mentioned, these hinge points can be bent during run time using the XML/T-COM specification (see XML discussion earlier).

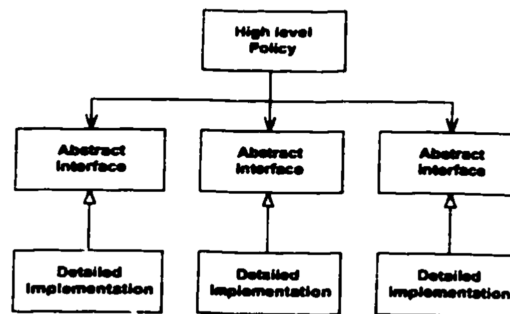


Figure 4-4: Dependency structure of an Object Oriented Framework[25]

This diagram depicts the concept of abstraction through interfaces. Implementing the required interfaces creates the detailed and replaceable implementation of a framework.

ParticlePropertyDatabase

One object common to all modules, and the basis for all aspects of simulation data generation and collection, is the Particle object. The Particle object and the ParticlePropertyDatabase (PPD) handle particle representation within the simulation. The

PPD is a singleton object that maintains particle attributes that are common to all particles.

Figure 4-5: PPD XML Node

One can add multiple particle species to the PPD node of the XML file.

The PPD was implemented by extending a Standard Template Library (STL) vector. Utilizing the STL vector allows the database fast associative lookups. as well as low penalties for end insertion.

A static pointer to the PPD is provided to the Particle object during initialization. Thus all Particles have direct access to the per species properties. All attributes of an argon particle can be accessed directly from the Particle without having to maintain a separate pointer to the PPD. This reduces the total amount of information a Particle must contain (typically millions of Particle instantiations per phase of execution), yet full access to all per species data is maintained in all areas of the simulation.

ParticleBuckets

ParticleBuckets are containers that hold and maintain Particles. Again, a STL vector was extended to give the vector's desirable features to this container. The ParticleBucket class

implements the ISourceParticles interface to abstract the principle of particle generation. ParticleBuckets can be added and scaled.

Grids

One key goal of this project was to provide the ability to simulate magnetron-sputtering processes using a discretized chamber volume. Thus a discussion of the Grids that are part of this project is relevant. The Grid class, designed and implemented by Dr. Loran Friedrich, Dr. Steven K. Dew, and the author of this thesis, provides the simulation with a data structure that can, not only hold unique data that describes a cell based view of the simulation, but also refine its structure to more accurately describe a physical attribute of the simulation. For any given cell, if the client determines that the change in value from one cell to the next is too extreme, the cell can be divided to allow for further refinement of the physical property description. Each cell can be divided into 8 smaller cells, and the data is distributed amongst the newly created cells. With interpolation routines, the inter-cell transition can be refined. The only limit to the cell division process is the host system's amount of memory.

These adaptable Grids give the framework the ability to describe properties such as a spatially varying magnetic field more efficiently. Large cells can describe places where the magnetic field is slowly varying. Where magnetic field lines are closer together, the grid can be refined. This reduces the total memory and computation load on the simulation.

The Grid object was designed as a C++ template to allow flexibility in the data structures appearing in each cell. If a developer implements the IGridData interface, the ability to output postscript slices is made available.

Currently the framework has three grids in use:

- Density grid (maintains density and temperature information)
- BField grid (represents the magnetic field)
- EField grid (represents the electric field)

Each of these three grids can be accessed through the Simulation object. A fourth grid is used by the ObstructionProcessor (discussed later) to optimize the hit calculation process.

4.2 ObstructionProcessor

The Obstruction Processor module represents all physical obstructions within the simulation. The set of physical obstructions contains all objects that will not change position within the chamber as a direct result of particle bombardment. Typical obstructions are targets, substrates, chamber walls, etc. This module is responsible for maintaining a representation of all properties, static or dynamic, of the internal obstructions throughout the simulation. All processes involved in updating the state of an obstruction, collecting location specific statistics upon an obstruction, obstruction modification, obstruction removal, obstruction adding, physical modification of an

obstruction, and obstruction movement will be contained within this module. This module will give obstruction-specific data to any module upon demand.

The bulk of the work done on the simulation product so far has been focused on the Obstruction Processor. Some key features of this module include the following:

- Full representation and specification of complex 3-D structures
- Visualization of internal structures using gnuplot[26] output
- Detailed per surface collection facilities:
    - Histogram generation
    - User-specified collection regions
    - Default data collection facilities
- Particle Generation using collected histogram data or run-time histogram specification
- Particle-Surface stopping using grid optimization for efficiency
- Extensibility

The ObstructionProcessor module is classified as a ProcessModule (mentioned earlier). All modules in the simulation environment have access to the ObstructionProcessor module. Thus the Simulation object is responsible for constructing the ObstructionProcessor. The majority of all simulation-physical structure interactions is conducted through some interface with this module. This module is responsible for collecting and outputting data collected during process-physical structure interactions.

During construction, the ObstructionProcessor requires a pointer to the Simulation environment.

For initialization, this module has its own DOM_Document processor. Using the XML file, the user can quickly create unique physical parameters.

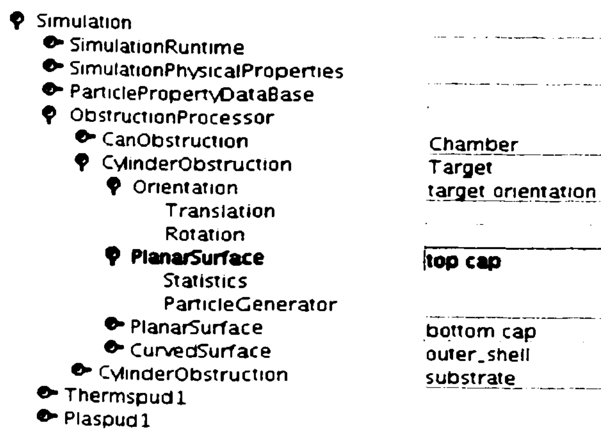


Figure 4-6: ObstructionProcessor XML node

The ObstructionProcessor is fully configurable via the XML file. Manipulating sub nodes of the ObstructionProcessor in the XML file configures CollectionRegions, geometry, particle generation, and orientation.

The details of the Obstruction Processor will be discussed from a bottom up approach. The discussion will focus on the small elements then examine the elements that can be combined to represent the physical obstructions in the simulation.

Orientation

The orientation object abstracts all positional data for the Obstruction Processor. The orientation object maintains all translation and rotational data. The orientation object has

a sense of 'local' and 'global' coordinate frames. The origin location of the orientation is considered 'local' coordinate frame and where it currently resides is considered the 'global' coordinate frame. The orientation can move particles, points, and vectors to and from local and global coordinate frames. Particles are tracked in the 'global' coordinate frame, whereas data is collected in the 'local' coordinate frame. With this method of position abstraction, any object can have a sense of position in the simulation. Orientation is used for surfaces, obstructions, and collection regions.

CollectionRegion

The CollectionRegion object is a user-specified masked 2-D histogram. The shape of the mask is determined by the collection region's subclass. The collection region's origin is the position of the Histograms (0,0), relative to the external coordinate frame. The Collection region has two orientations, one for the physical orientation of the region and one for the orientation of the histogram with respect to the collection region's hit mask. Thus it is possible to have collection regions that have uniquely situated masks and independently positioned histograms within the mask.

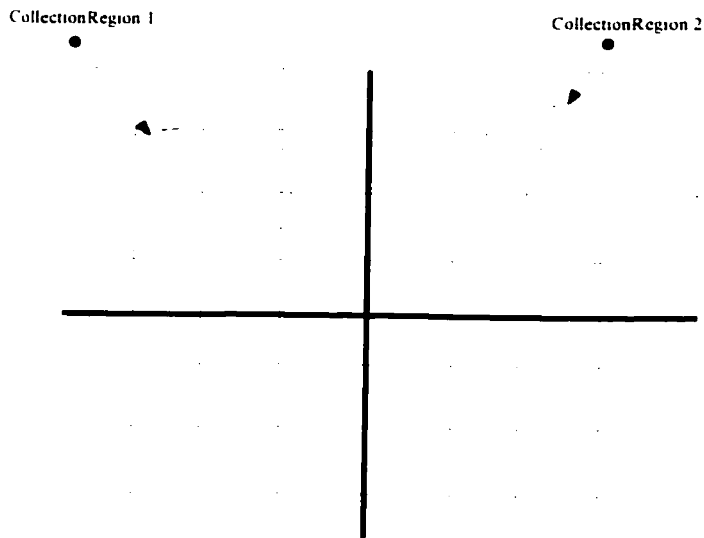


Figure 4-7: CollectionRegions

CollectionRegion 1 is a rotated and translated region with a synced coordinate frame. CollectionRegion 2 is a rotated and translated region with an independent coordinate frame.

Currently there are circular and rectangular hit masks. The user can specify positional data or directional data to be collected.

LinearProfile

Element	Name
Simulation	
SimulationRuntime	
SimulationPhysicalProperties	
ParticlePropertyDataBase	
SimulationProcessModules	
ObstructionProcessor	
CanObstruction	Chamber
CylinderObstruction	Target
CylinderObstruction	substrate
Orientation	substrate orientation
PlanarSurface	top cap
PlanarSurface	substrate cap
Statistics	
LinearProfile	**SubstrateProfile**
CurvedSurface	outer_shell
Thermspud1	
Plaspud1	

SubstrateProfil...

Field	Value
name:	SubstrateProfile
x2:	1
width:	1E-3
FilterSpecies:	false
x1:	0
numBins:	100
y2:	0
y1:	0
SpeciesFilter:	

Figure 4-8: LinearProfile XML node

A LinearProfile is specified in XML by two points. It collects data along the specified line.

A special form of collection region known as a LinearProfile can be used to collect data along a line. A LinearProfile is a Histogram that collects along a line specified by two endpoints and a width.

Statistics

Statistics is a grouping of statistical data that can be collected as one or two-dimensional Histograms or CollectionRegions. All statistical data collection is conducted through a Statistics object. This objects maintains several default histograms and collection parameters. All data collected and maintained by the statistics module are related to particle interaction in some way. The data collected by a Statistics object by default is as follows:

- Particle hit data (as a function of position)
- Directional particle hit data
- Energy particle hit data

All data are collected per species.

Any number of Collection regions or LinearProfiles can be attached to a Statistics object. This is usually done in the XML file on a per-Surface basis.

Surface:

The basic building block of any structure is the Surface. A Surface has an Orientation object (as mentioned) to keep track of its position and to transform all incoming particles to and from a local coordinate frame. All Surfaces begin life in a specific orientation in the global coordinate frame, then are rotated and translated into position to create Obstructions. This approach was selected to keep the data collection manageable. Using this design, the user can add CollectionRegions at arbitrary positions on a surface and determine the reference frame with which the results are aligned. All Surface objects thus have a local 'origin' from which all data collection is referenced. The user must consult the documentation to know the specific surface origin and rotation in relation to a specific Obstruction for correct data interpretation (see discussion on page 73).

Each Surface has a two Statistics objects. The first allows collection of all hit-specific data. For an entire simulation, the first Statistics object collects data of all particles that hit a surface. All user defined CollectionRegion objects are maintained by this Statistics object.

The second Statistics object is used for particle generation. One of the features of a surface is its ability to generate particles (see discussion on page 70). This second Statistics object collects data on all particles that hit a surface with enough energy to create a sputter event. This Statistics object is referred to as the genStatistics object. This object maintains no collection regions at this time.

The Simulation framework is designed to use a discretized chamber volume. There is need for Cell-based data to be extracted from a Surface. Currently, all Surface objects have the ability to calculate the area of the Surface bounded by a Cell.

Each Surface also has the ability to stop particles and has gnuplot visualization routines. The visualization routines give the user important data as to how the Surfaces are aligned in 3-D space.

Surfaces are divided into two basic categories: planar surfaces and curved surfaces.

Planar Surfaces

The planar surface group is further sub-classed into basic shapes:

- Multilateral (squares, rectangles, convex polygons)
- Circles and rings
- Ellipses and elliptical rings

It is important to note that all planar surfaces can only be hit from one side. It is assumed that each surface will be a part of a set of surfaces that will create an enclosed object, with the outer face having the ability to stop particles and collect data. This design decision was an optimization technique employed to reduce the total number of floating point calculations during hit processing.

Curved Surfaces

The only curved surface is the cylinder shell. This is a basic cylinder surface. As an artifact of the impact calculations, the cylinder shell surface is the only surface that can be hit from either the inside or the outside.

ParticleGenerator

Instead of storing millions of sputtered particles during a specific phase of a simulation, a statistical representation of the sputtered particles is collected to reduce the memory load and allow for independent module runs using previously collected data. A ParticleGenerator is a ProcessModule object that can generate particles using previously generated histograms. This object is attached to a specific Surface, thus creating a sputter target. A ParticleGenerator is responsible for generating only one specific species type. Thus if a multiple species target is to be constructed, several ParticleGenerators are required.

The ParticleGenerator has an XML node that allows the user to specify the attributes of each ParticleGenerator. Currently, the ObstructionProcessor handles the ParticleGenerator's XML data extraction.

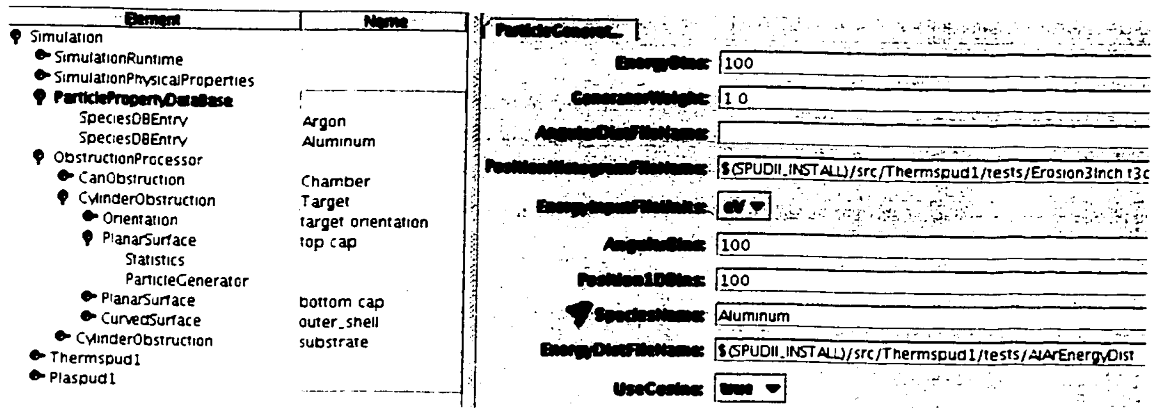


Figure 4-9: ParticleGenerator XML node

A target is merely a surface with one or more ParticleGenerators. The required distributions are specified under the ParticleGenerator node of the XML file. This node can be added to any PlanarSurface.

For each particle generated. the ParticleGenerator generates the subsequent information:

- Position
- Trajectory
- Energy

The specified attributes are generated for a specific Particle species.

Position generation is calculated using a random number generator that is capable of generating x, y coordinates by utilizing a two-dimensional (x, y, probability) Histogram. The random number generator was designed and implemented by Dr. Steven Dew and his associates and thus will not be discussed in this document. The Histogram can either be passed as a file parameter during initialization or can be generated by collecting sputtered particles through other simulation phases. This Histogram is usually an erosion profile that describes a specific target.

A ParticleGenerator is tightly coupled to a single Surface object. The ParticleGenerator uses a pointer to the host Surface to transform particles into the global coordinate frame. Although this can be viewed as poor software engineering design, the decision was made to reduce total communications, and thus tie the ParticleGenerator to a specific surface.

The trajectory is generated in one of two ways. First, a 2-D distribution can be supplied (via a file or a Histogram created during a separate phase of execution) that describes the angular distribution. With a random number generator, the in-plane angle (beta) and azimuth angle (phi) are generated in the local coordinate frame. These angles are later transformed into the global (lab) coordinate frame.

Alternatively a cosine distribution can be used. This was one of the models employed by SIMSPUD for emitting particles. The user can select a particular model by modifying the appropriate node in the XML file.

The energy at which the particle leaves the Surface is determined by a random number generated using a 1D Histogram that describes the probability of a specific energy occurring. This Histogram is either created from a user supplied file or a Histogram generated during a previous execution phase.

The ParticleGenerator does not maintain physical surface geometry (size, extents, etc); it assumes that all histograms used to generate particles have this data represented correctly.

Obstructions

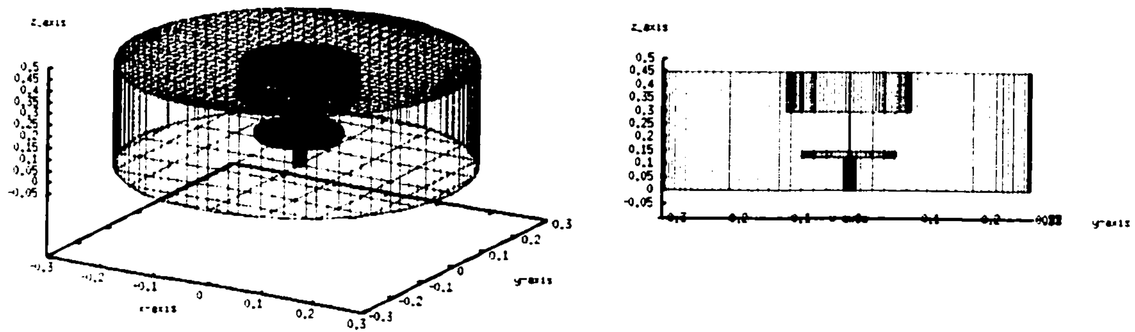


Figure 4-10: Chamber wire-frame representation

This depicts a chamber setup with a target on the top and a substrate on the bottom.

An Obstruction object is essentially a collection of surfaces rotated into the global coordinate frame. As was mentioned, each Surface begins in a local coordinate frame and then is rotated and translated into the global coordinate frame. To reduce the complexity of data collection specification, basic Obstruction objects that depict various geometric shapes are provided.

The basic shapes that are provided are the following:

- Cylinder
- Can
- Hollow Pipe
- Cube
- Box

It is important to note that due to the inaccuracy of PI approximations, all obstructions may NOT be sealed units. There will be small gaps at the boundaries of the surfaces. This can be fixed at a later time; it was a design decision to keep the orientation consistent

with the surface placement. Surface 'fuzziness' has been employed to reduce the margin of error and the possibility of particles escaping. The only exception is the Can Obstruction. This exception ensured that no particles would leave a chamber object. It will not reduce overall performance if one bounds a non-cylindrically shaped chamber with a Can Obstruction.

Each Obstruction is a container of Surface objects. The specific ordering of the shapes is important, as is the rotations and translations to get the surface into 3-D space. The end user needs to be aware of these details in order to make sense of the data collected during execution, as well as for meaningful orientation of CollectionRegion objects on specific Surfaces. The details of each Obstruction's Surface ordering and orientation follow. For the most part, the selected ordering for Obstruction creation is somewhat arbitrary.

CylinderObstruction

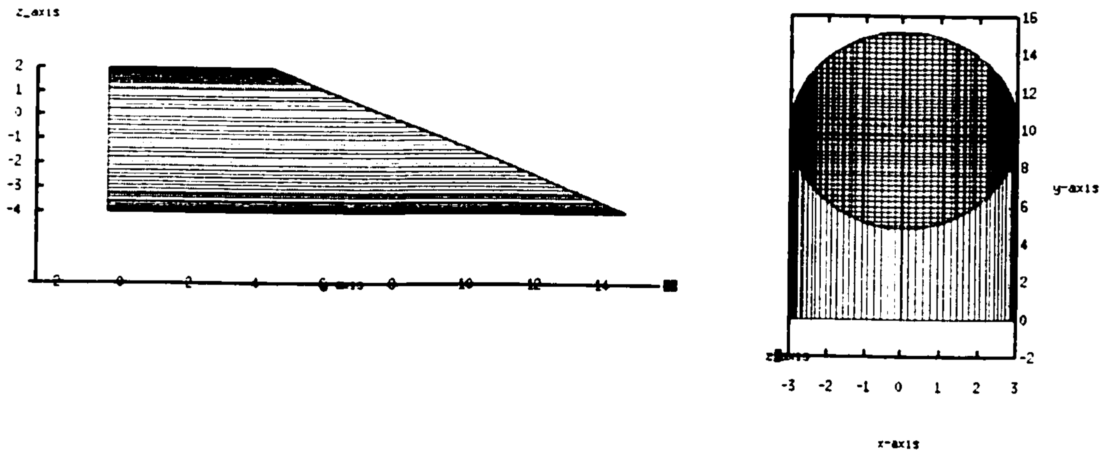


Figure 4-11: Cylinder Obstruction wire-frame representation

This depicts a side and front view of a cylinder with a tilted cap.

The CylinderObstruction is the basic Cylinder object. It can be hit from the outside only. It has an EllipticalPlanarSurface for a top cap (allows rotation and tilting), one CylinderShellCurvedSurface, and a CircularPlanarSurface bottom. If a cap must be rotated and tilted, this must occur on the top cap. The top cap can only be tilted so that it does NOT touch the bottom cap.

The Origin is at the center of the base cap; the normal points up towards the top cap from the origin. Surfaces are constructed at the origin in this order:

0: top cap,

1: cylinder shell,

2: bottom cap

The top cap is built on the origin and translated into place. The cylinder shell is not translated or rotated. The bottom cap is built at the origin and then rotated PI around the x-axis.

CanObstruction

The CanObstruction is the basic canister obstruction. This is one of the few obstructions built to be hit from the INSIDE only. A common use for this obstruction is the chamber obstruction. The top cap is pointing down; the bottom cap is facing up. The ordering of the surfaces is identical to the Cylinder object. Rotation of the top cap to put it into place is achieved by a rotation about the x-axis. The top cap cannot be tilted or rotated.

PipeObstruction

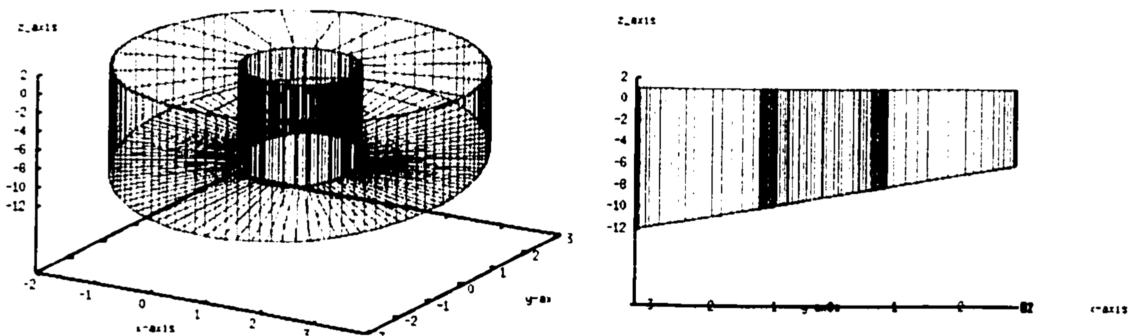


Figure 4-12: PipeObstruction wire-frame representation
The PipeObstruction is another of the basic building blocs available.

The PipeObstruction is the basic hollow pipe object. It can be hit from the outside only. It has a RingEllipticalPlanarSurface for a top cap (allows rotation and tilting), two

CylinderShellCurvedSurfaces (inner and outer), and a RingCircularPlanarSurface bottom. If a cap must be rotated (an operation that is valid for a PipeObstruction), the top cap only is able to be rotated and only around the x-axis.

The Origin is at the center of the base cap, the normal is pointing up towards the top cap from the origin. This class adds a second radius to the CylinderObstruction base class. This new radius is the inner radius of the obstruction.

0: top cap,

1: outer cylinder shell,

2: bottom cap

3: inner cylinder shell,

The Surface construction order and rotations are the same as the other cylinder based objects discussed earlier.

CubeObstruction

The cube obstruction is the basic cube. All surfaces are constructed at the origin, in the positive xy quadrant (<000>...<0L0>), and then rotated and translated into their final placement.

The surface ordering is as follows:

0: xy(+Length in z direction)

1: -xy

2: yz(+Length in x direction)

3: -yz

4: xz(+Length in x direction)

5: -xz

At construction, if the cube is not rotated, the origin of each surface is defined as follows:

0: (0,0,Length),

1: (0, Length,0),

2: (Length,0,0),

3: (0, Length,0),

4: (Length, Length,0),

5: (0,0,0).

BoxObstruction

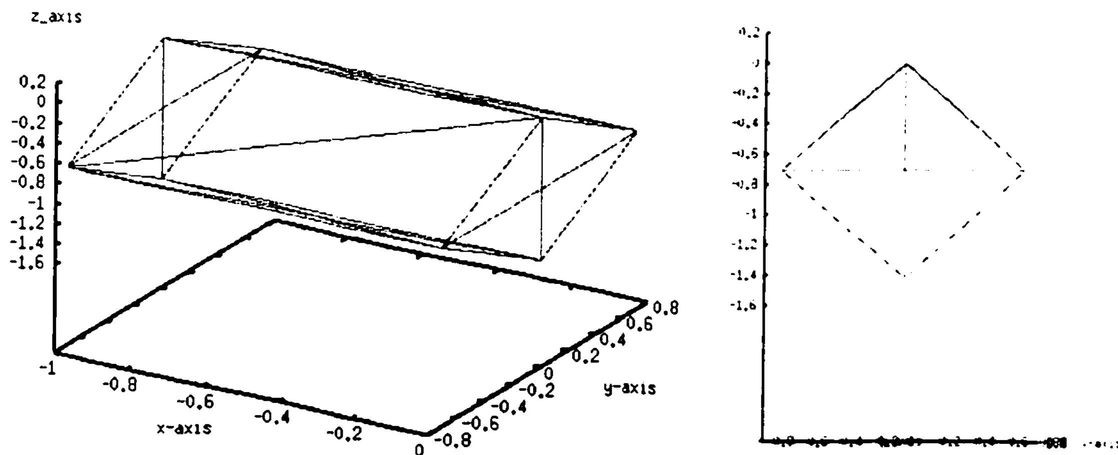


Figure 4-13: Box Obstruction wire-frame representation

The BoxObstruction is the basic box. All surfaces are constructed at the origin. in the positive xy quadrant (<000>...<0W0>). and then rotated and translated into their final placement.

The surface order is

0: xy (+Length in z direction)

1: -xy

2: yz (+Length in x direction)

3: -yz

4: xz (+Length in y direction)

5: -xz

At construction. if the box is not rotated, the origin of each surface is defined as follows:

0: (0.0.H).

1: (0.W.0).

2: (L.0.0).

3: (0.W.0),

4: (L.W.0).

5: (0.0.0).

4.3 Thermspud1

This module is responsible for the transport of neutral species particles within the chamber. This module encompasses a more sophisticated version of SIMSPUD.

Thermspud1 is a three-dimensional Monte Carlo model that follows the life cycle of energetic neutral species particles generated from a set of particle sources to a set of particle sinks. Particle sources can be any particle-generating surface (target) or particle bucket that has collected particles energized through other processes in the simulation. Particle sinks are typically any surface that obstructs the path of a particle. Another particle sink is the thermalization module (Thermspud2). Any particle that falls below an externally specified energy threshold is deposited into the thermalization module. Particle processing progresses until all externally attached completion modules (discussed later in this section) have reached statistical convergence.

Although Thermspud1 requires communication and information from Simulation-owned ProcessModules and ExecutionModules, Thermspud1 was designed to have no direct dependence on any external module. All connections are made through interfaces via externally instantiated 'set' methods. This design aids in extensibility of the Thermspud1 module. For example, in an attempt to parallelize Thermspud1's algorithm, one could abstract dependencies through interfaces and the externally instantiated 'set' methods. Thus all external modules could be replaced with proxy modules. For example, instead of connecting directly to Thermspud2 for thermal data collection, the Simulation, during initialization, sets a pointer to one of Thermspud2's interfaces: ICollectThermalData. Thus Thermspud1 will not need to 'know-a' Thermspud2, but only to 'know-a' ICollectThermalData.

Thermspud1 follows the classic SIMSPUD model in most respects. Key differences between SIMSPUD and Thermspud1 are:

- A more physically based collision cross-section model
- A soft potential scattering model
- Multiple gas species support
- An improved apparatus shadowing model
- Inhomogeneous gas density and temperature

SIMSPUD calculates the free path of a particle in motion using a Poisson distribution based on the mean free path between collisions[28]. SIMSPUD uses a binary hard sphere model that ignores long-range particle interactions. Thermspud1 uses the Ziegler-Biersack-Littermark (ZBL) universal potential and the so-called Magic Formula[30] to determine the collision behavior and the mean free path of a particle in motion (see later discussion). SIMSPUD calculates neutral particle - gas particle collisions assuming that the entire gas volume is homogeneous with respect to gas density and a single gas species population. Thermspud1 uses a discretized chamber volume to represent inhomogeneous densities of multiple species varying on a cell-by-cell basis. Multi-gas species densities are contained in an adaptable grid that can evolve to represent important structural features at higher resolution.

SIMSPUD does not take into consideration either gas heating or rarefaction and has limited support of shadowing by physical structures in the chamber. Although Thermspud1 does not directly calculate gas heating, it deposits particle energy throughout the volume. This energy data can be used by other modules (Thermspud2) to dynamically evolve the gas

properties. By using the ObstructionProcessor, Thermspud1 accounts for shadowing effects of the chamber apparatus. Any surface that obstructs the path of an in-transit particle will stop the particle and collect the spatial, angular, and energy data due to the impact.

The basic algorithm starts with a particle that is generated by one of the particle sources (typically a target). As soon as the particle is generated (see discussion on ParticleGenerators), its length of trajectory, or free path, is calculated. The particle follows a straight-line trajectory until one of the following contingencies occurs:

1. The particle collides with another particle (typically a gas species)
2. The particle collides with a surface
3. The particle energy, due to collisions with gas species and interaction with the simulation environment, falls below a predetermined threshold.

The details of each step will be discussed later in this section.

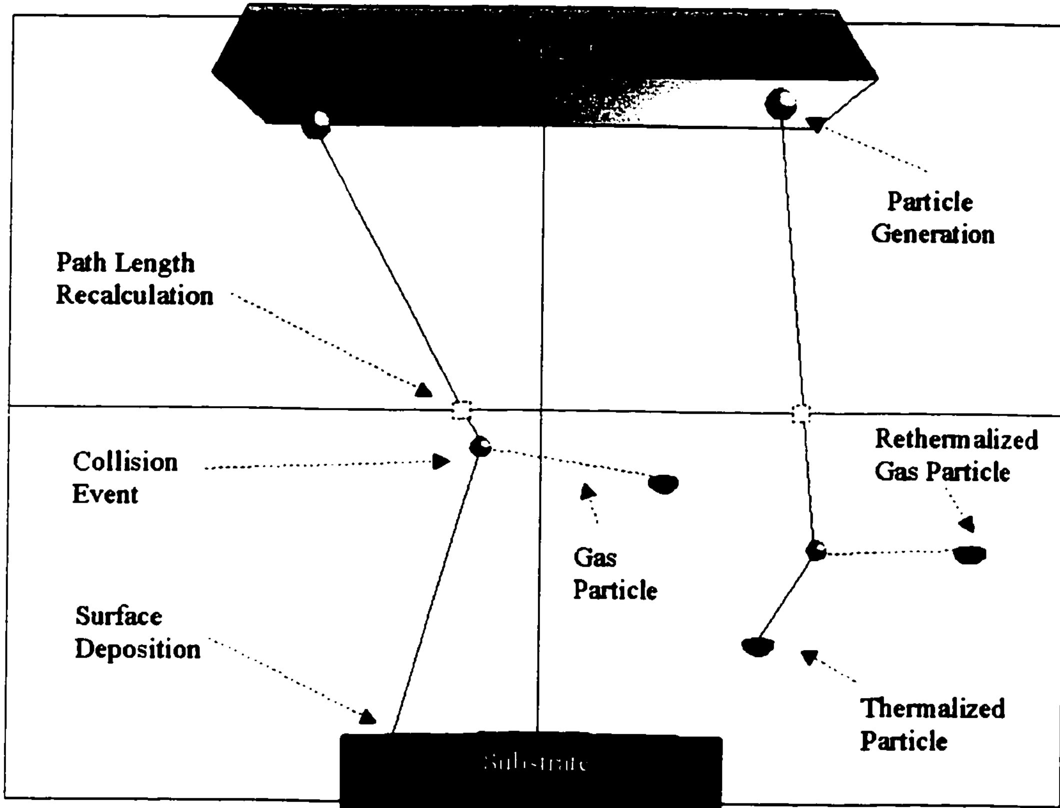


Figure 4-14: Thermspud1 algorithm

Thermspud1 uses a 3-D Monte Carlo algorithm to track neutral particle movements within a discretized chamber volume. Each 3-D cell represents a homogeneous gas density for the cell-contained volume. As a particle moves from cell to cell, the path length is recalculated. All particles are tracked until they deposit on a physical structure or fall below an externally set energy threshold and deposit into the chamber volume. The grid is actually a 3-D grid of much higher resolution than depicted in the Figure. The 2x2 size is for demonstration purposes only.

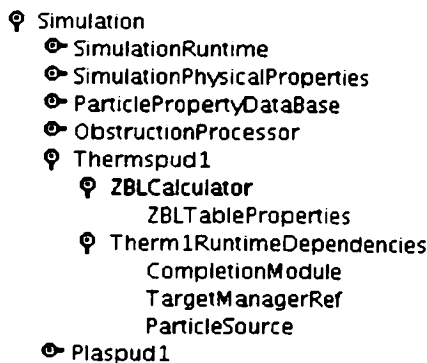


Figure 4-15: Thermspud1 XML node

The Thermspud1 XML node contains a runtime dependencies sub-node that allows for the user to specify which simulation modules will provide services. As shown, this configuration has Thermspud1 referencing a CompletionModule, a TargetManager, and only one external ParticleSource

Particle Sources

Since Thermspud1 begins its execution algorithm by generating a particle, this process will be discussed first. Thermspud1 has been designed to know as little about the particle sources as possible. During initialization, references to all particle sources are added, one at a time, to Thermspud1. These sources can be any object that implements the ISourceParticles interface. Each particle source has a relative weighting that determines which particle source is more likely to generate a particle. This abstracts the particle source selection away from Thermspud1. Thermspud1 creates a normalized weighted map of the particle sources that have a weight greater than zero. The particle source selection is then made based upon a random number.

During particle generation, the source is checked to see if it is still capable of generating a particle. If a particle source has a weight of zero, or can no longer generate a particle, the source is removed from the local data structure and thus will not be called upon to generate particles. This abstracts the responsibility maintaining the particle source lifecycle away from Thermspud1.

Particle Transport Model

After a particle has been generated, the straight-line distance the particle can travel before a collision with a gas particle has to be calculated. The distance a particle can travel before a collision is a function of the gas species densities. SIMSPUD assumes a homogeneous gas density. Thermspud1 assumes a pseudo inhomogeneous gas density model by assuming a homogeneous gas density and temperature on a per-cell basis,

which is updated whenever a new cell is entered. The Simulation uses a density grid that maintains the per-species gas densities.

The distance a particle can travel between collisions is referred to as the free path. The mean free path is the average distance a particle can travel before a collision occurs. The free path, intuitively, is related to the population densities of the particles, a measure that is quantified in gas particle species densities. The free path between collisions, λ, is randomly generated from a Poisson distribution[28].

$$f_{\lambda}(\lambda) = \frac{d\lambda}{\lambda_m} e^{-\lambda/\lambda_m} \qquad \text{Equation 4-1}$$

where λ_m is the mean free path. The mean free path is calculated using the subsequent information:

$$\lambda_m = \frac{1}{\sum_{i=GasSpecies_0}^{GasSpecies_n} n_i \sigma_i} \qquad \text{Equation 4-2}$$

This equation takes into account the local densities, n_i of each gas species i and the energy-dependent collision cross section σ (see ZBL discussion) of the particle in the gas species i.

The particle travel distance is thus calculated and assumed to be valid as long as the particle remains within the current cell. If it has been determined that the particle is able

to traverse the entire cell without hitting an obstruction or colliding with another particle, the free path distance is recalculated using the gas species densities that the new cell represents. If a neutral species particle enters a new cell that is more densely populated with gas particles than that of the previous cell, one could surmise that the in-transit neutral species would not be able to travel the full allotted distance calculated in the previous cell. Looking at Figure 4-16, we see a reduction in the total calculated path length once the particle moves into the right hand cell.

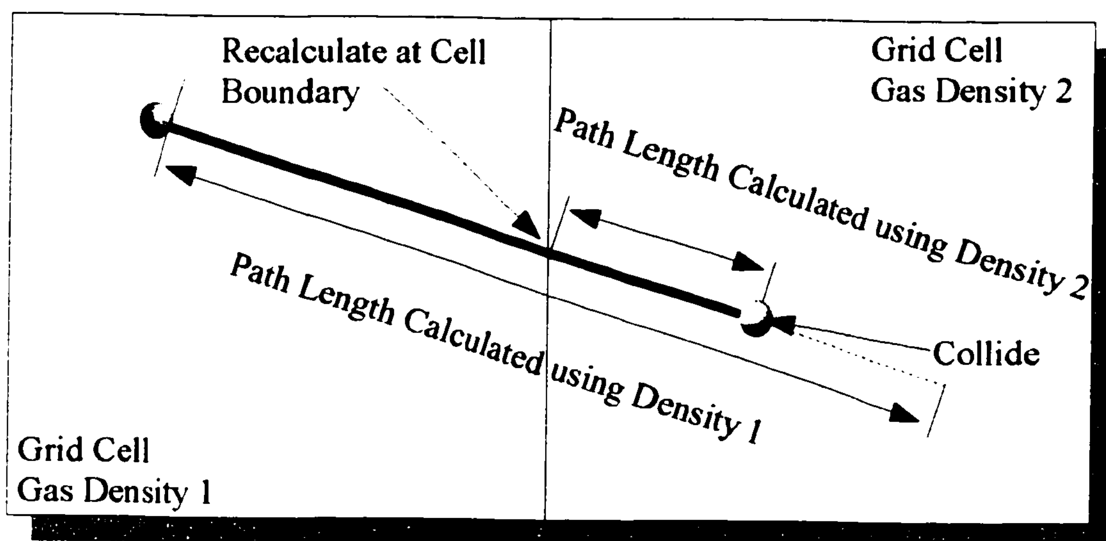


Figure 4-16: Particle path length calculations

A neutral species particle travels from a low gas density cell 1 to a high gas density cell 2. The total travel distance before collision calculated in cell 1 cannot be achieved in cell 2, so the path length is reduced.

As an optimization technique, each cell also contains a flag that states whether or not an obstruction is within the cell. This reduces the computationally intensive task of checking for collisions with surfaces if no surfaces are within the current cell.

Gas Particle Generation

Once a particle has traveled the computed distance before a collision (free path), the particle will collide with a gas species. Thus a gas particle must be generated to 'take part in the collision'. The weighted average density data structure extracted from the current cell is used to determine randomly the appropriate species of the gas particle.

To give the generated gas particle a trajectory, one assumes that the gas particle will have a uniform probability of any direction being generated, thus the trajectory is generated randomly. To calculate the energy of the gas species, one assumes that the gas particle will have a monochromatic speed. Speed is thus calculated using the temperature in the current cell:

$$\text{Gas speed} = \sqrt{\frac{2 \cdot kT}{m_{gass}}} \qquad \text{Equation 4-3}$$

Other approaches such as using a maxwellian speed distribution are possible.

ZBLCalculator

Collision cross-section calculation and collision event processing are two services that Thermspud1 requires external modules to provide. The modules that provide these services are attached using the Thermspud1 XML node. If no Simulation-owned ProcessModules are referenced in Thermspud1's XML node that provide these services, Thermspud1 defaults to using the ZBLCalculator to provide the collision-cross section

calculation and collision event processing. The inter-related models that are used by the ZBLCalculator are based upon the work presented by Ziegler, Biersack and Littermark[29]. Ziegler, Biersack and Littermark developed a formula, the ZBL universal potential, known for its ability to accurately describe interatomic potentials for a very wide range of atoms (see Figure 4-18):

$$V_{ZBL}(r) = \frac{Z_1 Z_2 e^2}{4\pi\varepsilon_0 r} \sum_{i=1}^{4} c_i \exp(-d_i \frac{r}{a}) \qquad \text{Equation 4-4}$$

where: c_1= 0.2817, d_1=0.20162,
c_2= 0.2802, d_2=0.40290,
c_3= 0.50986, d_3=0.94229,
c_4= 0.18175, d_4=3.19980

and

$$a = \frac{0.8854 * .529}{Z_1^{2/3} + Z_2^{2/3}} \qquad \text{Equation 4-5}$$

Taking the classical approach of treating the two-body collision as a central-force field scattering in the center-of-mass coordinate frame, one can give the scattering angle[29]

$$\Phi = \pi - 2 \int_{r_{min}}^{\infty} \frac{p \cdot dr}{r^2 \sqrt{1 - \frac{V(r)}{E_c} - \frac{p^2}{r^2}}} \qquad \text{Equation 4-6}$$

where p is the impact parameter which defines whether the collision is head-on or merely glancing.

The difficulty of using the universal potential directly in this calculation arises from the lack of an analytic solution to the integral. Ziegler et al present an empirical formula called the 'Magic Formula' that produces acceptable results to the integration of the universal potential formula[29].

$$\cos(\frac{\Phi}{2}) = \frac{B + R_c + \Delta}{R_o + R_c}$$ Equation 4-7

where R_o is the distance of closest approach defined as the solution of the scattering equation integrand denominator:

$$\left[\left(\frac{p}{r_0}\right)^2 + \frac{V(r_0)}{E_c}\right] - 1 = 0$$ Equation 4-8

where:

p is the impact parameter.
E_c is the center-of-mass energy.

$$B = \frac{P}{a}$$

Equation 4-9

$$R_c = \frac{\rho}{a}$$

a is the screening length.
Δ is a correction term
ρ is the radius of curvature of the two particle trajectories at the point of closest approach and is calculated from:

$$\rho = -2\frac{E_c - V(r_0)}{\frac{d}{dr}V(r_0)}$$

$$\Delta = A\frac{R_0 - B}{1+G}$$

$$A = 2 \cdot \alpha \cdot \varepsilon \cdot B^{\beta}$$

$$G = \gamma \big/ (\sqrt{1 + A^2} - A)$$

$$\alpha = 1 + C_1 \big/ \sqrt{\varepsilon}$$

$$\beta = \frac{C_2 + \sqrt{\varepsilon}}{C_3 + \sqrt{\varepsilon}}$$

$$\gamma = \frac{C_4 + \varepsilon}{C_5 + \varepsilon}$$

Equation 4-10

with C_1 to C_5 being fitting coefficients to the interatomic potential selected:

$C_1 = 0.9923$

$C_2 = 0.01162$

$C_3 = 0.007122$

$C_4 = 9.307$

$C_5 = 14.81$

and ε is the center-of-mass energy E_c divided by $Z_1Z_2e^2/a$.

This Magic Formula's strength is that it can be used to calculate the scattering angle for a wide range of interacting species with acceptable results. The magic formula provides the basis for Thermspud1's scattering model.

The concept of a collision cross section, or effective collision area, captures the probability density of an in-transit particle colliding with a gas species. The collision cross section is given by:

$$\sigma = 4 \cdot \pi \cdot r^2 \qquad \text{Equation 4-11}$$

where $2 \cdot r$ is the distance at which a meaningful interaction occurs between two particles.

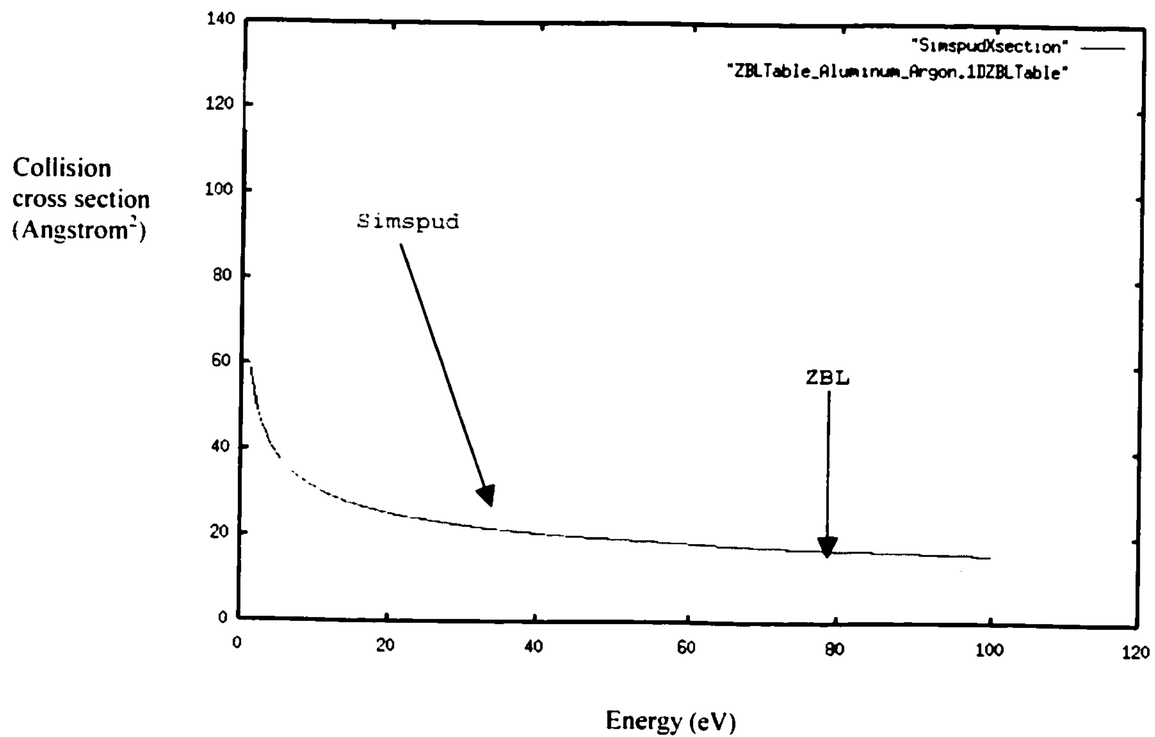


Figure 4-17: ZBL sigma vs. energy

This figure shows the dependence of collision cross-section on energy in the regime of interest to sputtering according to the ZBL model, in comparison to the experimentally calibrated SIMSPUD model.

Image removed due to copyright restrictions. The image depicts the correlation between the magic formula and the integral of the universal potential formula. (see [30])

Figure 4-18 Magic formula fit with integrated universal potential[30]

"The scattering of two atoms can be reduced to a function of the [reduced] collision energy [ε] and the impact parameter. The final scattering angle [Φ] can be calculated with the integral orbit equation [...] A greatly simplified analytic formula [is] called the Magic Formula for scattering. Shown above with small circles are the solutions of the complete orbit equation evaluated using the universal screening function [...] Also shown as solid lines are the results of fitting the detailed scattering points with the Magic Formula [.]" [30]

The Magic Formula can thus be used to calculate the collision cross-sectional area. If we define a meaningful interaction between particles as one in which a scattering deflection greater than 2° results, one can calculate the cross-sectional area by solving the Magic Formula for this value. By solving this value for all interaction sets at various energies, one can produce lookup tables of collision cross-section as a function of energy. This model incorporates the effects of long-range interactions that are ignored by the hard-sphere model currently employed in SIMSPUD. In Figure 4-17 one sees the dependence of the relative cross section on energy.

Completion Modules

It is assumed that during Thermspud1's execution phase, other external modules are collecting data as a side effect of particle interaction with the simulation environment. To keep the coupling with these modules to a minimum, Thermspud1 will continue iterating until one of these external modules notifies Thermspud1 that some completion criteria or convergence has occurred. This architectural pattern can be referred to as the Observer design pattern. This feature abstracts the total number and knowledge of the completion modules away from Thermspud1 and thus reduces inter-module coupling.

4.4 Software Support Systems

Regression Testing

In the code tree, each module directory has a tests subdirectory. To provide some manner of regression testing, most classes have a test driver. To aid the future developer, a template for test drivers is provided. This driver template driver.cpp has the class name "XXXX" to be tested. The driver compares result files token by token with doubles being equal if they are within Constants::EPS of each other. Thus if the developer creates tests from which output results, the driver file will compare the results of a current test with the results of a previous test (if the results exist) or else put the results in the results directory. This provides a test harness for all files created. If a developer chooses to modify existing classes, the test driver for this class is run to see if any functionality has been broken. Once it has been deemed that the original functionality is still valid, the

developer can add new tests to verify the changes made. This method of test/validation is called regression testing. For example, if a developer were to work on Plaspud1, the Plaspud1/Plaspud1.cpp would be edited; the Plaspud1/tests/Plaspud1Test.cpp would be used to test the modifications to the Plaspud1.cpp file. This driver file (Plaspud1Test.cpp) is currently in the Plaspud1/tests directory, and any output should be in the current Plaspud1/tests/results directory. Any class that is 'owned' by Plaspud1 would also reside in the Plaspud1/ directory. These 'owned' classes would have regression tests in the Plaspud1/tests directory. Makefiles have been supplied in the tests directories to make this possible.

The Automated Code Build System

To ensure timely feedback to the developer, an automated build system was developed. The entire framework is built and tested nightly with the automated code build system. This system consists of a series of Perl and bash scripts that keep the state of the code base visible at all times. The scripts 'cvs checkout' the code from the repository each night. This check is always conducted from a clean directory, so the state of a clean build is tested each night. The scripts then proceed to build the module libraries and a specific XML-specified application binary. The output of the build process is captured. If an error occurs, the error text is stored on an html document. If the tree builds successfully and the binary runs without problems, the entire set of regression tests is run. Each regression test has specific output that indicates successful execution or which tests fail. A Perl script is used to parse out just the errors of the regressions tests. These errors are stored in an html document.

After the build scripts are run, the code, binaries, and libraries are deleted. The error documents are then checked into the documentation repository. The documentation repository is mirrored nightly on the development web site. A detailed account of the build process is always displayed on the development web site. A sample of the web output is as follows:

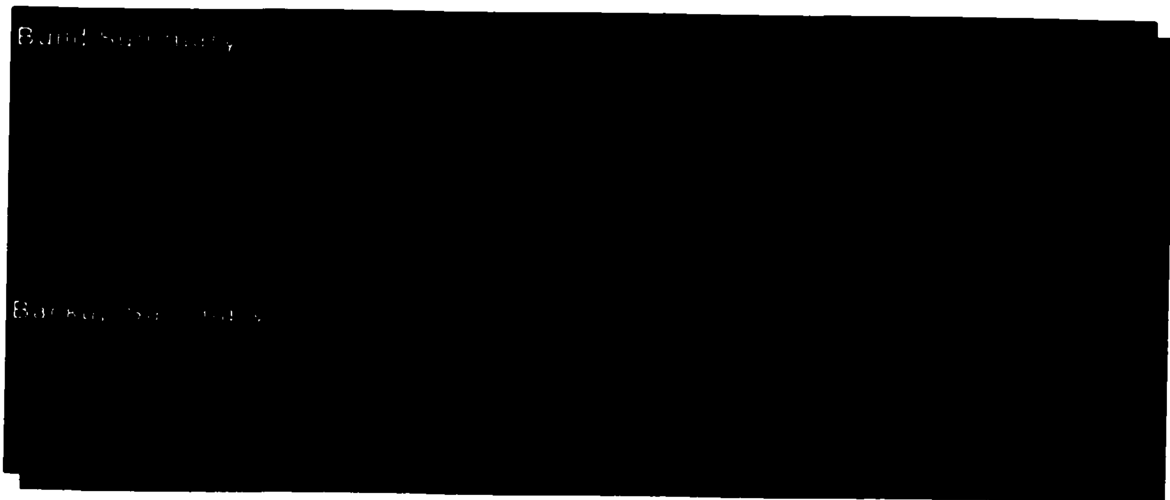


Figure 4-19: Build results

Results of the nightly build are displayed on the project web page. The code is built, tested and backed up to CDrom by an automated system.

Bug Tracking

To give the developer a sense of the state of the software system, a bug-tracking package was set up on the development web site. This bug-tracking package uses an SQL database to store bug data. This allows for assigning responsibility to software problems, as well as for keeping software problems from being forgotten. When a person finds a new bug, he/she logs it into the database. An Email is then sent to every developer on the developer mailing list. Once the bug has been resolved, appropriate emails are sent to the people involved. This keeps all developers current on the state of the software package.

The package also provides a front end for storing and presenting news items to the development group. This provides a record of the communications and development of the software.

5. Problem Domain: Results

5.1 ObstructionProcessor

On of the key services of the ObstructionProcessor is to provide physical stopping of particles. Figure 5-1 and Figure 5-2 demonstrate a pipe obstruction that has collected particle impacts.

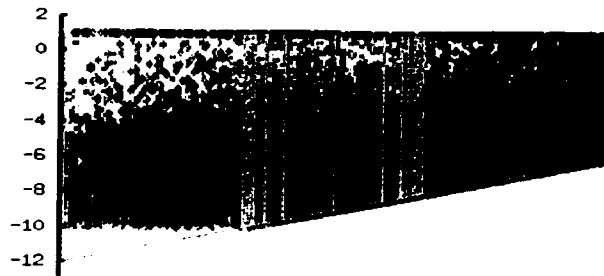


Figure 5-1: Particle stopping of a pipe with a tilted cap (side view)

This side view of a pipe obstruction shows particles collecting upon the surfaces. This pipe shows the cap tilted. This demonstrates not only the functionality of the pipe obstruction, but its ability to stop particles.

The ObstructionProcessor is able to output visual data to aid in the interpretation of the collected results. Although the curved surfaces appear to be created out of several planar surfaces in the visualizations, this is only an artifact of the visualization algorithm. The surfaces actually stop particles and collect data as true curved surfaces. Looking at the top view of the pipe obstruction in Figure 5-2, one can see the particles actually stop on the curve of the surface. In Figure 5-3, we see that the particle stopping stays consistent with translated and rotated obstruction geometry.

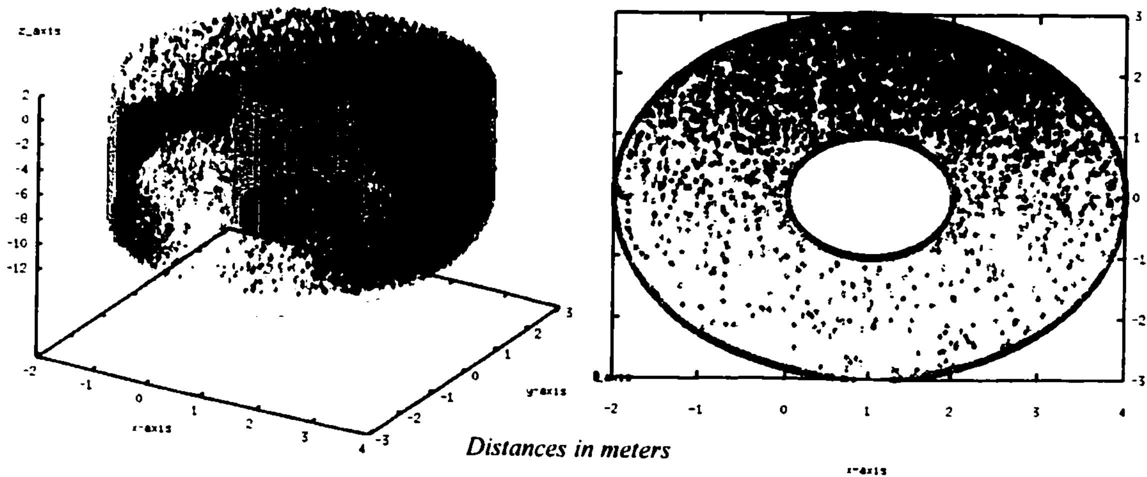


Figure 5-2: Particle stopping of a pipe with a tilted cap (oblique and top view)

This oblique and top view of a pipe obstruction shows particles collecting upon the surfaces. It is important to note that the segmenting of the curved surfaces as shown in the diagram is only an artifact of the visualization algorithm. As one can see from the top view, the segmentation on the pipe is not replicated in the particle collections.

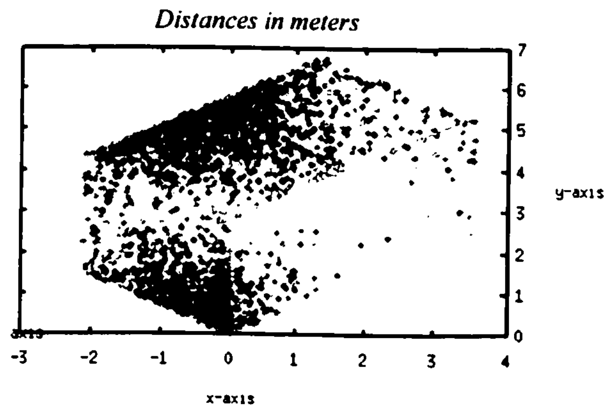


Figure 5-3: Particle stopping of a box

This top view of a rotated and tilted box demonstrates that translated and rotated obstructions accurately stop particles in relation to their geometry.

5.2 Thermspud1

The Thermspud1 algorithm entails tracking the life of a particle, from a particle source to surface or volume deposition. Typically, the particle source will be a target. Currently the target is merely a Surface that has an attached ParticleGenerator, which is capable of generating particles either by using an externally provided Histogram or by building a Histogram from an ion bombardment phase. Figure 5-4 and Figure 5-5 give views of a chamber, target, and substrate structure, as well as a view of a tracking of generated particles. One can see that the particles emanate from the top of the lower internal cylinder.

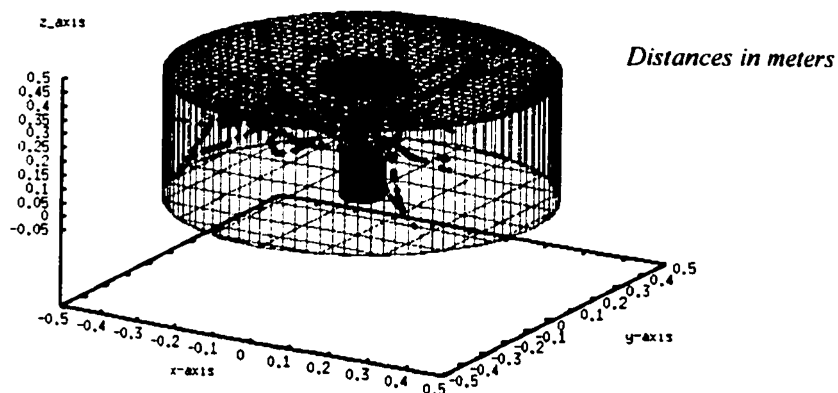


Figure 5-4: Particle paths (oblique view)

This is a visualization of a chamber setup. The lower, inner cylinder is a target object. The upper inner cylinder is a substrate object. The outer cylinder represents the chamber that bounds the simulation.

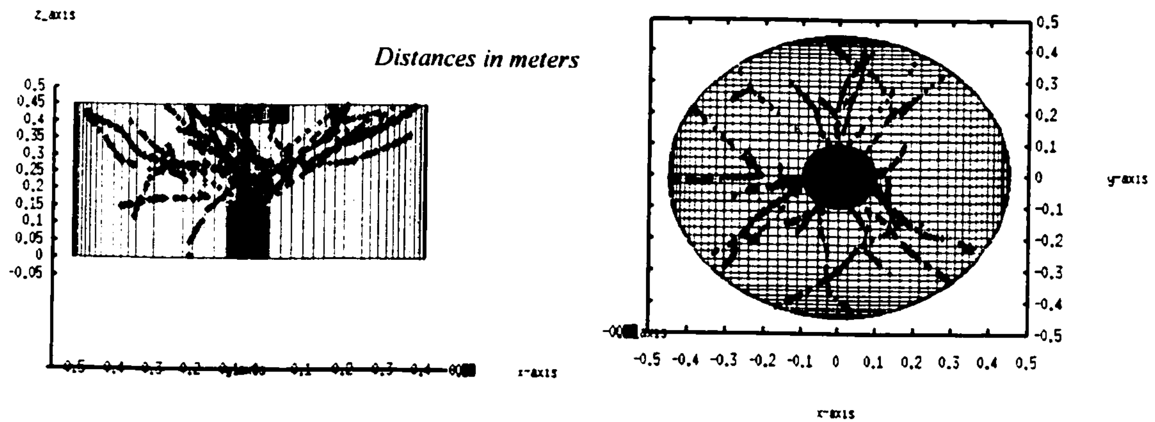


Figure 5-5: Particle paths (side and top view)

This is a side and top view of Figure 5-4. The particles originate from the lower cylinder (the target) and proceed until they deposit thermal energy into the gas or deposit onto a surface.

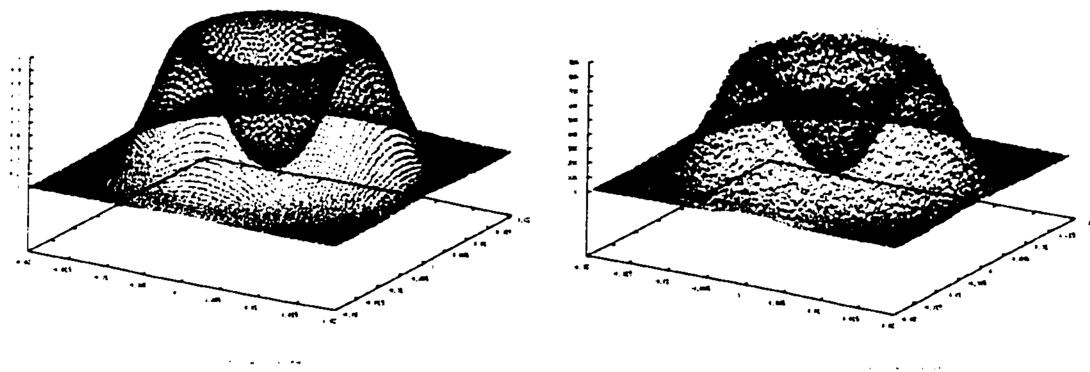


Figure 5-6: Particle generation from a histogram

The picture on the right is a distribution of all particles generated from a 20 cm target surface. The target surface's ParticleGenerator generated particle positions using the erosion profile on the left.

The left hand graph in Figure 5-6 shows the histogram used as an erosion profile to govern the random position generation on the target surface. The right hand graph of Figure 5-6 depicts the positions of generated particles using the erosion profile. The pattern, as expected, follows the outline of the erosion profile very closely.

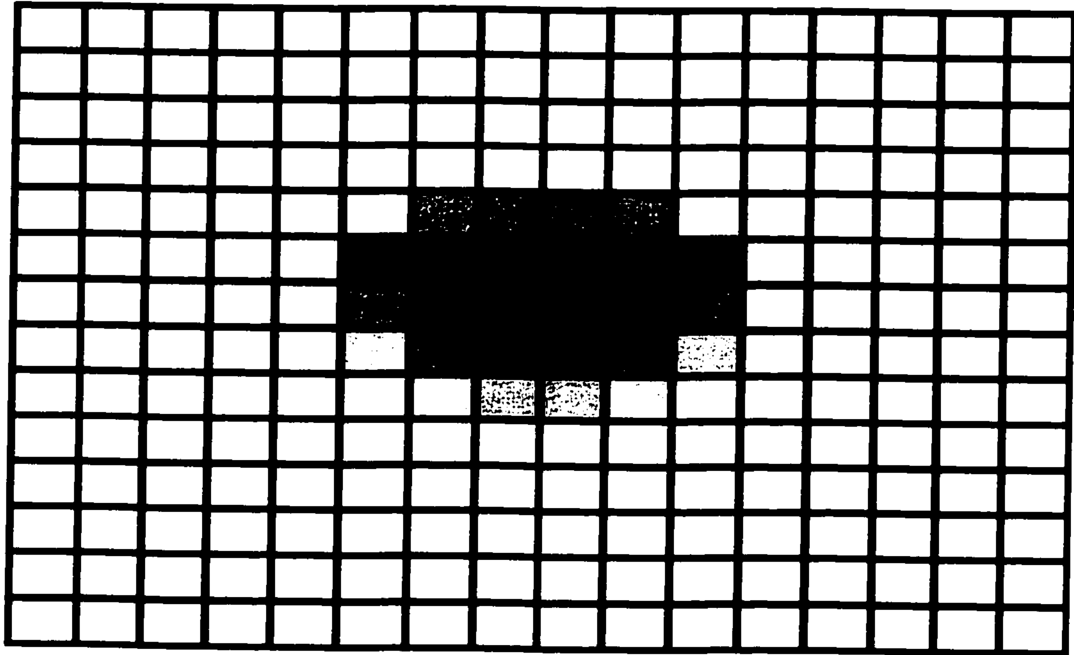


Figure 5-7: Thermal data slice

As particles fall below a predetermined energy threshold, the particles deposit thermal energy into a thermal grid. Slices of this grid can be output in postscript form for analysis. The thermal grid depicted here shows extreme heating above the target region (shown as dark cells). One can also see the shadowing effects of the substrate and the target objects. The grid image has been cropped to include only the cells that are within the chamber volume.

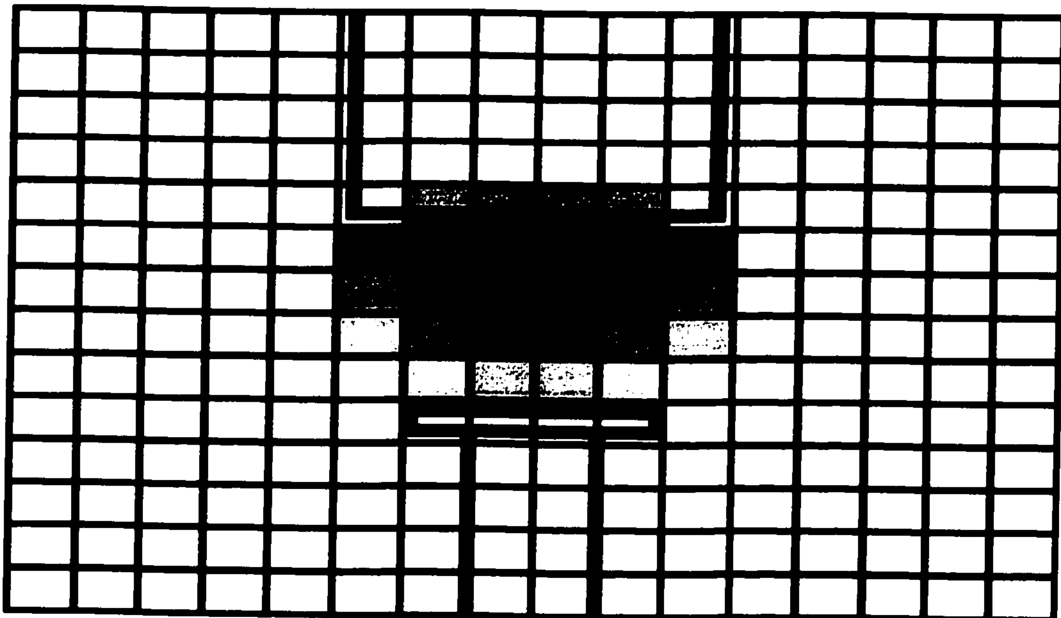


Figure 5-8: Thermal slice with obstruction overlay

This shows the positioning of the obstructions with respect to the energy deposit of Figure 5-7. This demonstrates the shadowing effects of energy deposition. Note the target and substrate partially fill cells. Thus there appears to be heat collected within the target and substrate (which is not the case).

In the Thermspud1 algorithm, when a particle falls below an externally set energy thermalization threshold (typically 1 eV), the particle's remaining energy is deposited into a thermal grid (this information is required by the Thermspud2 continuum thermalized transport model for further calculation). Figure 5-7 is a profile of the thermal deposition that results from slicing the thermal grid of a chamber setup similarly to that in Figure 4-10, through the XZ plane at y=0. Figure 5-8 adds the outline of the target and substrate to the image of Figure 5-7 to demonstrate effects of shadowing. This diagram was created as a result of thermalizing 30,000,000 energetic and gas particles into the chamber volume. Looking at the darker cells, which represent higher energy deposition, one can see the high number of particles that deposit energy into the thermal grid just below the target. This result is reasonable considering the higher probability of particles leaving the target at low energies (see Figure 5-9). Thermspud1's path algorithm follows a Poisson process, which, by definition, indicates that a particle has an exponentially decreasing probability of a long path length. This translates to a low number of high-energy particles that are able to travel out of the target area, and either deposit into the volume or onto a surface. The dependence of the collision-cross sectional area on energy, as depicted in Figure 4-17, also lends support to low energy particles having a lower probability of traveling a great distance. The probability of a particle colliding with a gas molecule is significantly high. Most collisions involve an energy transfer from the energetic particle; thus there will be a tendency for the gas to be heated near the target region, as observed.

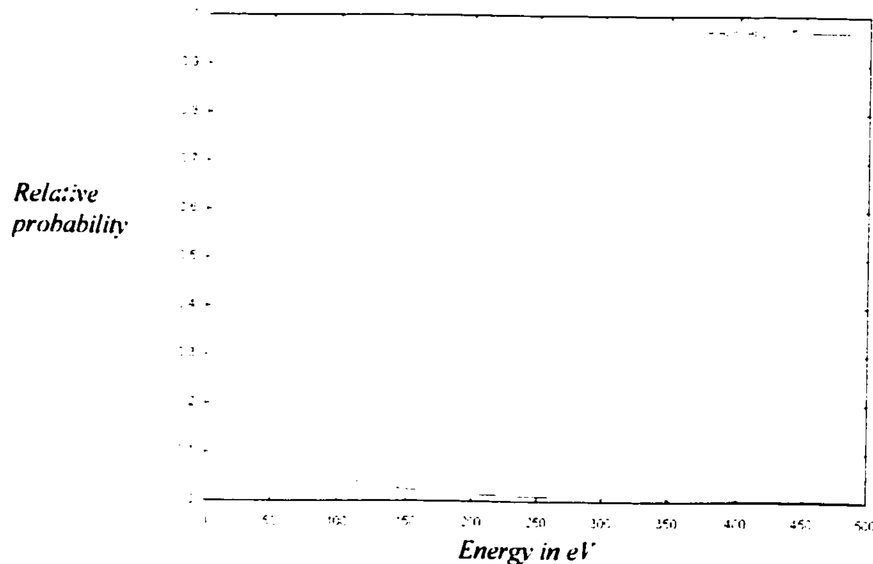


Figure 5-9: Energy distribution

This energy distribution (based upon the Thompson model) was used for particle generation at the target.

The deposition rate affects throughput as well as film quality (see Figure 5-10). Using Thermspud1, one examines the relationship between deposition rate and target-substrate distance. Using a 5 mTorr initial argon pressure, a cosine angular distribution for particle trajectories, the annular erosion profile depicted in Figure 5-11, and an energy distribution based upon the Thompson model[31] (see Figure 5-9), it was possible to tabulate relative flux results. The 20 cm aluminum target and 15 cm substrate used in the simulation depicted in Figure 4-10. The flux rate results were collected on a 2 cm radius circular CollectionRegion at the center of the substrate. These results are compared with the results from the same experiment conducted using SIMSPUD[28]. Although Thermspud1 is capable of calculations using energetic gas particles, tests were conducted using stationary gas particles. SIMSPUD assumes gas particles are stationary, thus the same conditions must be used to compare the output of SIMSPUD and Thermspud1.

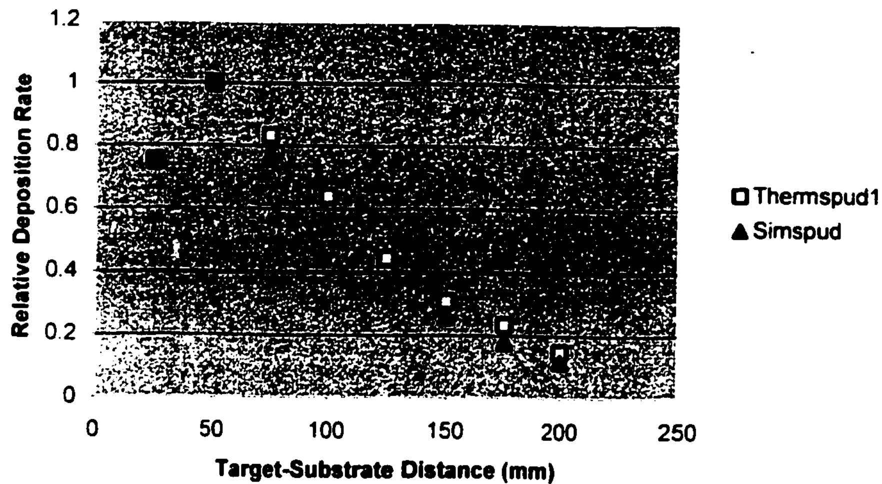


Figure 5-10: The effects of target-substrate distance on relative deposition rate
The effects of target-substrate distance are examined using Thermspud1 and SIMSPUD.

Initially, one observes an increase in relative rate with distance due to the little flux that makes it to the center of the substrate from the erosion ring at close distances. As the substrate-target separation is increased further, a non-linear decrease in deposition flux density is observed. As the substrate moves away from the target, more particles are lost to the chamber walls or lose energy due to collisions with the background gas and thus do not make it to the substrate. One can see that Thermspud1 output correlates closely to the SIMSPUD output. Some differences can be attributed to the physical chamber setup, but the different scattering model is an important factor. The hard-sphere model SIMSPUD uses tends to overestimate the likelihood of large scattering. Also, Thermspud1, as mentioned, uses a model that represents a target and substrate as physical three-dimensional entities as opposed to SIMSPUD's two-dimensional region on the chamber ends. Particles that cross the chamber elevations that correspond to either the target or substrate are removed from the simulation in SIMSPUD. These particles continue being

processed in Thermspud1. As the target-substrate separation becomes greater, these contributions become more pronounced.

Target-substrate separation also has an effect on the substrate coverage uniformity. Uniformity is a concern in microelectronics fabrication because deviations in thickness can reduce yield and reliability. Looking at Figure 5-12, one sees as the target-substrate separation is increased, the predicted uniformity initially increases, but then starts to fall off. It appears that, for the stated specifications, the optimal target-substrate separation is near 60 mm. This result is also predicted by SIMSPUD (see Figure 5-13).

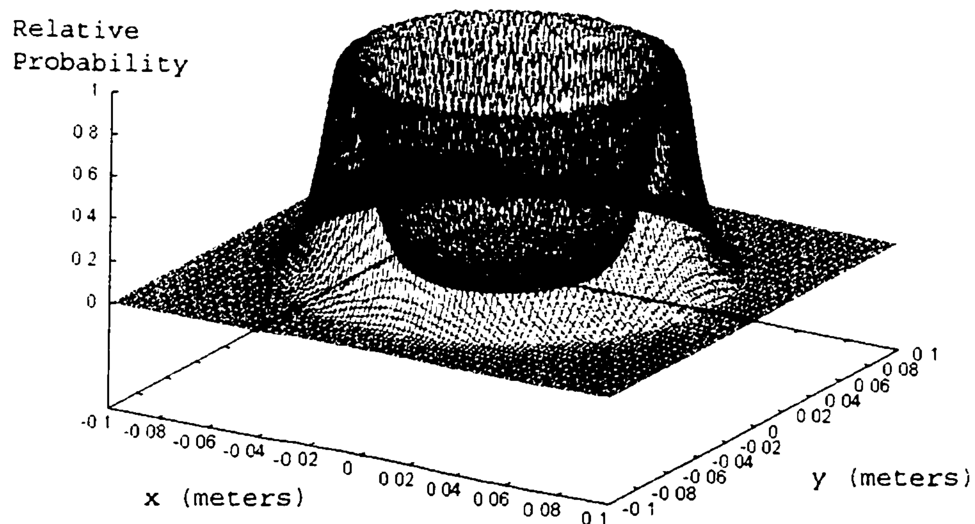


Figure 5-11: Erosion profile

This erosion profile was converted from a SIMSPUD 2D profile in order to do comparisons with Thermspud1.

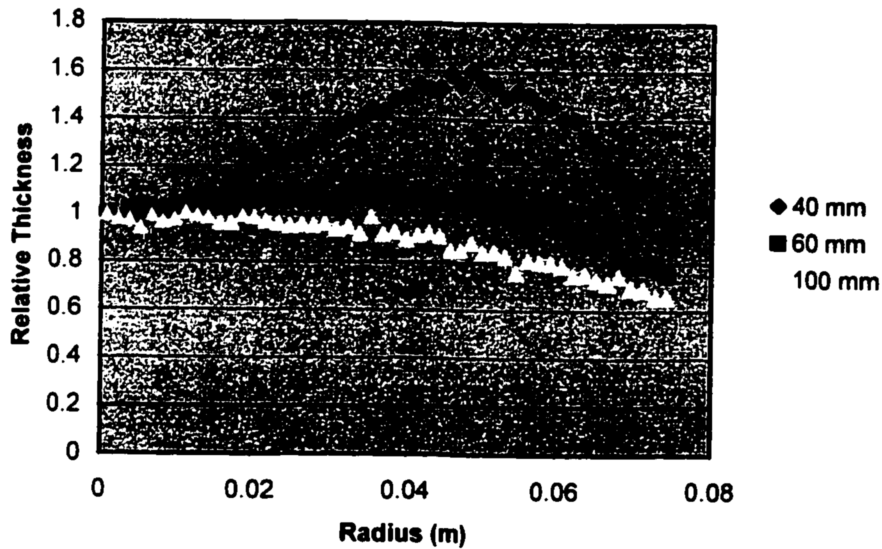


Figure 5-12: Radial profiles

This figure depicts the radial film thickness profiles for a 20 cm magnetron source at 5 mTorr for three different target-substrate separations across a 15 cm wafer, in relation to the target erosion profile. The values are normalized to the wafer center.

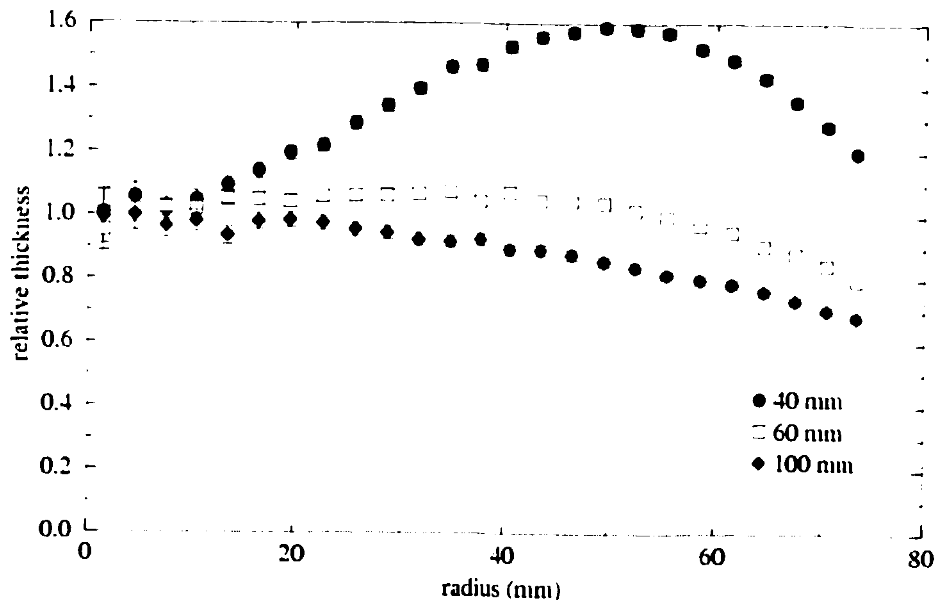


Figure 5-13: SIMSPUD radial profiles[28]

In regards to substrate coverage uniformity, SIMSPUD predicts the ideal target-substrate separation is near 60 mm.

Another quantity that varies with target-substrate separation is the average energy of incident atoms. Using the assumptions and specifications of the previous tests, the average energy deposited on the substrate using Thermspud1 was compared to output using SIMSPUD. Figure 5-14 shows similar trends between the SIMSPUD and Thermspud1 results. However, Thermspud1's use of the ZBL scatter model, as mentioned, results in less energy lost during collisions, thus resulting in a higher average energy being retained.

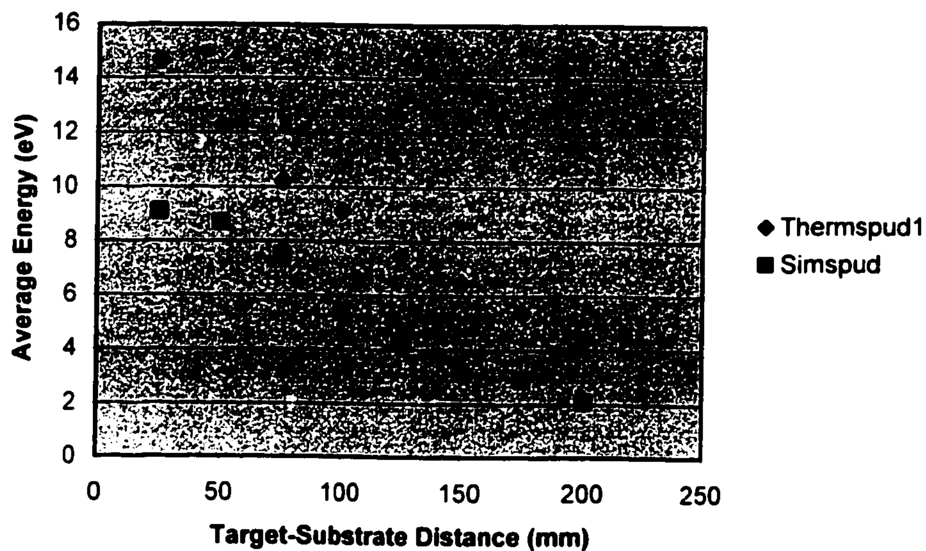


Figure 5-14: The effects of target-substrate distance on average deposited energy.
The average energy decreases with distance as the average number of collisions with gas molecules increases.

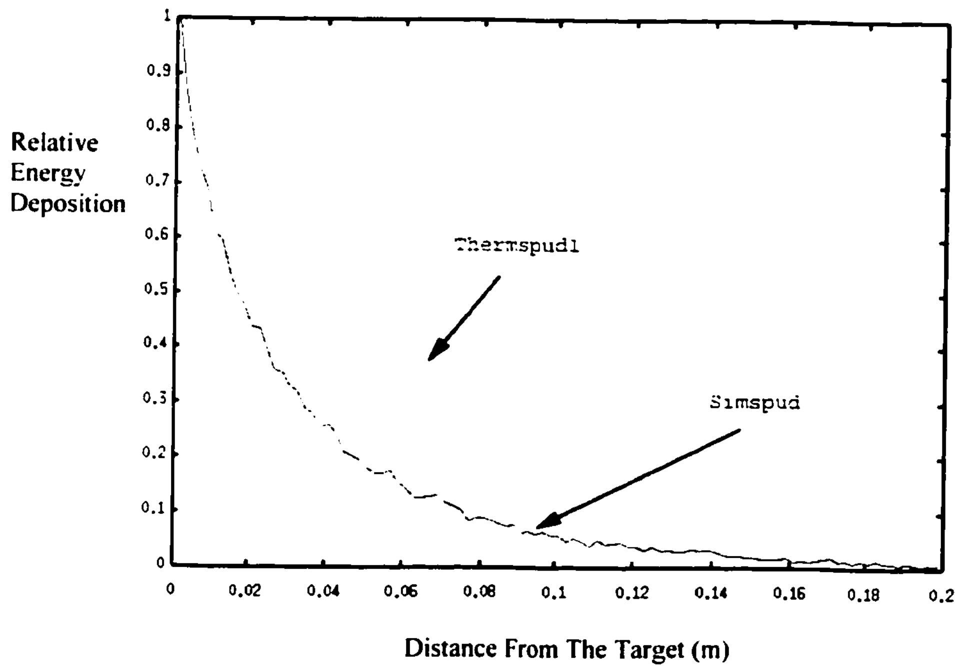


Figure 5-15: Energy deposition profiles

This figure compares Thermspud1 and SIMSPUD'S energy deposition profiles. As a result of particle-gas collisions, energy is transferred from particles into the background gas.

An important difference between SIMSPUD and Thermspud1, as mentioned, is SIMSPUD's use of the hard-sphere model. The overstatement of scattering angles by this model has an effect on the energy transferred during collisions. More energy is lost per collision in the SIMSPUD algorithm, thus particles have less energy to deposit further away from the target. This accounts for the differences observed in Figure 5-15.

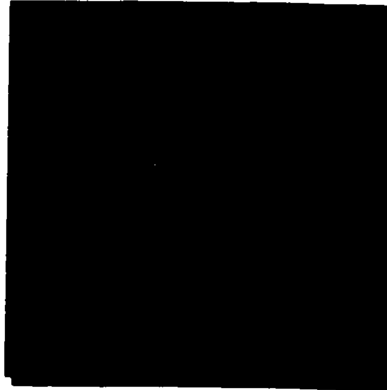


Figure 5-16: Refined grid

The grids that are part of the framework core have the ability to adapt to the problem domain. This grid has the lower right cell refined into 4 additional cells. The lower left cell of the refined quadrant has also been refined. There is no algorithmic limit on the level of refinement any cell can have.

One of the key facilities of the framework is the adaptable grid. These template containers provide the user with a very flexible solution to physical property discretization. Looking at Figure 5-16, we see an example of this type of container. All cells have the ability to divide into 8 sub-cells. Figure 5-16 depicts a 2-D view of a 3-D grid; thus each refined cell appears to only divide into 4, but is actually refined into 8 cells. Each of these, in turn, can be further divided.

6. Conclusions and Recommendations

6.1 Summary

This thesis presented an argument for creating a framework for reactor scale PVD simulation. This led to the development of SIMSPUDII, a product that not only is a study in framework creation and design, but also provides the PVD research community with a core foundation for future simulation development in this problem domain.

The encompassing framework was built using object technologies to ensure long-term growth and evolution of the product. A system of runtime application specification and creation using XML was presented and demonstrated. As well, this study presents a feature-rich message routing facility that introduced a method for utilizing multiple-mapped iostreams. This framework combined with detailed documentation and software support systems, provides the future developer with a strong foundation upon which to build.

After presenting a case for the creation of a framework for this problem domain, a test of the viability and extensibility of the framework was conducted by implementing two key modules in the problem domain: Thermspud1, a module that encompasses a more sophisticated simulator than the industry standard reactor scale simulator SIMSPUD, and the ObstructionProcessor, a highly configurable physical modeling and data collection module. Through implementation of these two modules, the framework has proven to be

a robust solution that provides the necessary abstractions and facilities to make extension of this framework straightforward.

In the simulations that were presented for this thesis, Thermspud1 produced results comparable to SIMSPUD. The predicted relationships between target-substrate separations were quite similar. Although Thermspud1 stops processing particles below a thermalization threshold, the difference can be removed for comparison by setting the thermalization threshold to zero. One can argue that when Thermspud2 (the thermalization ProcessModule) is implemented, the effects of particle thermalization will produce a more realistic Thermspud1 output. Although Thermspud1 currently takes into account the per-cell variances in gas density and temperature, the lack of a thermalization-processing module (Thermspud2) leaves this dimension unexploited. Results presented in this thesis show that Thermspud1 is capable of depositing thermal energy within a discretized chamber volume, a capability that is not present in SIMSPUD. Thermspud1's more realistic particle-particle interaction system (ZBL based), combined with the ObstructionProcessor for complex physical chamber geometry, provides the basis for a highly configurable neutral particle transport model.

The intended use of this framework is as a tool for exploring the effects of reactor scale variables on feature scale properties of PVD processes. The long-term vision for this product is to become a feature-rich simulator capable of simulating the plasma, the target, and the transport through the gas with rarefaction and heating, and film growth considering the effects of all of these. The modules presented and the software systems in

place are the first steps along this long path to this vision's fulfillment. A framework, by definition, is a work in progress. Despite this, the software is already capable of producing results that can be combined with feature scale simulators like SIMBAD to explore feature scale film phenomenon. SIMSPUDII has the capability of simulating target particle generation, and neutral particle transport through a background gas that is dynamically heated, as well as possessing the facilities to collect energy, positional, and angular distributions on any surface through a sophisticated obstruction module.

At the time this thesis was written, the framework consists of roughly 150 classes, 300 files, over 40,000 lines of code, and 30,000 lines of comments. The code is built and tested nightly by an automated build system. The software package has a revision control system, a regression test harness, a detailed Application Programming Interface (API), and a defect tracking system. A comprehensive web site provides future developers with extensive design and coding aids, reference manuals, development guides, and a history of the framework development process. Specifically, the web site offers tutorials on how to add new ProcessModules, ExecutionModules, and how to extend the XML parameter space. The software is automatically backed up to compact disk weekly.

6.2 Future Work

One of the goals stated in the introduction addressed program efficiency. Throughout the development of the framework of this work, extensibility and form received higher

priority than computational efficiency. It was felt that if program correctness could be achieved, efficiency could be attained with a code optimization process. It is a less involved task to optimize a well-designed product than to redesign an efficient product. More use of C++ inline functions and computational shortcuts could have made the current code run faster. The use of profiling tools to identify code bottlenecks would probably be prudent. Now that the framework foundation has been created, with a solid test harness to validate program state, a developer could spend time focusing on optimizing runtime behavior.

Another issue that could be streamlined is the concept of module 'ownership' discussed earlier. The current solution places the responsibility of memory management on the developer who adds modules to a DOM_DocumentProcessor. Instead of the assignment of ownership to the module that has the IUnknown interface, a more robust method of 'smart pointers' could be implemented. Microsoft's COM assigns memory management responsibility to the dynamically linked library. When the last reference to the object is deleted, the object removes itself from memory. This solution would simplify the memory management issues, but may add a level of complexity in the overall framework. The current solution works well, only requiring that the framework developer be aware of the memory management issues.

The framework, at the time this thesis was written, contains an extensive API documentation system. All code is well documented and a tool is currently used to extract the documentation and place it on the web page. Some software templates, 'howtos', and

a FAQ are provided for future developers. Although a great deal of documentation is provided, not enough documentation exists that describes how to interact with and extend the framework. This thesis provides a starting point for future developers, but the package would benefit from a well-written development guidebook. A user's manual on how to use the software is also an essential addition.

Although the use of XML has greatly simplified the framework design and offers great flexibility in future extensions to the framework, the amount of data in the XML file is very large and will definitely grow. A graphical user interface (GUI) to hide the complexities of the large XML file would be a great asset to the product.

A dedicated graphical application-building tool would aid the end user in creating a tailored application. As mentioned, the framework employs a runtime application specification system using XML. It would not be difficult to create a visual application builder. Relationships between modules could be achieved by dragging links. Removing the ID management from the user would simplify the application creation procedure.

Another application of a GUI that would benefit the project would be a visual chamber specification tool. All objects are currently specified via XML. A simple graphical tool that allows visualization and modification of the obstructions would aid the end user in verifying whether the simulation is indeed set up correctly. The tool could create the ObstructionProcessor node in the XML document.

Finally, as mentioned, Thermspud1's full potential is currently untapped. A proposed thermalization module (currently referred to as Thermspud2) will have an effect on the results of Thermspud1. Although Thermspud1 takes into account all per-cell differences in gas temperature and density, calculating these dependencies relies on modules being developed by others (i.e. Thermspud2). All tests in this thesis use a homogeneous gas density and temperature. When a thermalization module is completed, one will be able to observe what effect this has on Thermspud1's results.

Bibliography

[1] Michael Riordan. Lillian Hoddeson. *Birth of an Era*, Scientific American. Solid-State Century. pp. 10

[2] Gordon E. Moore, *Cramming more components onto integrated circuits*, Electronics, Vol. 38. No. 8

[3] http://www.bccresearch.com/editors/RGB-186E.html

[4] Loran J. Friedrich. Steven K. Dew. Michael J. Brett. and Tom Smy. *A Simulation Study of Copper Reflow Characteristics in Vias*, IEEE Transactions on Semiconductor Manufacturing. Vol. 12. No. 3. August 1999

[5] Tom Smy. Liang Tan. K. Chan. R. N. Tait. James N. Broughton. Steven K. Dew. and Michael J. Brett. *A Simulation Study of Long Throw Sputtering for Diffusion Barrier Deposition into High Aspect Vias and Contacts*. IEEE Transactions on Electron Devices, Vol. 45. No. 7. July 1998

[6] Ouyang. M.X.. Kinney. L.D., *Onyiriuka. E.C.. Computer simulation of Nb2O5/SiO2 sputtering process for narrow band optical filter*. Fourth International Conference on Thin Film Physics and Applications, Vol. 4086. pp. 458-61

[7] http://banyan.cie.rpi.edu/~evolve/

[8] Peter L. O'Sullivan. Frieder H. Buamann. Geourge H. Gilmer. *Simulation of Physical vapor deposition into trenches and vias: Validation and comparison with experiment*. Journal of Applied physics. Vol. 88. No.7. October 2000

[9] R. Alan McCoy. Yuefan Deng. *Parallel Particle Simulations of Thin-Film Deposition*. International Journal of High Performance Computing Applications. Vol. 13, No.1. Spring 1999. pp. 16-32

[10] E. Suzuki. Y. Hoshi. *Mechanism of Composition Change in Sputter Deposition of Barium Ferrite Films with Sputtering Gas Pressure*. Journal of Applied Physics. Vol. 83. No. 11. June 1998

[11] Shenglong Zhu. Fuhui Wang. and Weitao Wu, *Simulations of reactive sputtering with constant voltage power supply*. Journal of Applied Physics, Vol. 84. No. 11, December 1998

[12] W. Pyka. S. Selberherr, V. Sukharev, *Incorporation of Equipment Simulation into Integrated Feature Scale Profile Evolution*.
http://www.iue.tuwien.ac.at/pdf/2001/V378_Pyka.pdf

[13] Michael A. Vyvoda, Cameron F. Abrams, David B. Graves, *Feature Evolution Simulation of I-PVD Copper Films*

[14] Richard Lajoie, Rudolf K. Keller, *Design and Reuse of Object Oriented Frameworks: Patterns, Contracts and Motifs in Concert*, Centre de recherché informatique de Montreal

[15] Garry Froehlich, H. James Hoover, Ling Liu, Paul G. Sorenson, *Reusing Application Frameworks through Hooks*, Froehlich/Reusing Application Frameworks Through Hooks AF1051-1

[16] Garry Froehlich, H. James Hoover, Ling Liu, Paul G. Sorenson, *Designing Object-Oriented Frameworks*, CRC Handbook of Object Technology, CRC Press, 1998, in press.

[17] Julian C. Cummings, James A. Crotinger, Scott W. Haney, William F. Humphrey, Steve R., Karmesin, John V.W. Reynders, Stephen A. Smith, Timothy J. Williams, *Rapid Application Development and Enhanced Code Interoperability using the POOMA Framework*, SIAM Workshop on Object-Oriented Methods and Code Inter-operability in Scientific and Engineering Computing: OO98

[18] http://www.acl.lanl.gov/pooma/index.html

[19]M. Griebel, G. Zumbusch, *Parallel multigrid in an adaptive PDE solver based on hashing.* , Proceedings of ParCo '97, 1998, pp. 589-599.

[20] S.M. Rossnagel, *"Sputtered Atom Transport Processes"*, IEEE Transactions on Plasma Science" , Vol. 16, No. 6, 1990, p. 878.

[21] Garry Froehlich, H. James Hoover, Ling Liu, Paul G. Sorenson, *Hooking into Object-Oriented Application Frameworks*

[22] John V. W. Reynders, David W. Forslund, Paul J. Hinker, Marydel Throuburn, David G. Kilman, William F. Humpfrey, *OOPS, An Object-Oreinted Particle Simulation Class Library for Distributed Architectures*

[23] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, Inc, Englewood Cliffs, NJ (1990)

[24] A. A. R. Cockburn, *The Impact of Object Orientation on Application Development*, IBM Systems Journal, Vol. 38, Nos. 2&3, 1999

[25] Robert C. Martin, *Design Principles and Design Patterns*, www.objectmentor.com

[26] John R. Cary, Svetlana G. Shasharina, Julian C. Cummings, John V. W. Reynders, Palu J. Hinker, *Comparison of C++ and Fortran 90 for Object-Oriented Scientific Programming*,
http://www.amath.washington.edu/~lf/software/CompCPP_F90SciOOP.html

[27] http://www.gnuplot.info/

[28] Steven K. Dew, *Processes and Simulation for Advanced Integrated Circuit Metallization*, Doctoral Thesis, 1992

[29] J.F. Ziegler, J.P. Biersack, U. Littermark, *The Stopping and Range of Ions in Solids*, Vol. 1, New York: Pergamon Press, 1985

[30] J.F. Ziegler, J.P. Biersack, U. Littermark, *The Stopping and Range of Ions in Solids*, Vol. 1, New York: Pergamon Press, 1985, pp. 59

[31] M.A. Vidal, R. Asomoza, *Monte Carlo Simulations of the Transport Process in the Growth of a-Si:H Prepared by Cathodic Sputtering*, Journal of Applied Physics, Vol. 67, No. 1, 1990, p. 477.