FINITE-ELEMENT METHODS FOR PROCESS SIMULATION APPLICATION TO SILICON OXIDATION

by

Pantas Sutardja

Memorandum No. UCB/ERL M88/26

2 May 1988

COVER PAGE

FINITE-ELEMENT METHODS FOR PROCESS SIMULATION APPLICATION TO SILICON OXIDATION

by

Pantas Sutardja

Memorandum No. UCB/ERL M88/26

2 May 1988

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

FINITE-ELEMENT METHODS FOR PROCESS SIMULATION APPLICATION TO SILICON OXIDATION

by

Pantas Sutardja

Memorandum No. UCB/ERL M88/26

2 May 1988

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Finite-Element Methods for Process Simulation Application to Silicon Oxidation

Ph.D. Pantas Sutardja EECS Department

ABSTRACT

Two-dimensional (2D) process simulation has become increasingly important for process design as the lateral dimensions of the devices in integrated circuits (IC) shrink to sizes comparable to their vertical dimension. In this dissertation, we explore the applicability of the finite-element numerical method in IC process simulation.

A new 2D process simulator has been developed and named the CREEP program. The modeling and simulation of silicon oxidation are used as the test vehicles in this research. Models to account for the stress-effects in silicon oxidation are developed. The finite-element numerical algorithms required to solve the model equations are then developed to test the models against experimental observation.

A new data structure for handling 2D geometric structures has also been built into CREEP so that the program can now handle a large class of oxidation related problems. The data structure is sufficiently general to allow the implementation of other process simulation capabilities into the program in the future.

The practical acceptance of the use of the finite-element method in IC process simulation would be difficult if the grid or mesh required for the finite-element discretization cannot be generated automatically. The new geometric data structure has made possible the design and implementation of a robust automatic mesh-generator. This mesh-generator demonstrates the feasibility of reliably automating the finite-element discretization process.

The stress-dependent models developed for silicon oxidation have been verified to accurately predict the retardation of oxidation rate on cylindrical silicon surfaces. The CREEP program can now simulate various types of LOCOS processes, oxidation of silicon gate and trench structures, glass reflow (a subset problem of silicon oxidation) and so on. However, the development of a framework for a general purpose finite-element based process simulator, the CREEP program itself, is the major contribution of this project.

[illegible]

Committee Chairman

Dedicated to my parents

Acknowledgement

I would like to thank my advisor Professor William G. Oldham for his encouragement and guidance throughout the course of this work. I would also like to thank Professor A.R. Neureuther and Professor R.L. Taylor for serving on my dissertation committee. Discussions with them have further enhanced my technical understanding.

It has been a great pleasure working with my colleagues in the IC processing and simulation groups at UC Berkeley. It is not possible to list all their names here. However, I would especially like to thank Gino Addiego, Carl Galewski and Pei-Lin Pai for their friendship and the many technical discussions we had. My technical presentation skill has been greatly influenced by them through the critical but informal training they provided. This has perhaps been the most important learning experience during my graduate study.

The financial support provided by the IBM Corporation and the Semiconductor Research Corporation is gratefully acknowledged.

I would like to thank my parents for their love, support and patience through the years. The sacrifice made on their part in order to provide me with excellent education is greatly appreciated. Last, but not the least, I thank my fiancee, Ting Chuk, for her love and moral support in the past few years.

Table of Contents

CHAPTER 1.

INTRODUCTION

Various types of computer aided design (CAD) tools have become indispensable in the design and fabrication of today's integrated circuits (IC). Circuit simulators such as SPICE [1] were among the first to be developed and gain popular acceptance. Then, process simulators, logic and timing simulators and various types of automatic synthesis, layout and placement tools were developed. Yet, after more than a decade of continuing development, process simulators have not gained as much acceptance as the other CAD tools by the IC industry.

The slower acceptance of process simulators stems primarily from a lack of confidence in the accuracy of most of the available process simulators. There are very few process models that match the accuracy and qualities of models used in, for example, circuit simulations. A few process simulators that are fairly accurate can only simulate a limited class of processes. What is needed is a complete process simulator package or a collection of process simulators that provide the capabilities of accurately predicting the important processes encountered in IC fabrication technology. In order to do this, it is necessary to step up our effort to obtain more accurate and scalable physical models. Scalable models are models which provide accurate prediction of processes occurring in physical dimensions (size and shape) different from those used to characterize the models. These models are more likely to be physically-based than empirical. However, verifying such models is often difficult, as analytical solutions to many of them are generally not available. Numerical simulations are often needed to solve the model-equations so that the models can be tested against experimental observations. At present, unfortunately, the numerical techniques for solving continuum-equations, which are needed in most process simulation, are not as well developed as the numerical-techniques used in solving discrete-systems, such as those used in circuit simulations. The models needed to accurately describe most of the IC processes are usually difficult enough to solve in one-dimensional systems. As the device dimensions become smaller, it becomes necessary to solve the model-equations in two-dimensional (2D) or even three-dimensional systems. The complexity of process simulation thus increases dramatically.

Certainly, the model-verification step requires systematic experimental data that can be readily quantified, and it is often difficult to design and perform the experimental procedures to obtain these data. In this work, however, the focus is more on the issue of simulation. Since it is impossible for a single person to develop all the simulation capabilities that process-designers are interested in, this work is restricted to the development of a framework for a general purpose 2D process simulator. It is hoped that other simulation capabilities can be added into the program without too much difficulty in the future. The modeling and simulation of silicon oxidation in wet ambient are chosen as the vehicles for the development of this program. The choice should prove to be interesting and useful as the algorithms needed to solve the models for silicon oxidation are quite difficult and comprehensive. Because silicon oxidation is a creep-flow problem, the program developed in this work is named CREEP.

The historical events that led to the development of the CREEP program is described in chapter 2. Various aspects related to the modeling of silicon oxidation will be given in chapter 3. The numerical discretization algorithms for solving the silicon oxidation models will then be described in chapter 4. A special geometric data structure developed for CREEP, and the finite-element mesh generation algorithms used will be described in chapter 5 and 6 respectively. Finally, some application examples of the CREEP program will be given in chapter 7 before concluding this dissertation in chapter 8.

CHAPTER 2.

HISTORICAL ACCOUNT

This work started in 1984 with the attempt to explore new silicon isolation technology. Many different isolation schemes were already proposed by other researchers in the field. But there was a general lack of physical understanding for non-planar oxidation. Hence, instead of performing many trial-experiments in search of yet another isolation technique, we thought that it would be fruitful to pursue for a physical understanding of non-planar oxidation. We hoped that a better understanding of silicon oxidation would eventually help us in identifying better isolation technologies, or the inherent limitation of silicon oxidation technology.

As in any other modeling work, it is necessary to verify any proposed silicon oxidation models against experimental observation. By definition, non-planar oxidation is at least two-dimensional (2D) in nature. Unfortunately, there is generally no analytical solution for even the simplest 2D extension of the Deal-Grove models of oxidation* in arbitrarily structured 2D domains. It was clear that the use of 2D numerical simulation would be required to verify any model which might be proposed.

Two oxidation simulators were available at that time: SOAP[2], from Stanford University, and NEOVEDA[3], from NEC of Japan (several other oxidation simulators were developed more recently by Poncet[4], Boruki et al[5], Tung et al[6], and Rafferty et al[7].). SOAP used a viscous-incompressible oxide model while NEOVEDA used a linear visco-elastic oxide model. But both programs used a 2D extension of the Deal-Grove models for the oxidation kinetics. The simulation results of LOCOS oxidation from both programs generally differed from the experimental observations. The disagreements between the simulated and experimental results were more noticeable when large silicon-curvature or thick nitride layers existed in the structures. Assuming that the two programs solved the intended physical and mathematical models

* The 2D Deal-Grove models are described in chapter 3.

properly, we concluded that the simple 2D Deal-Grove models were not accurate, regardless of the mechanical model used for the oxide. A careful comparison of the simulated profiles and experimental results led us to suspect that mechanical-stress in the oxide modifies some of the parameters in the Deal-Grove models. The experimental results from Marcus and Sheng[8], which show clear oxidation rate retardation on both convex and concave corners of a silicon trench, provided more graphical evidence of the stress-effects in silicon oxidation. It was then our goal to determine the functional dependence of the Deal-Grove parameters on the various stress-components in the oxide or silicon.

The modification of one of the existing simulators to include some stress-dependent oxidation models seemed to be the simplest approach to study the stress-effects in silicon oxidation. A feasibility study was made. A major consideration was the frequent no-convergence problems that the existing programs suffered. Coupled with the slow speed of the programs (typically several hours computer time on a minicomputer for one complete simulation of a LOCOS process), we had no confidence that we could achieve a workable enhancement of either program. Another consideration, which turned out to be more fundamental, is the numerical approach used in both programs. Both programs (and more recently the work of Tung et al[6]) used some form of boundary-element methods for the numerical discretization of the oxide deformation and oxidant diffusion. As a result, material non-homogeneity or non-linearity within the bulk of the material could not be handled easily. Since we did not rule out the possibility of testing non-linear bulk properties (such as oxidant diffusivity), we had to choose numerical techniques that discretize both the bulk and boundaries of the domain of interest. It was quite apparent that the required modification of the existing programs would be a major undertaking that offer little or no advantage over starting a new program. We thus decided to write a new program, which was subsequently called the CREEP program.

Among the two well established techniques, the finite-difference and the finite-element methods, we chose the latter for several reasons. First, the finite-element method can be easily applied to irregularly shaped domains. Next, the size and shape of the finite-elements can vary drastically over a short distance in a domain, making it possible to allocate dense mesh (discretization) only in regions of high variation for the unknown variables or the material properties. On the other hand, there is no clear way of formulating the finite-difference method for an arbitrarily irregular grid. The computation time for a program using the finite-element method can thus be greatly reduced by using a proper mesh-generation or discretization. Lastly, the finite-element method is a widely established technique used in the civil and mechanical engineering disciplines. This is a practical consideration for us since the problems are similar and knowledge acquired from the other disciplines might speed the development of the CREEP program.

To take advantage of the finite-element method clearly requires the ability to generate a reasonable mesh of elements. The mesh-generation can be done either manually by the user, or automatically by the program. For silicon oxidation which is a moving boundary problem, repeated mesh-generation may be needed throughout the whole simulation process. User-aided mesh-generation may become unacceptably tedious. We thus decided to build an automatic mesh-generator. The finite-element algorithms needed to solve silicon oxidation models were assumed to be available by adapting certain well known techniques used in the civil engineering discipline.

Many different element types are in used in the field of the finite-element analysis. The quadrilateral elements are the most commonly used elements. We chose to use the triangular elements mainly for the reason of the relative ease in implementing automatic triangular mesh-generator. It also turned out that when the meshes are highly irregular, triangular elements often provide superior results compared to the quadrilateral elements[9].

Because of our inexperience in using and building an automatic mesh-generator, we were concerned that the mesh-generation process might be time-consuming. We concentrated our effort on building an efficient instead of a robust code for the mesh-generator. After a first version of our mesh-generator was tested in the finite-element analysis of silicon oxidation, we realized that the speed of the program was primarily determined by the speed of the finite-element discretization process and the linear-system solution process. The computer time used by the mesh-generation turned out to be negligible. As expected, the computer time involved is strongly dependent on the the number of nodes in the system. But it was not expected that adequate accuracy can often be obtained with a relatively low node-count in the discretized system.

To take advantage of this situation, the mesh-generator should run with a low mesh-density and yet produce a reasonably good mesh. The first mesh-generator built was unfortunately not robust enough when running under low mesh-density. It was then clear that we could afford to have a slower but more reliable mesh-generator.

A second mesh-generator was then written. This mesh-generator was more robust than the first one, but still not satisfactory. Occasionally, zero-area elements were formed, causing the program to crash. Although the problems were identified and could have been fixed, we were more concerned about the inability of the program to handle more than one polygonal structure. Before the third mesh-generator was written, we determined that we need a general data structure for the efficient manipulation of arbitrarily shaped and connected geometries. Such a data structure will not only allow the program to handle a larger class of geometric structures, it will also simplify the design of the mesh-generator. The third version of the mesh-generator was then built around this data structure. This data structure is presented in chapter 5. The mesh-generator is described in chapter 6.

The various support routines needed for the finite-element discretization of silicon oxidation models are also applicable to other process simulation. The routines that handle the data structure, the mesh-generator, the linear-system solvers and the input interpreter can actually provide a basis for other process simulation capabilities. Only the routines for the finite-element discretization of the silicon oxidation models are particular to local oxidation problems. Simulation of bulk processes, such as impurity diffusion in solids, requires the same set of support routines. Topographical process simulation such as etching and deposition requires a robust and general data structure for profile description. It then appears that many other important process simulation capabilities can be added into the CREEP program with relatively minor additional effort. Thus the various parts of the program are implemented in a modular fashion to make the CREEP program a framework for a general purpose process simulator.

CHAPTER 3.

MODELING OF SILICON OXIDATION

3.1. INTRODUCTION

Silicon oxidation is a moving boundary problem in which the oxide is continuously generated (ie, no conservation of volume). Although the numerical algorithms used in this work is described in chapter 4, it is worth discussing certain aspects of the numerical approach used at this early stage. The formulation and numerical solution of the models for silicon oxidation in this dissertation closedly follow the approach often used in the solution of metal forming processes (extrusion, rolling, etc). Metal forming is a moving boundary problem in which creep flow is often assumed for the deformation of the material. A simple time-stepping scheme often used in updating the profile of the material at time $t = t_0$ to another profile at $t = t_0 + \Delta t$ involves the displacement of every point in the material by an amount equals to Δt times the velocity of that point. The time-stepping scheme used in this work follows a similar strategy as that used in metal forming. At any time $t = t_0$ during the oxidation process, the model equations for silicon oxidation are formulated and solved, using the configuration/profile of the materials at $t = t_0$. The velocity field is obtained and the profile is updated by moving every point by Δt times the velocity at that point.

Although the results of the numerical simulation performed so far appear to be correct, no attempt has been made to prove that the numerical solution converges to the true solution as the spatial discretization and the Δt used in the time-stepping scheme approach zero. This is particularly a concern for the time-stepping process. In this work, oxide is assumed as a viscous fluid, and the formulation for the oxide deformation is based on the linear (infinitesimal) strain-rate theory. However, silicon oxidation is clearly a large scale deformation problem. More rigorous formulation using the large strain/displacement (finite deformation) theory may be needed to avoid possible accumulation of error in the deformation that does not vanish even as Δt approaches zero. The numerical algorithms used in this work is only a first implementation of the algorithms for silicon oxidation. Future work should consider the application of the finite deformation theory in silicon oxidation. This is especially crucial if visco-elastic and/or elasto-plastic models are to be used in oxidation modeling, as the stress in the material may also accumulate incorrectly if a proper finite-deformation formulation is not used.

3.2. 2D DEAL-GROVE MODELS

The Deal-Grove models of oxidation[10] represent the first major break-through in the modeling of silicon-oxidation. The models state that the oxidant diffuses from the ambient into the oxide, and react with the silicon at the silicon/silicon-dioxide interface. The Deal-Grove models are usually applied to one-dimensional (planar) oxidation, although the extension to two- or three-dimensional systems appears to be straight-forward. The first attempt to solve the model in two-dimensional (2D) systems was made by Chin et al[2]. In the set of the original (one-dimensional) Deal-Grove models, the mechanical properties of the oxide are not considered since the oxide flows only in one direction, normal to the planar-surface. In a two-dimensional system, the mechanical properties of the oxide must be explicitly included in the models. In Chin's program, the oxide is modeled as a viscous fluid. Because the models for oxidant diffusion and reaction are essentially due to those of Deal and Grove, we will consider Chin's models as 2D Deal-Grove models.

To explain the 2D Deal-Grove models, consider the common LOCOS process illustrated in Fig. 3.2.1. Since the oxidation rate is usually slow compared to the time constant for oxidant diffusion, the oxidant diffusion is quasi-steady-state :

$$\nabla \cdot (D \nabla C) = 0 \qquad \text{in } \Omega \tag{3.2.1}$$

where D and C are the oxidant diffusivity and concentration respectively in the bulk of the oxide (Ω).

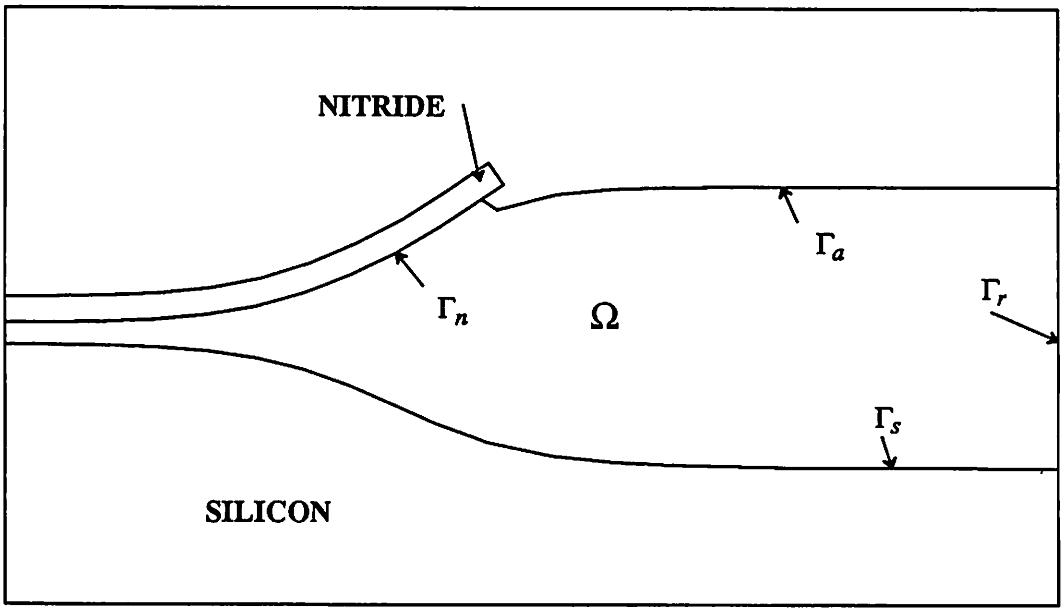


Fig. 3.2.1. A typical oxide profile from a LOCOS process.

The oxidation rate at the Si/SiO$_2$ interface is assumed to be proportional to the oxidant concentration :

$$N\frac{dV_{ox}}{dt} = k_s C \qquad \text{on } \Gamma_s \tag{3.2.2}$$

where N is the number of oxygen atoms per unit volume of oxide, V_{ox} is the volume of oxide formed, k_s is a proportionality factor known as the surface reaction rate parameter and Γ_s is the Si/SiO$_2$ interface.

Given the model described by Eqn. 3.2.2, we must have

$$-D\frac{\partial C}{\partial n} = k_s C \qquad \text{on } \Gamma_s \tag{3.2.3}$$

as the boundary condition for the oxidant diffusion at Γ_s. In the above, n is the normal direction outward from the oxide region.

The ambient phase gas transport coefficient for the oxidant (which is H_2O in our case) was found to be sufficiently high[10] that we can accurately approximate the boundary condition at the SiO_2/ambient interface (Γ_a) by setting the oxidant concentration there to the maximum oxidant solubility in the oxide at the given ambient conditions. This maximum solubility is called the equilibrium oxidant concentration, C^* :

$$C = C^* \qquad \text{on } \Gamma_a \tag{3.2.4}$$

Nitride (Si_3N_4) is assumed to be a perfect oxidant mask and thus no oxidant flux can go through the SiO_2/Si_3N_4 interface (Γ_n). The boundaries with the simulation window (Γ_r) are assumed to be reflection boundaries. The reflective symmetry of the structures requires that no flux crosses any of the reflection boundaries. Thus, we have

$$-D\frac{\partial C}{\partial n} = 0 \qquad \text{on } \Gamma_r \text{ and } \Gamma_n \tag{3.2.5}$$

Eqn. 3.2.1 through 3.2.5 determine the oxidant diffusion in the oxide and the reaction at the Si/SiO_2 interface. The reaction generates new oxide at the Si/SiO_2 interface which is about 2.2 times the volume of silicon consumed. This newly formed oxide layer "pushes" the existing oxide layer away from the silicon region. Assuming that the Si/SiO_2 interface is smooth, then the displacement of the existing oxide at the Si/SiO_2 interface is normal to the interface. The displacement of the existing oxide regions as a whole depends on the mechanical properties of the oxide. The mechanical properties of a material are usually described by the stress-strain relationship for the material, commonly known as the material's constitutive relationship. In Chin's work, the oxide is assumed to be a viscous incompressible fluid at the oxidation temperature. The constitutive relationship for the oxide is given by

$$\sigma_s = 2\eta\dot{\varepsilon}_s \tag{3.2.6}$$

and

$$\varepsilon_v = 0 \tag{3.2.7}$$

In the above, σ_s is the shear stress and $\dot{\varepsilon}_s$ is the shear strain-rate in the oxide. The proportionality factor η is the viscosity of the oxide. The incompressibility condition is specified by setting the volumetric strain (ε_v) to zero, regardless of the hydrostatic pressure in the oxide.

3.3. LIMITATION OF THE 2D DEAL-GROVE MODELS

Certainly, the models described by Eqn. 3.2.6 and 3.2.7 are an idealization of the complex behavior of oxides. Other models for the mechanical properties of oxide are possible. Both visco-elastic[3, 6] and elastic[4] models were used in other oxidation simulators. However, because all the modeling work used the same Deal-Grove diffusion and reaction rate models, we will classify these models under the 2D Deal-Grove models.

Most of the tests for the 2D Deal-Grove models were performed on LOCOS processes. Simulation results from several programs, including the earlier version of our program, showed that the simple 2D Deal-Grove models provide qualitatively good predictions of LOCOS processes, if the nitride masks used were thin enough. The actual mechanical properties of the oxide used have little effect on the predicted-profiles. However, the model-prediction degrades as the thickness of the nitride mask used in the LOCOS process is increased. The model-prediction for oxide grown on a silicon trench was even less successful. The experimental results of Marcus and Sheng[8] showed that oxide grown around both the convex and concave corners of a silicon trench are thinner than oxide grown on flat silicon surfaces. Their results are illustrated in Fig. 3.3.1. The apparent retardation in the oxidation rate cannot be explained by the simple Deal-Grove models of oxidant diffusion and reaction, nor by any reasonable constitutive relationship for oxide. It may be expected that silicon oxidation is influenced by the mechanical stress generated in the oxide. The discrepancy between the predictions and the experimental results of thick-nitride LOCOS process can then be qualitatively explained, as higher stresses are expected to exist in LOCOS structures with thicker nitride-masks.

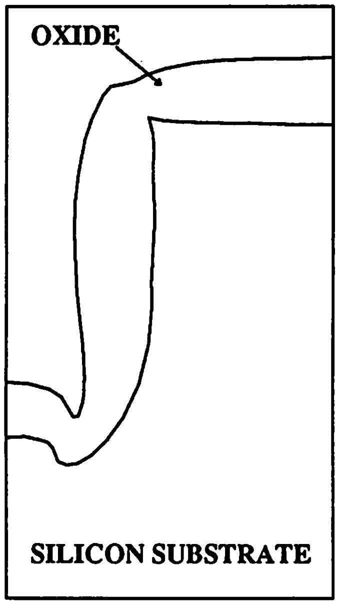


Fig. 3.3.1. Retardation of oxidation around both the convex (upper) corner and concave (lower) corner of a silicon trench.

3.4. CHARACTERIZATION OF STRESS-EFFECTS

Marcus recognized that the stress effects in silicon oxidation might be quantitatively characterized by studying the oxidation of cylindrical silicon structures of different radii[11]. The experiments were performed by Kao et al at Stanford University[12]. In Kao's experiments, cylindrical ring structures with heights of about 3μm were fabricated on (100) silicon wafers (see Fig. 3.4.1). The wafers were oxidized at various temperatures for various times so that all the oxide films grown on flat (110) surfaces were approximately 0.5 μm thick. The wafers were then coated with poly-silicon for protection, and lapped by about 1.5μm to reveal the cross-section of the cylindrical structures. The thickness of oxide grown on the different-sized cylinders and at the various temperatures was measured and compared with the thickness of oxide grown on flat (110) surfaces at the corresponding temperatures.

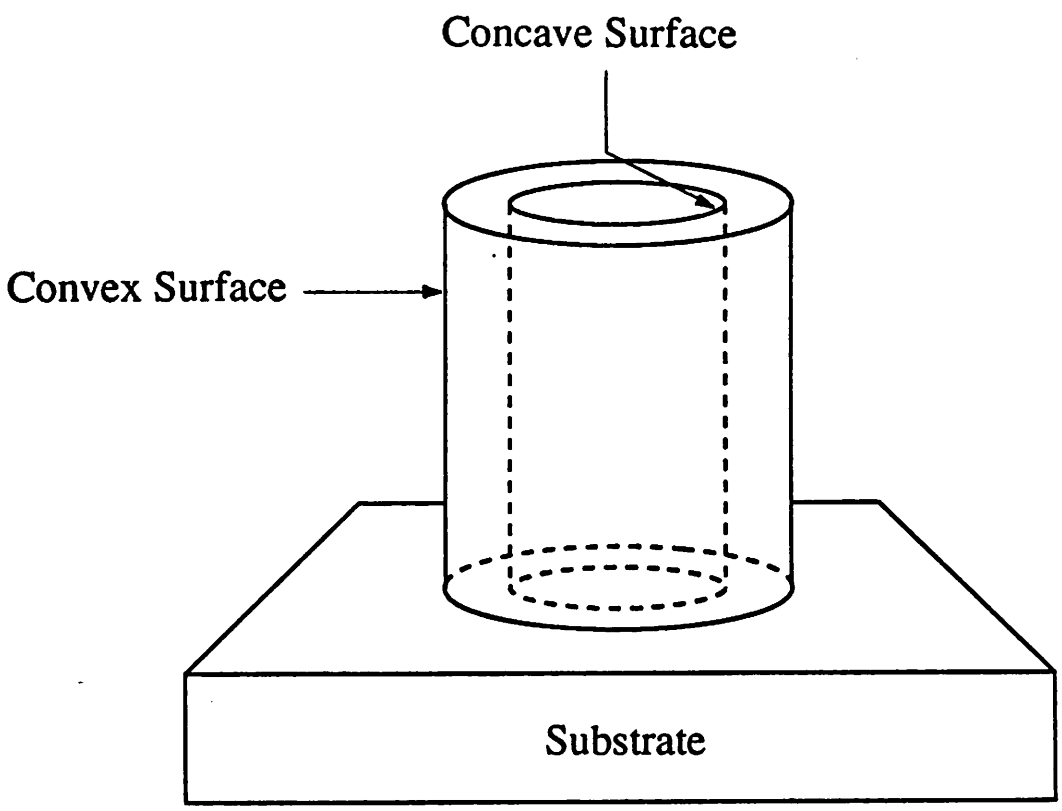


Fig. 3.4.1. Cylindrical silicon rings used in the oxidation experiments performed by Kao et al[12].

3.5. EFFECTS OF CRYSTAL-ORIENTATION

The cylindrical structures were used in the experiments because they offer considerable symmetry to simplify the interpretation and analysis of the data. However, the oxidation rate varies significantly with the crystal-orientation of the silicon surface, especially in the lower temperature range. For wet thermal oxidation in the temperature range of 800 to 900 °C , the relative magnitudes of the reaction rate on the different crystal surfaces are approximately given by

$$\frac{k_s(111)}{k_s(100)} \approx 1.68 \qquad (3.5.1)$$

$$\frac{k_s(110)}{k_s(100)} \approx 1.45 \tag{3.5.2}$$

For cylindrical silicon structures fabricated on (100) oriented wafers, the faces of the cylinders are oriented between (110) and (100), as illustrated in the following figure:

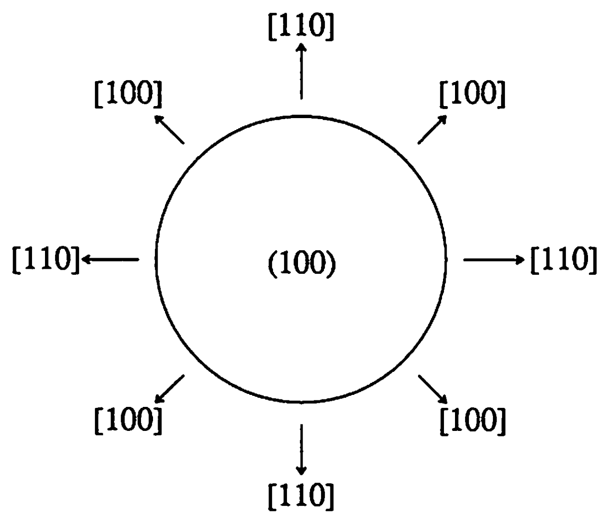
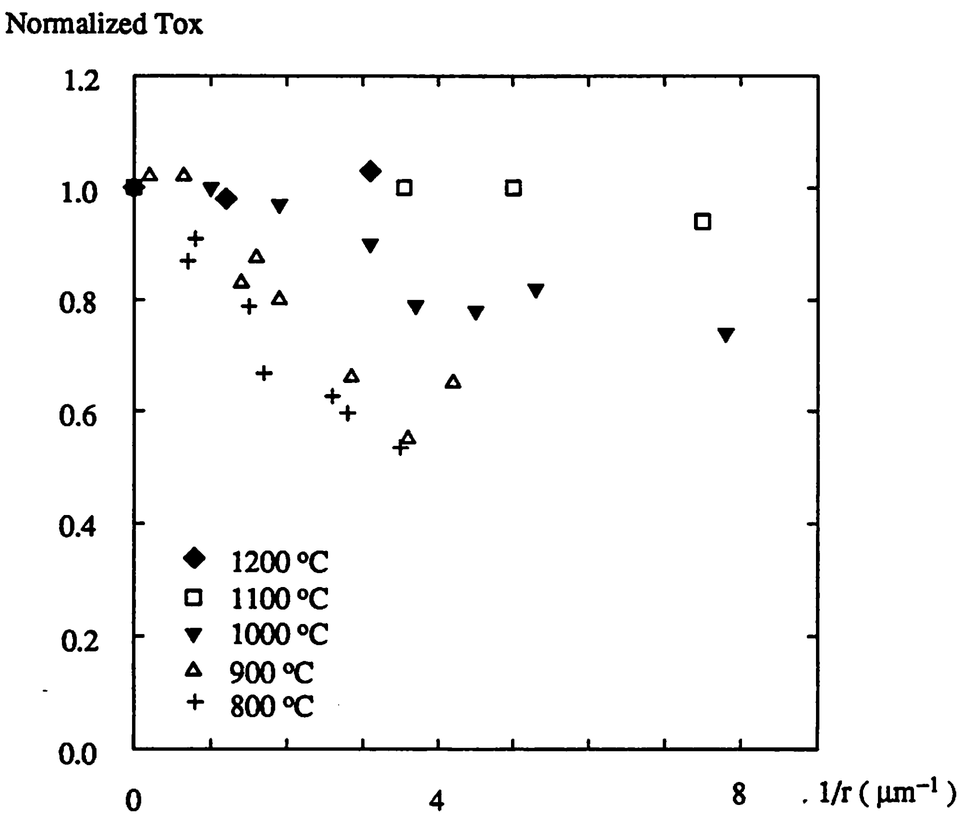


Fig. 3.5.1. The surface orientation on a cylindrical structure etched in (100) oriented wafers.

Despite the fact that the starting silicon structures were cylindrical, the oxidized profiles had four-fold symmetry instead of full cylindrical symmetry. To collect a set of consistent data, all measurements were made along the [110] direction. The measured oxide thickness was normalized to the thickness of oxide grown on flat (110) surfaces. The distance between the center of the structure to the Si/SiO_2 interface at the end of the oxidation process was also measured along the [110] direction. This distance provided an estimate for the radius of curvature of the Si/SiO_2 interface at the end of the oxidation process, and thus was designated as r. The normalized oxide thickness was then plotted against the reciprocal of r. The results for both the convex and concave structures, reproduced from the work of Kao et al, are shown in Fig. 3.5.2 and 3.5.3.



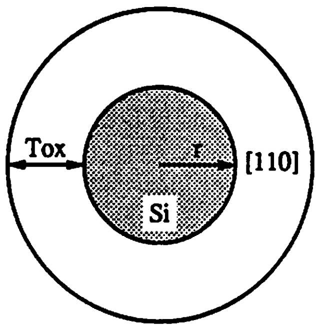
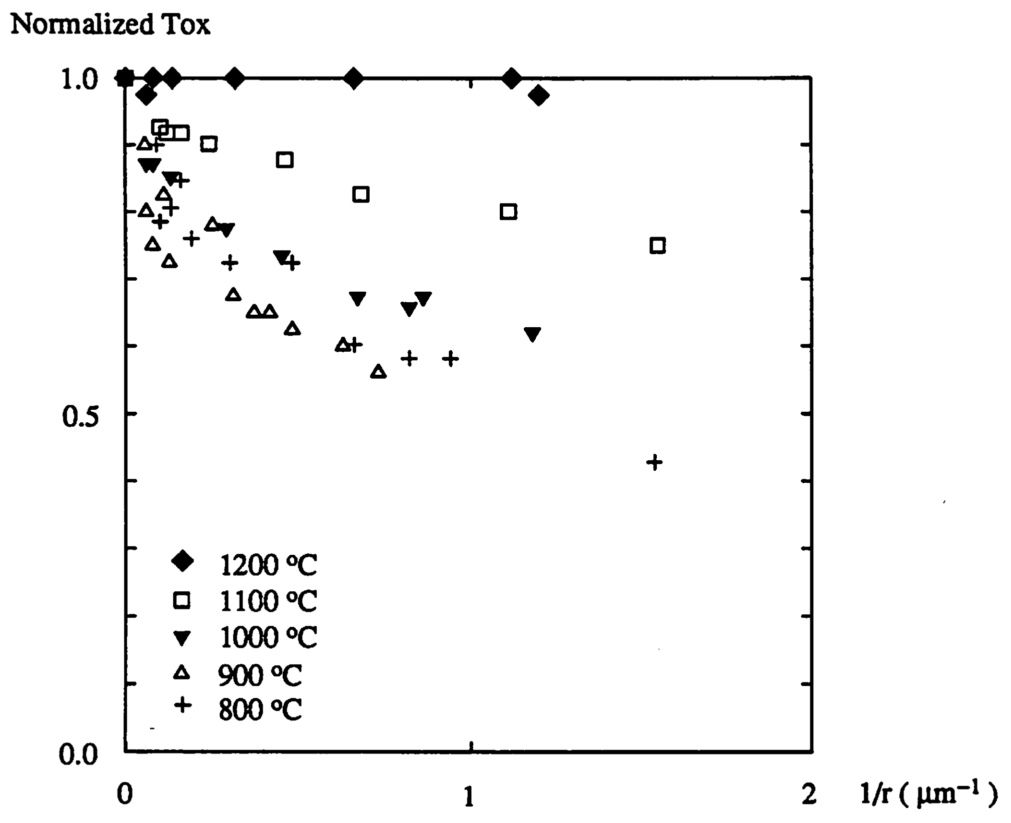


Fig. 3.5.2. Convex experimental data. All measurements were made along the [110] direction.



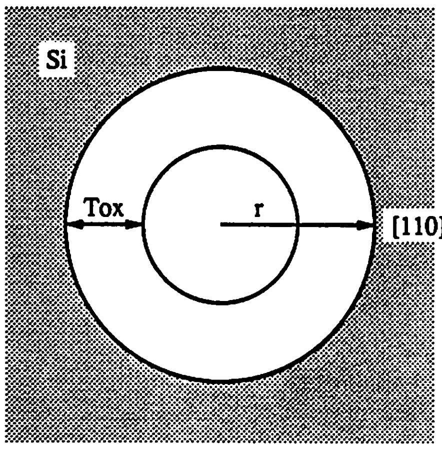


Fig. 3.5.3. Concave experimental data. All measurements were made along the [110] direction.

3.6. KAO'S MODELS FOR STRESS-EFFECTS IN OXIDATION

Kao modeled the stress-effects in silicon oxidation in terms of a set of stress-dependent Deal-Grove parameters and oxide viscosity. The models are :

$$k_s = k_{s0}(T)\exp(\sigma_{nn} V_k / kT) \tag{3.6.1}$$

$$D = D_0(T)\exp(-pV_d / kT) \tag{3.6.2}$$

$$C^* = C_0^*(T)\exp(-pV_c / kT) \tag{3.6.3}$$

$$\eta = \eta_0(T)\exp(\alpha(T)p) \tag{3.6.4}$$

In the above, k is the Boltzman constant and T is the temperature. k_{s0} , D_0 , C_0^* and η_0 are the zero-stress reaction rate parameter, oxidant diffusivity, equilibrium concentration and oxide viscosity respectively.

The reaction rate parameter was assumed to be retarded by a normal compressive stress ($\sigma_{nn} < 0$). The volume parameter V_k has the physical meaning of the reaction jump-volume and was argued to be 25 Å^3, the difference between the volume of an SiO_2 molecule and a silicon atom. Eqn. 3.6.1 states that extra work needs to be done in the oxidation reaction to move a newly formed and expanding SiO_2 molecule against a normal compressive force field acting on the molecule.

No significant physical basis is available for the other three models. As the oxidant diffusivity, equilibrium concentration and oxide viscosity are bulk properties, they are assumed to be dependent on the hydrostatic pressure in the oxide. Both D and C^* are assumed to be retarded by a compressive hydrostatic pressure ($p > 0$), while η is assumed to increase with increasing compression. Because these models are empirical, the parameters V_d , V_c and $\alpha(T)$ cannot be predicted and thus must be extracted from experimental data.

The low-stress viscosity η_0 for wet thermal oxide is not well known and thus must also be extracted from model fitting to the experimental results. The parameter k_{s0} can be inferred from the Deal-Grove parameters obtained for planar oxidation[10]. The values for D_0 and C_0^* can also be inferred from the Deal-Grove parameters for planar oxidation, even though planar oxides actually grow under compressive stress condition in a certain temperature range. Slight errors may be introduced into the values for D_0 and C_0* at a lower temperature range. But these errors can be minimized by using their extrapolated values from a high temperature range where oxides grow under a relatively low stress condition.

3.7. LIMITATION OF KAO'S MODELS

Careful examination of the behavior of Kao's models would reveal that the model for the oxide viscosity renders the whole set of models non-physical under certain circumstances. In the oxidation of cylindrical silicon structures, compressive (positive) hydrostatic pressure exists in the oxide grown on inner cylindrical surfaces (concave structures) and tensile or negative hydrostatic pressure in the oxide grown on outer cylindrical surfaces (convex structures). In concave structures, the positive hydrostatic pressure increases the oxide viscosity, which in turn increases the hydrostatic pressure. A positive feedback between the stress and the viscosity is thus established. This leads to a possibility that there may be no physical solution to the models, if the Deal-Grove parameters cannot be sufficiently reduced by the stress to counter the trend of the positive feedback. Under such a situation, numerical computation of the oxidation equations will always diverge.

The model for the C^* is also inconsistent with some observed oxidation phenomena. Deal and Grove found that the value of C^* for wet thermal oxide is almost constant between 900 to 1200 $°C$[10]. EerNisse found that there is an "intrinsic" stress during thermal oxidation of silicon[13]. For wet thermal oxidation below about 950 $°C$, the oxide is under a compressive stress of the order of 1e9 $dyne/cm^2$. Above that temperature, there is almost no "intrinsic" stress (actually, the stress is below the 1e9 $dyne/cm^2$ that was needed to accurately observe the bending of silicon wafers). If C^* is significantly dependent on the hydrostatic pressure, the measured value for C^* obtained by Deal and Grove should be temperature-dependent. The lack of an obvious "breakpoint" for the activation energy of the linear-rate constant (the parameter $B/A \approx k_s C^*/N$ in the original Deal-Grove model) also suggests that C^* should not depend much on the oxide-stress.

The remaining two models (Eqn. 3.6.1 and 3.6.2) were tested against the available experimental data by using 2D numerical simulation. Our simulation results indicated that all the convex data points can be quantitatively explained by the two stress-dependent models, if the volume parameter V_d is allowed to be temperature-dependent. However, the concave data set cannot be fitted by any combination of V_k, V_d and η_0. The concave data set shows an initial rapid decrease in the thickness of the oxide with increasing curvature. However, at higher curvature, the "rate" of retardation with increasing curvature decreases. If we attempt to fit the data points in the low-curvature regime, we invariably over-estimate the amount of retardation at higher curvature. Fig. 3.7.1 illustrates this difficulty.

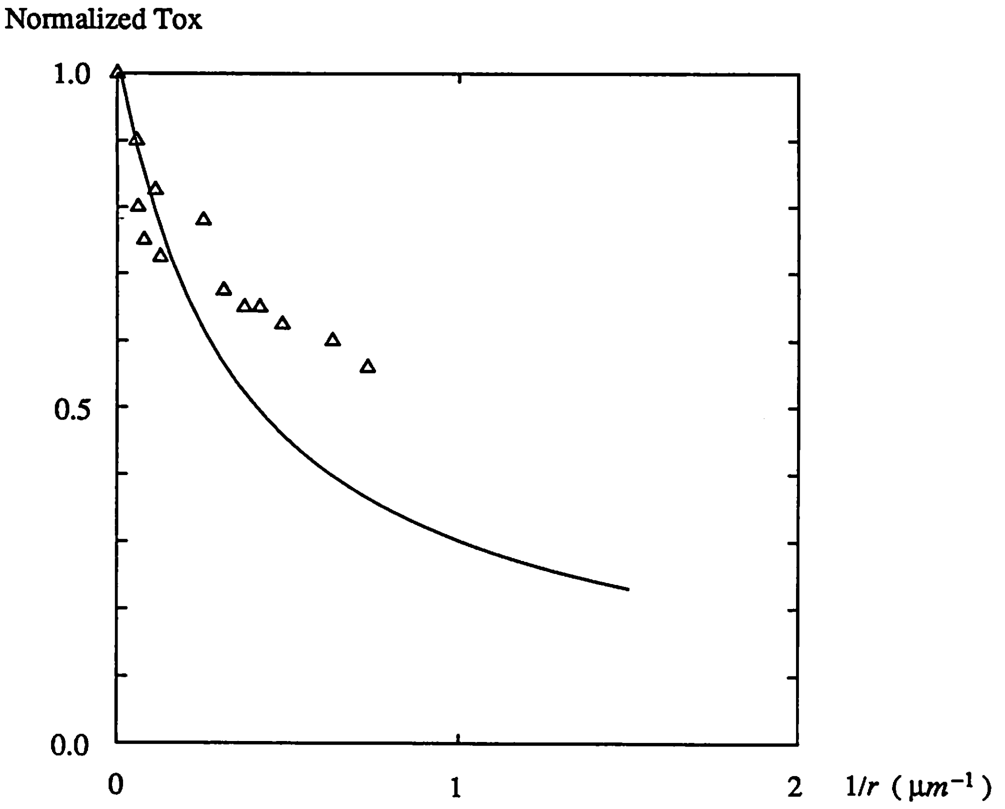


Fig. 3.7.1. Model fit to the 900 °*C* concave experimental data using only the stress-dependent reaction-rate and diffusivity models of Kao.

3.8. IMPROVED STRESS-DEPENDENT PARAMETERS

It is apparent that Kao's models must be revised or replaced to improve the model-fit to the experimental results. We propose a revision of the following form (the removal of the stress dependent model for C^* is explained in section 3.7) :

$$k_s = k_{s0}(T) \exp(\sigma_{nn} V_k / kT) \tag{3.8.1}$$

$$D = D_0(T) \exp(-pV_d / kT) \quad , \quad D < D_{\max}(T) \tag{3.8.2}$$

$$\eta = \eta_0(T) \frac{\frac{\sigma_s V_0}{2kT}}{\sinh(\frac{\sigma_s V_0}{2kT})} \tag{3.8.3}$$

3.8.1. Surface Reaction Rate Model

This model is physically appealing and thus is unmodified. But we now argue that the reaction jump-volume V_k should be the volume expansion involved in every single oxygen atom attachment to an Si–Si bond. This means that V_k should be about 12.5Å^3 , instead of 25Å^3. However, other values for this parameter will be tested during the model-fitting process to find out if 12.5Å^3 is the only number that can fit the experimental results.

3.8.2. Oxidant Diffusivity Model

The oxidant diffusivity, being a bulk quantity, is likely to depend on the micro-structure of the oxide. This means that it should in some way depend on the density of the oxide. It is also possible that the diffusivity depends anisotropically on the the direction of the shear-strain in the oxide. Presently, it is difficult enough to characterize the dependence of the diffusivity on the density of the oxide. Thus, any higher order effect will not be considered at the moment. Characterizing the effect of the oxide-density on the oxidant diffusivity is difficult because the change in the density is usually very small (typically about 1%). It is much easier to measure and calculate the hydrostatic pressure in the oxide. Unfortunately, in the usual range of oxidation temperature, oxide behaves as a visco-elastic material. The density of the oxide can have a delayed-response on the applied hydrostatic pressure. The phenomena is called dilatational creep. The only data on the dilatational creep of oxide are probably the measurements of Corsaro[14]. The experiment used a mercury-filled acoustic dilatometer to measure the dilatational compliance (ie, the dilatational response to a step change in the hydrostatic pressure) of B_2O_3 in the temperature range of 227 to 282 $°C$. The dilatational compliance for B_2O_3 was found to be well described by the equation

$$J_D(t) = \frac{1}{3K_0} + \left(\frac{1}{3K_\infty} - \frac{1}{3K_0} \right) \left\{ 1 - \exp\left[-(t/\tau_D)^{0.6} \right] \right\} , \qquad (3.8.2.1)$$

where K_0 and K_∞ are the instantaneous and final values, respectively, for the bulk modulus of elasticity. The term τ_D is a time constant for the dilatational creep. The experimental procedure is obviously unsuitable for measuring the dilatational compliance of thermal oxide in the temperature range of interest (800 to 1200 $°C$). In the absence of any experimental evidence of the dilatational creep of thermal oxide, we will assume that τ_D for oxide is sufficiently short so that the density changes with the applied hydrostatic pressure without delay. This assumption allows us to use the hydrostatic pressure for further characterization of the stress-effect on the oxidant diffusivity of oxide.

In the absence of a physical model for the dependence of the oxidant diffusivity on the density or hydrostatic pressure, we propose a slightly modified version of Eqn. 3.6.2 , given in Eqn. 3.8.2. In the equation, $D_{\max}$ is the maximum value the oxidant diffusivity can reach under tensile hydrostatic pressure. It is intuitively appealing that the diffusivity cannot increase indefinitely under tensile pressure. It is likely that the actual transition to the maximum value is smooth. However, the present simple model is used in the absence of any specific diffusivity data. Further modification to the model can be made as dictated by experimental data.

3.8.3. Oxide Viscosity Model

The viscosity model described in Eqn. 3.8.3 represents the major modification to Kao's models. Here, the viscosity is assumed to be dependent on the shear-stress (σ_s) in the oxide. This model was actually developed in the 1930's from the theory of rate processes[15], and was meant to describe the behavior of the viscosity of general fluids under high shear-stress. Fig. 3.8.3.1 illustrates the behavior of this model. The model predicts that the viscosity remains constant over a wide range of low stress levels. But the viscosity drops rapidly at higher stress levels, creating the appearance of the existence of a critical stress. Li et al[16] showed that the model agrees reasonably well with the measured viscosity of glass fibers containing 8% of Rb_2O. A reproduction of their experimental results is given in Fig. 3.8.3.2. The results show that at an applied stress of less than about 1e9 $dyne/cm^2$, the viscosity of the oxide is a constant value at a constant temperature. This is indicated by the unit slope of the log-strain-rate v.s. log-stress curves in the lower stress regions. Although a similar measurement for pure silica is not yet available, it is still worth investigating the applicability of the model in describing the viscosity of pure silica. Also, the apparent critical stress value in the range of $10^9 dyne/cm^2$ observed in Li's experiment correlates well with the intrinsic stress observed in the oxidation of planar silicon surface[13]. Later, we will show that this model indeed can explain the intrinsic stress levels in planar oxidation.

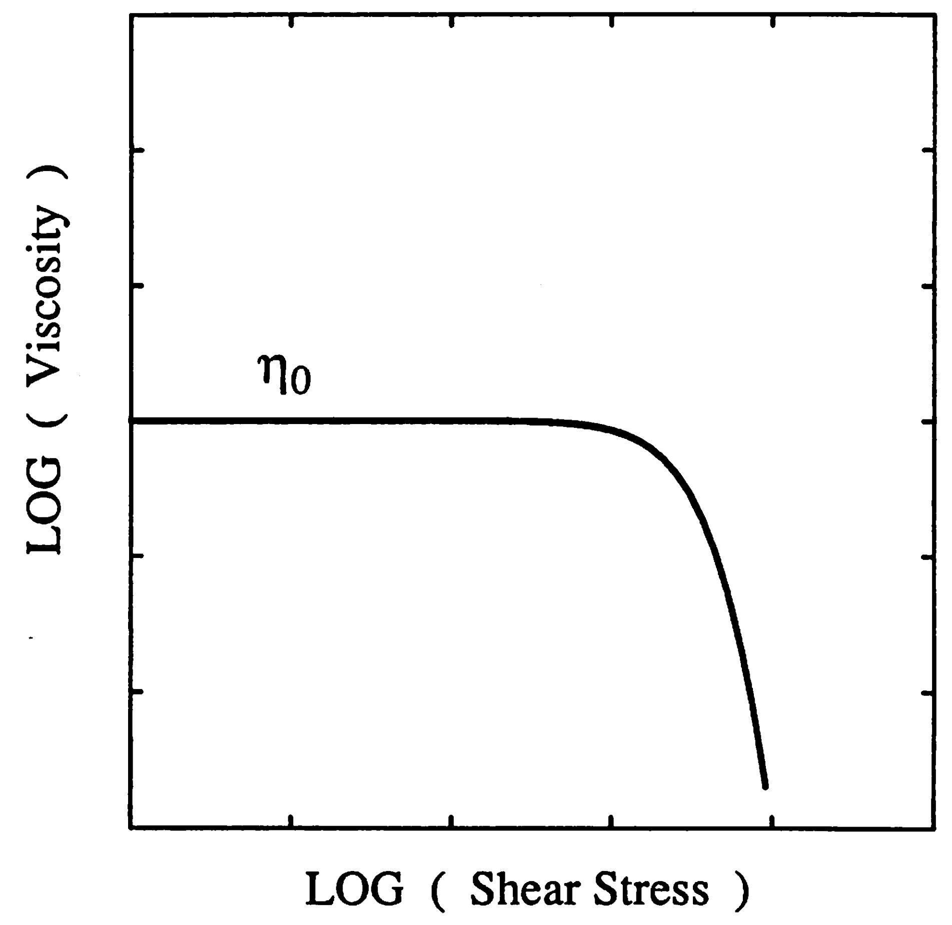


Fig. 3.8.3.1. The behavior of viscosity as a function of shear stress, as described by Eqn. 3.8.3.

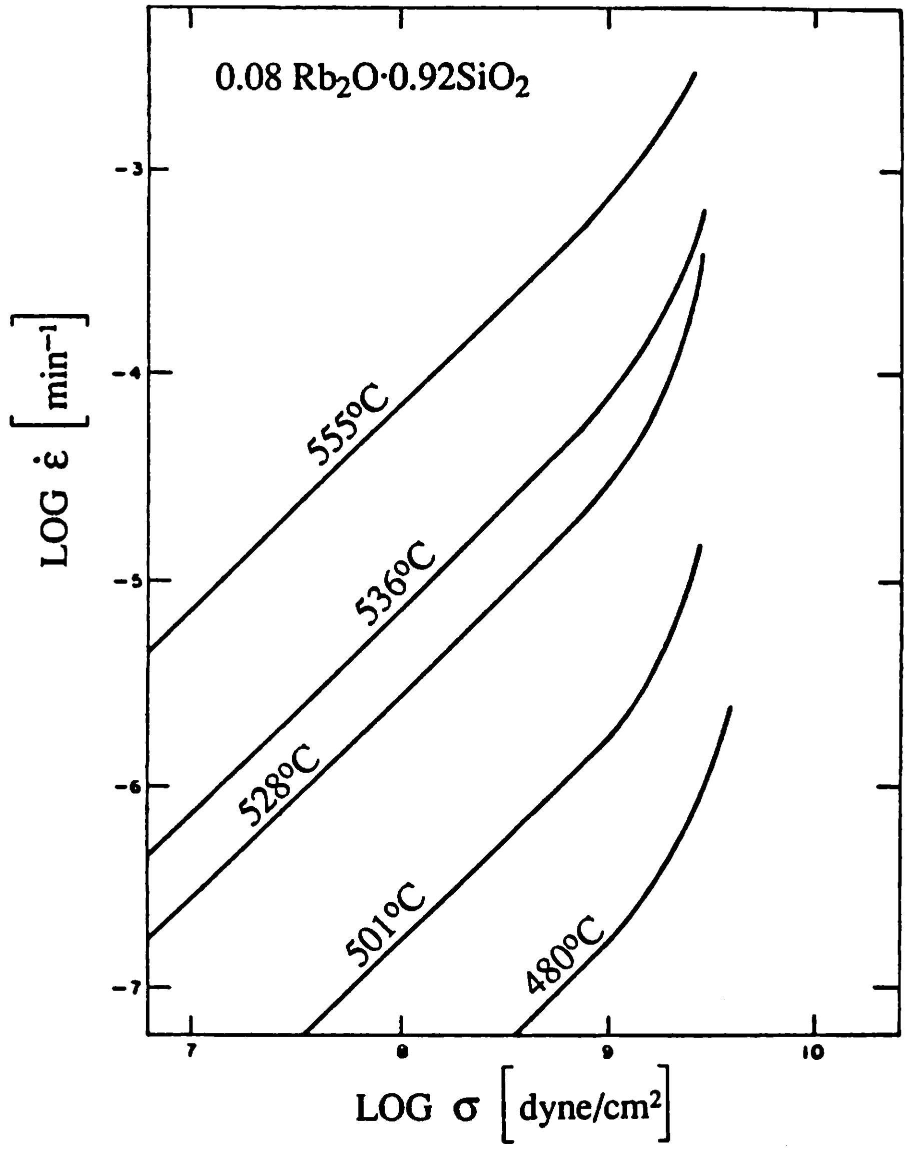


Fig. 3.8.3.2. The log-strain-rate v.s. log-stress curves for $0.08Rb_2O \cdot 0.92SiO_2$ glass fiber (reproduced from the work of Li et al[16]).

3.9. MODEL FIT TO THE EXPERIMENTAL DATA

The revised models were implemented into the CREEP process simulator. Many trial runs of the simulator using different values of V_k, V_d , D_{max}/D_0 , η_0 and V_0 were performed. The experimental data of Kao was successfully fitted within the error range of the measurement processes. Fig. 3.9.1 and 3.9.2 show the model-fit to the experimental data of convex and concave structures.

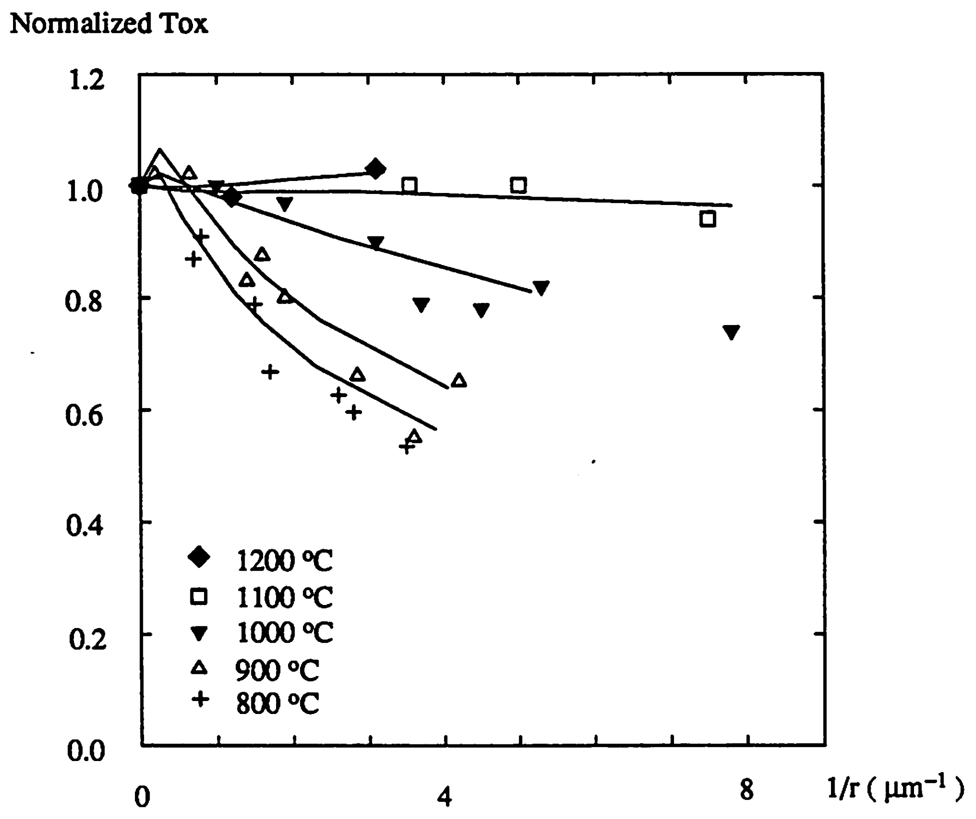


Fig. 3.9.1. Model fit to the convex experimental data. The solid lines are the model predictions.

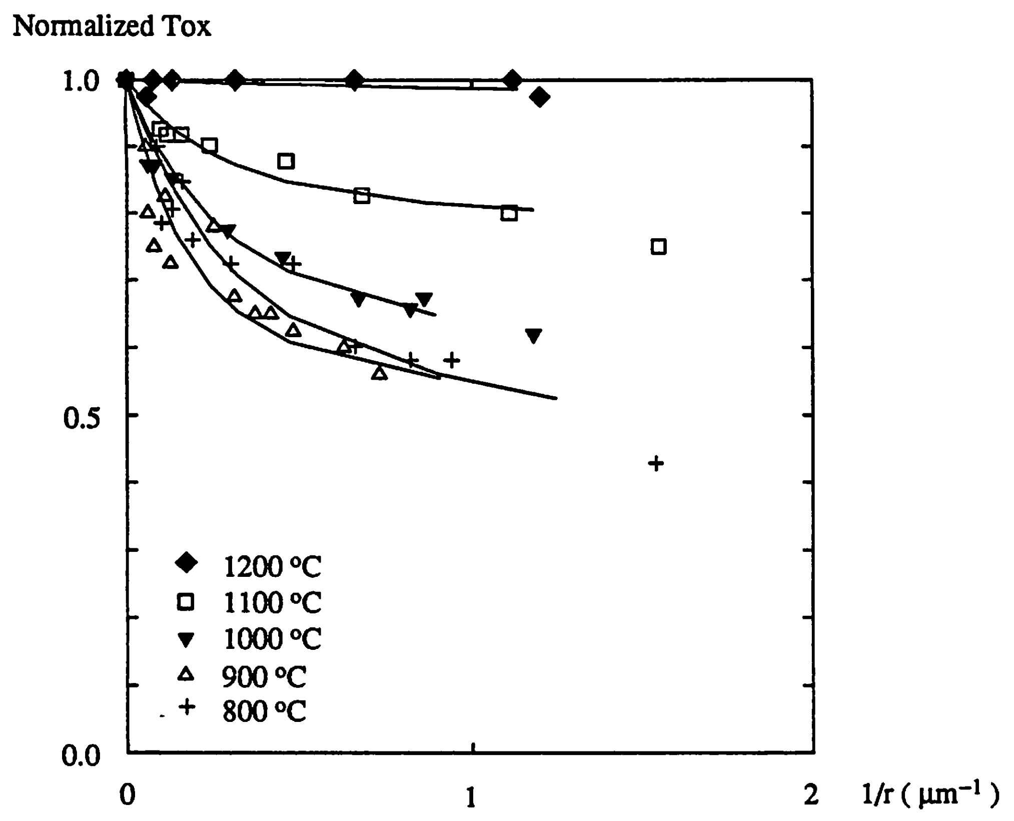


Fig. 3.9.2. Model fit to the concave experimental data. The solid lines are the model predictions.

Table 3.9.1 lists the values of the parameters used to obtain the fit shown in Fig. 3.9.1 and 3.9.2. The major finding is that the value for V_k and V_d are almost constant. In addition, the value for V_k is verified to be closed to 12.5 $\mathring{A}^3$, with an uncertainty of about $\pm 2\mathring{A}^3$. Trial runs using values of V_k that significantly differ from 12.5 $\mathring{A}^3$ have not been successful. The value of V_d is found to be closed to 75 $\mathring{A}^3$, with an uncertainty of about $\pm$ 10 $\mathring{A}^3$.

Fitting Parameters					
T (^oC)	Viscosity Parameters		Diffusivity and Surface Reaction Rate Parameters		
	η_0 (*poise*)	V_0 ($\AA^3$)	$\frac{D_{max}}{D_0}$	V_d ($\AA^3$)	V_k ($\AA^3$)
1200	5e11	800	1.0	75	12.5
1100	1.5e13	800	1.0		
1000	7e13	340	1.1		
900	5e14	220	1.6		
800	2e15	180	2.0		

Table 3.9.1. Model parameters used to obtain the results shown in Fig. 3.9.1 and 3.9.2.

The other parameters are temperature dependent. The volume parameter for the viscosity, V_0, varies strongly with temperature. This behavior is not understood. The temperature dependency of D_{max}/D_0 is actually not surprising. As we will detail in section 3.10 , intrinsic stress is known to exist in oxides grown on planar silicon surfaces. Planar oxides grown at higher temperature are in a relatively relaxed state and thus the oxidant diffusivity cannot be significantly increased by the application of a tensile pressure. At lower temperature, planar oxides are in a relatively compressed state and thus can be relaxed more by the tensile pressure. This account for the progressively increasing value for D_{max}/D_0 at lower temperature.

The extracted linear viscosity, η_0, varies strongly with temperature, which is expected. Our extracted viscosity data is compared with other independently measured viscosity for vitreous silica in section 3.11.

3.10. INTRINSIC STRESS IN THERMAL OXIDATION

EerNisse [13] showed in his wet O_2 experiments that a lateral compressive stress of about 7e9 $dyne/cm^2$ exists in oxide grown at 900 °C. This stress is shown as σ_{xx} in Fig. 3.10.1 (Note the reversal of the sign convention for σ_{xx}) :

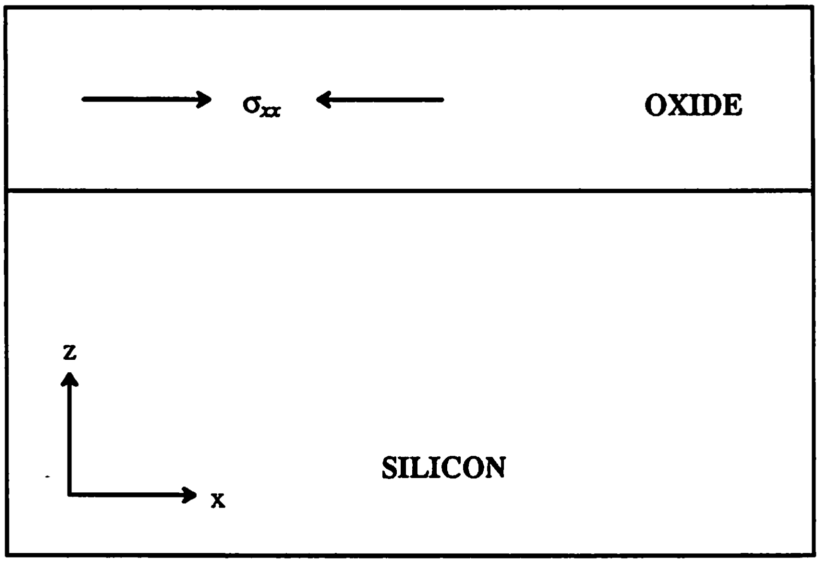


Fig. 3.10.1. The lateral stress observed in planar oxide films during oxidation. Note the reversal in the sign convention for σ_{xx}. There is also another component , σ_{yy} , in the direction perpendicular to the plane of the figure.

For oxides grown at and beyond 975 °C, the stress is smaller than the resolution of the measurement process, which was about 1e9 $dyne/cm^2$. The presence of this "intrinsic" stress in planar oxidation was attributed to the volumetric expansion and viscous flow during silicon-oxidation. We shall attempt to estimate the intrinsic stress using the shear-stress dependent viscosity model of Eqn. 3.8.3. Consider the volume expansion during the oxidation of a silicon atom, as shown in Fig. 3.10.2 (the "Manhattan" shapes are only for illustrative purpose) :

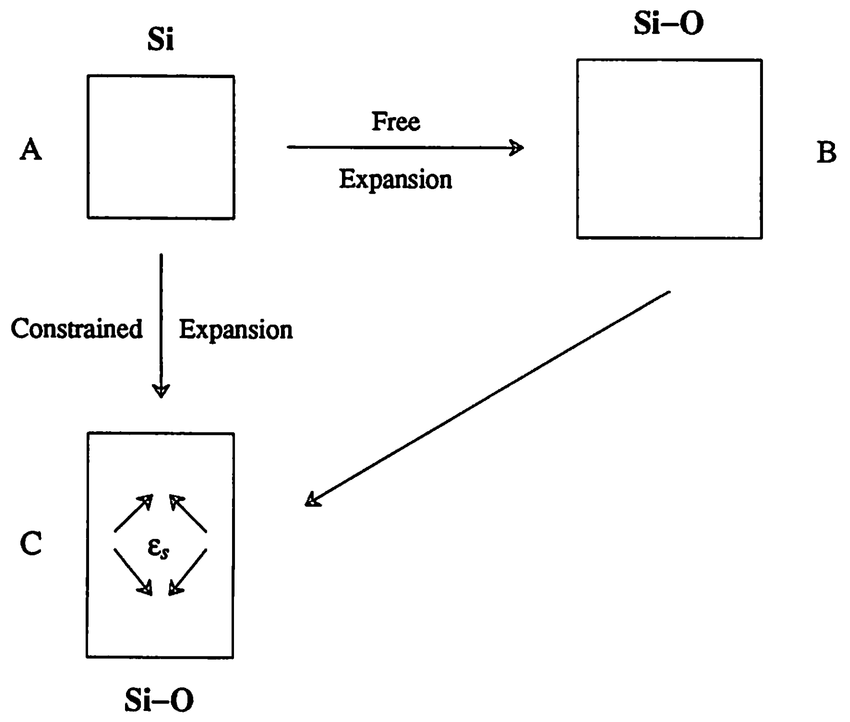


Fig. 3.10.2. Volume expansion during silicon oxidation. The shear-strain system (ε_s) involved in the B→C transition is shown by the four arrows in box-C.

If a silicon atom (A) is allowed to be oxidized in a stress-free condition, it should on average expand in all direction, as illustrated by the larger square on the right (B). However, the silicon substrate constrains the volume expansion in the lateral direction, forcing the oxide element to be elongated in the vertical direction (C). This process involves a large amount of strain, which can be estimated by the transformation from B to C in Fig. 3.10.2. A crude estimate gives a shear-strain of approximately 0.3. The direction of the shear-strain is given in Fig. 3.10.2, with another component of the same magnitude in a vertical plane perpendicular to the drawing. The initial shear-stress within the new oxide element can then be estimated by $2G\varepsilon_s$, where G is the shear modulus of elasticity and ε_s is the shear-strain. It is difficult to determine if the G of silicon or silicon dioxide should be used. In either case, the estimated value for the initial shear-stress should be in the range of 1e10 to 1e11 $dyne/cm^2$. At this range of high stress, the exact value is not important. The reason will become clear by examining the dynamics of the stress relaxation process.

Using a simple visco-elastic model for oxide, we can estimate the shear-stress relaxation in the oxide after its formation. Consider the mechanical analog for the simple visco-elastic model, using a spring and a dash-pot connected in series :

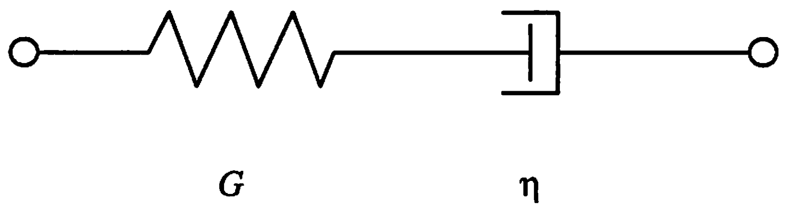


Fig. 3.10.3. A mechanical analog of a visco-elastic material. The spring represents the elastic component (G), and the dash-pot represents the viscous or dissipative component (η).

In the above, we can associate G with the spring, and η with the dash-pot. Given an initial stress σ_s applied on the visco-elastic element, the stress then relaxes at a rate given by

$$\frac{d\sigma_s}{dt} = \frac{\sigma_s}{\tau} \quad , \tag{3.10.1}$$

where

$$\tau = \frac{\eta(\sigma_s)}{G} \tag{3.10.2}$$

It turns out that when $\eta(\sigma_s)$ is given by Eqn. 3.8.3, Eqn. 3.10.1 can be solved analytically to yield

$$\sigma_s(t) = \frac{4kT}{V_0} \tanh^{-1}(\tanh(\frac{\sigma_{s0} V_0}{4kT}) \exp(\frac{-t}{\tau_0})) \quad , \tag{3.10.3}$$

where

$$\tau_0 = \frac{\eta_0}{G} \tag{3.10.4}$$

and η_0 is the linear viscosity value. To make a comparison of this shear-stress relaxation with the experimental results, we need to determine the shear-stress in the oxide. As indicated in

Fig. 3.10.1, the stress observed in EerNisse's experiment is a lateral stress in the oxide film. This stress system can be decomposed into two components: a shear-stress system and a hydrostatic component p, as shown in Fig. 3.10.4.

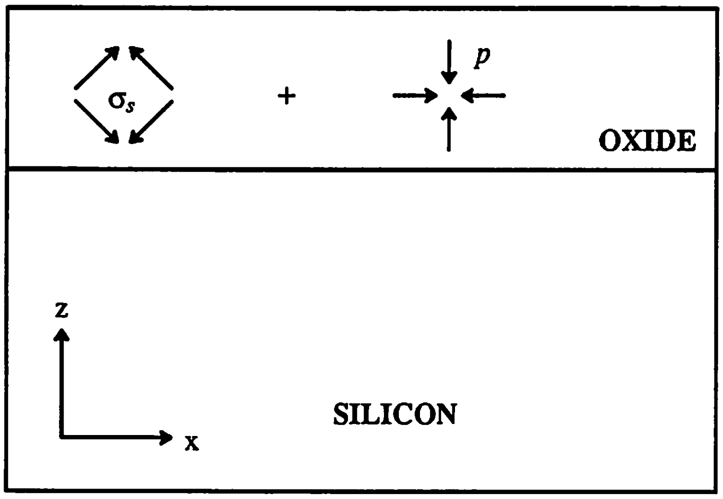


Fig. 3.10.4. Decomposition of the lateral stress system found in planar oxidation into a shear system (σ_s) and a hydrostatic pressure system (p).

The values of the different stress components are given by

$$p = \frac{\sigma_{xx} + \sigma_{yy}}{3} = \frac{2\sigma_{xx}}{3} \qquad (3.10.5)$$

and

$$\sigma_s = \sigma_{xx} - p = \frac{\sigma_{xx}}{3} \qquad (3.10.6)$$

Now, the measured value for σ_{xx} is the average value across the oxide thickness. Eqn. 3.10.3 only gives the shear-stress relaxation in an oxide film after it is formed. To calculate the average shear-stress across the oxide element, we can calculate the growth rate of the oxide as a function of time, and integrate Eqn. 3.10.3 appropriately. However, we shall only make an approximation by using the time average of $\sigma_s(t)$:

$$\overline{\sigma_s}(t) = \frac{\int_0^t \sigma_s(t')\, dt'}{t} \qquad (3.10.7)$$

This estimate will be exact if the oxidation rate is constant. The actual oxidation of planar silicon surfaces follow a linear-parabolic law[10], which is not a strong non-linear function. Thus the time-average should give a reasonable estimate of the film-average.

Using the extracted viscosity parameters, we plot in Fig. 3.10.5 the average shear-stress as a function of time. On the same figure, the range of the experimentally inferred shear-stress value for 900 $°C$ oxide is also given, showing a reasonable match between the model prediction and the experimental observation. The stress at the several different oxidation temperatures was initially quite high, but quickly relaxed to a lower level. Hence the actual values of the initial stresses are not important. For oxidation at and beyond 1000 $°C$, the stress level falls rapidly, making it difficult to measure the value from wafer bending experiments. The overall model prediction is consistent with the observations made by EerNisse.

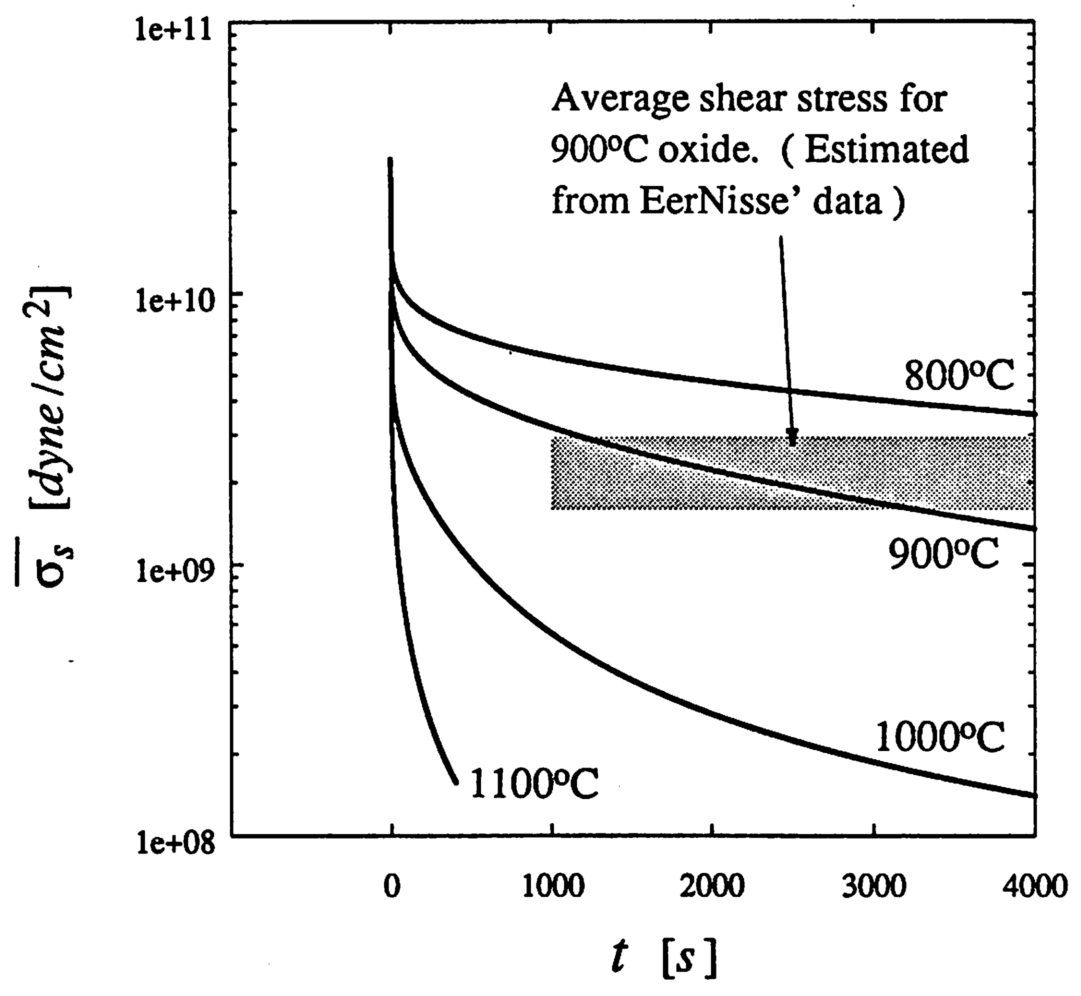


Fig. 3.10.5. Average shear-stress relaxation in planar oxide films during oxidation.

3.11. VISCOSITY OF VITREOUS SILICA

Hetherington et al measured the equilibrium viscosity of vitreous silica as a function of temperature and hydroxyl content[17]. The results are reproduced in Fig. 3.11.1.

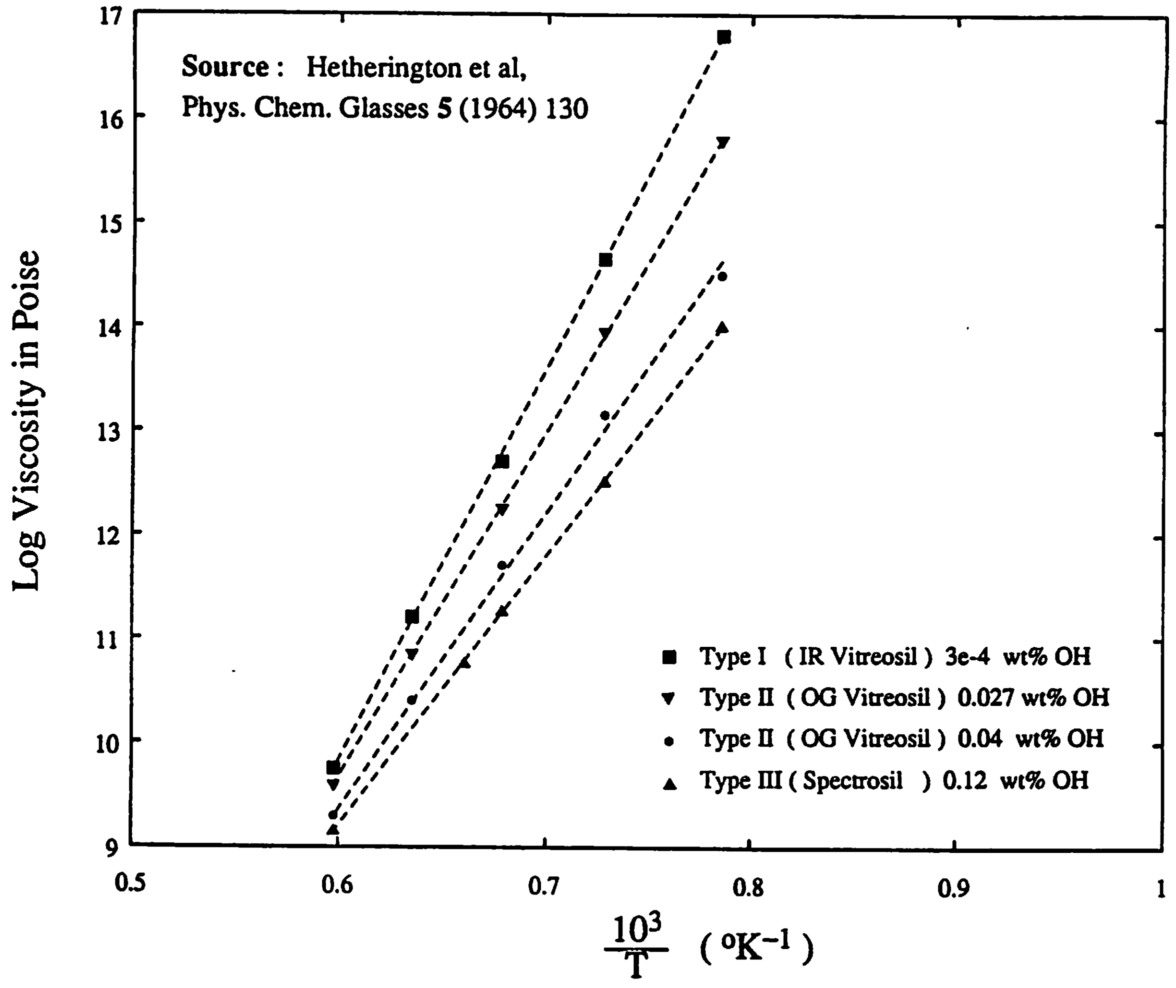


Fig. 3.11.1. Equilibrium viscosity of silica.

The different types of silica contain different levels of impurities, with type I silica containing the largest amount of metallic ions (about 30 to 100 ppm by weight of Al and 4 ppm of Na) and type III silica containing the smallest amount of metallic impurities. However, type I silica contains the least amount of hydroxyl groups (3 ppm by weight of OH groups), and type III contains the largest amount of hydroxyl groups (1200 ppm by weight of OH groups). It is known that impurities in silica generally loosen the glass-network and thus reduce the viscosity[17,18]. The data shown in Fig. 3.11.1 show that type I silica has the highest viscosity values, and type III has the lowest viscosity values. This reduction in the equilibrium viscosity should be attributed mainly to the level of hydroxyl content in the the silica. Hence, we need to estimate the hydroxyl content in wet thermal oxides in order to meaningfully compare this set of data to our extracted values. Unfortunately, the water content in a growing oxide is a function of location. Fig. 3.11.2 illustrates the distribution of the oxidant within the oxide during the oxidation process.

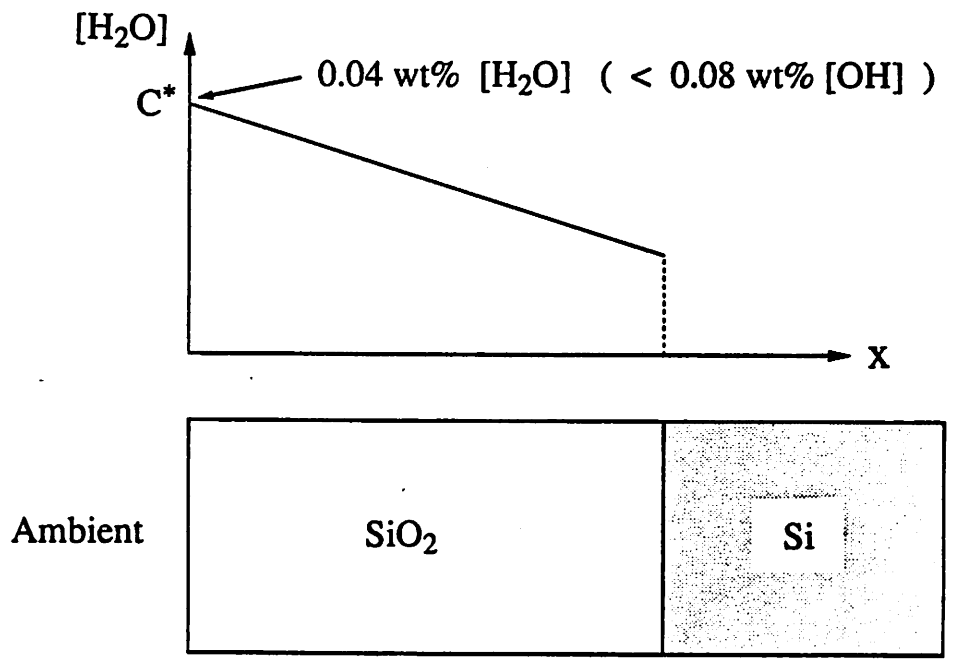


Fig. 3.11.2. A typical oxidant concentration profile in the oxide during oxidation.

For oxides grown under an H_2O partial pressure of 0.84 atm (a typical value used in most atmospheric pressure steam oxidation), the equilibrium oxidant concentration C^* is about 3e19 $molecules/cm^3$. This corresponds to about 400 ppm by weight of H_2O content near the surface of the oxide. Note that H_2O may or may not be in a molecular form. Experimental works on high pressure oxidation [19,20] show that the linear and parabolic rate parameters for steam oxidation are proportional to the H_2O partial pressure. Using the conventional interpretation of Henry's Law, the results indicate that the H_2O molecules do not dissociate in the oxide.

However, infra-red spectroscopy [21] and isotope-tracing experiments [22,23] clearly established that almost all of the H_2O molecules interact with the glass-network to form Si-OH bonds. The apparently contradicting experimental evidence are not well understood at present. In any case, assuming that the H_2O molecules fully interact with the glass network to form Si-OH bonds, there will be about 800 ppm by weight of hydroxyl content in the oxide at the ambient interface, and progressively lower hydroxyl content towards the silicon interface. It is difficult to define a meaningful average value for the hydroxyl content. But such an average value should be less than 800 ppm by weight of hydroxyl content. The type II silica should then be closest to the atmospheric steam oxide in terms of the amount of hydroxyl content. In fact, the agreement between the equilibrium viscosity of the type II silica with 400 ppm of hydroxyl content and our extracted linear viscosity values is surprisingly good at high temperature. The comparison is shown in the following figure :

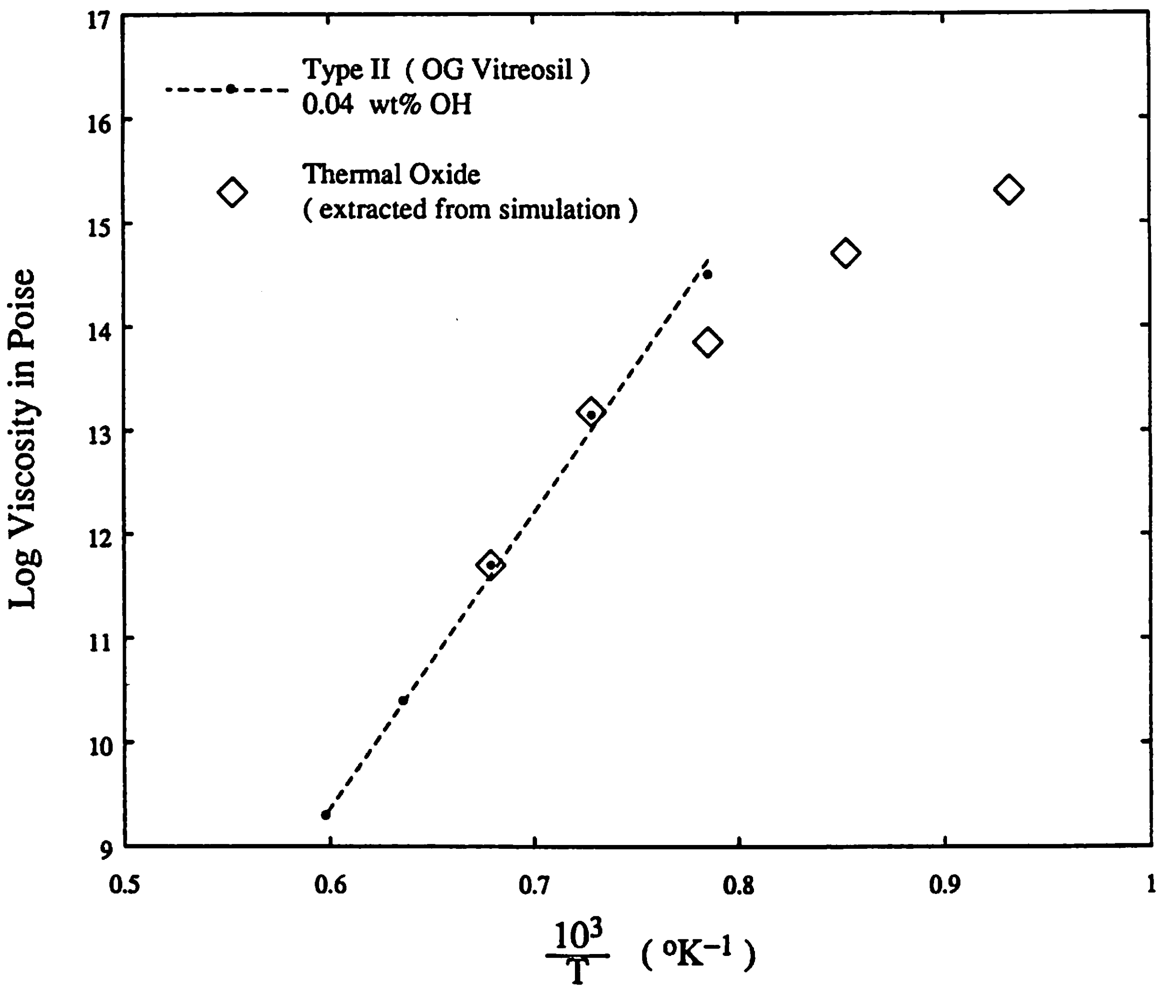


Fig. 3.11.3. Comparison of our extracted linear viscosity values with the equilibrium viscosity of type II silica containing 400 ppm by weight of OH groups.

The two sets of data almost coincide at 1100 °C and 1200 °C. At lower temperature, our extracted linear viscosity values deviate from the Arrhenius temperature dependence that the equilibrium viscosity values exhibit. This deviation is not surprising since it is highly unlikely that a growing oxide which has undergone large deformation at the time of its formation can exhibit equilibrium viscosity values at low temperature. It is a well known fact from the field of glass technology that glassy materials exhibit a range of viscosity values at the same temperature, depending on their thermal history[18]. At high temperature, typically higher than the glass transition temperature, glasses usually exhibit equilibrium viscosity values that follow a simple Arrhenius temperature dependence. However, the equilibrium viscosity at lower temperature can only be reached through slow cooling from high temperature. A typical plot of the log of non-equilibrium viscosity v.s. the reciprocal temperature for glasses is given in Fig 3.11.4:

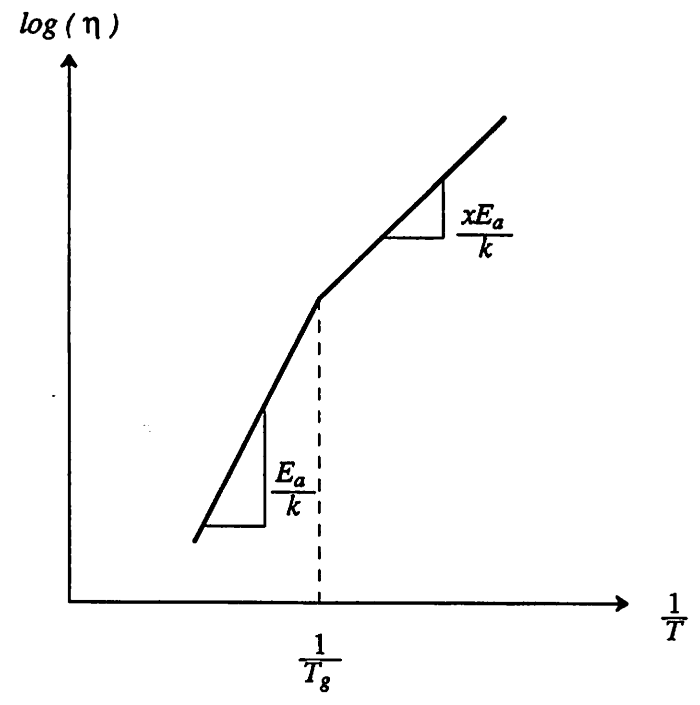


Fig. 3.11.4. A typical temperature dependence-curve for the viscosity of glasses.

In the above, T_g is called the glass transition temperature*. The apparent activation energy of the curve at lower temperature is a fraction of that at higher temperature (typically $x \approx 0.5$).[24] A glass material is essentially a liquid at a temperature well beyond T_g. The glass transition is a region of temperature in which molecular rearrangements occur on a scale of minutes or hours, so that the properties of the material change at a rate which can be conveniently observed. If the material does not have enough time to make complete structural

See page 120 of reference [24] for its technical definition.

rearrangement, it is in a non-equilibrium, glassy state. Without going into the technical definition of the T_g, we can qualitatively explain why our extracted data deviate significantly at 1000 °C and below. We shall first assume that the equilibrium viscosity for the wet thermal oxide under investigation is given by the equilibrium viscosity for type II silica with 400 ppm hydroxyl content. Define

$$\tau^* = \frac{\eta^*}{G} \tag{3.11.1}$$

where η^* is the equilibrium viscosity, and G is the shear modulus of elasticity for oxide, which is around 3e11 $dyne/cm^2$. Then τ^* should give a crude but useful estimate of the relaxation time constant for the structural rearrangement. The ratio of the total oxidation time over τ^* will then give an indication of the amount of structural rearrangement in the oxide. The larger the ratio is, the more complete is the structural rearrangement. The following table summarizes the results:

Temp	800°C	900°C	1000°C	1100°C	1200°C
Oxidation Time t	1440 min	300 min	96 min	40 min	24 min
τ^*	3700 hr	29 hr	18 min	50 s	2 s
$\frac{t}{\tau^*}$	0.006	0.17	5.3	48	720

Table 3.11.1. Ratio of the total oxidation time v.s. the characteristic relaxation time. The oxidation time at the various temperature was given by Kao[25].

For 1200 and 1100 °C oxidations, structural rearrangement should be easily accomplished well within the time scale of the oxidation process. Equilibrium viscosity should then be exhibited. For 900 and 800 °C oxidation, the oxide has little time to make structural rearrangement. Non-equilibrium (lower viscosity) values are thus expected. For 1000 °C oxidation, partial structural rearrangement occurs, which explains the slight deviation of our extracted data from the equilibrium viscosity data.

3.12. OTHER MODELS AND APPLICATIONS

One of the objectives of this project is to develop an ability to simulate oxidation processes that are encountered in IC fabrication. The models presented thus far are not enough to allow us to handle more interesting structures other than the cylindrical silicon structures discussed. In the following two sections we discuss some needed models that have not been fully understood or developed.

3.12.1. Oxidation of Sharp Silicon Corners.

In our earlier discussion, we have assumed that the oxidized silicon surface is relatively smooth so that we can meaningfully define a normal vector for the motion of the existing oxide region. However, we often encounter sharp silicon corners in some critical oxidation steps, such as the oxidation of silicon trenches. The oxidation rate around sharp corners is not well understood. During silicon oxidation, sharp concave silicon corners tend to smooth out. A slight error in the advancement of a sharp concave corner does not pose too serious a problem since the error diminishes after several time steps of the numerical computation, as the profile becomes smoother. However, a sharp convex silicon corner can become smoother, remain as sharp, or become sharper, depending on the oxidation condition. As a result, errors in the profile advancement algorithms may accumulate. Approximation of a sharp corner by small radius of curvature is impractical as it increases the computation time significantly. It actually does not avoid the problem of sharp convex corners over a longer run of computational steps. The numerical advancement of the Si/SiO$_2$ interface will eventually create a sharp corner at a high convex curvature region after several time steps of numerical computation. This behavior is inherent in string (discretized surface) advancement algorithms that employ deloop procedures. The solution to this problem is to find out how sharp convex corners actually advance during the oxidation process.

For lack of a systematic data on sharp-corner-oxidation, we currently approximate the advancement of the Si/SiO$_2$ interface using the surface-motion algorithm, which is illustrated in the following figure:

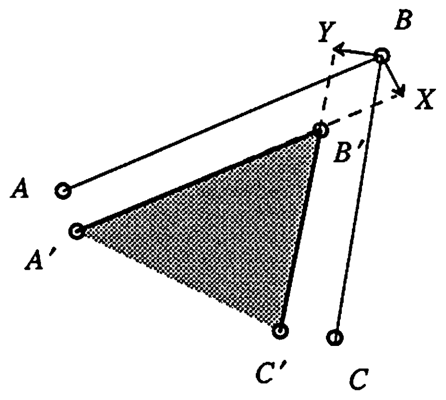


Fig. 3.12.1.1. Advancement of segmented surface by advancing the segments and calculating the intersection point.

The segment $\overline{AB}$ and $\overline{BC}$ represents the initial Si/SiO$_2$ interface near the sharp convex silicon corner (point B). Point B moves to B' after certain amount of oxidation. The coordinate of B' is located at the intersection between the line segments $\overline{A'X}$ and $\overline{YC'}$, which are moved from the segments $\overline{AB}$ and $\overline{BC}$ respectively. The amount of motion of the line segments is determined by the oxidant concentration and the reaction rate parameter at the Si/SiO$_2$ interface.

Several simulation runs for the oxidation of convex silicon corner at different temperatures were performed to check the accuracy of the program. Fig. 3.12.1.2 shows a typical example of the simulation results. Our simulation results do not agree very well with the limited experimental data provided by Marcus and Sheng[8]. Both the simulation results and the experimental data show sharpening of sharp silicon corners when the structures are oxidized at a relatively low temperature (800 to 1000 °C). However, the simulation results consistently show more oxidation retardation at sharp convex silicon corners than experimentally observed. At present, we believe that the error is mainly attributable to the basic surface-motion algorithm used. A more refined string-motion algorithm may be needed to improve the accuracy of the simulation results.

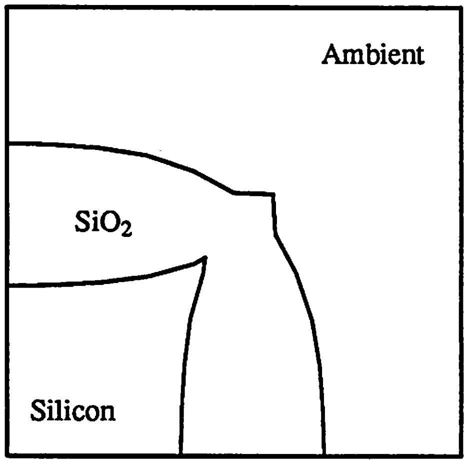


Fig. 3.12.1.2. A typical simulated result for the oxidation of a convex silicon corner.

3.12.2. Other Material Problems

Most interesting examples of silicon oxidation involve other materials such as silicon nitride (used mainly as oxidant mask) and poly-silicon layers. Examples of these oxidation processes are the various forms of the LOCOS process and the oxidation of the poly-silicon gate in MOS processes. These materials must also be modeled to simulate the oxidation of structures that contain them. Both silicon and silicon nitride are commonly assumed to be elastic. This assumption is reasonable when the materials are under low to moderate stress levels. However, in the oxidation of non-planar structures, the stress levels in the oxidized structures can easily be in the range of 1e9 to 1e10 $dyne/cm^2$. Many elastic materials deform plastically at this high stress level, especially at high temperature. Little work has been done in this area of modeling. Currently, nitride and silicon are modeled in CREEP as quasi-visco-elastic materials, in the sense that the materials are assumed to be elastic in every time-step of the numerical computation, assuming no accumulation of stress from the previous time-step. This is only a first implementation to allow us to generate some initial results on the LOCOS and poly-gate oxidation processes for further discussion.

Consider a LOCOS simulation illustrated in Fig. 3.12.2.1. The nitride layer is seen to be bent during the oxidation process. It has been shown that when the grown oxide is etched away, the bent nitride layer does not always return fully to its shape before the oxidation[26,27]. This suggests that either plastic deformation has occurred or the nitride is a non-linear visco-elastic material, with the viscosity of nitride probably behaving in a similar manner as that for oxide.

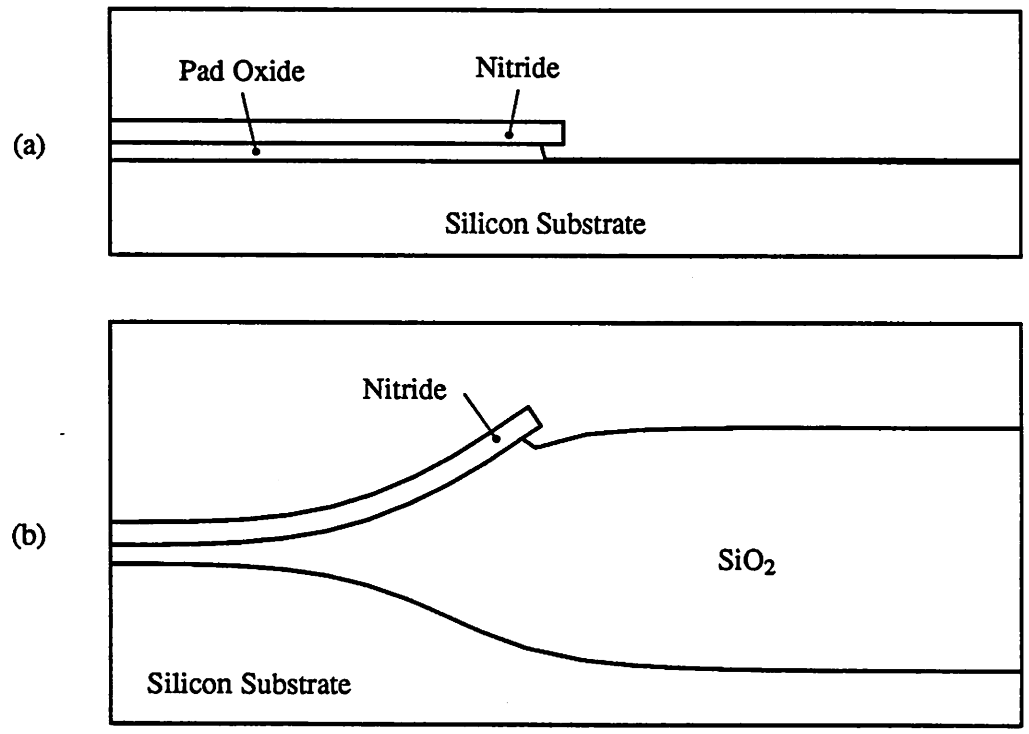
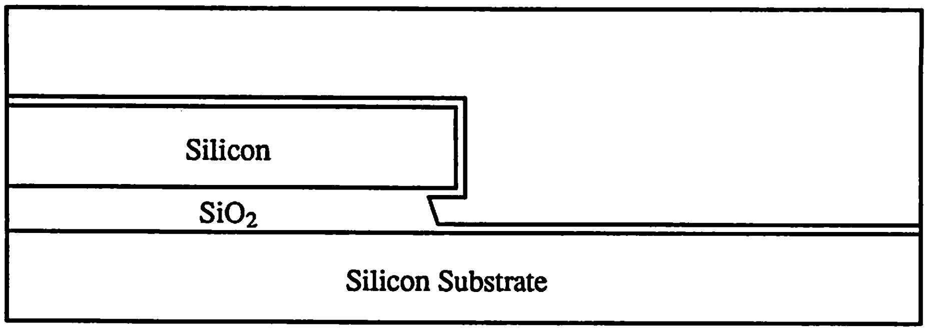


Fig. 3.12.2.1. Simulation of a LOCOS process: (a) the initial structure and (b) the structure after oxidation.

Consider the oxidation of a silicon gate. Fig. 3.12.2.2 shows a simulation example of such a structure. For simplicity, the silicon gate is assumed to be single crystalline with the same crystal orientation as the substrate. The simulation shows that the silicon gate is noticeably bent after a certain amount of oxidation. This bending has been observed in real experiments[28]. The amount of strain in the bent-region is in the range of several percent, indicating that plastic deformation is likely to occur. Aside from the possible anisotropy that results from the crystalline structure of silicon, it is possible that the mechanical behavior of silicon is similar to that of silicon nitride.

(a)



(b)

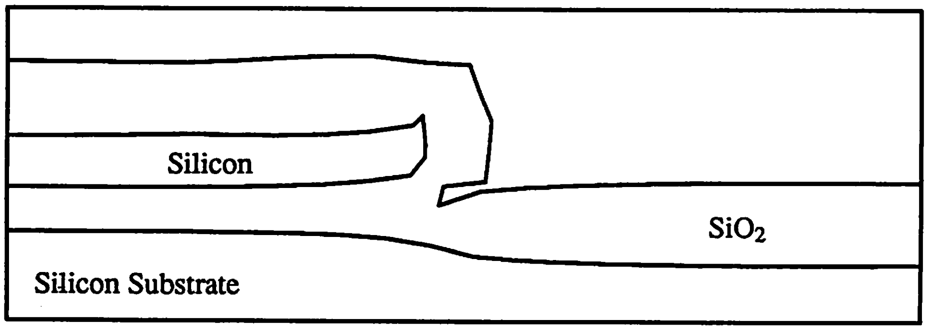


Fig. 3.12.2.2. Oxidation of silicon gate structure : (a) the initial structure and (b) the structure after oxidation.

Clearly, the modeling of the mechanical properties of both silicon and silicon nitride provides further challenges in the modeling of silicon oxidation.

3.13. CONCLUDING REMARKS

In this chapter, we have described a set of stress-dependent Deal-Grove parameters and an oxide viscosity model. By further introducing the visco-elastic model, we have successfully applied the shear-stress dependent viscosity model to explain the observed stress in planar oxide films during silicon oxidation. The results suggest that the non-linear visco-elastic model should be used to improve the accuracy of the program.

CHAPTER 4.

NUMERICAL ALGORITHMS

4.1. INTRODUCTION

Silicon oxidation is a coupled process of oxidant diffusion and oxide deformation. In general, the diffusion and deformation equations should be solved simultaneously. However, for the simple 2D Deal-Grove models which are not stress-dependent, the oxidant diffusion and oxide-deformation can be considered uncoupled and separately solved using the following steps :

1. Calculate the oxidant diffusion everywhere in the oxide.
2. Use the concentration at the Si/SiO_2 interface to calculate the oxidation-rate at that interface. This gives the velocity of the Si/SiO_2 interface and the velocity of the existing oxide layer at the original Si/SiO_2 interface, assuming that the silicon regions are rigid bodies.
3. Use the velocity of the existing oxide at the original Si/SiO_2 interface to calculate the deformation of the existing oxide region.
4. Use a small time-step Δt to calculate the profile of the newly formed oxide at the Si/SiO_2 interface, as well as the deformed profile of the existing oxide region.
5. Merge the new and old oxide regions together. This gives the new oxide profile.
6. Repeat the above 5 steps as many times as it takes to reach the total oxidation time desired.

It would be difficult to immediately explain the algorithms required to solve the stress-coupled oxidation models. Instead, we shall first explain the discretization algorithms required to solve the uncoupled problem. Section 4.2 describes the finite-element discretization of the virtual work equation for the flow/deformation of oxide. Section 4.3 describes some of the important aspects in the discretization of the oxidant diffusion equation. With the numerical tools for the uncoupled problem developed, the extension of the algorithms to solve the stress-coupled oxidation/deformation models of Eqn. 3.8.1 through 3.8.3 will then be described in section 4.4 . In section 4.5, the numerical algorithms for handling nitride and floating silicon regions will be described, and the assumption of the silicon as a rigid body will be removed. Methods to handle special geometries will be discussed in section 4.6 .

4.2. VIRTUAL WORK FORMULATION FOR THE OXIDE DEFORMATION

The laws of mechanics (eg, balance of force/momentum) can be expressed in many ways. The integral form provided by the virtual work statement is suitable for the finite-element discretization of continuum systems :

$$\int_{\Omega} \delta\boldsymbol{\varepsilon}\cdot\boldsymbol{\sigma}\, d\Omega = \int_{\Gamma} \delta\mathbf{u}\cdot\mathbf{f}\, d\Gamma + \int_{\Omega} \delta\mathbf{u}\cdot\mathbf{b}\, d\Omega \qquad (4.2.1)$$

In the above equation, $\boldsymbol{\varepsilon}$ is the strain tensor, $\boldsymbol{\sigma}$ is the stress tensor, **u** is the displacement of the system , **f** is the traction/stress applied on the boundary (Γ) of the system, and **b** is the body force applied within the bulk (Ω) of the system. $\delta\mathbf{u}$ represents an arbitrary or virtual displacement of each point in the system about its real displacement, and $\delta\boldsymbol{\varepsilon}$ represents the virtual strain due to the virtual displacement. The virtual work principle states that if the virtual displacements go through the real forces that exist in the system, the resulting internal virtual energy (the left hand term above) must be equal to the external virtual energy (the right hand terms). That is, Eqn. 4.2.1 is a statement of conservation of virtual energy*.

It is also possible to use another form of the virtual work equation, if instead of the displacement, the velocity is the quantity of interest:

$$\int_{\Omega} \delta\dot{\boldsymbol{\varepsilon}}\cdot\boldsymbol{\sigma}\, d\Omega = \int_{\Gamma} \delta\mathbf{v}\cdot\mathbf{f}\, d\Gamma + \int_{\Omega} \delta\mathbf{v}\cdot\mathbf{b}\, d\Omega \qquad (4.2.2)$$

* See page 58 of reference [29] for another interpretation as a weak form of equilibrium conditions.

The above equation is actually a virtual power statement, as **v** is the velocity-distribution of the system, and $\dot{\boldsymbol{\varepsilon}}$ is the strain-rate of the system. Nevertheless, it shall also be called the virtual work equation, and it will be used as the basis of our analysis in the next few sections.

Before we proceed to describe the discretization algorithms for solving Eqn. 4.2.2, it is worth making a final remark about the generality of the virtual work statements. Although Eqn. 4.2.1 seems to apply only to static problems, and Eqn. 4.2.2 to creep-flow problems, they both can be generalized to describe the dynamic behavior of continuum systems by invoking the d'Alembert Principle*. This is done by replacing the body force **b** of Eqn. 4.2.1 by ($\mathbf{b} - \rho\frac{d^2\mathbf{u}}{dt^2}$) , or that of Eqn. 4.2.2 by ($\mathbf{b} - \rho\frac{d\mathbf{v}}{dt}$), where ρ is the density of the material. The generalized equations are generally much harder to solve. Fortunately, Eqn. 4.2.2 is adequate for our modeling purpose as silicon-oxidation is a creep-flow problem.

4.2.1. Discretization of the Velocity

The velocity vector in a continuum system is a function of position (ie, a velocity field). We assume that if the velocities at a number of selected points in the system are known, then we know the velocity everywhere in some approximate sense. The selected points are called the grid-points or nodes of the system. In finite-element analysis, the approximation must be explicitly specified. The most common way of specifying this approximation is to subdivide the domain of interest into simple-shaped "elements". Each element has a certain number and pattern of nodes. Interpolation functions are then set up within each element to specify the velocity anywhere within the element as a function of the velocities at the grid-points:

$$\mathrm{v}(x,y) = f\,(\,\mathrm{V}_0, \mathrm{V}_1, \ldots, \mathrm{V}_{n-1}\,) \quad , \qquad (4.2.1.1)$$

where $\mathrm{V}_i = \begin{bmatrix} \mathrm{V}_{xi} \\ \mathrm{V}_{yi} \end{bmatrix}$ is the velocity of the i-th node, and n is the number of nodes on the element. The indices $(0,1,\ldots,n-1)$ are the local-node-numbers of the nodes with respect to the element, since there can be many other nodes outside the element. These indices corresponds to some

* See, for example, page 5 of reference [30].

other numbers in the global node-numbering system.

For simplicity of the analysis, the interpolation function within each element is assumed to be linearly dependent on the nodal velocities :

$$\mathbf{v}(x,y) = \sum_i \mathbf{N}_i(x,y)\mathbf{V}_i \quad , \qquad (4.2.1.2a)$$

where

$$\mathbf{N}_i(x,y) = \begin{bmatrix} N_i(x,y) & 0 \\ 0 & N_i(x,y) \end{bmatrix} \quad , \qquad (4.2.1.2b)$$

and each $N_i(x,y)$ is a scalar interpolation function that will be described shortly. In a more compact form the above equation can be represented in terms of matrix quantities :

$$\mathbf{v}(x,y) = \mathbf{N}\mathbf{V} \quad , \qquad (4.2.1.2c)$$

where

$$\mathbf{N} = [\mathbf{N}_0, \mathbf{N}_1, \ldots] \qquad (4.2.1.2d)$$

and

$$\mathbf{V} = \begin{bmatrix} \mathbf{V}_0 \\ \mathbf{V}_1 \\ \cdot \\ \cdot \end{bmatrix} \qquad (4.2.1.2e)$$

The interpolation functions for a 3-node triangular element are linear in x and y. To obtain the interpolation functions, we first consider the linear triangle of Fig. 4.2.1.1 :

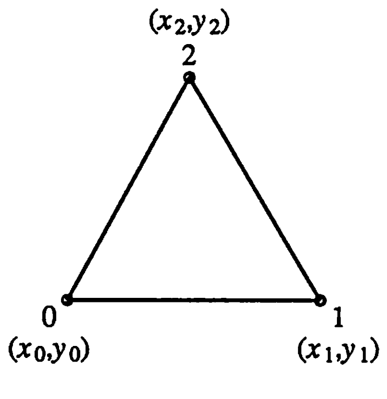


Fig. 4.2.1.1. A 3-node triangle.

Assume that $\phi(x,y)$ is a scalar profile to be interpolated over the element, using the nodal values ϕ_0, ϕ_1 and ϕ_2. In the form of Eqn. 4.2.1.2c, we write

$$\phi(x,y) = [\, N_0(x,y)\,,\; N_1(x,y)\,,\; N_2(x,y)\,] \begin{bmatrix} \phi_0 \\ \phi_1 \\ \phi_2 \end{bmatrix} \equiv \mathbf{N\Phi}\ , \tag{4.2.1.3}$$

bearing in mind that the N here is not the same as the N of Eqn. 4.2.1.2d because we now assume a single variable per node. Since $\phi(x,y)$ is linearly interpolated, it can be written in the form

$$\phi(x,y) = a + bx + cy \tag{4.2.1.4}$$

To get the values of a, b and c, we write

$$\begin{aligned} \phi_0 &= a + bx_0 + cy_0 \\ \phi_1 &= a + bx_1 + cy_1 \\ \phi_2 &= a + bx_2 + cy_2 \end{aligned} \tag{4.2.1.5}$$

or

$$\begin{bmatrix}\phi_0\\ \phi_1\\ \phi_2\end{bmatrix} = \mathrm{A}\begin{bmatrix}a\\ b\\ c\end{bmatrix} \quad , \tag{4.2.1.6}$$

where

$$\mathrm{A} = \begin{bmatrix}1 & x_0 & y_0\\ 1 & x_1 & y_1\\ 1 & x_2 & y_2\end{bmatrix} \tag{4.2.1.7}$$

We thus have

$$\begin{bmatrix}a\\ b\\ c\end{bmatrix} = \mathrm{A}^{-1}\begin{bmatrix}\phi_0\\ \phi_1\\ \phi_2\end{bmatrix} = \mathrm{A}^{-1}\mathbf{\Phi} \tag{4.2.1.8}$$

Hence

$$\phi(x,y) = [1\ x\ y]\begin{bmatrix}a\\ b\\ c\end{bmatrix} = [1\ x\ y]\mathrm{A}^{-1}\mathbf{\Phi} \tag{4.2.1.9}$$

Comparing the above with Eqn. 4.2.1.3 , we get

$$[\,N_0(x,y)\,,N_1(x,y)\,,N_2(x,y)\,] = [1\ x\ y]\mathrm{A}^{-1} \tag{4.2.1.10}$$

or

$$N_i = [1\ x\ y]\mathrm{col}_i(\mathrm{A}^{-1}) \quad , \qquad i = 1,2,3 \tag{4.2.1.11}$$

where $\mathrm{col}_i(\mathrm{A}^{-1})$ is the i-th column of A^{-1} . Solving A^{-1} symbolically, we obtain

$$N_0 = (\alpha_0 + \beta_0 x + \gamma_0 y)/2\Delta \tag{4.2.1.12a}$$

where $$\alpha_0 = x_1 y_2 - x_2 y_1 \tag{4.2.1.12b}$$

$$\beta_0 = y_1 - y_2 \tag{4.2.1.12c}$$

$$\gamma_0 = x_2 - x_1 \tag{4.2.1.12d}$$

Δ = Area of triangle

$$= (x_1 y_2 + x_0 y_1 + x_2 y_0 - x_2 y_1 - x_1 y_0 - x_0 y_2)\,/\,2 \tag{4.2.1.12e}$$

and so on. All the N_i's have the following properties :

$$N_i(x_i, y_i) = 1.0 \qquad (4.2.1.13a)$$

$$N_i(x_j, y_j) = 0.0 \quad , \quad i \neq j \qquad (4.2.1.13b)$$

Fig. 4.2.1.2 depicts the shape of the various N_i's for the linear triangle.

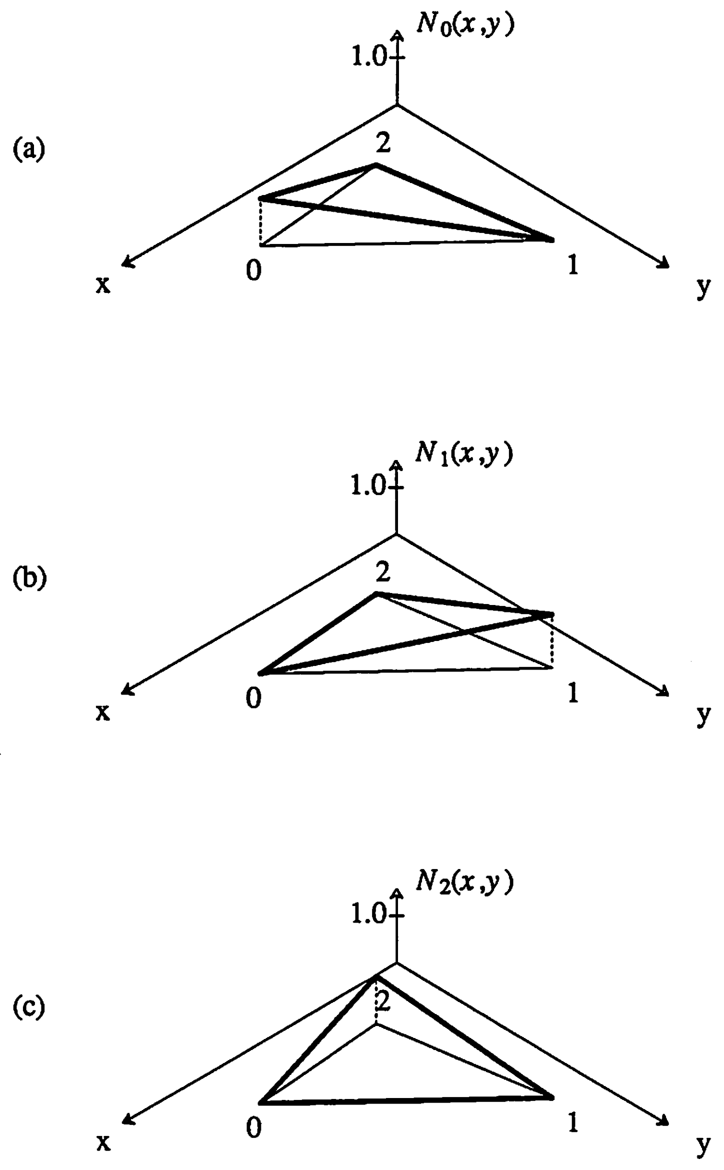


Fig. 4.2.1.2. Shape functions of a 3-node triangle: (a) N_0, (b) N_1 and (c) N_2.

It is also possible to evaluate the interpolation functions using the area-coordinate formulation. Consider the areas (Δ_0 , Δ_1 and Δ_2) of the various sub-regions in the following triangle :

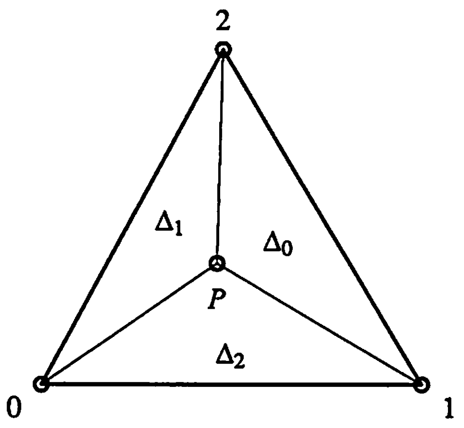


Fig. 4.2.1.3. Sub-regions of a triangle formed by drawing lines from a point P (x,y) to the vertices of the triangle.

As in Eqn. 4.2.1.12e, let Δ be the area of the triangle. Define the area-coordinates :

$$L_0 = \Delta_0 / \Delta$$

$$L_1 = \Delta_1 / \Delta \qquad (4.2.1.14)$$

$$L_2 = \Delta_2 / \Delta$$

An alternative way of specifying the point (x,y) is by using (L_0,L_1,L_2) , or simply (L_0,L_1) since $L_0+L_1+L_2 = 1.$

It can be easily seen that

$$L_i(x_i,y_i) = 1.0 \qquad (4.2.1.15a)$$

$$L_i(x_j,y_j) = 0.0 \quad , \quad i \neq j \qquad (4.2.1.15b)$$

Since all the L_i's are linear in x and y, we can identify them as the linear interpolation functions for a triangular element, which can be evaluated using Eqn. 4.2.1.12 .

The area-coordinates are often used in finite-element analyses where triangular elements are used. One of the reasons for this is that most numerical integration schemes samples the integrand at certain locations that are stationary if expressed in area-coordinates, regardless of the shape of the triangle*. Another reason is that the use of the area-coordinates simplifies the formulation of the interpolation functions for higher order triangular elements. Consider the second order triangle of the following figure.

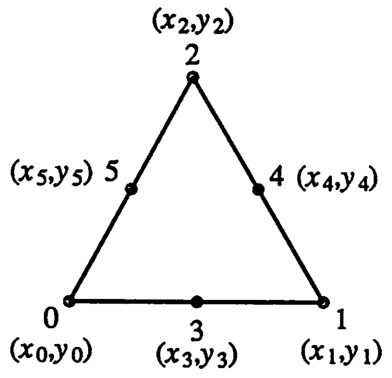


Fig. 4.2.1.4. A 6-node (quadratic) triangle

If we had to go through a similar procedure that we applied to the linear triangle, we would be facing a difficult task of inverting the matrix A of Eqn. 4.2.1.6, which will now be of size 6×6 . However, by observing that the interpolation functions must have the properties described by Eqn. 4.2.1.13, and that they are quadratic polynomials in x and y, we can generate them by inspecting their roots :

$$N_0 = L_0(2L_0 - 1) \tag{4.2.1.16a}$$

$$N_1 = L_1(2L_1 - 1) \tag{4.2.1.16b}$$

$$N_2 = L_2(2L_2 - 1) \tag{4.2.1.16c}$$

$$N_3 = 4L_0L_1 \tag{4.2.1.16d}$$

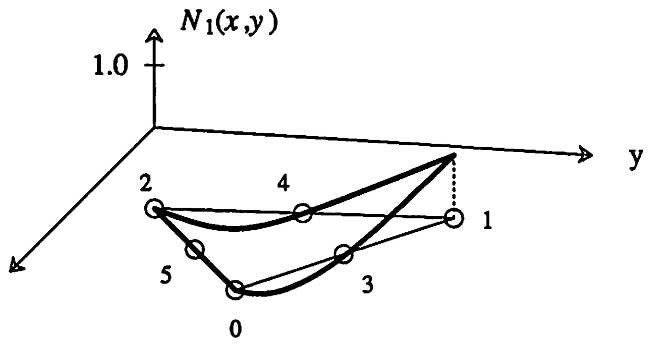
$$N_4 = 4L_1L_2 \tag{4.2.1.16e}$$

$$N_5 = 4L_2L_0 \tag{4.2.1.16f}$$

* See page 201 of reference [29] for some examples of numerical integration on triangular elements.

Fig. 4.2.1.5 shows some of the interpolation functions for a 6-node triangle :

(a)



(b)

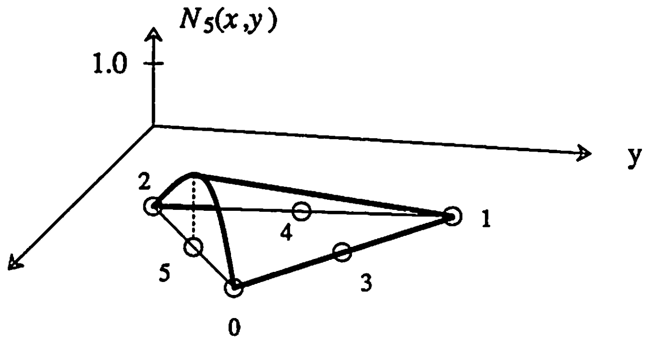


Fig. 4.2.1.5. Some of the interpolation functions of a 6-node triangle.

Other types of elements, not restricted to the family of triangular elements, are also possible. Triangular elements are used in the CREEP program because it is easier to write an automatic mesh-generator for triangular elements than for other elements such as the quadrilateral elements. Both the linear- and quadratic-triangles are used in oxidation simulation. The quadratic-triangles are needed for the analysis of incompressible-flow. This will be explained in section 4.2.6. Either the linear- or the quadratic-triangle can be used for the discretization of the diffusion equation. But for reasons of numerical-stability that will be explained in section

4.2.6 and 4.3, we use the linear-triangles for the diffusion equation.

4.2.2. Formulation and Discretization of the Strain-Rate

In a three-dimensional space, the strain-rate is generally a tensor defined as

$$\dot{\varepsilon} = \begin{bmatrix} \dot{\varepsilon}_{xx} & \dot{\varepsilon}_{xy} & \dot{\varepsilon}_{xz} \\ \dot{\varepsilon}_{xy} & \dot{\varepsilon}_{yy} & \dot{\varepsilon}_{yz} \\ \dot{\varepsilon}_{xz} & \dot{\varepsilon}_{yz} & \dot{\varepsilon}_{zz} \end{bmatrix} \tag{4.2.2.1}$$

In the virtual-work formulation, the strain-rate is usually written in a vector form :

$$\dot{\varepsilon} = \begin{bmatrix} \dot{\varepsilon}_{xx} \\ \dot{\varepsilon}_{yy} \\ \dot{\varepsilon}_{zz} \\ 2\dot{\varepsilon}_{xy} \\ 2\dot{\varepsilon}_{xz} \\ 2\dot{\varepsilon}_{yz} \end{bmatrix} \equiv \begin{bmatrix} \dot{\varepsilon}_{xx} \\ \dot{\varepsilon}_{yy} \\ \dot{\varepsilon}_{zz} \\ \dot{\gamma}_{xy} \\ \dot{\gamma}_{xz} \\ \dot{\gamma}_{yz} \end{bmatrix} \tag{4.2.2.2a}$$

For 2D plane-strain problems in the xy-plane (ie, the material is infinitely deep in the z-axis), the components $\dot{\varepsilon}_{xz} = \dot{\varepsilon}_{yz} = \dot{\varepsilon}_{zz} = 0$. It is not necessary to include all the zero terms in the strain-rate vector. The following form for the strain-rate vector will be used from now on:

$$\dot{\varepsilon} = \begin{bmatrix} \dot{\varepsilon}_{xx} \\ \dot{\varepsilon}_{yy} \\ \dot{\varepsilon}_{zz} \\ \dot{\gamma}_{xy} \end{bmatrix} = \begin{bmatrix} \dot{\varepsilon}_{xx} \\ \dot{\varepsilon}_{yy} \\ 0 \\ \dot{\gamma}_{xy} \end{bmatrix} \tag{4.2.2.2b}$$

The various components of the strain-rate are defined as:

$$\dot{\varepsilon}_{xx} = \frac{\partial v_x}{\partial x} \quad , \quad \dot{\varepsilon}_{yy} = \frac{\partial v_y}{\partial y} \quad , \quad \dot{\gamma}_{xy} = \frac{\partial v_x}{\partial y} + \frac{\partial v_y}{\partial x} \tag{4.2.2.2c}$$

Eqn. 4.2.2.2a includes $\dot{\varepsilon}_{zz}$ even though $\dot{\varepsilon}_{zz} = 0$. This is done because we would like to decompose the strain-rate vector into two orthogonal components :

$$\dot{\boldsymbol{\varepsilon}} = \dot{\boldsymbol{\varepsilon}}' + \dot{\boldsymbol{\varepsilon}}'' \tag{4.2.2.3}$$

where $\dot{\boldsymbol{\varepsilon}}'$ is the shear component and $\dot{\boldsymbol{\varepsilon}}''$ is the dilatational/volumetric component of $\dot{\boldsymbol{\varepsilon}}$. This decomposition is made partly because the shear and volumetric strain-rates can often be related to their stress-counterparts in a form given by:

$$\boldsymbol{\sigma}' = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} P\dot{\boldsymbol{\varepsilon}}' \quad , \quad \boldsymbol{\sigma}'' = Q\dot{\boldsymbol{\varepsilon}}'' \tag{4.2.2.4}$$

where P and Q are scalar differential-operators, $\boldsymbol{\sigma}'$ and $\boldsymbol{\sigma}''$ are the shear and hydrostatic stress-vectors respectively. This decomposition makes $\boldsymbol{\sigma}'$ also orthogonal to $\dot{\boldsymbol{\varepsilon}}''$ and $\boldsymbol{\sigma}''$ to $\dot{\boldsymbol{\varepsilon}}'$, allowing some simplification in the virtual work formulation (see section 4.2.4). The decomposition is also necessary because the numerical method we have chosen requires that the shear and the hydrostatic terms in the virtual work equation be integrated differently when the material is nearly incompressible. Further discussion on this is given in section 4.2.6.

The volumetric strain-rate is given by:

$$\dot{\boldsymbol{\varepsilon}}'' = \begin{bmatrix} \dot{\varepsilon}_v \\ \dot{\varepsilon}_v \\ \dot{\varepsilon}_v \\ 0 \end{bmatrix} \tag{4.2.2.5a}$$

$$\dot{\varepsilon}_v = \frac{\dot{\varepsilon}_{xx}+\dot{\varepsilon}_{yy}+\dot{\varepsilon}_{zz}}{3} = \frac{\dot{\varepsilon}_{xx}+\dot{\varepsilon}_{yy}}{3} \tag{4.2.2.5b}$$

Therefore

$$\dot{\boldsymbol{\varepsilon}}' = \begin{bmatrix} \dot{\varepsilon}_{xx} - \dot{\varepsilon}_v \\ \dot{\varepsilon}_{yy} - \dot{\varepsilon}_v \\ -\dot{\varepsilon}_v \\ \dot{\gamma}_{xy} \end{bmatrix} = \begin{bmatrix} \frac{2}{3}\dot{\varepsilon}_{xx} - \frac{1}{3}\dot{\varepsilon}_{yy} \\ \frac{-1}{3}\dot{\varepsilon}_{xx} + \frac{2}{3}\dot{\varepsilon}_{yy} \\ \frac{-1}{3}\dot{\varepsilon}_{xx} - \frac{1}{3}\dot{\varepsilon}_{yy} \\ \dot{\gamma}_{xy} \end{bmatrix}$$

$$= \begin{bmatrix} \frac{2}{3}\frac{\partial v_x}{\partial x} - \frac{1}{3}\frac{\partial v_y}{\partial y} \\ \frac{-1}{3}\frac{\partial v_x}{\partial x} + \frac{2}{3}\frac{\partial v_y}{\partial y} \\ \frac{-1}{3}\frac{\partial v_x}{\partial x} - \frac{1}{3}\frac{\partial v_y}{\partial y} \\ \frac{\partial v_x}{\partial y} + \frac{\partial v_y}{\partial x} \end{bmatrix}$$

$$= \begin{bmatrix} \frac{2}{3}\frac{\partial}{\partial x} , & \frac{-1}{3}\frac{\partial}{\partial y} \\ \frac{-1}{3}\frac{\partial}{\partial x} , & \frac{2}{3}\frac{\partial}{\partial y} \\ \frac{-1}{3}\frac{\partial}{\partial x} , & \frac{-1}{3}\frac{\partial}{\partial y} \\ \frac{\partial}{\partial y} , & \frac{\partial}{\partial x} \end{bmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix} \equiv \mathbf{L}'\mathbf{v} \qquad (4.2.2.6)$$

where $\mathbf{L}'$ is the differential operator for the shear strain-rate.

Similarly, we have

$$\dot{\boldsymbol{\varepsilon}}'' = \begin{bmatrix} \dot{\varepsilon}_v \\ \dot{\varepsilon}_v \\ \dot{\varepsilon}_v \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{3}\frac{\partial}{\partial x} & \frac{1}{3}\frac{\partial}{\partial y} \\ \frac{1}{3}\frac{\partial}{\partial x} & \frac{1}{3}\frac{\partial}{\partial y} \\ \frac{1}{3}\frac{\partial}{\partial x} & \frac{1}{3}\frac{\partial}{\partial x} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix} \equiv \mathbf{L}''\mathbf{v} \qquad (4.2.2.7)$$

and $\mathbf{L}''$ is the differential operator for the volumetric strain-rate.

The discretized strain-rate vectors can now be expressed as

$$\dot{\boldsymbol{\varepsilon}}' = \mathbf{L}'\mathbf{v} = \mathbf{L}'\mathbf{N}\mathbf{V} \qquad (4.2.2.8)$$

$$\dot{\boldsymbol{\varepsilon}}'' = \mathbf{L}''\mathbf{v} = \mathbf{L}''\mathbf{N}\mathbf{V} \qquad (4.2.2.9)$$

4.2.3. Formulation and Discretization of the Stress

The stress-tensor is similar to the strain-tensor and is generally expressed as

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{xy} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{xz} & \sigma_{yz} & \sigma_{zz} \end{bmatrix} \tag{4.2.3.1}$$

In 2D plane-strain problems , $\sigma_{xz} = \sigma_{yz} = 0$ and $\sigma_{zz} \neq 0$ in general. Hence, we rewrite the stress into a vector form with the following components :

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{zz} \\ \sigma_{xy} \end{bmatrix} \tag{4.2.3.2}$$

Again, $\boldsymbol{\sigma}$ is decomposed into a pure shear and pure hydrostatic term :

$$\boldsymbol{\sigma} = \boldsymbol{\sigma}' + \boldsymbol{\sigma}'' \tag{4.2.3.3a}$$

where
$$\boldsymbol{\sigma}'' = \begin{bmatrix} -p \\ -p \\ -p \\ 0 \end{bmatrix} \tag{4.2.3.3b}$$

and $p = -\frac{(\sigma_{xx} + \sigma_{yy} + \sigma_{zz})}{3}$ is the hydrostatic pressure. (4.2.3.3c)

For incompressible analysis in 2D plane-strain problems, $\sigma_{zz} = \frac{(\sigma_{xx} + \sigma_{yy})}{2}$ and so

$$p = \frac{-(\sigma_{xx} + \sigma_{yy})}{2} \tag{4.2.3.3d}$$

The shear term of the stress vector is thus given by

$$\boldsymbol{\sigma}' = \begin{bmatrix} \sigma_{xx} + p \\ \sigma_{yy} + p \\ \sigma_{zz} + p \\ \sigma_{xy} \end{bmatrix} \tag{4.2.3.4}$$

At this point, we need to relate the stress with the strain-rate. Currently, silicon dioxide is modeled in the CREEP program as a viscous fluid which is nearly incompressible. This model is given by:

$$\sigma' = \eta \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \dot{\varepsilon}' \equiv \eta D \dot{\varepsilon}' \qquad (4.2.3.5)$$

$$\sigma'' = \alpha \dot{\varepsilon}'' \quad , \qquad \alpha \gg \eta \qquad (4.2.3.6)$$

where η is the viscosity and α is a number much larger than η so as to approximate the incompressible condition. Normally, the hydrostatic term σ'' is related to the volumetric strain through the bulk modulus. However, in the limit that $\alpha \gg \eta$, the volumetric strain can be replaced by the the volumetric strain-rate $\dot{\varepsilon}''$ without much change in the results.

The discretized version of the stress vectors are then given by

$$\sigma' = \eta D\, L'NV \qquad (4.2.3.7)$$

$$\sigma'' = \alpha\, L''NV \qquad (4.2.3.8)$$

4.2.4. Discretization of the Virtual Work Equation

The virtual work equation can now be discretized. First, Eqn. 4.2.2 is rewritten using the column vector form of stress and strain-rate :

$$\int_\Omega \delta\dot{\varepsilon}^T \sigma \, d\Omega = \int_\Gamma \delta v^T f \, d\Gamma \qquad (4.2.4.1)$$

For silicon oxidation simulation, the body force term of Eqn. 4.2.2 is negligible compared to the other 2 terms. It has thus been omitted in the above equation.

Now, consider the first term

$$\int_\Omega \delta\dot{\varepsilon}^T \sigma \, d\Omega = \int_\Omega (\, \delta\dot{\varepsilon}'^T + \delta\dot{\varepsilon}''^T)(\, \sigma' + \sigma'' \,) \, d\Omega \qquad (4.2.4.2)$$

It can be shown that the cross-terms $\delta\dot{\varepsilon}'^T\sigma'' = \delta\dot{\varepsilon}''^T\sigma' = 0$ by explicitly evaluating the cross-terms. Thus we have :

$$\int_\Omega \delta\dot{\varepsilon}^T \sigma \, d\Omega = \int_\Omega \delta\dot{\varepsilon}'^T \sigma' \, d\Omega + \int_\Omega \delta\dot{\varepsilon}''^T \sigma'' \, d\Omega$$

$$= \int_\Omega \delta(L'NV)^T \eta D \,(L'NV)\, d\Omega + \int_\Omega \delta(L''NV)^T \alpha \,(L''NV)\, d\Omega$$

$$= \int_\Omega \delta V^T (L'N)^T \eta D\, (L'N)\, d\Omega\, V + \int_\Omega \delta V^T (L''N)^T \alpha\, (L''N)\, d\Omega\, V$$

$$= \delta V^T \left[\int_\Omega (L'N)^T \eta D\, (L'N) + (L''N)^T \alpha\, (L''N)\ d\Omega \right] V \qquad (4.2.4.3)$$

For the right hand term of Eqn. 4.2.4.1, we have

$$\int_\Gamma \delta v^T f\, d\Gamma = \int_\Gamma \delta (NV)^T f\, d\Gamma = \delta V^T \int_\Gamma N^T f\, d\Gamma \qquad (4.2.4.4)$$

Since the virtual work equation must hold true for any arbitrary variation of δv , δV^T can be factored out from the virtual work equation :

$$\left[\int_\Omega (L'N)^T \eta D\, (L'N) + (L''N)^T \alpha\, (L''N)\ d\Omega \right] V = \int_\Gamma N^T f\, d\Gamma \qquad (4.2.4.5)$$

or

$$KV = F_l \qquad (4.2.4.6a)$$

where

$$K = K' + K'' \quad , \qquad (4.2.4.6b)$$

$$K' = \int_\Omega (L'N)^T \eta D\, (L'N)\, d\Omega \quad , \qquad (4.2.4.6c)$$

$$K'' = \int_\Omega (L''N)^T \alpha\, (L''N)\ d\Omega \quad , \qquad (4.2.4.6d)$$

$$F_l = \int_\Gamma N^T f\, d\Gamma \qquad (4.2.4.6e)$$

The matrix **K** is known as the stiffness-matrix of the discretized system, and F_l is called the equivalent nodal-load vector. For a system with n nodes, K is a $2n \times 2n$ matrix, and F_l is a column vector with $2n$ entries, since each node has 2 variables : v_x and v_y. But it is convenient to look at K as an $n \times n$ matrix with entires of 2×2 sub-matrices, because it simplifies the indexing of the matrix-entries. By doing this, we were able to write a direct-solver that solves Eqn. 4.2.4.6a at a rate of approximately 3 times faster than a general purpose direct-solver. Naturally, F_l is also grouped into a column-vector with n entries of 2-component column vectors. In this way, the individual entries of K are given by

$$K_{ij} = \int_{\Omega} (L'N_i)^T \eta D (L'N_j) + (L''N_i)^T \alpha (L''N_j) \, d\Omega \tag{4.2.4.7}$$

and of F_l :

$$F_{li} = \int_{\Gamma} N_i^T f \, d\Gamma \tag{4.2.4.8}$$

4.2.5. Assembly of the Stiffness Matrix

Since L' and L'' are first order differential operators, $L'N$ and $L''N$ can only contain finite-jump discontinuity between elements. Hence, it is possible to assemble **K** from the individual element stiffness matrix K_e :

$$K = \sum_e K_e \tag{4.2.5.1}$$

$$K_e = \int_{\Omega_e} (L'N)^T \eta D (L'N) + (L''N)^T \alpha (L''N) \, d\Omega \tag{4.2.5.2}$$

Note that the assembled matrix equation is singular, as the Dirichlet boundary condition of the system has not been specified. For silicon-oxidation, the Dirichlet boundary is the boundary where the velocity of the system is known (ie, the Si/SiO_2 interface). Specification of the known velocities in the discretized equation is performed by modifying the appropriate equations contained in Eqn. 4.2.4.6a. For example, if the velocity of node m is known to be equal to V_{m0} , the original equation in Eqn. 4.2.4.6a corresponding to node m (row m of **K** and F_l) can be changed to the equation

$$V_m = V_{m0} \tag{4.2.5.3}$$

by simply setting K_{mm} to a 2×2 identity matrix, the other entries of row m to null-matrices , and F_{lm} to V_{m0}. The resulting matrix-equation will be slightly different from Eqn. 4.2.4.6a and non-singular. The solution of which yields an approximate velocity profile in the oxide.

4.2.6. Difficulty in the Incompressible Formulation

It turns out that if each of the $\mathbf{K}_{ij}$ is integrated exactly from Eqn. 4.2.4.7, the solution to Eqn. 4.2.4.6a gives a poor approximation to the solution of the continuum system. This phenomena has been reported many times in the finite-element literature. The problem is attributed to an excessive discretization error from the finite-element interpolation scheme, when used in near-incompressible analysis. Simply stated, there is no possible value of $\mathbf{V}$ that makes the velocity distribution approximate the actual viscous-incompressible solution if either the linear or the quadratic elements are used. Mathematically, the poor approximation is attributed to an inadequate number of zero energy modes for the hydrostatic term $\mathbf{K}''$ of Eqn. 4.2.4.6d. To overcome this problem, we have to settle for a good approximation of the velocity, but not the incompressibility condition, since a slight error in the velocity distribution can easily lead to a large error in the incompressibility condition (in terms of energy consideration). This can be accomplished by using low-order numerical integration to evaluate $\mathbf{K}''$, so as to artificially increase the number of its zero-energy modes. Here lies the reason that a linear-triangle cannot be used in incompressible analysis: the lowest order integration scheme (1 point quadrature at the centroid) for a triangle will integrate both $\mathbf{K}'$ and $\mathbf{K}''$ of Eqn. 4.2.4.6b exactly, if a linear-triangle is used. If a quadratic-element is used, we can use quadratic integration (3 point quadrature which leads to exact integration if the viscosity is constant within the element) for $\mathbf{K}'$ and linear-integration (1 point quadrature) for $\mathbf{K}''$, as shown in following figure :

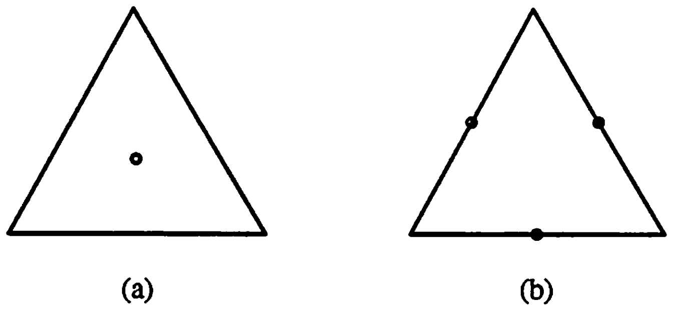


Fig. 4.2.6.1. Numerical integration points for triangular elements : (a) centroid for first order (linear) integration and (b) mid-side-nodes for second order (quadratic) integration.

The resulting solution of Eqn. 4.2.4.6a when the mixed integration scheme is employed is well behaved and converges to the exact solution when the size of the elements decreases. One way to interpret the effect of reduced-integration is that it enforces incompressibility condition only at the reduced integration-point (centroid of the triangles), thus placing little constraint on the overall shear-deformation of the discretized system. The implication is that the hydrostatic pressure should only be evaluated at the centroids of the triangles. Since the hydrostatic pressure can be directly evaluated only at the centroid of the triangles, and the oxidant diffusivity depends on the hydrostatic pressure in our stress-dependent oxidation models, we should use the linear-triangles to discretize the oxidant diffusion equation. We shall elaborate on this later in the next section.

4.3. DISCRETIZATION OF THE OXIDANT DIFFUSION EQUATIONS

The finite-element formulation for the oxidant diffusion equations of Eqn. 3.2.1 through 3.2.5 is given by :

$$\left[\int_{\Omega} (\nabla \mathbf{N})^T D_{ox} (\nabla \mathbf{N})\ d\Omega + \int_{\Gamma_s} \mathbf{N}^T k_s \mathbf{N}\ d\Gamma\right] \mathbf{C} = 0 \tag{4.3.1}$$

or

$$\mathbf{K_c C} = \mathbf{0} \tag{4.3.2}$$

where $\mathbf{N}$ is the appropriate interpolation function, D_{ox} is the oxidant diffusivity, k_s is the surface reaction rate parameter and Γ_s is the Si/SiO$_2$ boundary. The assembled matrix $\mathbf{K_c}$ is singular since the Dirichlet boundary condition of the diffusion equation (in this case, the oxide/ambient boundary) has not been specified. By a similar procedure described in section 4.2.5, we can enforce the Dirichlet boundary condition and form a new matrix equation that we shall write as

$$\mathbf{K_c C} = \mathbf{F_c} \tag{4.3.3}$$

where the two $\mathbf{K_c}$ in Eqn. 4.3.2 and 4.3.3 are different. In subsequent sections, when we refer to $\mathbf{K_c}$, we shall mean the one in Eqn. 4.3.3.

The derivation of Eqn. 4.3.1 and the assembly process can be found in many textbooks on FEM (eg, reference [29]) and also in reference [31]. We will not repeat the derivation here but make a special note concerning the evaluation of the term $\int_{\Gamma_s} \mathbf{N}^T k_s \mathbf{N}\ d\Gamma$. First, let us call this term the loss term, since it represents a "sink" of oxidant. Consider the oxidation of a LOCOS structure with a uniformly thin pad-oxide, as illustrated in Fig. 4.3.1 . Then, the oxidant concentration under the oxide region covered by the nitride can be calculated using the Fourier-Series analysis, assuming that the structure is periodic in the x-direction. The oxidant concentration is then approximately given by

$$C(x,y) = C_0 \cos(\sqrt{k_s/t_{ox}D_{ox}}(t_{ox} - y))\exp(-x\sqrt{k_s/t_{ox}D_{ox}}) \tag{4.3.4}$$

Where C_0 is the oxidant concentration in the oxide at the Si/SiO$_2$ interface and t_{ox} is the oxide thickness. The approximation approaches the exact solution as t_{ox} decreases. For very small

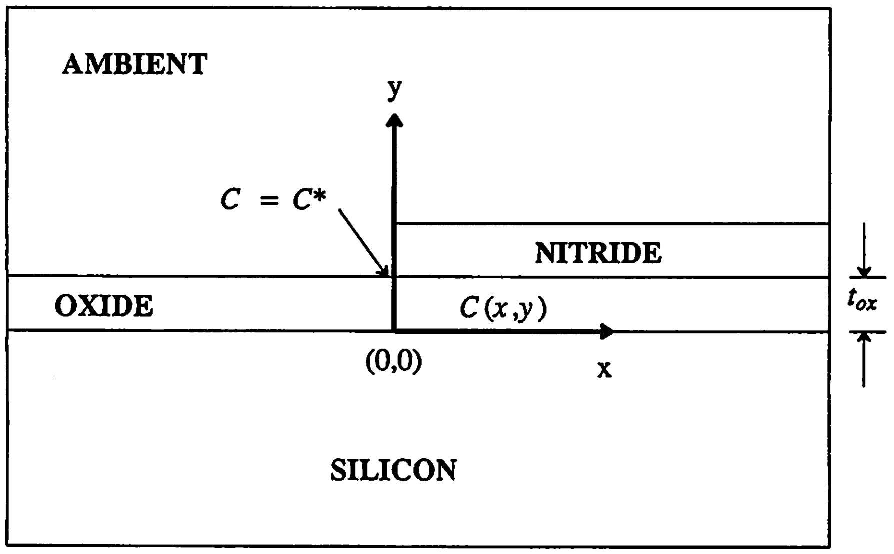


Fig. 4.3.1. A locos structure with thin pad oxide at the beginning of the oxidation process. Eqn. 4.3.4 describes the oxidant concentration within the oxide to the right of the y-axis.

t_{ox}, the solution is practically given by

$$C(x,y) = C_0 \exp(-x \sqrt{k_s/t_{ox}D_{ox}}) \tag{4.3.5}$$

Unfortunately, if the loss term is evaluated exactly (using quadratic numerical integration if linear-elements are used), the finite-element solution of the oxidant concentration under the nitride region can become oscillatory. The solution of oxidant concentration oscillates between positive and negative values in such regions, creating a non-physical condition for the oxidation rate. This oscillatory solution is obtained when the product of $\sqrt{k_s/t_{ox}D_{ox}}$ and the element length exceeds some critical value. The oscillation is caused by an excessive discretization error when the element length along the x-direction is too coarse to approximate the rapid exponential drop of the oxidant concentration along the x-direction. The situation can be analyzed by noting that the thin-oxide system is effectively a one dimensional system. First, the flux of oxidant across an arbitrary vertical line at location x (under the nitride region) is approximately given by

$$\text{Flux} = -t_{ox}D_{ox}\frac{\partial C}{\partial x} \tag{4.3.6}$$

The loss of oxidant molecules per unit length along the x-direction due to oxidation at the Si/SiO$_2$ interface is $-k_s C$. The continuity requirement states that

$$\nabla\cdot\text{Flux} = -k_s C \tag{4.3.7}$$

or, for constant values of D_{ox} and t_{ox}, we have

$$t_{ox}D_{ox}\frac{\partial^2 C}{\partial x^2} = k_s C \tag{4.3.8}$$

Notice that the analytical solution of Eqn. 4.3.8 yields Eqn. 4.3.5. Now, Using a one-dimensional finite-element discretization, as illustrated in Fig. 4.3.2, one can set up a difference equation

$$\frac{t_{ox}D_{ox}}{\Delta x^2}\left(C_{n-1} + C_{n+1} - 2C_n\right) = k_s\left[\frac{C_{n-1}}{6} + \frac{C_n}{3} + \frac{C_{n+1}}{6}\right] \tag{4.3.9}$$

or

$$\left[\frac{t_{ox}D_{ox}}{\Delta x^2} - \frac{k_s}{6}\right]C_{n+1} - 2\left[\frac{t_{ox}D_{ox}}{\Delta x^2} + \frac{k_s}{6}\right]C_n + \left[\frac{t_{ox}D_{ox}}{\Delta x^2} - \frac{k_s}{6}\right]C_{n-1} = 0 \tag{4.3.10}$$

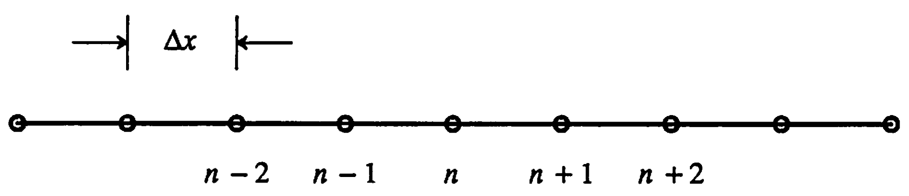


Fig. 4.3.2. One-dimensional approximation to the oxide region of the LOCOS structure shown in Fig. 4.3.1. The spatial discretization is assumed to be uniform to simplify the stability analysis.

The general solution to the above difference equation becomes oscillatory when

$$\frac{\frac{t_{ox}D_{ox}}{\Delta x^2} + \frac{k_s}{6} \pm \sqrt{(\frac{t_{ox}D_{ox}}{\Delta x^2} + \frac{k_s}{6})^2 - (\frac{t_{ox}D_{ox}}{\Delta x^2} - \frac{k_s}{6})^2}}{\frac{t_{ox}D_{ox}}{\Delta x^2} - \frac{k_s}{6}} < 0 \tag{4.3.11}$$

or when

$$\frac{(\frac{t_{ox}D_{ox}}{\Delta x^2} + \frac{k_s}{6}) \pm 2\sqrt{\frac{t_{ox}D_{ox}}{\Delta x^2}\frac{k_s}{6}}}{(\frac{t_{ox}D_{ox}}{\Delta x^2} - \frac{k_s}{6})} < 0 \tag{4.3.12}$$

The numerator in the above inequality is always non-negative and thus the above inequality can be reduced to

$$\frac{t_{ox}D_{ox}}{\Delta x^2} - \frac{k_s}{6} < 0 \tag{4.3.13}$$

or

$$\frac{k_s \Delta x^2}{t_{ox}D_{ox}} > 6 \tag{4.3.14}$$

We see that unless we are willing to decrease the element size, oscillation cannot be avoided when t_{ox} is small. The problem with decreasing element size is that excessively many grid-points can be generated thus increasing the total computational time. To avoid this problem, we propose a less accurate one-point integration scheme that samples at the grid-point. This scheme, when applied to Eqn. 4.3.1, results in a difference equation of the form

$$\frac{t_{ox}D_{ox}}{\Delta x^2}(C_{n-1} + C_{n+1} - 2C_n) = k_s C_n \tag{4.3.15}$$

The solution of which is unconditionally stable, reasonably smooth and always positive. This method, compared to the exact integration method, results in an only slightly less accurate solution for structures with small variation of the oxidant concentration. In relative magnitude, the solution in the thin oxide regions under a nitride mask will be noticeably different from the exact solution. Fortunately, since the concentration in such regions are very small. a large relative error in the solution will not cause a significant error in the calculated oxide profile around such regions.

Finally, we should also elaborate on the need to use linear-triangles in our analysis. As mentioned in section 4.2.6, the numerical techniques we used to solve oxide flow allow us to evaluate the hydrostatic-pressure, and thus also the oxidant diffusivity, only at the centroid of the triangles. This is not a problem if we use linear-triangles, since it will be sufficient to evaluate the bulk-diffusion term $\int_{\Omega} (\nabla N)^T D_{ox} (\nabla N)\ d\Omega$ using a 1-point integration scheme that samples at the centroid, even if D_{ox} varies linearly within each element. If quadratic-triangles are used to discretize the diffusion equation, it is impossible to evaluate the bulk-diffusion term accurately. If the oxidant-diffusivity is assumed to be constant within every-element, the accuracy of the solution obtained will not be better than that obtained using linear-triangles. Nonetheless, the major objection to using quadratic-triangles is that the overall numerical algorithm as described in section 4.4 becomes less stable, when the stress-models of Eqn. 3.8.1 through 3.8.3 are implemented.

4.4. IMPLEMENTATION OF THE STRESS-DEPENDENT OXIDATION MODELS

The stress-dependent oxidation models described by Eqn. 3.8.1 through 3.8.3 depend on 2 components of the stress: the hydrostatic pressure (p) and the normal stress (σ_{nn}) acting at the Si/SiO$_2$ interface. Hence, explicit expressions to calculate both p and σ_{nn} must be available before algorithms for the non-linearly coupled oxidation models can be implemented.

4.4.1. Calculation of the Hydrostatic Pressure

The incompressibility condition for oxide was modeled in section 4.2.3 by Eqn. 4.2.3.6 :

$$\boldsymbol{\sigma}'' = \alpha\, \dot{\boldsymbol{\varepsilon}}'' \quad , \tag{4.4.1.1}$$

where $\boldsymbol{\sigma}''$ contains the hydrostatic pressure. The hydrostatic pressure should then be consistently calculated from Eqn. 4.2.3.8 , and evaluated only at the centroid of the triangles :

$$p = -\alpha \frac{\dot{\varepsilon}_{xx} + \dot{\varepsilon}_{yy}}{3} \tag{4.4.1.2}$$

$$= -\frac{\alpha}{3}\left(\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} \right) = -\frac{\alpha}{3}\begin{bmatrix} \frac{\partial}{\partial x} & \frac{\partial}{\partial y} \end{bmatrix} \mathrm{NV} \tag{4.4.1.3}$$

4.4.2. Calculation of the Normal Interface Stress

The normal-stress σ_{nn} can usually be evaluated using

$$\sigma_{nn} = \sigma'_{nn} - p \quad , \tag{4.4.2.1}$$

where σ'_{nn} is the normal stress obtained from the pure-shear stress vector $\boldsymbol{\sigma}'$:

$$\sigma'_{nn} = [\, n_x^2 \quad n_y^2 \quad 0 \quad 2n_x n_y \,]\boldsymbol{\sigma}' \tag{4.4.2.2}$$

and $[n_x \, , n_y]^{\mathrm{T}}$ is the unit normal vector.

The problem with this approach, however, is that the hydrostatic pressure p can only be directly evaluated at the centroid of the triangular elements. To obtain σ_{nn} at the Si/SiO$_2$

interface, the value of p must be extrapolated to the interface, which is a difficult procedure if an irregular mesh is used. Fortunately, there is another way of calculating the interface traction. We first note that $\sigma_{nn} = \hat{\mathbf{n}}^T\mathbf{f}$, where $\mathbf{f}$ is the surface-traction, and $\hat{\mathbf{n}}$ is the outward unit normal vector at the boundary. Then, we note that the nodal load vector $\mathbf{F}_l$ of Eqn. 4.2.4.6a is assembled using Eqn. 4.2.4.8

$$\mathbf{F}_{li} = \int_\Gamma \mathbf{N}_i^T\mathbf{f}\, d\Gamma$$

When numerical integration is used,

$$\mathbf{F}_{li} = \sum_j \mathbf{b}_{ij}\mathbf{f}_j \quad , \tag{4.4.2.3}$$

where $\mathbf{f}_j$ = traction at node j , and the $\mathbf{b}_{ij}$ are the coefficients obtained from the numerical integration of Eqn. 4.2.4.8. If $\mathbf{V}$ is known, we may perform a reverse calculation for $\mathbf{f}_j$ from Eqn. 4.4.2.3 and Eqn. 4.2.4.8 . Unfortunately, there is a uniqueness problem here. There can be several different $\mathbf{f}$ that generate a particular $\mathbf{F}_l$ for a given mesh configuration. The calculation for $\mathbf{f}$ using Eqn. 4.4.2.3 automatically assumes that $\mathbf{f}$ is smooth over Γ . This assumption is made at present, although it may not be reasonable to assume that $\mathbf{f}$ is smooth around the sharp corners of the interface Γ.

Certainly, it is possible to calculate only for the stress at the Si/SiO$_2$ interface. This is done by first splitting $\mathbf{F}_l$ into 2 parts:

$$\mathbf{F}_l = \mathbf{F}_{l0} + \mathbf{F}_{ls} \quad , \tag{4.4.2.4}$$

where $\mathbf{F}_{ls}$ is the contribution of the traction at the Si/SiO$_2$ interface to $\mathbf{F}_l$, and $\mathbf{F}_{l0}$ is the contribution from the other boundaries.

Now instead of destroying the rows of $\mathbf{K}$ corresponding to the nodes on the Si/SiO$_2$ interface to specify the Dirichlet boundary condition of the system, we propose the following arrangement:

$$\begin{bmatrix} \mathbf{K} \ , & \mathbf{K}_s \\ \mathbf{I}_s \ , & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{V} \\ \mathbf{F}_s \end{bmatrix} = \begin{bmatrix} \mathbf{F}_{l0} \\ \mathbf{V}_{s0} \end{bmatrix} \tag{4.4.2.5}$$

In the above, $\mathbf{F}_s$ contains the traction variables of the nodes on the Si/SiO$_2$ interface. Let n be the total number of nodes in the system, and n_s be the number of nodes at the Si/SiO$_2$ interface. Then $\mathbf{K_s}$ is a matrix of size $2n \times 2n_s$. The matrix $\mathbf{K_s}$ is assembled from the coefficients b_{ij} of Eqn. 4.4.2.3 for the nodes on the Si/SiO$_2$ interface. In effect, we make the nodal traction at the Si/SiO$_2$ interface part of the unknown variables of the matrix equation. The specification of the Dirichlet boundary condition is given by

$$\mathbf{I_s} = \mathbf{V_{s0}} \tag{4.4.2.6}$$

where $\mathbf{V_{s0}}$ is the interface velocity as calculated from the oxidation rate at the Si/SiO$_2$ interface, and $\mathbf{I_s}$ is an identity matrix of size $2n_s \times 2n_s$ (scalar entries).

4.4.3. Assembly of the Matrix K_s

The solution to Eqn. 4.4.2.5 provides the nodal velocities $\mathbf{V}$ and the surface traction at the Si/SiO$_2$ interface simultaneously. But the solution for $\mathbf{F_s}$ strongly depends on how the matrix $\mathbf{K_s}$ is assembled. By making the nodal surface traction $\mathbf{F_s}$ an unknown variable of the matrix equation, we are essentially discretizing the surface traction $\mathbf{f}$ at the Si/SiO$_2$ interface. Since quadratic-elements are used in our analysis, we may be tempted to use a quadratic interpolation for the discretization of $\mathbf{f}$. But this choice is found to yield an oscillatory solution for $\mathbf{f}_i$. To avoid oscillation, a linear interpolation function must be used. For the quadratic triangular elements, this means that the mid-side-node is not used in the interpolation of the surface-traction. The need to use a linear-interpolation function can be intuitively explained by the fact that the stress is a first order derivative function of velocity. Hence, to be consistent with the quadratic distribution of velocity used, a linear stress distribution on every element should be assumed.

A special note needs to be made regarding a possible no-convergence problem in the above approach. If the quadratic-elements are used to discretize the oxidant diffusion equations, and the oxidation-rate at the mid-side-node of the triangle calculated using the oxidant concentration at that node, the solution for $\mathbf{F_s}$ depends strongly on the shape (aspect ratio) of the elements at the Si/SiO$_2$ interface. If the oxidation-rate or velocity at the mid-side-node is set to the mean value of their neighboring vertex-velocities, the convergence of the calculated stress improves drastically. The reason for this behavior is not known at present and may require further investigation. But it may be guessed that the use of reduced-integration scheme to evaluate the hydrostatic term $\mathbf{K}''$ of Eqn. 4.2.4.6d implies that the velocity distribution is actually quasi-quadratic. Hence, to get an accurate stress calculation, another level of constraint must be imposed on the interpolation function of some of the discretized variables, which in this case happens to be the velocity distribution at the Si/SiO_2 interface. The requirements for the convergence of the calculated normal interface-traction thus provide us another reason to use the linear-triangles in the discretization of the oxidant diffusion equation.

Certainly, the assembly of Eqn. 4.4.2.5 must be modified to reflect the linear-velocity variation used at the Si/SiO_2 interface. This is accomplished by modifying part of Eqn. 4.4.2.6. But we shall continue to use the form of Eqn. 4.4.2.5 in our future discussion, with the assumption that the appropriate modification to Eqn. 4.4.2.5 has been made.

4.4.4. Accuracy of the Calculated Normal Stress

Once the nodal interface traction $\mathbf{F}_s$ is obtained, the normal interface traction at node i (σ_{nni}) can be equated using

$$\sigma_{nni} = \hat{\mathbf{n}}^T \mathbf{f}_i \qquad (4.4.4.1)$$

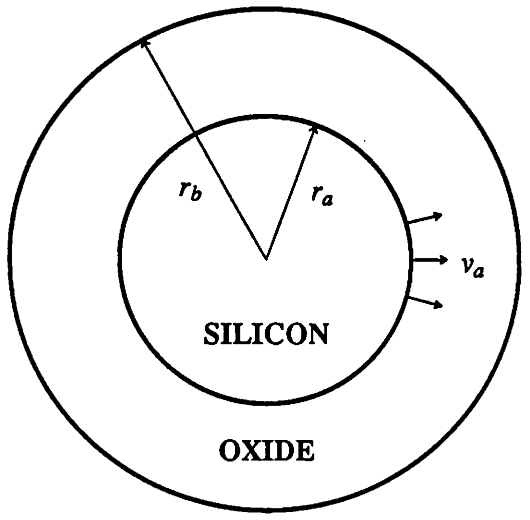
where $\hat{\mathbf{n}}$ is the outward unit normal vector at node i.

In general, it is not possible to verify the accuracy of the calculated interface stress for arbitrary structures. We may only check the convergence of the calculated stress with increasing mesh density. However, it is still possible to obtain some level of confidence by studying some simple structures which have known analytical solution for the interface stress. A cylindrical structure is convenient for such a study. Consider the test case shown in Fig. 4.4.4.1. Assume that the inner surface of the cylinder (at radius r_a) represents the Si/SiO_2, and the oxidation-rate is uniform along the interface. Let the oxide velocity at the Si/SiO_2 interface = v_a. Then, for a viscous-incompressible oxide model, the normal surface-traction at the Si/SiO_2

interface is given by

$$\sigma_{nn} = -2\eta v_a \left(\frac{1}{r_a^2} - \frac{1}{r_b^2} \right) \tag{4.4.4.2}$$

where r_b is the outer-radius (at the oxide/ambient interface). The numerical solution for σ_{nn} at the Si/SiO$_2$ interface can then be compared with the analytical solution given above. In general, the numerical solution for σ_{nn} is not exactly constant over the Si/SiO$_2$ interface. Hence, the maximum deviation of the numerically computed σ_{nn} from the analytical solution is used as an error indicator. Fig. 4.4.4.2 shows the dependence of this error on the mesh-density used (the mesh-density is represented by the square root of the number of nodes used). One can see that reasonably good accuracy (error of 8%) can be obtained with the coarser mesh shown. A finer mesh gives higher accuracy but requires significantly more computation time.



$$\sigma_{nn} = -2\eta v_a \left(\frac{1}{r_a^2} - \frac{1}{r_b^2} \right)$$

Fig. 4.4.4.1. Analytical solution to the normal stress acting on the Si/SiO$_2$ interface of a convex cylindrical silicon structure, assuming no orientation effect on the reaction rate.

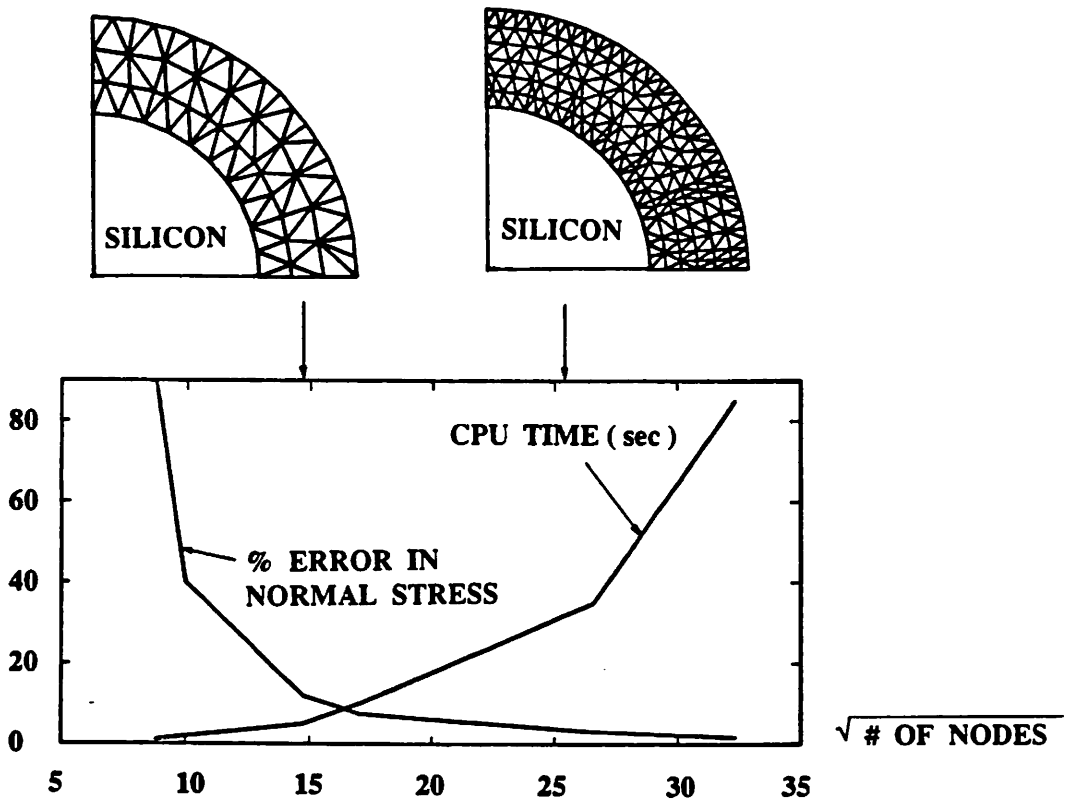


Fig. 4.4.4.2. The dependence of the accuracy of the calculated normal interface traction on the mesh-density used. Also shown is the cpu time needed to solve Eqn. 4.4.2.5 on a VAX 11/780 computer. The vertical scale is in percent error and seconds cpu-time. The calculation is performed on a quadrant of the structure by exploiting the cylindrical symmetry.

It should be noted that further tests on the accuracy and convergence of the program is needed. The cylindrical structure is too simple in the sense that it is essentially a one-dimensional problem. Further tests should use truly 2D structures which can be verified.

4.4.5. Algorithms for the Stress-Dependent Oxidation Models

With the algorithms for individually solving the oxidant diffusion and oxide deformation, it is now possible to implement the stress-dependent oxidation models of Eqn. 3.8.1 through 3.8.3. Since V_{s0} of Eqn. 4.4.2.5 is calculated from the oxidant concentration at the Si/SiO$_2$ interface, it is possible to write a matrix equation relating V_{s0} and C :

$$V_{s0} = K_{cv}C \qquad (4.4.5.1)$$

Combining Eqn. 4.4.2.5 and 4.4.5.1 together:

$$\begin{bmatrix} K\ , & K_s\ , & 0 \\ I_s\ , & 0\ , & -K_{cv} \\ 0\ , & 0\ , & K_c \end{bmatrix} \begin{bmatrix} V \\ F_s \\ C \end{bmatrix} = \begin{bmatrix} F_{10} \\ 0 \\ F_c \end{bmatrix} \qquad (4.4.5.2)$$

To introduce the stress-models, we first note that

$$K = K(\dot{\varepsilon}') = K(V) \qquad (4.4.5.2a)$$

$$K_{cv} = K_{cv}(F_s) \qquad (4.4.5.2b)$$

$$K_c = K_c(p) = K_c(V) \qquad (4.4.5.2c)$$

Thus, we obtain an equation of the form

$$\begin{bmatrix} K(V)\ , & K_s\ , & 0 \\ I_s\ , & 0\ , & -K_{cv}(F_s) \\ 0\ , & 0\ , & K_c(V) \end{bmatrix} \begin{bmatrix} V \\ F_s \\ C \end{bmatrix} - \begin{bmatrix} F_{10} \\ 0 \\ F_c \end{bmatrix} = 0 \qquad (4.4.5.3)$$

This system of non-linear equation can be solved using the Newton-Raphson iteration algorithm. Let us first call the solution to the above equation as

$$X = \begin{bmatrix} V \\ F_s \\ C \end{bmatrix} , \qquad (4.4.5.4)$$

Also, let us set the left terms of Eqn. 4.4.5.3 to E(X). Assuming that X_k is the solution of the system at the k-th Newton-iteration, then the solution obtained at the next iteration is given by

$$\mathbf{X}_{k+1} = \mathbf{X}_k - \mathbf{J}^{-1}(\mathbf{X}_k)\mathbf{E}(\mathbf{X}_k) \tag{4.4.5.5}$$

where **J** is the Jacobian of **E** with respect to **X**. The assembly process of **J** is tedious but straight-forward, and thus will not be presented here. In our solution process, **J** takes the form :

$$\mathbf{J} = \begin{bmatrix} \mathbf{K}^J\ , & \mathbf{K}_s\ , & 0 \\ \mathbf{I}_s\ , & \mathbf{K}_1^J\ , & -\mathbf{K}_{cv}^J \\ \mathbf{K}_2^J\ , & 0\ , & \mathbf{K}_c^J \end{bmatrix} \tag{4.4.5.6}$$

where

$\mathbf{K}^J$	=	Jacobian of **K** w.r.t. **V** ,
$\mathbf{K}_{cv}^J$	=	Jacobian of $\mathbf{K}_{cv}$ w.r.t. **C** ,
$\mathbf{K}_c^J$	=	Jacobian of $\mathbf{K}_c$ w.r.t. **C** ,
$\mathbf{K}_1^J$	=	Jacobian of $\mathbf{K}_{cv}$ w.r.t. $\mathbf{F}_s$,
$\mathbf{K}_2^J$	=	Jacobian of $\mathbf{K}_c$ w.r.t. **V**

A general purpose direct sparse matrix solver is used to solve the Newton iteration of Eqn. 4.4.5.5 since **J** is not symmetric and may contain diagonal zero. The iteration process is generally stable unless the volume parameter V_d of Eqn. 3.8.2 becomes too large. The possible instability is often due to the inadequate accuracy in the discretization of the continuum. A higher density mesh should be used for stronger non-linearity or stress-coupling.

It is important to keep the time-steps in the profile advancement small enough so that the normal interface stress σ_{nn} at the beginning and the end of the time-step (actually the beginning of the next time-step) does not differ by very much. This is to ensure that σ_{nn} does not change significantly during the time-step to make the oxidation-rate stay approximately constant during the time-step.

It is also appropriate to make a final remark about the size of the matrix $\mathbf{K}_c^J$. As noted in section 4.2.4, grouping the entries of **K** in Eqn. 4.2.4.6a into 2×2 sub-matrices provides us an approximately 3 fold advantage in computational speed in solving Eqn. 4.2.4.6a. For the same reason, we should organize the entries of **J** in Eqn. 4.4.5.6 into 2×2 sub-matrices. The presence of $\mathbf{K}_c^J$ in **J** may introduce some difficulties into the 2×2 organization process because there is no obvious or preferred way of grouping the entries of **C**, and the number of vertex-nodes (n_c) can be odd. In the CREEP program, the variables in **C** are grouped in the form $(C_{0,} C_1)$, $(C_{2,} C_3)$, ... , etc. In case n_c is odd, a dummy concentration variable and a dummy equation is added to the last row of Eqn. 4.4.5.3, so as to complete the 2×2-block organization of **J**.

4.5. OTHER MATERIAL PROBLEMS

The algorithms described in section 4.2 and 4.4 are developed for the growth of oxide on a rigid, immovable silicon substrate. To handle a more interesting class of oxidation problems that involve the presence of silicon nitride, we also need to discretize the nitride regions. Also, silicon and poly-silicon layers that are either floating or thin cannot be assumed to be rigid bodies. In section 3.12.2, we have indicated that both silicon and silicon nitride are elasto-plastic or non-linear visco-elastic materials. In the present version of the CREEP program, however, we have assumed them to be elastic materials that relax all the stress after every time step. In the following discussion, we will describe some algorithms that should properly handle the behavior of these materials.

4.5.1. Linear Visco-Elastic Formulation

The finite-element algorithms for handling silicon and silicon nitride can be made similar to that for oxide regions, if the constitutive relationship for the two materials can be written in forms similar to that in Eqn. 4.2.3.5 and 4.2.3.6. Assuming that silicon and silicon nitride are first-order linear visco-elastic materials, their stress-strain relationship are then given by:

$$\frac{\partial \boldsymbol{\sigma}'}{\partial t} + \frac{\boldsymbol{\sigma}'}{\tau} = G\,\mathbf{D}\,\dot{\boldsymbol{\varepsilon}}' \ , \ \tau = \frac{\eta}{G} \qquad (4.5.1.1)$$

and

$$\boldsymbol{\sigma}'' = 3K\,\boldsymbol{\varepsilon}'' \qquad (4.5.1.2)$$

where η, G and K are the viscosity, the elastic shear modulus and the elastic bulk modulus respectively. **D** is the same matrix as that given in Eqn. 4.2.3.5. Note that $\boldsymbol{\varepsilon}''$ is the total volumetric strain vector. Hence, if $\boldsymbol{\sigma}'' = \boldsymbol{\sigma}_0''$ at time = t_0, $\boldsymbol{\sigma}''$ at time = $t_0 + \Delta t$ can be written as

$$\boldsymbol{\sigma}'' = 3K\,\Delta\boldsymbol{\varepsilon}'' + \boldsymbol{\sigma}_0'' \qquad (4.5.1.3)$$

where $\Delta\boldsymbol{\varepsilon}''$ is the incremental volumetric strain vector during Δt. If Δt is small, $\Delta\boldsymbol{\varepsilon}'' \approx \Delta t\,\dot{\boldsymbol{\varepsilon}}''$ and thus Eqn. 4.5.1.3 can be further written as

$$\sigma'' = 3K\Delta t\, \dot{\varepsilon}'' + \sigma_0'' \qquad (4.5.1.4)$$

By letting $\alpha = 3K\Delta t$, Eqn. 4.5.1.4 becomes similar to Eqn. 4.2.3.6, except for the "memory" term σ_0''. A similar equation involving a memory term can also be derived for Eqn. 4.5.1.1. We first note that Eqn. 4.5.1.1 is a first-order ordinary differential equation. Thus we can solve it to yield

$$\left.\sigma'(t)e^{\frac{t}{\tau}}\right]_{t_0}^{t_0+\Delta t} = \int_{t_0}^{t_0+\Delta t} G\,\mathbf{D}\,\dot{\varepsilon}'(t)e^{\frac{t}{\tau}}\,dt \qquad (4.5.1.5)$$

Letting $\sigma'(t_0) = \sigma_0'$ and assuming that Δt is small enough so that $\dot{\varepsilon}'$ does not change significantly during the time Δt , we get

$$\sigma'(t_0+\Delta t) = G\mathbf{D}\tau\,\dot{\varepsilon}'\,(\,1 - e^{\frac{-\Delta t}{\tau}}\,) + \sigma_0'\,e^{\frac{-\Delta t}{\tau}} \qquad (4.5.1.6)$$

By writing $\sigma'(t_0+\Delta t)$ as σ' and simplifying the above expression, we get

$$\sigma' = \eta\mathbf{D}\,(\,1 - e^{\frac{-\Delta t}{\tau}}\,)\,\dot{\varepsilon}' + \sigma_0'\,e^{\frac{-\Delta t}{\tau}} \qquad (4.5.1.7)$$

We can now identify the term $\eta\mathbf{D}\,(\,1 - e^{\frac{-\Delta t}{\tau}}\,)$ as the effective viscosity of the material within the time step Δt . The term $\sigma_0'\,e^{\frac{-\Delta t}{\tau}}$ represents the residual stress that decays from the previous time step. By using Eqn. 4.5.1.4 and 4.5.1.7 in Eqn. 4.2.4.1 , and employing the orthogonality between the shear and hydrostatic deformation modes, we get the following virtual work equation :

$$\int_\Omega \delta\dot{\varepsilon}'^T\Delta\sigma'\,d\Omega + \int_\Omega \delta\dot{\varepsilon}''^T\Delta\sigma''\,d\Omega =$$

$$\int_\Gamma \delta\mathbf{v}^T\mathbf{f}\,d\Gamma - \int_\Omega \delta\dot{\varepsilon}'^T\sigma_0'\,e^{\frac{-\Delta t}{\tau}}\,d\Omega - \int_\Omega \delta\dot{\varepsilon}''^T\sigma_0''\,d\Omega \qquad (4.5.1.8)$$

where
$$\Delta\sigma' = \eta\mathbf{D}\,(\,1 - e^{\frac{-\Delta t}{\tau}}\,)\,\dot{\varepsilon}' \qquad (4.5.1.9)$$

and
$$\Delta\sigma'' = 3K\Delta t\,\dot{\varepsilon}'' \qquad (4.5.1.10)$$

The discretization process is then similar to that for silicon dioxide, except for the extra two terms on the right of Eqn. 4.5.1.8. However, special care needs to be taken when evaluating σ_0'

. In every time-step, the bulk of the material, and thus the elements used in the discretization deform in some way that is determined by the numerical computation. As mentioned earlier, the stresses should only be evaluated at certain "optimal" sampling points. These points (actually every point) also move with the elements. Furthermore, there can be non-zero rotation associated with every point. In some LOCOS simulation using the viscous model in this work, the rotation was found to be substantial (in the range of a few degrees per time-step at some point in the oxide). Thus, to avoid the accumulation of error in the stress, the calculated stress tensor at every point should also be "rotated" (using similarity transformation) before the stress is integrated in the next time-step.

4.5.2. Non-Linear Visco-Elastic Formulation

The preceding procedures are also applicable to non-linear visco-elastic materials, as long as the time step Δt is small enough so that the variation of $\tau = \eta / G$ over the time period Δt is small enough to justify the use of Eqn. 4.5.1.5. Unfortunately, severe difficulties may arise when τ changes rapidly with time. This may happen when the viscosity of the material depends strongly on the shear stress, such as those described by Eqn. 3.8.3 for the viscosity of oxide. The computational time required to solve the problem as a result of the small Δt used may become unacceptably high. In principle, this difficulty may be overcome by solving the differential equation of Eqn. 4.5.1.1 for stress-dependent τ . Unfortunately, for most cases of stress-dependent τ, there is no closed-form analytical solution to Eqn. 4.5.1.1.

The most common catastrophic behavior for elastic materials is the plastic deformation. We may, to the first order, model these materials as having very high viscosity when the shear-stress is below the critical plastic-flow-stress, and rapidly falling viscosity values beyond this stress level. Applying large enough strain and strain-rate, we would expect the stress in the material to be pinned around the plastic stress level, as illustrated in the following figure.

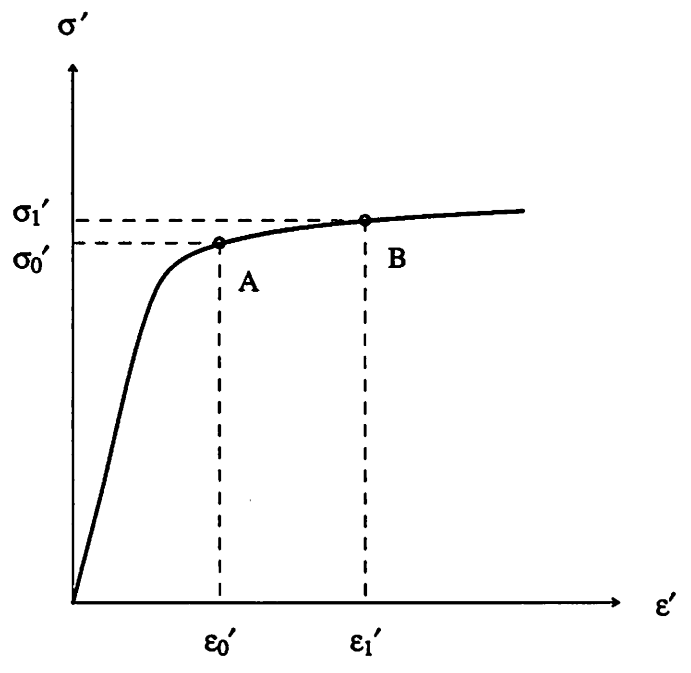


Fig. 4.5.2.1. Strain v.s. loading curve of a plastic material. In the plastic deformation regime, the stress changes slowly with increasing strain/strain-rate.

Assume that the material is in a state described by point A of Fig. 4.5.2.1 at time $t = t_0$, where the stress in the material is σ_0'. When the material is further strained to point B at time $t = t_0 + \Delta t$, we expect little change in the stress in the material. In the plastic deformation regime, the effective viscosity of the material is low, yielding a low value for τ. If the Δt used is large compared to τ, the calculated shear stress from Eqn. 4.5.1.7 can become very small, possibly much smaller than σ_0'. This is an unacceptable error. The major source of this error comes from the residual-stress $\sigma_0' e^{\frac{-\Delta t}{\tau}}$. During the time duration Δt, the viscosity of the material changes rapidly to a higher value in a certain fraction of Δt. Hence the residual-stress should not fall very much. A heuristic remedy for this situation is provided by calculating this residual stress from Eqn. 4.5.1.1, assuming that the strain-rate $\dot{\varepsilon}'$ is zero within the time duration Δt. The computation may still be difficult and require numerical integration. But at least it is easier than solving Eqn. 4.5.1.1, for general (unknown) value of $\dot{\varepsilon}'$. In the case the viscosity is described by Eqn. 3.8.3, and the shear modulus of elasticity G is a constant, we may even use an analytical solution for the residual stress, which is given by Eqn. 3.10.3.

The concept of "yield surface" is another and more common way of describing plastic behavior of materials[32]. At present, we prefer to model silicon and silicon nitride as non-linear visco-elastic material. Hence the stress/stain-rate formulation described in this section should be more appropriate for our modeling work.

4.6. BOUNDARY CONDITIONS

Most of the interfaces and boundaries that occur in silicon oxidation simulation require special treatment. The number of different material interfaces depends on the number of different materials we are willing to handle. The following sub-sections describe some of the interfaces and boundaries that need special attention, assuming that there are only silicon, silicon dioxide, silicon nitride and ambient regions in the simulation domain.

4.6.1. Reflection Boundary

This is an artificial boundary which arises when the simulation domain is limited to a finite size. It is actually a "partial" Dirichlet boundary, in the sense that one component of the nodal-velocity is known. For example, on a rectangular simulation window, the reflection boundaries are located at the sides of the window. The x-component of the velocity for nodes on the vertical boundaries and the y-component of the velocity for nodes on the horizontal boundaries should be set to zero.

4.6.2. Oxide/Silicon Interface

In the previous sections, silicon was assumed to be a rigid material that would not move or deform. This approximation is good only if the silicon region is the supporting substrate of large size. In cases where either the silicon is thin or floating, we need to take into account the deformation/motion of the silicon material. To handle the oxidation reaction that occurs at the interface, we need to split every node on the Si/SiO_2 interface into two nodes, one for the silicon region, and the other for the oxide region. The nodes are then separately numbered. The two regions are coupled by specifying that the tractions applied on the oxide and the silicon are equal in magnitude but opposite in sign. Further, the normal velocity of the existing oxide at the interface point is equal to the normal velocity of its corresponding silicon node plus the growth velocity that was described in section 4.1. The tangential velocities of the split-nodes are set to be equal. After solving the system of the discretized problem, the nodes in the discretized system are then moved by amounts determined by their velocities. The final Si/SiO_2

interface is then approximated by converting a layer of the silicon material to SiO_2, by an amount proportional to the oxide growth rate at the interface.

4.6.3. Oxide/Nitride Interface

CREEP currently assumes that there is no reaction occurring at this interface. In reality, oxidation does occur slowly at this interface. But there is almost no volume expansion involved in the conversion of Si_3N_4 to SiO_2. Hence the oxidation of Si_3N_4 can be handled without the use of node-splitting technique. The solution involves the simple conversion of an amount of Si_3N_4 to SiO_2 by an amount proportional to the oxidant concentration at the SiO_2/Si_3N_4 interface. However, the out-diffusion of the nitrogen or ammonia released from the reaction and its possible effect on the oxidation kinetics have not been adequately understood. Further investigation is needed.

4.6.4. Silicon/Ambient and Nitride/Ambient Interfaces

These interfaces can only exist at the first time step of the simulation. After that they will be covered with SiO_2. These interfaces are not yet handled by the CREEP program. However, they can be handled by simply converting a layer of the material to SiO_2, by an amount proportional to the oxidant concentration in the ambient. For silicon, volume expansion during the oxidation is also involved.

4.6.5. Silicon/Nitride/Ambient Triple Point

Little physical understanding of this interface-point is available. We propose the use of the following heuristic scheme, in which a small amount of the silicon material is converted to SiO_2 in a small time step :

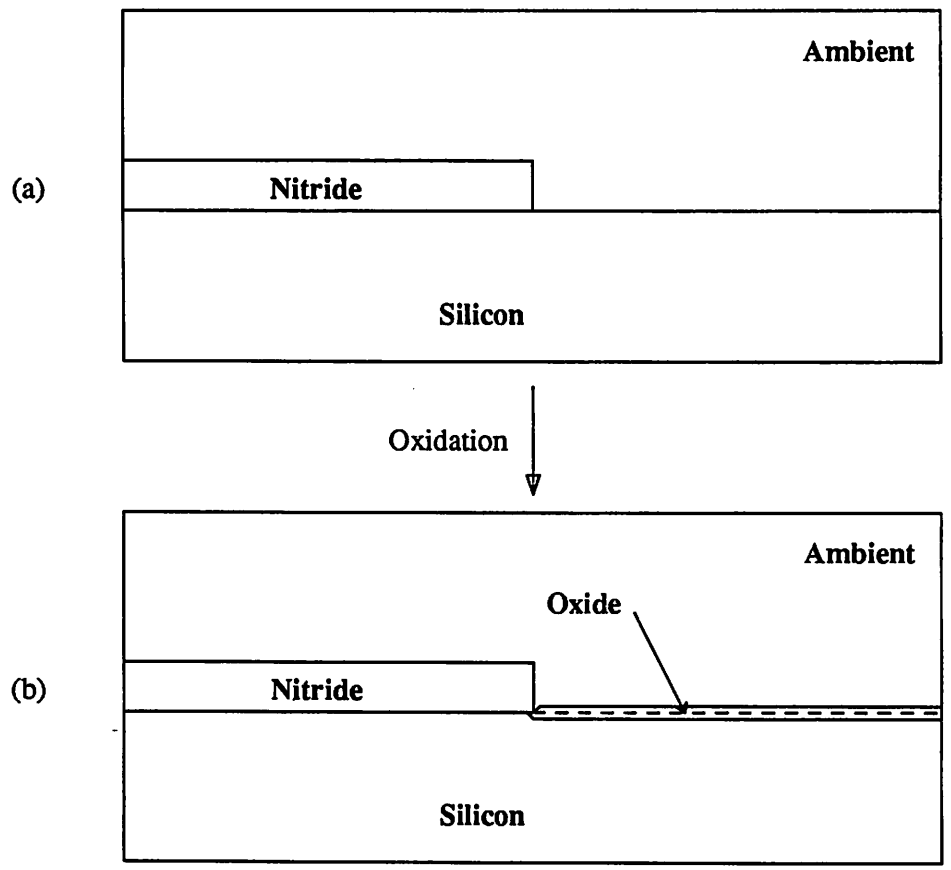


Fig. 4.6.5.1. Heuristic solution to the oxidation of a thin layer of silicon into silicon dioxide near a silicon/ambient/nitride triple point.

The time step should be kept small so that there will be no excessive conversion of the silicon under the nitride to SiO_2. Further oxidation of this region should be determined by the computation performed in the subsequent time-steps, so that the stress-effects in the oxidation process can be as closely accounted as possible.

4.6.6. Oxide/Nitride/Silicon Triple Point

There is also little understanding of this interface-point at present. Normally, the oxide/nitride interface is assumed to be a non-slip boundary, while the oxide and the silicon at the oxide/silicon interface are assumed to be moving apart from each other in a direction normal to the interface. Obviously the two assumptions are incompatible at this triple point. One or both of the assumptions must be violated as a result of the high stress generated around the triple point. We propose to overcome this dilemma by allocating only one node at the triple point in the discretization process, even though oxidation occurs at this point. This scheme effectively set the velocity of the existing oxide region at the triple point zero relative to the point. However, the etch-conversion of the silicon and the Si_3N_4 materials is performed in the same way described in section 4.6.2 and 4.6.3 , thus allowing the advancement (oxidation) of the triple point.

4.6.7. Oxide/Ambient Interface

Normally, nothing needs to be done for this interface. However, in the annealing of glasses with low viscosity, where the surface tension is a significant deformation driving force, the surface tension effect should be accounted for. The surface tension is simply a surface traction force that will be included in the the right hand term of Eqn. 4.2.4.1.

Surface tension appears as a normal stress that is proportional in magnitude to the curvature of the oxide surface. For low curvature surfaces, the surface tension can be easily estimated from the discretized (segmented) surface. But this method cannot meaningfully estimate the surface tension around a sharp corner, where the stress approaches a singular value. Fortunately, the finite element method uses the concept of nodal lumped forces in the discretized equation. We first note that surface tension is really a tangential force along the oxide surface, which results in a normal stress acting on the oxide surface when the surface is curved. Hence, instead of attempting to estimate the surface tension forces everywhere on the surface, we propose to calculate the lumped forces on the nodes directly by taking advantage of the segmented oxide surface. This is illustrated in the following figure :

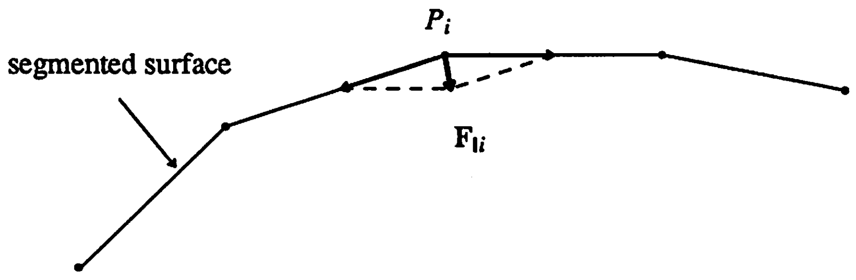


Fig. 4.6.7.1. Calculation of the equivalent nodal lumped-forces $\mathbf{F}_{\text{I}i}$ due to the surface tension at the point P_i directly from the segmented surface. The direction of $\mathbf{F}_{\text{I}i}$ bisects the vertex at the point P_i .

The above approach is applicable to smooth surfaces and sharp corners. The only problem is that the straight-line segments are curved after every time-step of the computation because of the use of the quadratic-elements. Our discretization algorithms require the use of straight segments in every time-step and thus reconditioning of the surface to straight line segments is needed. In the CREEP program, this is done by moving the mid-side-nodes of the elements back to their center position after updating the discretized profiles. The problem with this approach is that truncation error may accumulate. But this is not a difficulty peculiar to surface tension problems. If higher accuracy is needed, we propose to re-interpolate the surface to straight-line segments using some kind of least-square algorithm.

CHAPTER 5.

DATA STRUCTURE

5.1. INTRODUCTION

As stated in chapter 1 and 2, one of the goals of this project is to develop a framework for a general purpose 2D process simulator. To achieve this goal, we need to define and implement a data structure for the description of general 2D geometric structures. Access and manipulation of the geometric information should be simple and flexible. This data structure will also be crucial for the successful implementation of an automatic triangular mesh generator. This chapter is devoted to the description of the data structure currently used in CREEP.

5.2. NODE-SEGMENT REPRESENTATION

By definition, a general 2D geometric data structure must be able to represent arbitrarily complex planar graphs. For the purpose of process simulation, it is often sufficient to represent only straight-line segmented planar graphs. It is clear that all straight-line segmented planar graphs must and can be represented by nodes and segments. Hence, it is natural to also base our data structure on node and segment representation.

While it is possible that all segmented planar graphs can be represented by nodes and segments, it is obvious that infinitely long and open regions cannot be represented by planar graphs. This case is not important in process simulation and thus will not be considered. We will restrict ourselves to the handling of geometric structures with only closed regions. We further restrict that all material regions represented by the data structure must be enclosed in a rectangular "simulation window".

5.3. DATA FILE FORMAT

The first step in defining such a data structure is to define the format for the data file that stores the geometric information. The access of data files by the CREEP program is usually infrequent. Thus the format for the data files may be relatively simple. Our choice of the data file format is shown in the following :

nodes
0 x_0 y_0
1 x_1 y_1
2 x_2 y_2
3 x_3 y_3
4 x_4 y_4
5 x_5 y_5
.
.
.
N x_N y_N
segments
0 $n_{0,1}$ $n_{0,2}$ $m_{0,1}$ $m_{0,2}$
1 $n_{1,1}$ $n_{1,2}$ $m_{1,1}$ $m_{1,2}$
2 $n_{2,1}$ $n_{2,2}$ $m_{2,1}$ $m_{2,2}$
3 $n_{3,1}$ $n_{3,2}$ $m_{3,1}$ $m_{3,2}$
.
.
.
S $n_{S,1}$ $n_{S,2}$ $m_{S,1}$ $m_{S,2}$

Fig. 5.3.1. File format for geometric information.

The data file contains the information of all the nodes and segments that exist in the geometric structure. In the above, the entry "nodes" signifies the start of the node-list. Each node-entry contains 3 components : the node-number and the x- and y-coordinates of the node. Node-numbers must always start from zero. In addition, the first four nodes must be used to specify the four corners of the simulation window. Node 0 , 1 , 2 and 3 are respectively located at the lower-left , lower-right , upper-right and upper-left corners of the window :

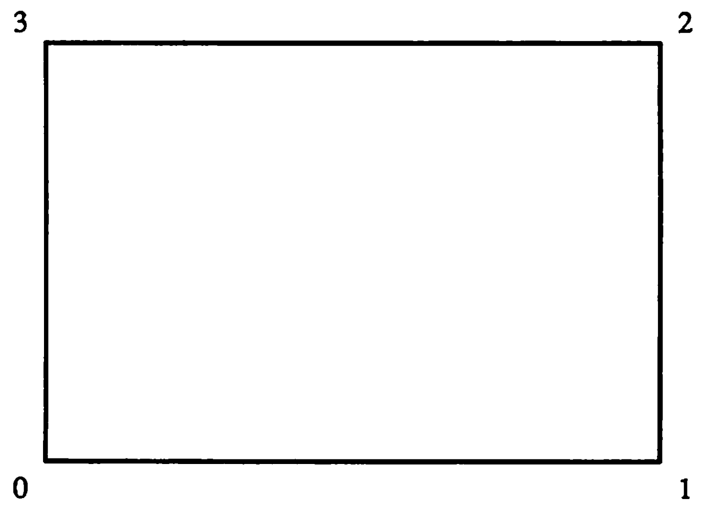


Fig. 5.3.2. The node-numbers for the corners of the simulation window.

The other nodes in the structure should be located within or on the boundaries of the simulation window. They may be numbered in any sequence, as long as they are all positively and distinctly numbered. The node-numbers are specified only for the references made by the the segment-entries. There is, however, a limit on the memory allocation within the program which limits the use of large node-numbers.

The segment-list starts immediately after the word "segments" in Fig. 5.3.1. Each segment-entry contains a segment-number, the node-numbers of the two end-points of the segment, and the identification numbers for the material types on both sides of the segment. For example, in the illustration of Fig. 5.3.1, the two end-nodes of segment number 3 are represented by $n_{3,1}$ and $n_{3,2}$. The material types on the two sides of the segment are represented by the numbers $m_{3,1}$ and $m_{3,2}$. The convention is such that the region spanned counter-clockwise from $n_{3,1}$ to $n_{3,2}$ is of type specified by $m_{3,1}$. This convention is illustrated in the following figure :

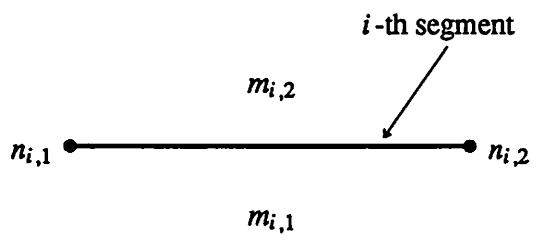


Fig. 5.3.3. The convention for the location of the material regions as specified by the structure file.

All segment-numbers must be distinct and non-negative. They may, however, be ordered in any sequence. The only limitation is imposed by the fixed size of memory space initially allocated within the CREEP program to store the largest numbered segment.

All material types are identified by their identification numbers, which must be positive. The data structure used in CREEP is currently configured to handle up to 127 different material types. The first five numbers from 1 through 5 have been tentatively assigned to label ambient regions, rigid substrate regions, silicon regions, oxide regions and nitride regions respectively. The oxidation module in CREEP assumes that the rigid substrate region is silicon, and the ambient region contains gaseous oxidizing species. However, we intend to leave the rigid substrate and the ambient regions open to interpretation, so that other simulation modules added into the CREEP program in the future will not be unnecessarily forced to deal with silicon region.

Regions outside the simulation window must also be labeled. They are labeled as number 0. Again, the meaning of the window boundaries is open to interpretation. The oxidation module in CREEP assumes that the window boundaries are reflection boundaries and hence the material outside a window segment is the same as the material inside the window at that segment.

5.4. TRI-DIRECTIONAL LINKED-LIST STRUCTURE

The data file format adequately describes most of the geometric structures that are of interest to us, but is certainly unsuitable for fast access and recognition within the CREEP program. Once the node-segment information of a geometric structure is read into the CREEP program, the program must reorganize the data using some internal data structure so that the geometric information can be easily accessed by other modules in the program, in such a way that every region within the geometric structure can be traversed easily, and that neighboring regions from any particular region can be found easily. The data structure proposed by Lee[33] provides some flexibility in representing a large class of geometric structure. However, only a fixed number of segments can be connected to a node. Further, when two or more adjacent regions have the same material type, it becomes difficult to traverse any of the regions. This is not much of a limitation for Lee's application, but a severe one for us since we always create connected regions with the same material type during mesh-generation. After much consideration, we decided to use a tri-directional linked-list structure to represent the nodes and segments within the program. This structure is designed to eliminate the two limitation of Lee's data structure. We shall call every tri-directional linked-list structure a "terminal". Every segment will be constructed of two terminals. In the C programming language, the terminal is defined as a C-structure :

```
typedef  struct  terminal_strct  {
       short                        node_no ;
       short                        l_node  ;
       struct   terminal_strct     *oppterm ;
       struct   terminal_strct     *fwdterm ;
       struct   terminal_strct     *bckterm ;
       double                       v_angle ;
       char                         cc_in, cc_out ;
       char                         marker1 , marker2 ;
       }  TERMINAL ;
```

In the CREEP program, the entries of the terminal structure are actually not arranged in the order shown. The order shown is not efficient in terms of the memory size needed to define the terminal structure. Each entry in the terminal structure will however be explained in the order shown.

The first entry *node_no* is a short integer variable used for storing the node-number of the terminal. The actual x- and y-coordinates of the node is stored in a separate node-table which will be described later.

The *l_node* entry is used to provide an alternative index number for a terminal, independent of the node-number of the terminal. It is mainly used in the re-numbering of the nodes and terminals.

The pointers *fwdterm*, *bckterm* and *oppterm* are described in the following pictorial representation of the terminal structure :

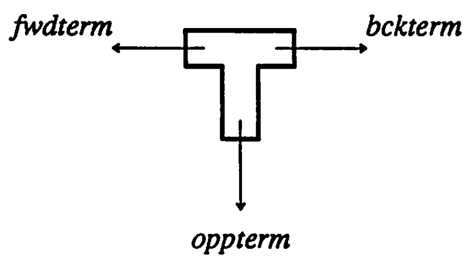
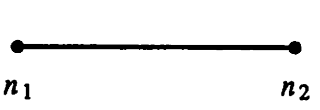


Fig. 5.4.1. A pictorial representation of a terminal structure.

The purpose of the 3 pointers is best explained by describing their usage in representing a segmented graph. We first consider the representation of a segment by 2 terminals :



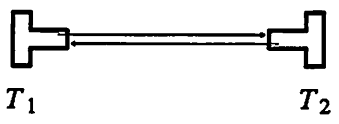
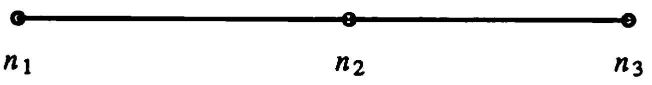


Fig. 5.4.2. (a) A line segment with end-nodes n_1 and n_2, and (b) the representation of the line segment by two terminals. Node n_1 is represented by terminal T_1 and n_2 by T_2.

In the above, the *oppterm* pointer on each terminal is used to point at the other terminal on the segment. Suppose that two segments are connected, as shown in Fig. 5.4.3a. Then the two segments are represented by 4 terminals :



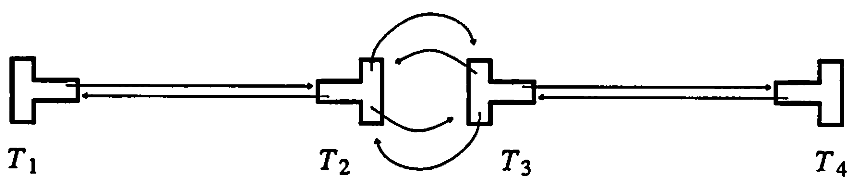


Fig. 5.4.3. (a) Two connected line segments and (b) their representation by 4 terminals. Node n_1 is represented by terminal T_1, n_2 by T_2 and T_3, and n_3 by T_4 .

That is, the *fwdterm* and *bckterm* pointers are used to link connected segments together. For 3 segments connected to a node, we have the following representation :

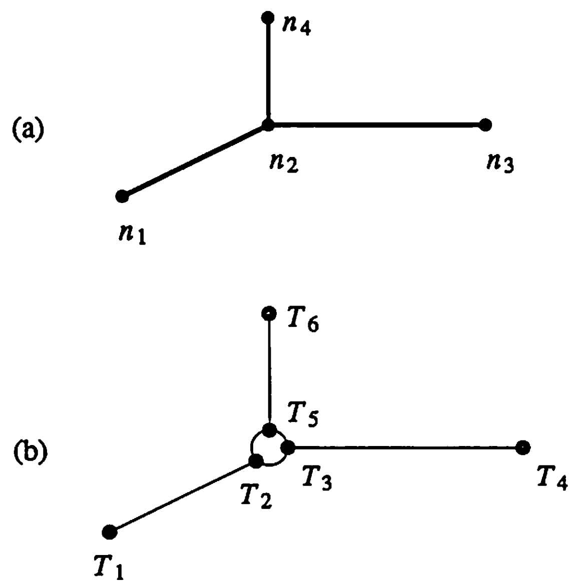


Fig. 5.4.4. (a) Three line segments connected to a node and (b) their representation by 6 terminals. This figure is simplified by using fine lines and filled circles to represent bi-directional links and terminals respectively.

In the above, all the arrows that are shown in Fig. 5.4.2 and 5.4.3 are omitted, every bi-directional link is drawn as a curved or straight line, and the terminal structures are simply represented by filled circles. This simplified representation will be used again when detailed explanation is not needed.

The representation of 4 or more segments connected to a single node is straight forward. There is no limit on the number of segments that can be connected to a single node. Note that the node-number (*node_no*) on every terminal connected to a single node should all be identical.

Since we have decided that we will only handle geometric structures that are closed, we are guaranteed that there will be no dangling *fwdterm* and *bckterm* pointers on any terminal in the whole data structure. As an illustration, Fig. 5.4.5a shows a simple but complete geometric structure bounded by a rectangular simulation window. The representation of the complete structure in terms of terminals are shown in Fig. 5.4.5b . We shall call the completed graph formed by the linked-list structures the map of the geometric structure.

(a)

(b)

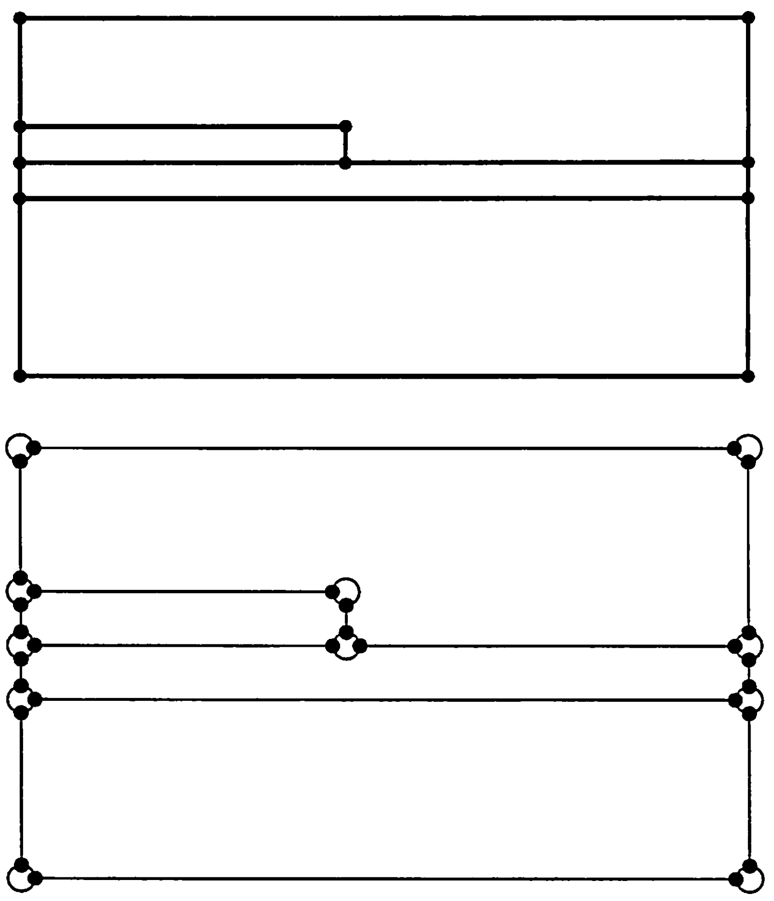


Fig. 5.4.5. (a) A simple but complete geometric structure and (b) its representation by terminals.

The tri-directional linked-list structure allows one to traverse any region in the geometric structure in any direction (this is the subject of the next section). In every polygonal region, we define the convention for traversal as forward motion for counter-clockwise and backward motion for clockwise traversal of the region. Note that the *fwdterm* and *bckterm* pointers on every terminal are defined according to this convention.

Each node in a geometric structure represents the vertices of several polygonal regions. Since there are as many terminals on a node as there are polygonal regions connected to the node, we assign each terminal on a node to represent the polygonal region immediately to the left of the terminal. The left hand side of a terminal is defined as the region on the side of the *bckterm* pointer of the terminal, as illustrated in Fig. 5.4.6 :

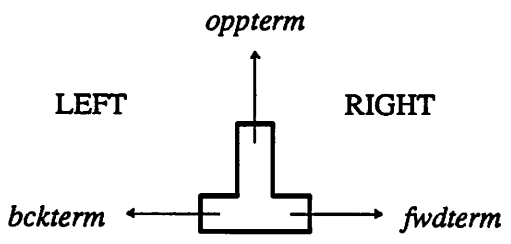


Fig. 5.4.6. The definition of the left and right hand sides of a terminal.

The double precision floating point variable *v_angle* of a terminal is used to store the internal vertex-angle of the polygon at the vertex represented by the terminal.

The *cc_in* and *cc_out* entries in the terminal structure are used to specify the material types on the left and right hand sides of the terminal respectively.

The *marker1* entry is used as a bit field for general purpose flags. Because different machine architectures may perform sign extension from a character variable to an integer variable differently, the most significant bit of *marker1* is not used to prevent any possible uncertainty due to sign-extension. A character is normally 1 byte and thus we have 7 bits for general purpose flags. If more flags are needed for every terminal, we may convert the *marker1* variable to a long integer type. On a 32-bit machine, this will give us 32 bits of flags for every terminal.

The *marker2* entry is a reserved variable that is not yet in used. No specific plan has been made as to how it will be used in the future. It is defined in the terminal structure because it doesn't cost the terminal structure extra memory space when the entries of the structure are properly ordered.

5.5. BASIC TRAVERSAL OPERATIONS

As mentioned in section 5.4 , the coordinates of the nodes are stored in a separate node-table. Each node-entry in this node-table also contains a pointer to one of the terminals that form the node. Given a node-number, the pointer in the node-table provides a quick way for locating the node in the map of the geometric structure. Once the structure map of the geometric structure is assembled, we can perform many searching operations on the map. The basic operation in most searching algorithms is the traversal of regions. We will thus attempt to explain the operations involve in the traversal algorithms.

First, let us call the polygonal regions defined by the nodes and segments the basic polygons of the map. To traverse the boundary of a basic polygon is simple. Let *p* be a pointer that points to a terminal that represents a vertex of a polygonal region. Then, in the C language notation, the operation

p = p->oppterm->fwdterm

moves the pointer *p* a step forward (counter-clockwise) to the next vertex in the basic polygon. Similarly, the operation

p = p->bckterm->oppterm

moves the pointer *p* a step backward (clockwise) to the previous vertex in the basic polygon. The operation is valid regardless of the number of polygons connected to the node pointed to by *p* .

The ability to traverse any basic polygon in a simple and unambiguous way is important but not sufficient for many purposes. Sometimes we may want to define a composite polygon that encompasses several basic polygons. For example, several connected basic polygons may be of the same material types and we would like to traverse the outer-boundary of all the regions with the same material type (this always happen when a material region is cut into many triangles by a finite-element mesh generator). In such a case, we need to first mark the boundary of the composite polygon. A bit-field in the *marker1* entry of the terminals at the vertices of the composite polygon may then be set to 1 to mark the boundary of the composite polygon. The traversal of the composite polygon will then involve the checking of the bit-field that is used to mark the composite polygon. But the operation is still simple and unambiguous.

For example, moving from one vertex of a composite polygon to the next (in the counter-clockwise direction) involves a jump through the *oppterm* pointer, then a jump through the *fwdterm* pointer with a check of the bit-field in question, and possibly several other jumps through the *fwdterm* pointers until a terminal is found for which the bit-field in question is set.

5.6. FUTURE ENHANCEMENTS

The data structure described thus far is able to represent a large class of geometric structures. However, the handling of other quantities such as doping profiles is also important in general process simulation. We have not adequately addressed the issue of handling distribution of quantities inside material or polygonal regions in a general way. A related issue is the difficulty in interpolating a quantity distribution from an old mesh to a new mesh, given that the mesh used is often irregular. Future development in the data structure should include the provision to simultaneously specify several quantity distributions in any polygonal regions, and the implementation of interpolation algorithms to map data from one mesh to another with a minimal loss of accuracy.

CHAPTER 6.

MESH GENERATION

6.1. INTRODUCTION

The mesh is the assembly of elements used to discretize a geometric structure in a finite-element analysis. The quality of the mesh often strongly affects the results of the finite-element analysis. Hence, mesh generation is an important process in finite-element analyses.

It is difficult to quantify the quality of a mesh. However, meshes that are qualitatively smooth and regular usually provide good results in finite-element analyses. Meshes formed of triangular elements that are nearly equilateral usually exhibit the smoothness and regular quality. Hence, whenever possible, we should attempt to make elements that are closed to equilateral triangles.

Mesh generation is a tedious process if done manually. Commercially available finite-element analysis packages often provide semi-automated mesh-generators that allow the user to generate meshes interactively using graphic terminals. For our purpose, that approach may still be inadequate. For problems with continuously expanding and deforming material regions, it is often necessary to generate a new mesh at a new time-step of the analysis. For problems that may take large amount of computing time, it is not convenient for the user to interact with the program continually until the end of the program execution. Hence, a fully automated mesh-generator is desirable.

User aided mesh-generation can be considered an art that improves with the users' experience. Thus, mesh-generation algorithms are also heuristic in nature. Mesh-generators that can discretize arbitrary 2D geometric structures can be difficult to implement if a data structure and algorithms for manipulating arbitrarily shaped geometric structures are not available. As mentioned in chapter 2, we developed three versions of the mesh-generator because the first two were unsatisfactory. It is worth making a remark here that one of the common difficulties involved in the development of the first two mesh-generators was in the tracking of connected polygonal regions. When triangulating a polygonal region, we would invariably add new nodes on the sides of the polygon. The earlier data structure used did not provide a way of automatically updating the boundaries of the polygonal regions connected to the polygon being triangulated. The difficulties involved in searching for the other affected polygons prevented us from successfully implementing a mesh generator that can handle geometric structures with multiple polygonal regions. In addition to providing simple an unambiguous traversal algorithms, the current geometric data structure used in CREEP treats a geometric structure as one unit, thus eliminating the problems of updating nodes on region boundaries.

There have been several mesh-generation algorithms published in the literature. The algorithms we used are derived from the work of Bykat[34], with the following 2-step strategy :

1. Convert all complex polygons to convex polygons.
2. Triangulate all convex polygons.

Complex polygons are polygons with some of the internal vertex-angles larger than 180° (they are also often referred to as concave polygons). Convex polygons are polygons with all their internal vertex-angles less than or equal to 180°. Complex polygons are first truncated into convex polygons. Triangulation of the remaining convex polygons is then simplified because any cut-line drawn across a convex polygon will only intersect the polygon at two points (unless the cut-line is tangential to the polygon, which can be easily avoided). We shall describe the algorithms we used in some detail in the next two sections.

6.2. CONVERSION OF COMPLEX TO CONVEX POLYGONS

Before attempting mesh-generation , we need to decide on the density of the mesh. We define an average length parameter l_{ave} so that the elements created during mesh-generation will tend to have side-lengths approximately equal to l_{ave} . This parameter will only be used as a guideline to divide a larger segment into several smaller sized segments. In general, we do not attempt to remove or move existing nodes in the geometric structure. Existing segments that are much shorter than l_{ave} will thus be left unchanged.

The complete division of long segments into segments with length approximately equal to l_{ave} is not performed during the reduction of a complex polygon into convex polygons. In the process of reducing complex polygons to convex polygons, long segments are cut into shorter segments only when needed to truncate a larger polygon into smaller polygons. The first step in the procedures of the complex to convex polygon-truncation process is to first search for a vertex with internal vertex-angle greater than 180°. Following a terminology used by Bykat, we call such a vertex a reflexive vertex. We draw an internal bisector line at the reflexive vertex, as illustrated in the following figure :

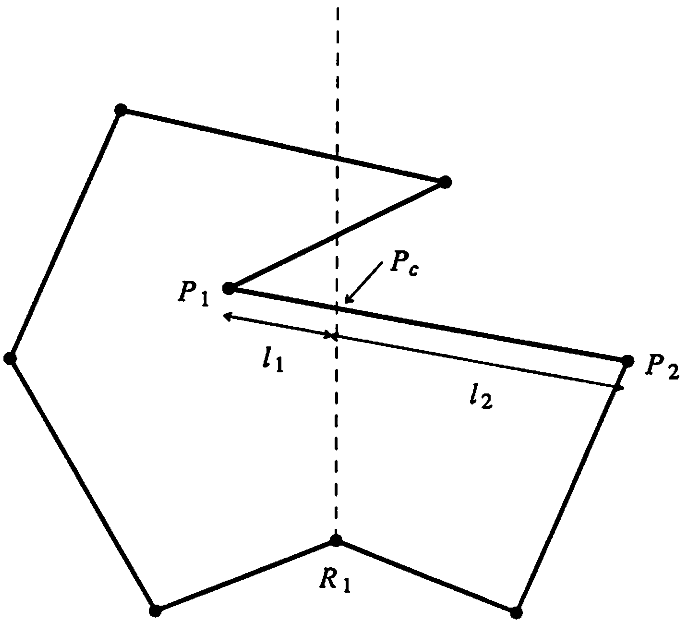


Fig. 6.2.1. Searching for a cut-point starting from a reflexive vertex R_1. The dash-line is the guiding cut-line, initially chosen as the angle-bisector at R_1. P_c is the closest point of intersection from R_1.

The bisector line is only used as a guiding cut-line. The intersection points of this guiding cut-line with the polygon are calculated. The intersection point closest to the reflexive vertex is noted and designated as P_c. The distances between this intersection point to the two vertices immediately before and after this intersection points are computed, and designated as l_1 and l_2

in Fig. 6.2.1. These two distances are compared.

If both l_1 and l_2 are greater than $0.7 l_{ave}$, we create a new node (vertex) at the intersection point P_c . A segment is then added across the reflexive vertex and the new vertex to cut the complex polygon into two polygons.

If the intersection point P_c happens to coincide with an existing vertex, a segment will be added across that vertex and the reflexive vertex. In practice, we define that such a coincidence occurs when the computed distance between the intersection point to the nearest vertex along the polygon boundary is shorter than a very small number (for example, a number smaller than 1e-10). This is to avoid problems that may arise due to the finite-precision of the computer architecture.

If the above conditions are not true but l_1 is smaller than l_2, we change the guiding cut-line to a line drawn from the reflexive vertex through the vertex P_1. All of the above procedures of finding the shortest intersection point is performed again. This is to detect the possibility that the rotated guiding cut-line may intersect at another point, such as in the case illustrated in Fig. 6.2.2 :

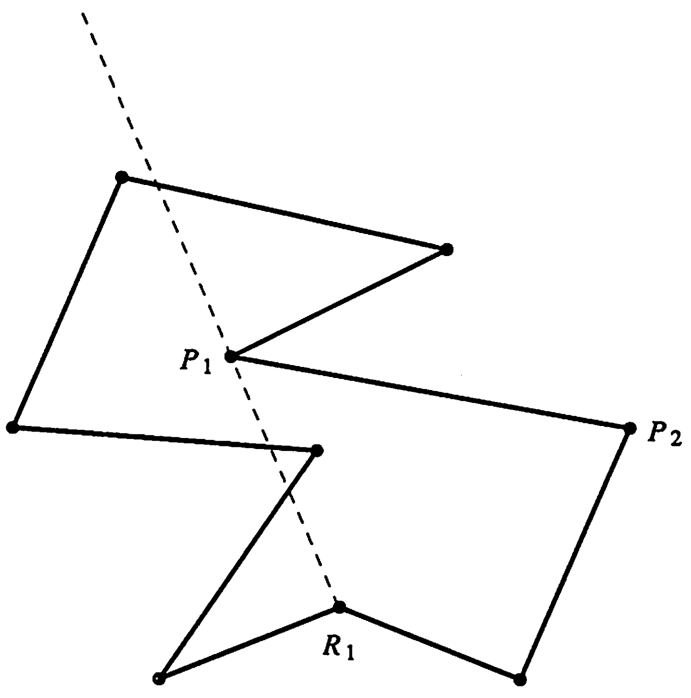


Fig. 6.2.2. A possible case in which the rotated guiding cut-line intersects at a closer point from the reflexive vertex R_1.

Note that if the rotated guiding cut-line does not intersect the polygon at a closer point, the coincidence test will pass at the next round of searching and testing.

If, however, l_2 is shorter than l_1, we change the guiding cut-line to a line drawn from the reflexive vertex through the vertex P_2. The remaining test procedures are then similar to that of the case of l_1 less than l_2.

There is a possibility that when we try to change the guiding cut-line, the cut-line may be tangential to one of the segments connected to the reflexive vertex, as illustrated in Fig. 6.2.3. In such a case, we are forced to create a new node (or vertex) mid-way between P_1 and P_2. A new guiding cut-line will then be drawn from the reflexive vertex through the new vertex. All the procedures listed above will then be repeated to detected the possibility of shorter intersection points.

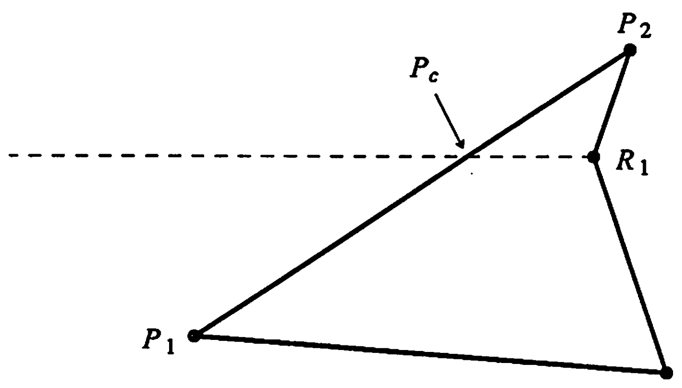


Fig. 6.2.3. An illustration of the possibility that when the guiding cut-line shown in dash is rotated clockwise to P_2, the new guiding cut-line will be tangential to a segment connected to the reflexive vertex R_1.

The resulting two polygons cut from the complex polygon will each go through the reduction procedures recursively until all of the polygons contain no reflexive vertex.

The procedures outlined thus far, if carefully implemented, should be fairly robust and capable of truncating most conceivable complex polygons into convex polygons. But there is a possibility that the following situation may occur :

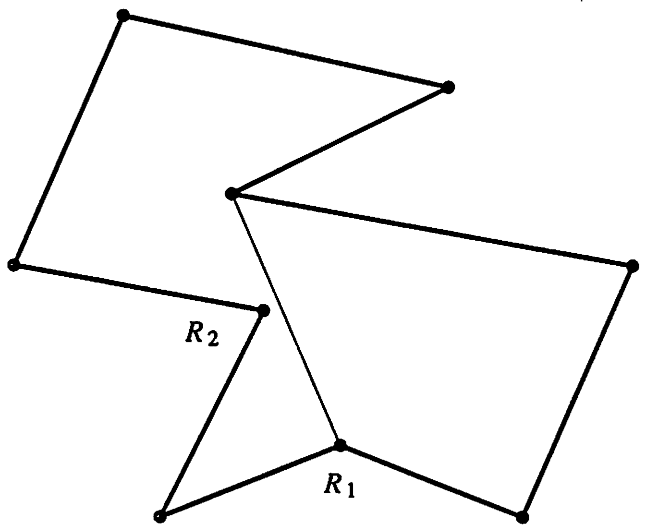


Fig. 6.2.4. A possible case in which a new cutting-segment is too close to another reflexive vertex R_2.

That is, a new segment added to truncate the polygon may be too close to another reflexive vertex (R_2 in Fig. 6.2.4). In Bykat's work, a proximity test is performed to check for such a condition and invalidate the cut-line. We choose not to do such a test, even though the mesh generated around the second reflexive vertex during the subsequent triangulation procedure may be ill shaped when compared to an equilateral triangle. This situation rarely occurs in process simulation problems. If it does, the effect is still dependent on the robustness of the finite-element algorithms. The effect of poorly formed mesh on the result of our oxidation simulation has not been fully characterized, although simulation results from CREEP appear to suggest that the finite-element algorithms used to discretize the oxidation equations are tolerant to poorly formed mesh. However, in case it is necessary to use a "good" mesh, the user can still manually place segments across appropriate nodes in the data-file of the geometric structure to remove some of the reflexive vertices.

6.3. TRIANGULATION OF CONVEX POLYGONS

After the reduction of complex polygons into convex polygons, all the sides of the convex polygons with length significantly larger than l_{ave} are divided into segments each with length approximately equal to l_{ave} (in CREEP, the threshold for such a division is about $1.5l_{ave}$). Triangulation procedure is then perfomed on each convex polygon.

If a polygon is already a 3-node triangle, no further triangulation is needed. Otherwise, we search for the vertex with the sharpest internal vertex-angle. Referring to Fig. 6.3.1, if the smallest vertex-angle is less than 60°, we draw a virtual line (ie, not a real segment) between the two vertices P_1 and P_2 immediately before and after the sharpest vertex P_0 :

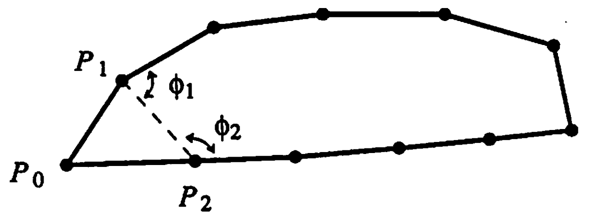


Fig. 6.3.1. Once the sharpest vertex P_0 is found, a line is drawn across its immediate neighbor vertices P_1 and P_2. The size of ϕ_1 and ϕ_2 determines if a segment will actually be placed across P_1 and P_2.

If either ϕ_1 or ϕ_2 of Fig. 6.3.1 is smaller than 45° , we consider the polygon a "small" polygon. The polygon is then sent through a procedure called *MG_TriangSmallPoly* for triangulation. This procedure will be described at the end of this section.

If both ϕ_1 and ϕ_2 are greater than 45°, a segment is added across P_1 and P_2 to truncate the polygon. The process for triangulation is then recursively applied to the remaining polygon.

If, however, the smallest intemal vertex-angle is greater than 135° , the polygon is cut into two by approximately bisecting the intemal angle of the sharpest vertex, as illustrated in Fig. 6.3.2 :

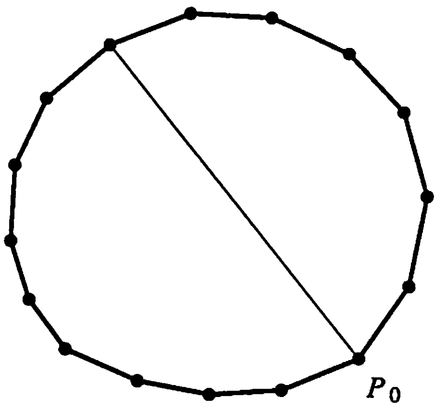


Fig. 6.3.2. If the internal angle at the sharpest vertex P_0 is greater than 135°, the polygon is cut into two, using a segment (shown as a fine-line) drawn from P_0 .

The truncation is done by using a procedure similar to but simpler than the procedure used for reducing a complex polygon into convex polygons.

If the sharpest vertex has an internal vertex-angle in the range of 60° to 135°, we search for a cut-line near the sharpest vertex. Consider the polygon shown in Fig. 6.3.3 :

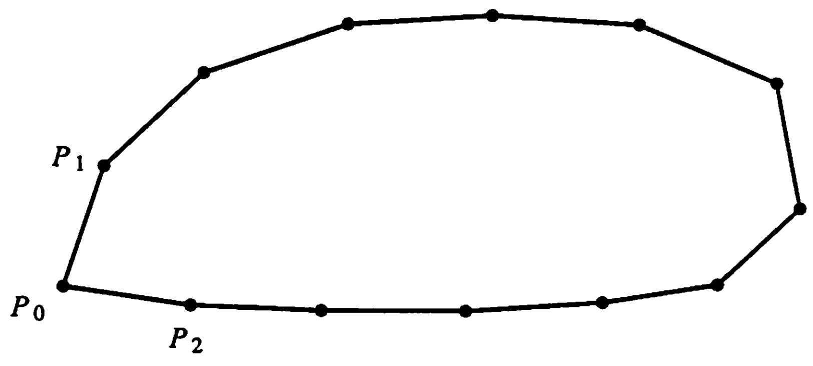


Fig. 6.3.3. A polygon with the internal angle of the sharpest vertex in the range of 60° and 135° .

Let P_0 be the sharpest vertex and P_1 and P_2 be the 2 neighboring vertices. First, we draw a virtual line between P_1 and another vertex P_f, such that the line $\overline{P_1P_f}$ is closest to being parallel to the segment $\overline{P_0P_2}$:

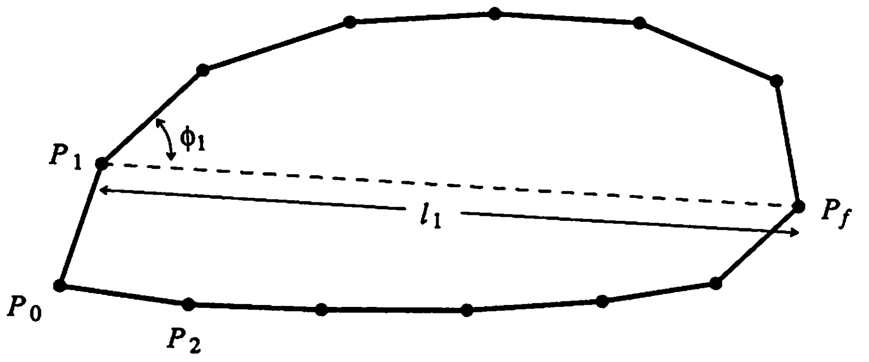


Fig. 6.3.4. Searching for a possible cutting point P_f from P_1 so that $\overline{P_1P_f}$ is nearly parallel to $\overline{P_0P_2}$.

The angle ϕ_1 and the length l_1 are calculated. Similarly, we draw a virtual line between P_2 and another vertex P_b, such that the line $\overline{P_2P_b}$ is closest to being parallel to the segment $\overline{P_0P_1}$:

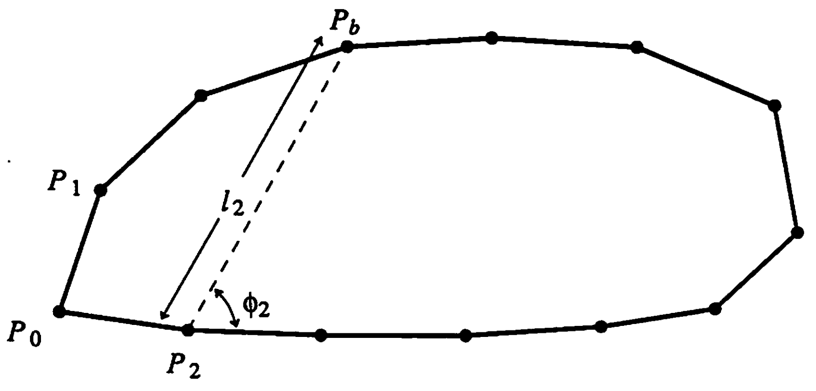


Fig. 6.3.5. Searching for a possible cutting point P_b from P_2 so that $\overline{P_2P_b}$ is nearly parallel to $\overline{P_0P_1}$.

The angle ϕ_2 and the length l_2 are also calculated.

If both ϕ_1 and ϕ_2 are smaller than 7.5°, we will consider the polygon a "small-polygon" and triangulate it with the procedure *MG_TriangSmallPoly*.

If $\phi_1 > 7.5^\circ$ and $\phi_2 < 7.5^\circ$, a segment is added across P_1 and P_f. If $\phi_2 > 7.5^\circ$ and $\phi_1 < 7.5^\circ$, the segment is added across P_2 and P_b instead. If $\phi_1 > 7.5^\circ$ and $\phi_2 > 7.5^\circ$, the segment is added across P_1 and P_f if $l_1 < l_2$, or the segment is added across P_2 and P_b if $l_1 > l_2$. For the example shown in Fig. 6.3.3 through 6.3.5, the cut line will be drawn across P_2 and P_b.

If the new (cutting) segment is appreciably longer than l_{ave}, it is divided into several segments, each with a length approximately equal to l_{ave}. The resulting polygon that contains the vertex P_0 is then sent through the procedure *MG_TriangSmallPoly* for triangulation. The whole process for triangulation will be applied recursively to the remaining polygon.

In general, a "small" polygon is either a polygon with only a few vertices or a long but thin polygon. They are triangulated by the procedure *MG_TriangSmallPoly* , which is described in the following. Consider the "small" polygon shown in Fig. 6.3.6 :

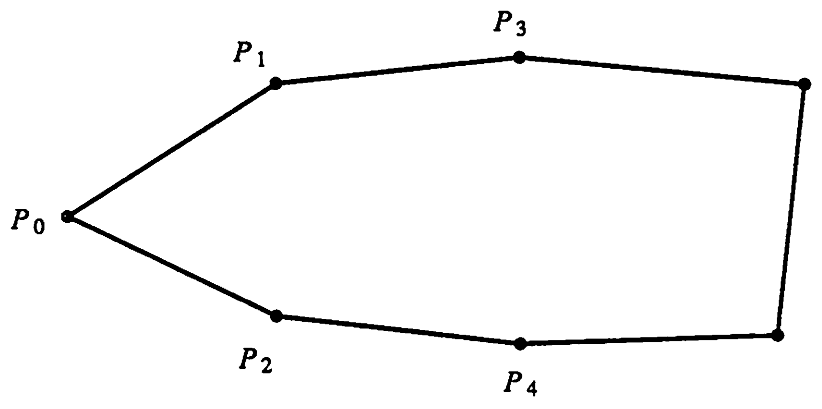


Fig. 6.3.6. A "small" polygon.

First, the sharpest vertex P_0 is searched for (note that P_3 and P_4 may actually be the same vertex if the polygon has only 4 vertices). Then, we try to draw three different virtual cut-lines. The first cut-line is shown in Fig. 6.3.7 :

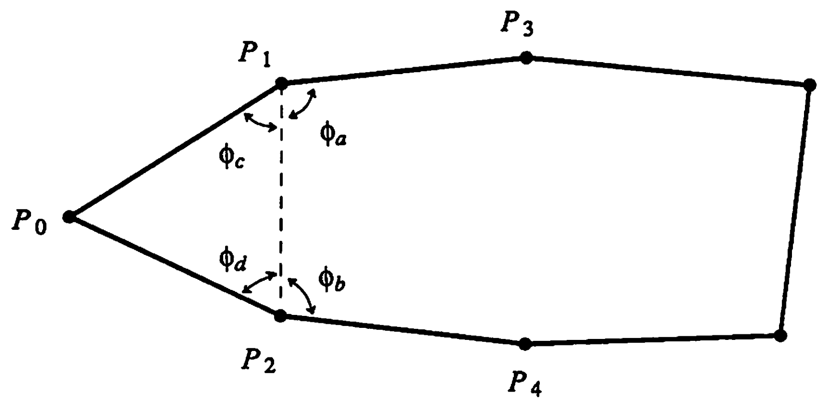


Fig. 6.3.7. The first trial cut.

The angles ϕ_a , ϕ_b , ϕ_c and ϕ_d are calculated. The smallest of them is then noted and defined as ϕ_1.

The second cut-line is shown in Fig. 6.3.8 :

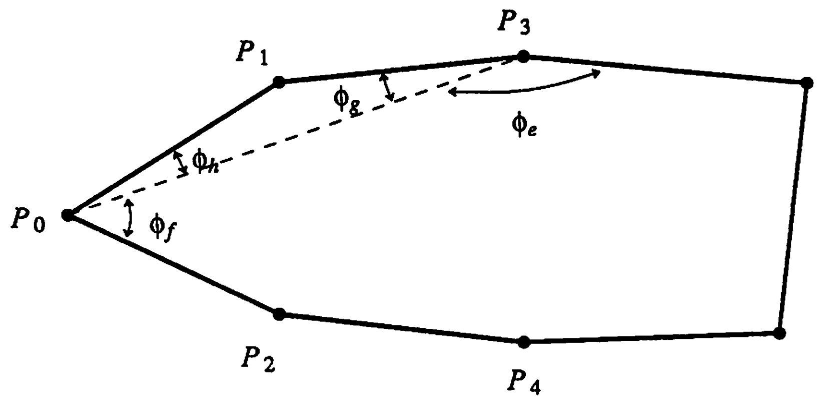


Fig. 6.3.8. The second trial cut.

The various angles are calculated and the smallest of them defined as ϕ_2.

The third cut-line is shown in Fig. 6.3.9 :

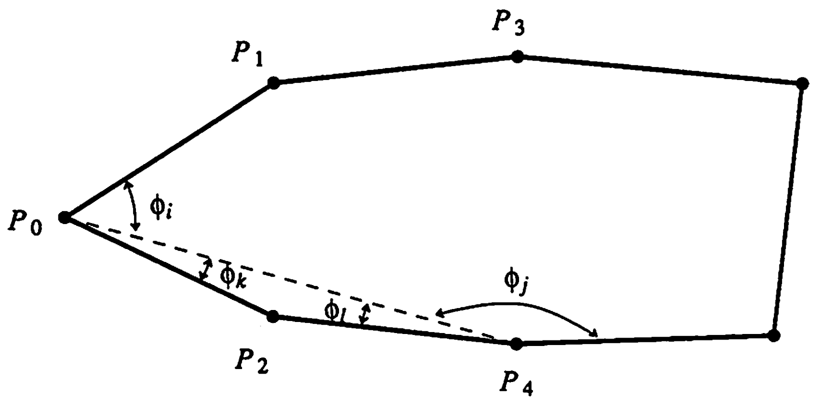


Fig. 6.3.9. The third trial cut.

Again, the various angles are calculated and the smallest of them defined as ϕ_3.

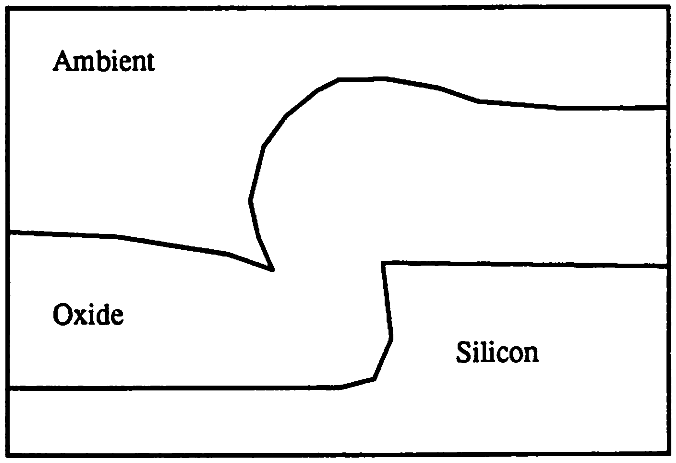
The three angles ϕ_1, ϕ_2 and ϕ_3 are compared. If ϕ_1 is the smallest among the 3 angles, the cut-line shown in Fig. 6.3.7 is used in the actual truncation process. If ϕ_2 is the smallest among the 3 angles, the cut-line shown in Fig. 6.3.8 is used. If ϕ_3 is the smallest angle, the cut-line shown in Fig. 6.3.9 is used. The remaining polygon then goes through the *MG_TriangSmallPoly* procedure again until the whole polygon is triangulated.

6.4. RESULTS AND PERFORMANCE

The mesh-generator implemented in the CREEP program performs quite well on a variety of test cases. In the following, we show the results of mesh-generation over some test structures.

1. Chemical-vapor-deposited oxide film over a silicon step :

(a)



(b)

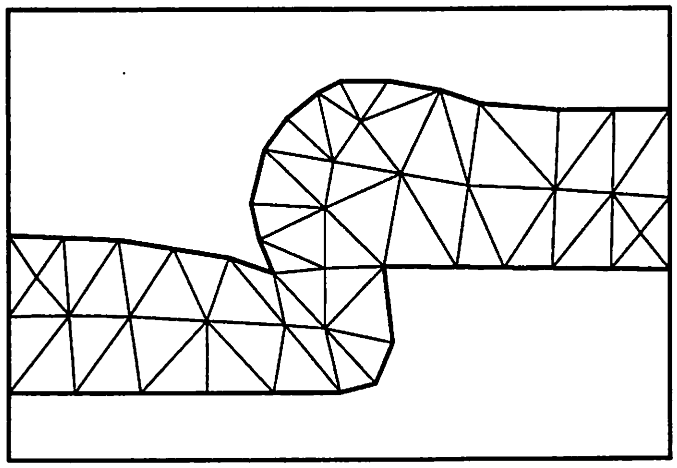
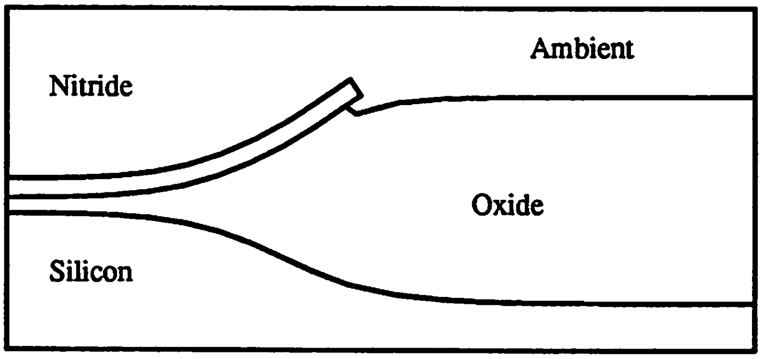


Fig. 6.4.1. (a) A CVD oxide film over a silicon step and (b) the mesh generated over the oxide region.

2. Local oxidation structure :

(a)



(b)

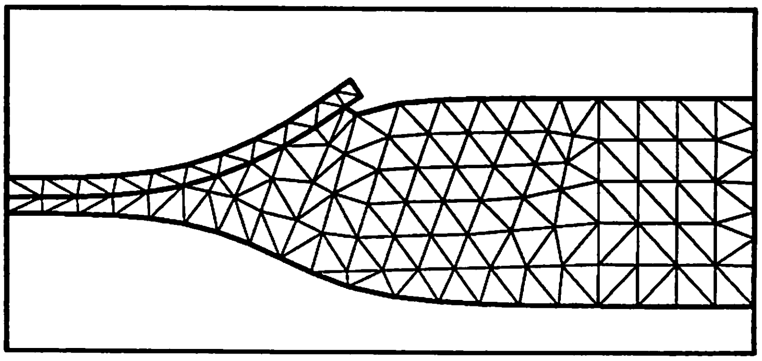
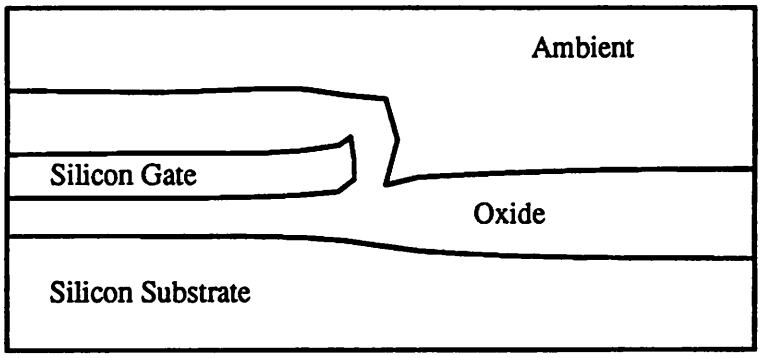


Fig. 6.4.2. (a) A LOCOS oxide and (b) the mesh generated over the oxide and nitride regions.

3. Silicon Gate Structure :

(a)



(b)

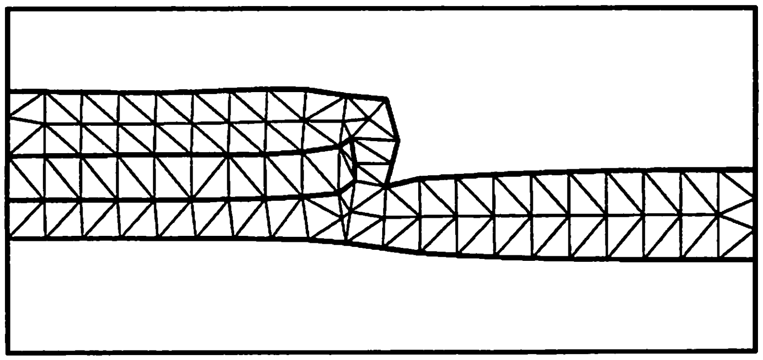
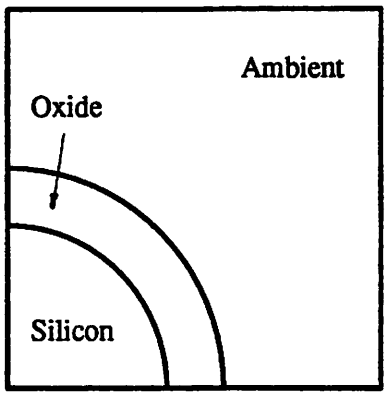


Fig. 6.4.3. (a) A silicon gate structure with oxide grown over it and (b) the mesh generated over the oxide and silicon gate regions.

4. Cylindrical structure (one quadrant) :

(a)



(b)

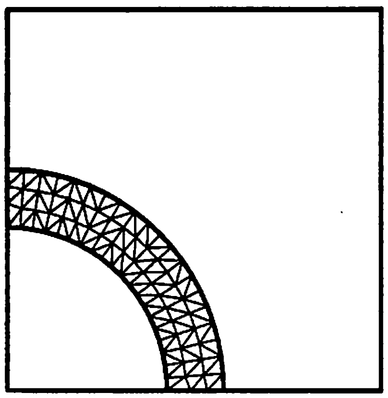


Fig. 6.4.2. (a) A quadrant of a cylindrical oxide region and (b) the mesh generated over the oxide region.

CHAPTER 7.

APPLICATION EXAMPLES

Although the modeling of silicon oxidation cannot be considered complete because of the lack of accurate mechanical models for silicon and silicon nitride, the CREEP program can still be used to investigate many silicon oxidation related problems. In this chapter we present some example runs of the CREEP program.

7.1. OXIDATION OF LOCOS STRUCTURE WITH THIN NITRIDE

Fig. 7.1.1 shows an initial LOCOS structure (before oxidation). The nitride mask is about 500 Å thick, and the initial oxide under the nitride mask is 400 Å thick. The simulation window is 2 μm wide by 1 μm high. The structure is oxidized at 1000 °*C* in wet ambient for 90 minutes, using 15 uniform time-steps. The input files for this simulation is given in Example 1 of Appendix B. The time evolution of the profile is shown in Fig. 7.1.2. The time evolution of the normal traction (positive values for compression) acting on the Si/SiO_2 interface is given in Fig. 7.1.3. The total CPU time required to run this example is about 7 minutes on the VAX 8800 computer.

The three figures are the actual hard-copy outputs (reduced in size to fit into this dissertation) of the CREEP program. Hence, the various material regions are not labeled in Fig. 7.1.1.

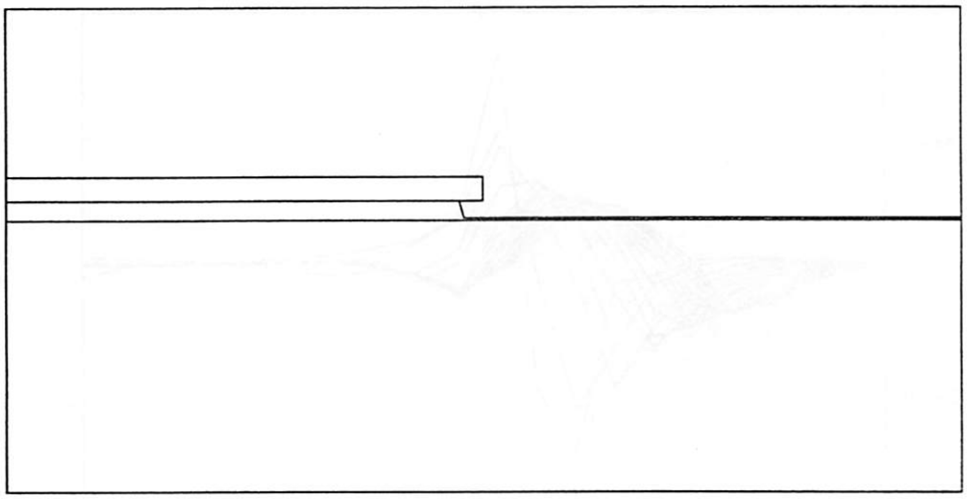


Fig. 7.1.1. Initial structure for a LOCOS oxide.

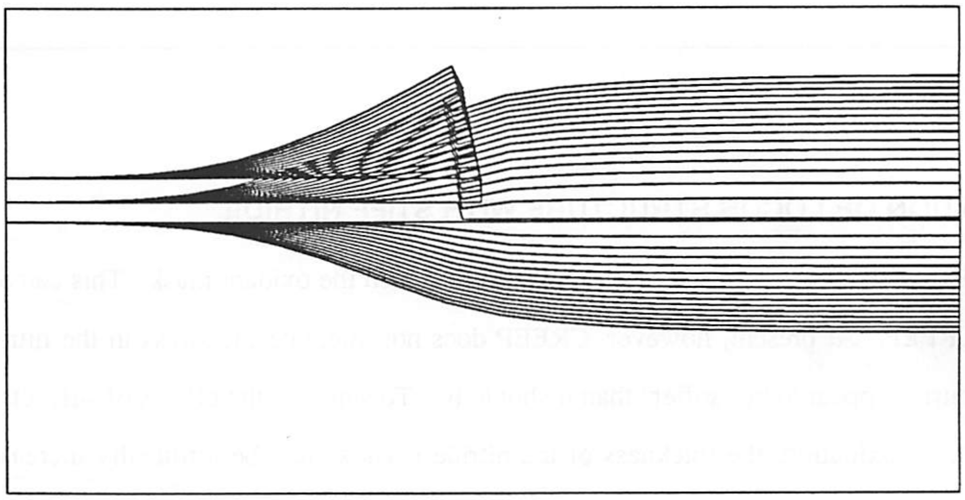


Fig. 7.1.2. Profile evolution of the LOCOS oxide.

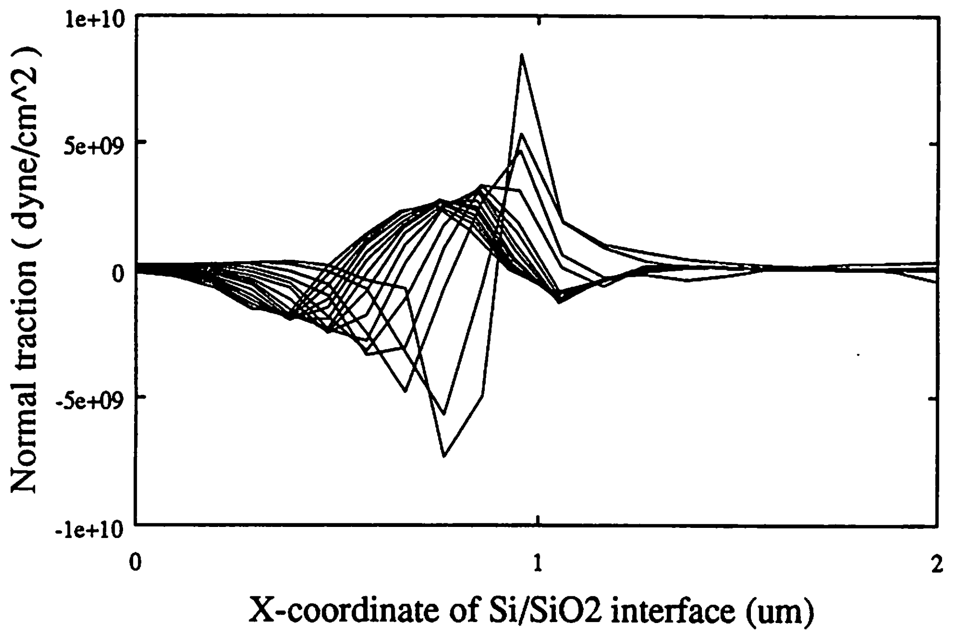


Fig. 7.1.2. Time evolution of the normal traction at the Si/SiO$_2$ interface.

7.2. OXIDATION OF LOCOS STRUCTURE WITH STIFF NITRIDE

In actual experiments, thicker nitride masks are used to stiffen the oxidant mask. This can also be done in CREEP. At present, however, CREEP does not integrate the stress in the nitride. This makes nitride appear to be "softer" than it should be. To simulate the effects of stiff nitride mask on LOCOS oxidation, the thickness of the nitride mask should be artificially increased. Unfortunately, the simulation time will become excessive when many grid points are generated in the large nitride mask. An alternative way of stiffening the nitride is to artificially increase the elastic moduli of nitride. This approach is taken in this example.

Fig. 7.2.1 shows the initial structure for the oxidation, Fig. 7.2.2 shows the time evolution of the oxidation process and Fig. 7.2.3 shows the time evolution of the normal traction at the Si/SiO$_2$ interface. The size of the simulation window, the oxidation temperature, the total oxidation time and the number of time-steps used are the same as those given in the previous example. The input files for this simulation are given in Example 2 of Appendix B. The total CPU time required to run this example is about 7 minutes on the VAX 8800 computer.

By comparing the results of this example with those of the previous one, we see that the lateral encroachment of the oxide under the mask region is reduced by using a stiffer nitride mask. However, the stress at the Si/SiO_2 interface is increased, which may result in an unacceptable level of defect generation in the silicon substrate. This simulation can thus be used to help determine the trade-off between a small lateral encroachment and a low defect generation in LOCOS processes.

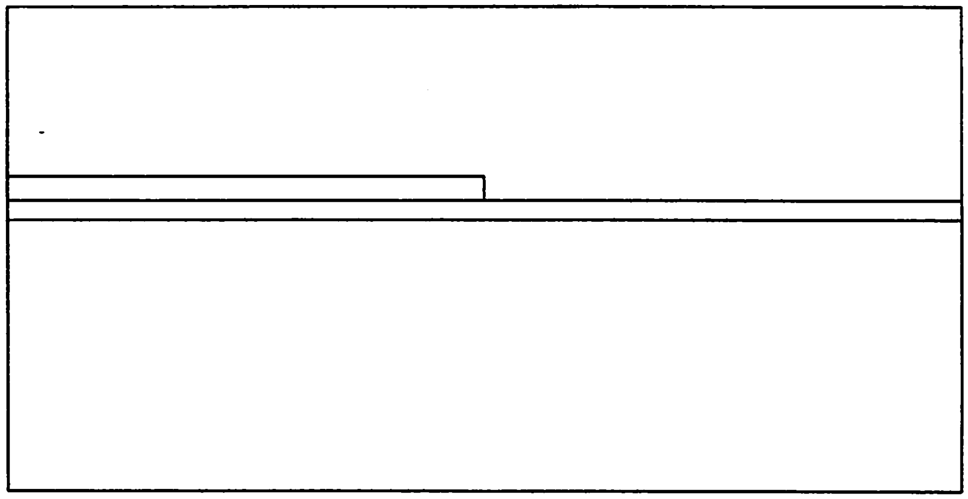


Fig. 7.2.1. Initial structure for a LOCOS oxide with stiff nitride.

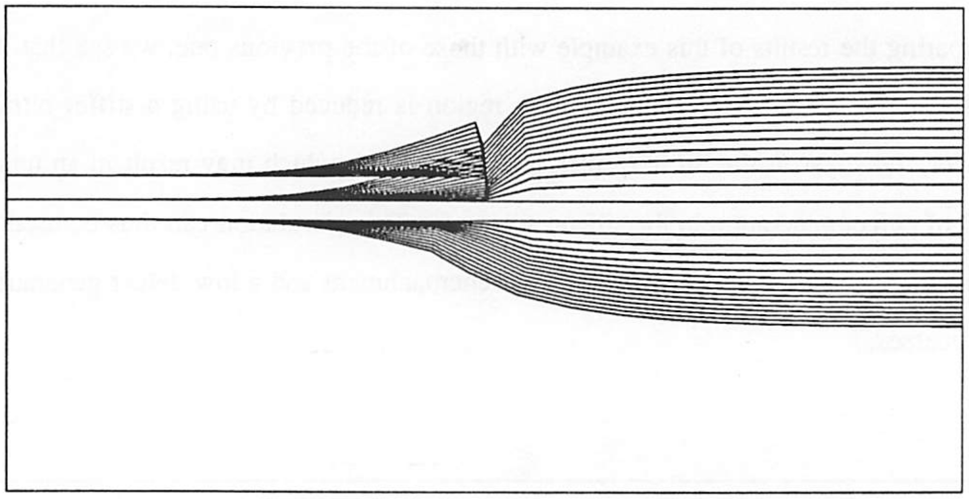


Fig. 7.2.2. Profile evolution of the LOCOS oxide with stiff nitride.

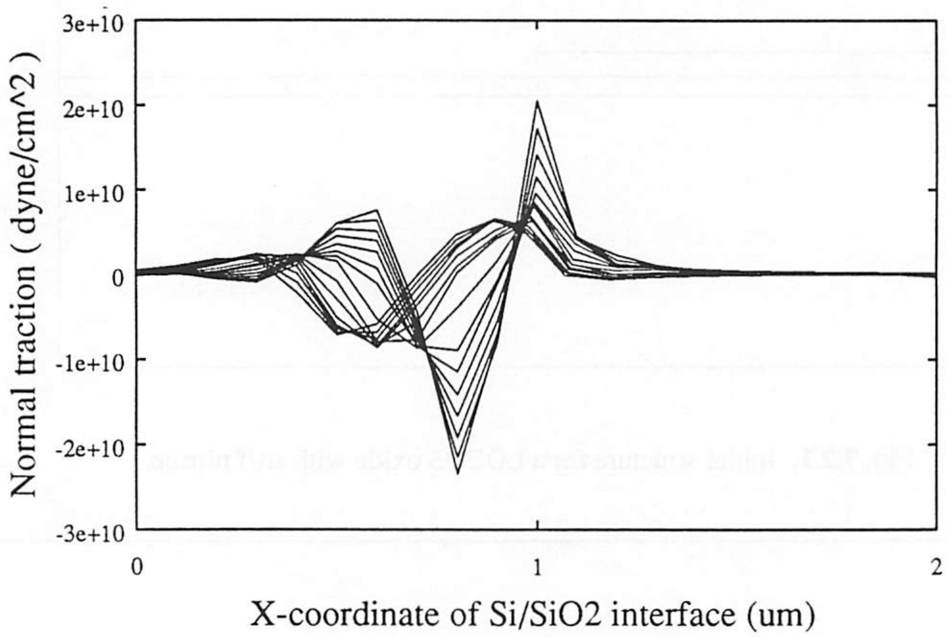


Fig. 7.2.2. Time evolution of the normal traction at the Si/SiO_2 interface.

7.3. OXIDATION OF SILICON GATE STRUCTURE

In this example we show the profile evolution of an MOS silicon gate during an oxidation process (see Example 3 of Appendix B for the input files). Fig. 7.3.1 shows the initial structure. The simulation window is 2 μm wide by 0.7 μm high. The structure is oxidized at 1000 °*C* for 24 minutes in wet ambient, using 6 uniform time-steps. The time evolution of the profile is shown in Fig. 7.3.2. The CPU time required to run this example is about 11 minutes on the VAX 8800 computer.

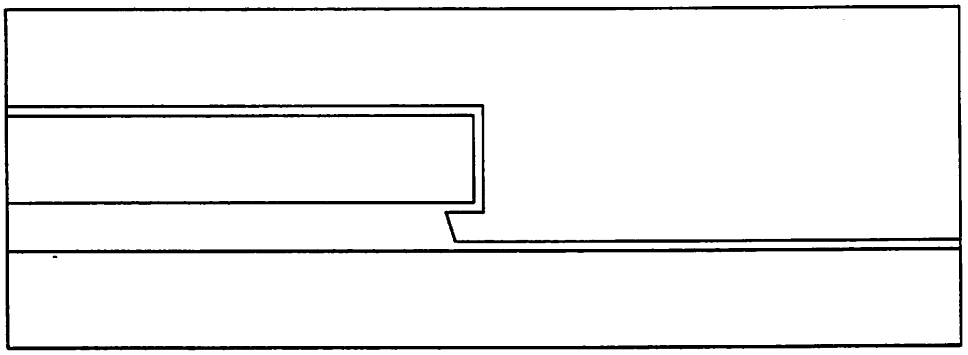


Fig. 7.3.1. Initial profile of a silicon gate structure.

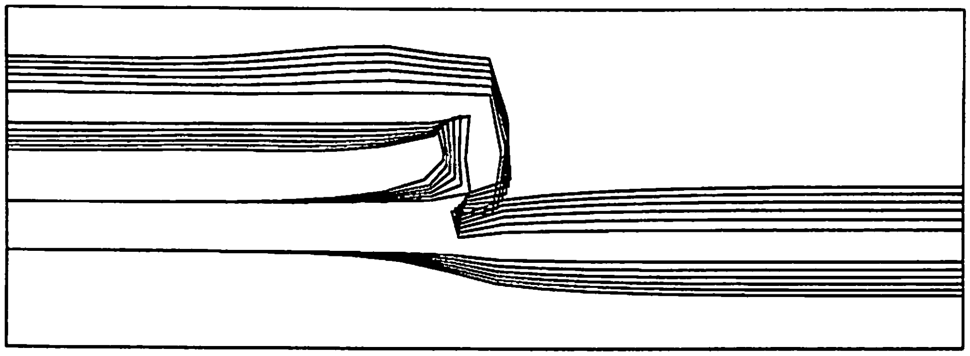


Fig. 7.3.2. Profile evolution during the oxidation of the silicon gate.

7.4. FLOW-ANNEAL OF GLASS

The flow-anneal of glass can be considered a subset problem of silicon oxidation which can be handled by the CREEP program. This is done by simply giving the oxide a lower viscosity and a certain value for its surface tension. In this example the oxidant partial pressure is set to zero to inhibit the oxidation process. The input files for this example are given in Example 4 of Appendix B.

Fig. 7.4.1 shows the (initial) profile of a phospho-silicate glass (PSG) as deposited over a silicon step. Fig. 7.4.2 shows the time evolution of the PSG profile. The total CPU time required to run this example is about 30 seconds on the VAX 8800 computer.

The anneal time and temperature is not relevant here since the viscosity and the surface tension of the oxide have been arbitrarily chosen in this example (see the input files for the values used). The implication is that the viscosity and surface tension of PSG should be experimentally characterized.

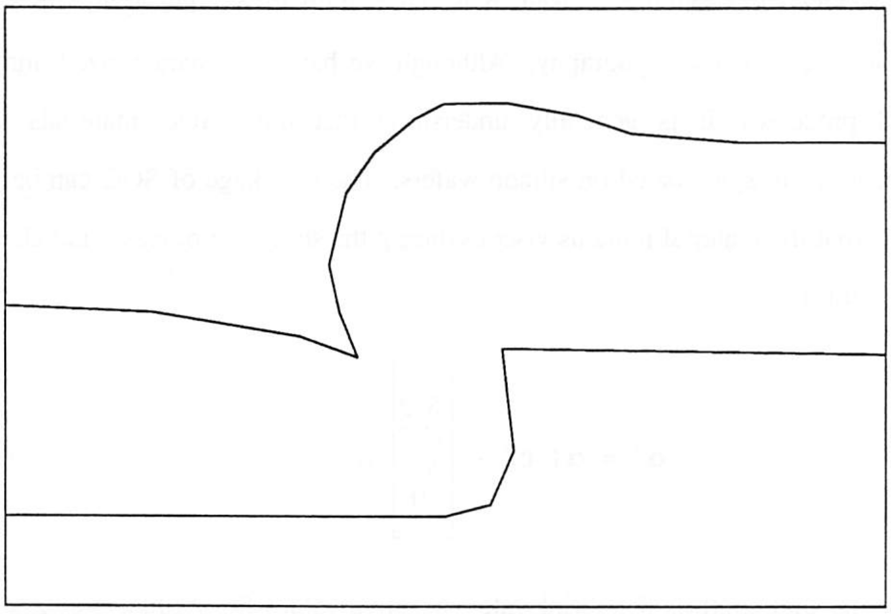


Fig. 7.4.1. Initial profile of a PSG film deposited over a silicon step

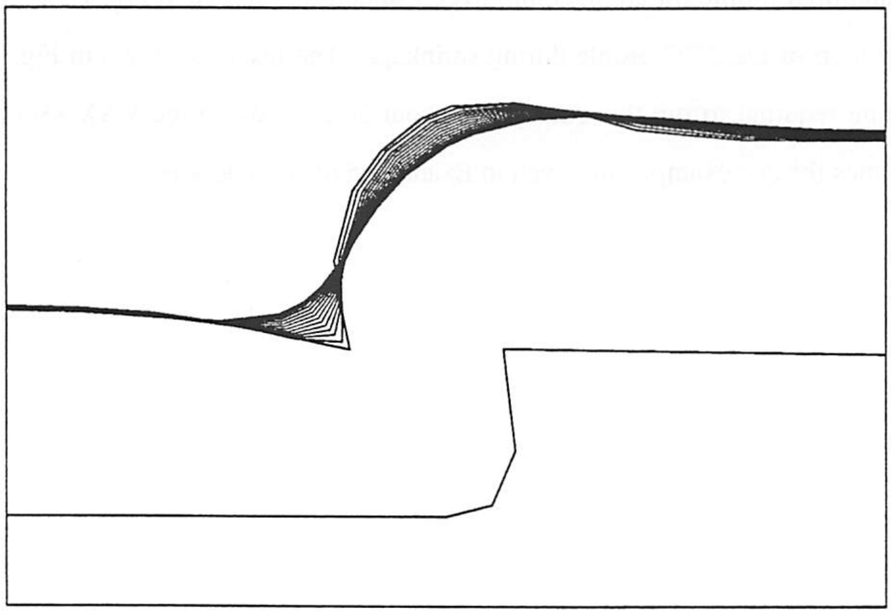


Fig. 7.4.2. Profile evolution of the PSG film during the anneal process.

7.5. SHRINKAGE OF SPIN-ON-GLASS

Spin-on-glasses (SOG) are sometimes used in IC fabrication to provide a smooth surface profile over an originally severe topography. Although we have not characterized and fully understood SOG processes, it is generally understood that most SOG materials shrink significantly after they are spin-coated on silicon wafers. The shrinkage of SOG can be simulated by assuming that the material remains viscous during the shrinking process, and changing Eqn. 4.4.3.6 to the form

$$\sigma'' = \alpha (\dot{\varepsilon}'' - \begin{bmatrix} S/3 \\ S/3 \\ S/3 \\ 0 \end{bmatrix}) \qquad (7.5.1)$$

where S is parameter that controls the shrink-rate (negative values for shrinkage and positive values for expansion). Here, S has the physical significance of the divergence of the velocity field everywhere in the oxide.

Assuming an initial profile for an SOG film over a silicon step shown in Fig. 7.5.1, we can calculate the evolution of the SOG profile during shrinkage. The result is shown in Fig. 7.5.2. The total CPU time required to run this example is about 30 seconds on the VAX 8800 computer. The input files for this example are given in Example 5 of Appendix B.

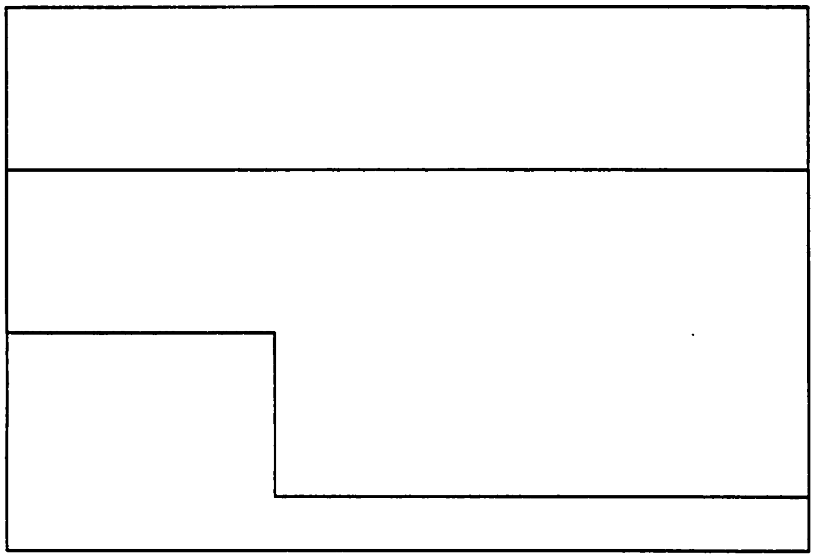


Fig. 7.5.1. Initial profile of an SOG film coated over a silicon step.

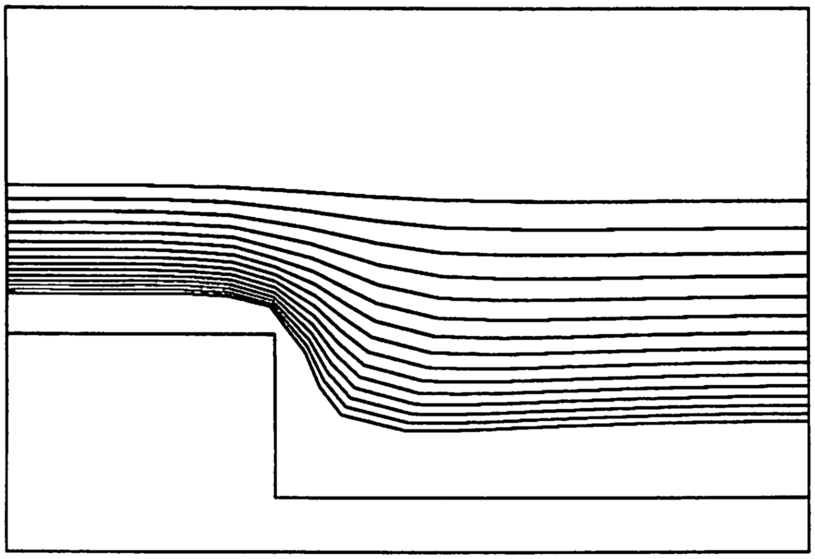


Fig. 7.5.2. Profile evolution of the SOG film during the anneal process.

CHAPTER 8.

CONCLUSIONS

The finite-element method has been applied to the simulation of silicon oxidation. Although not mentioned in any of the previous chapters, the success obtained so far in the modeling work depends very much on the success of the finite-element algorithms and the other support routines. In the course of this work, several different stress dependent oxidation models were tested. Without the ability to solve the model equations, we would not have been able to verify any of the proposed models with sufficient confidence. The speed/efficiency of the finite-element codes developed also helped us "debug" the CREEP program quickly, because test problems can be run almost in real time to allow the programmer to interact directly with the program.

In this research, we have demonstrated the viability of the finite-element algorithms applied to the CREEP process simulator. CREEP can be differentiated from other process simulators by its high level of automation in the mesh-generation and the efficiency of the overall computer codes. This has been achieved through a careful choice of the data structure and the finite-element algorithms. As far as we can learn from the literature, CREEP is the first program that successfully implemented stress-dependent oxidation models which agree well with the only available set of quantitative data of oxidation on cylindrical silicon structures. We would like to attribute this to its ability to provide stable stress calculation, using some finite-element algorithms that were adapted from the field of civil engineering. However, only perfectly cylindrical structures have been used to verify the accuracy of the calculated stress. Further tests are thus needed to verify the accuracy of the program for arbitrary 2D structures.

CREEP can now simulate a number of different oxidation problems, and some other creep-flow problems such as glass flow and spin-on-glass shrinkage processes. However, improvements are still needed even in the area of silicon oxidation. Further modeling work is needed to improve the prediction accuracy of LOCOS and other interesting structures. Non-linear visco-elastic and/or elasto-plastic models are needed to describe the mechanical properties of silicon dioxide, silicon and silicon nitride. Models that accurately describe the oxidation rate around a sharp convex silicon corner are also needed to accurately simulate the oxidation of, for example, trench like structures. Heuristic algorithms outlined in section 4.6.5 and 4.6.6 to handle triple points should be characterized. The algorithms outlined in section 4.5.1 and 4.5.2 for the visco-elastic formulation also need to be characterized. At every time-step in the visco-elastic formulation, the stress-values from the preceding time-step are required to calculate the stress-relaxation. Since a new mesh is generated at every time-step of the computation, the stress-values from the previous time-step must be interpolated to the new mesh. Thus, algorithms for accurately interpolating data from one mesh to another must be developed to implement the visco-elastic models.

It should be pointed out that once the interpolation algorithms have been developed, many other important simulation capabilities can be added to CREEP with relative ease. An important class of process simulation is the impurity diffusion simulation. Impurity diffusion always occurs simultaneously with silicon oxidation, and the two processes interact with each other. The finite-element algorithms for diffusion processes are quite well known, and we now have the major support routines (in particular, the data structure and the mesh-generator). The difficulty that can be foreseen at present is the interpolation of the impurity concentration data from one mesh to another mesh. Since CREEP is intended to be a framework for a general purpose process simulator, a high priority should be given to the development of the interpolation algorithms. The interpolation algorithms should be developed as an extension of the geometric data structure described in chapter 5.

Another two obvious capabilities that can be introduced into CREEP are etching and deposition simulations. The geometric data structure used in CREEP is ideal for handling multi-layer etching and deposition simulation, because most conceivable geometric structures encountered in process simulation can be represented by the data structure.

Finally, we should remark again on the ad hoc nature of applying a simple time-stepping scheme and linear strain-rate (infinitesimal deformation) theory to the large deformation problem of silicon oxidation. Although the results of the numerical simulation for silicon oxidation appear to be accurate within the constraint of the accuracy of the models used, no attempt has been made to prove the applicability of the linear strain-rate theory in silicon oxidation modeling. In the future, the present formulation should be compared to the formulation using the finite-deformation theory. Appropriate changes should be made if the present formulation is found to generate significant amount of errors. However, it should be noted that the field of numerical methods for finite-deformation is relatively new to civil and mechanical engineering, in the sense that it has only been a subject of intense research in the past two decades. Future research in process simulation clearly requires a thorough multidisciplinary understanding of integrated circuit processes, continuum mechanics of finite-deformation and the numerical algorithms required to solve the geometrically non-linear problems encountered in the finite-deformation of many different materials.

REFERENCES

1. L.W. Nagel, *SPICE2 – A Computer Program to Simulate Semiconductor Circuits,* ERL Memo No. ERL-M520, UC Berkeley, May 1975.
2. D. Chin, S.Y. Oh, S.M. Hu, R.W. Dutton, and J.L. Moll, "Two-Dimensional Oxidation," *IEEE Trans. Electron Devices*, vol. ED-30, p. 744, 1983.
3. H. Matsumoto and M. Fukuma, "A Two-Dimensional Si Oxidation Model Including Viscoelasticity," *IEDM Tech. digest*, p. 39, 1983.
4. A. Poncet, "Finite-Element Simulation of Local Oxidation of Silicon," *IEEE Trans. CAD*, vol. 4, no. 1, p. 41, Jan. 1985.
5. L. Borucki, H.H. Hansen, and K. Varahramyan, "FEDSS — A 2D Semiconductor Fabrication Process Simulator," *IBM Journal of Reserach and Development*, vol. 29, no. 3, p. 263, May 1985.
6. T.L. Tung and D.A. Antoniadis, "A Boundary Integral Equation Approach to Oxidation Modeling," *IEEE Trans. Electron Devices*, vol. ED-32, no. 10, p. 1954, Oct. 1985.
7. C. Rafferty, SRC-Berkeley-Stanford Technology Transfer Course, July 8 1986.
8. R.B. Marcus and T.T. Sheng, "The Oxidation of Shaped Silicon Surfaces," *J. Electro-Chemical Soc.*, vol. 129, no. 6, p. 1278, June 1982.
9. J.A. Stricken, W.S. Ho, E.Q. Richardson, and W.E. Haisler, "On Isoparametric vs Linear Strain Triangular Elements," *International Journal for Numerical Methods in Engineering*, vol. 11, p. 1041, 1977.
10. B.E. Deal and A.S. Grove, "General Relationship for the Thermal Oxidation of Silicon," *J. Applied Physics*, vol. 36, no. 12, p. 3770, Dec. 1965.
11. R.B. Marcus, personal communication, 1984.
12. D.B. Kao, J.P. McVittie, W.D. Nix, and K.C. Saraswat, "Two-dimensional silicon oxidation experiment and theory," *IEDM Tech. digest*, p. 388, 1985.

13. E.P. EerNisse, "Stress in thermal SiO_2 during growth," *Applied Physics Letters*, vol. 35, no. 1, p. 8, 1 July 1979.

14. R.D. Corsaro, "Volume relaxation of dry and wet boron trioxide in the glass transformation range following a sudden change of pressure," *Physics and Chemistry of Glasses*, vol. 17, no. 1, p. 13, 1976.

15. Glasstone, *The Theory of Rate Processes*, McGraw-Hill Book Company, 1941.

16. J.H. Li and D.R. Uhlman, "The Flow of Glass at High Stress Levels. I. Non Newtonian Behavior of Homogeneous $0.08Rb_2O \cdot 0.92SiO_2$ Glasses," *J. Non-Crystalline Solids*, vol. 3, p. 127, 1970.

17. G. Hetherington, K.H. Jack, and J.C. Kennedy, "The Viscosity of Vitreous Silica," *Physics and Chemistry of Glasses*, vol. 5, no. 5, p. 130, Oct. 1964.

18. R. Brückner, "Properties and Structure of Vitreous Silica. II," *J. Non-Crystalline Solids*, vol. 5, p. 177, 1971.

19. J.R. Ligenza, "Oxidation of Silicon by High Pressure Steam," *J. Electrochemical Soc.*, vol. 109, no. 2, p. 73, Feb. 1962.

20. R.R. Razouk, L.N. Lie, and B.E. Deal, "Kinetics of High Pressure Oxidation of Silicon in Pyrogenic Steam," *J. Electrochemical Soc.*, vol. 128, no. 10, p. 2214, Oct. 1981.

21. A.J. Moulson and J.P. Roberts, *Trans. Faraday Soc.*, vol. 57, p. 1208, 1961.

22. P.J. Burkhardt, "Tracer Evaluation of Hydrogen in Steam-Grown SiO_2 Films," *J. Electrochemical Soc.*, vol. 114, no. 2, p. 196, Feb. 1967.

23. G.J. Roberts and J.P. Roberts, "An Oxygen Tracer Investigation of the Diffusion of 'Water' in Silica Glass," *Physics and Chemistry of Glasses*, vol. 7, no. 3, p. 82, June 1966.

24. G.W. Scherer, *Relaxation in Glass and Composite*, Jonh-Wiley and Sons.

25. D.B. Kao, "Two-Dimensional Oxidation Effects In Silicon — Experiments and Theory," *Ph.D. Thesis (Stanford Univ.)*, June 1986.

26. J.G. Dil, J.W. Bartsen, R.D.J. Verhaar, and A.E.T. Kuiper, "Locos with Thick and Thin Nitride Spacing," *Philips Journal of Research*, vol. 40, no. 2, p. 72, 1985.

27. Kazuhito Sakuma, Yoshinobu Arita, and Masanobu Doken, "A New Self-Aligned Planar Oxidation Technology," *J. Electrochemical Soc.*, vol. 134, no. 6, p. 1503, June 1987.

28. N. Novedo , *Two Dimensional Characterization of Line Edge Profile of Poly-Crystalline Silicon Oxidation,* Master Thesis (UC Berkeley), Jan 1980.

29. O.C. Zienkiewicz, *The Finite Element Method* , McGraw-Hill Book Company.

30. E.P. Popov , *Introduction to Mechanics of Solids,* Prentice Hall, Inc..

31. P. Sutardja , *Finite element method in oxidation process simulation,* Master Thesis (UC Berkeley), May 1985.

32. C.R. Calladine, *Plasticity for Engineers,* Ellis Horwood Limited, 1985 .

33. K. Lee , *SIMPL-2 (Simulated Profiles From Layout — Version 2),* Ph.D. Thesis (UC Berkeley), July 1985.

34. A. Bykat, "Automatic Generation of Triangular Grid: I — Subdivision of a General Polygon into Convex Subregions. II — Triangulation of Convex Polygons," *International Journal for Numerical Methods in Engineering*, vol. 10, p. 1329, 1976.

APPENDIX A

CREEP User Manual

NAME

creep – A 2D Creep-Flow Process Simulator

SYNOPSIS

creep [file1 ... fileN ... –x command-string –t]

DESCRIPTION

Creep is a new 2D creep-flow process simulator originally designed to solve creep-flow related problems encountered in IC fabrication processes. Its current modeling capabilities include silicon-oxidation, glass-reflow and spin-on-glass (SOG) shrinkage processes. In anticipation for the future-expansion of the **creep** program, we have organized **creep** into modules running under an "arithmetic-shell", which we called the creep-shell. The emphasis of this "arithmetic-shell" is in number-programming, as opposed to the string-programming emphasized in the UNIX "csh". See CREEP-SH(1) for a more detailed discussion of the shell's capabilities.

Creep receives instructions either from the standard input (interactive-mode), from input-files, or directly from its arguments. If no argument is provided to **creep**, it reads instructions from the standard input. If file-names are given as its arguments, then the instructions contain in the files are executed in the order that the files are ordered in the argument-strings. Note that instructions contained in different files are not concatenated, and hence each file must contain a "complete set" of instructions (eg, the overall structure of conditional statements must be complete within each file). A non-fatal error-condition encountered in one file only affects the program execution for that file. If one of the name of the input-file given is a null-string, **creep** will read from the standard input in place of that file. To provide instructions to **creep** directly from **creep**'s arguments, use the arguments :

```
        –x  command-string
```

where command-string is the instruction to be executed by **creep**. Naturally only commands that do not occupy more than one line can be executed this way.

ENVIRONMENT PARAMETERS

Creep reads the environment parameter "TERM" to find out the terminal type the graphics output of the program is to be sent to. Currently **creep** only recognizes the hp2648 and the xterm (X-window). If an xterm is specified, **creep** will read another environment parameter "DISPLAY" to find out the name of the X-server. In UNIX, the environment parameter is set using the *setenv* command.

Occasionally, one might one to run the program with no graphics output. The –t option instructs creep to send all graphics output to a null device. Graphics output can be obtained again by using the **tset** command described in CREEP-SH(1).

AUTHOR

Pantas Sutardja

DIAGNOSTICS

Error messages are generally send to stderr, and interactive prompts to stdout.

BUGS

See the manual pages for the available commands.
Please report undocumented bugs to the author.

NAME

creep-shell – An arithmetic-shell for the CREEP program.

DESCRIPTION

The creep-shell is invoked automatically when **creep** runs. It is called an arithmetic-shell because it supports arithmetic operations and conditional/loop statements based on arithmetic results.

Lexical Structure

With the exception of the control statements (loops and conditional structures) and some very special custom-commands, one can insert several commands into one line. In such a case, the commands must be separated by a semicolon ";" (for this reason, we call the semicolon a shell meta-character). A line which contains one command need not be terminated by a semicolon.

Commands and Key-Words

The interpreter reads the first word in a command to determine the type of the command, and hence the way to execute the command input. The first word used is known as a key-word (other words in a command can also be a key-word). Key-words are restricted to 15 characters. Longer key-words will be truncated to 15 characters. Keywords should be composed of alphanumeric characters and the underscore character. Further, the first character of a key-word must be either the underscore character or one of the alpha-characters. Key-words are case-type sensitive.

A key-word can be either a command, an integer variable, a floating point variable, a string variable, a mathematical function, a module name, or a conditional/loop control command. The shell recognizes the different types of key-words and execute the command inputs according to their context.

Modules and Hierarchy

When the **creep** program is first initiated, it will enter its root module. Within the root module lies a set of commands, variables and functions that will be described later. Other modules can be installed under the root module during compilation time. At present, there are only two levels in the module hierarchy —— the root module and its sub-modules. There are currently several sub-modules installed. They are the data-structure module (**ds_mod**), the plotter module (**plotter**) and the oxidation/annealing simulation module (**annealer**). Each of these sub-modules contains its own set of commands, variables and strings that can be accessed externally. There are also several other modules installed within the **creep** program. But they are not directly accessible to the user. Their operations are controlled by other modules within the program.

To enter a sub-module, simply enter the name of the sub-module. To exit a sub-module, use **exitm** . It is possible to enter another sub-module from a current module. In such a case, each **exitm** command will cause a return to the previous module. Note that it is impossible to exit the first (root) module.

Search Paths

When a key-word is read, the shell searches the current module to determine its type. If the key-word is not found in the current module, the shell will then search the root module.

Extension of the search path is possible by the use of the **path** command, which will be described later.

Note that there can be identical key-words existing within different modules. Further, they can have meaning from one module to another. So there is a possibility of confusion when using a key-word that is defined in more than one module. It is up to the user to prevent the confusion.

The root module contains a set of key-words that are often used and meant to be unique (such as the loop-control and conditional commands). The programmer should not use these key-words in the sub-modules. Otherwise, it is impossible to access the command that resides in the root module when one enters a sub-module which contain the identical key-word. For added protection, the important commands that reside in the root-module are internally labeled to be unique. The user cannot externally assign an identical key-word for a new variable or string.

Variables and Strings

The following is a set of commands related to variables and strings :

float *variable_name*	create a floating point variable called *variable_name*
int *variable_name*	create an integer variable called *variable_name*
rm_var *variable_name*	remove a (**float** or **int**) variable *variable_name*
set *string_name* = *string_sequence*	create a string variable called *string_name*, and assign to it the string given by *string_sequence*. *String_sequence* must be enclosed by two double-quotes if it contains any blank character.
unset *string_name*	remove the string variable called *string_name*.

Arithmetic Operations

The contents of **int** or **float** variables can be assigned by the command :

variable_name = *arithmetic expression*

eg, Prob1 = 1 – exp(– lambda * x)

As mentioned earlier, the shell recognizes if the first word in a command is an **int**, a **float** or other types of commands. If the first word is an **int** or a **float**, then an assignment command is expected. If, for example, the = character is missing, the command will be considered erroneous and thus aborted.

There are a number of arithmetic operators supported. The evaluation of a complex expression is determined by the precedence of the operators. The operators supported are (starting with the one having the highest precedence) :

primary operators : ( ) and arithmetic functions

unary operators : + –

binary operators : * /

+ –

> < >= <=

```
        ==  !=
        &&
        ||
```

Notice that the precedence of the operators are the same as those used in the C programming language. The operators (> < >= <= == != && ||) are relational operators that result in 0 (false) or 1 (true). Thus 1 != 0 (1 not equal to 0) is 1 (true), and so on. They are mainly used in conditional statements. Since > and < are used as binary operators, they are not used as input/output redirection characters as in the UNIX sh and csh.

Mathematical Functions

A number of functions are provided for used in arithmetic expressions. They are (including their arguments):
sin(x) , cos(x) , tan(x) , asin(x) , acos(x) , atan(x) , atan2(x,y) ,
sinh(x) , cosh(x) , tanh(x) , asinh(x) , acosh(x) , atanh(x) ,
exp(x) , log(x) , alog10(x) , log10(x) , pow(x,y)
abs(x), ceil(x), floor(x), sqrt(x)
These functions are the same as those implemented in the UNIX math library for the C programming language.

Shell Meta-Characters

Meta-characters are characters with special meaning to the shell. We shall divide the meta-characters into 2 classes --- primary and secondary. The primary meta-characters are those characters which are interpreted and processed immediately by the shell. There are only a few of these : { ; \ $ } (not including the curly braces). As mentioned earlier the semicolon (;) is used as a command separator. The $ is used as a macro-expansion facility. Its usage is $*var* or $*string* , where *var* is a floating point or integer variable, and *string* is a string variable. If *var* or *string* exists, then $*var* is replaced by the value of *var* and $*string* substituted by its content. To get the literal meaning of ";" and "$", use the sequence \; or \$. To get the backslash character \ itself, use \\ . ie, "\" is also a meta-character. The backslash is also used to make two lines into one by placing it just before the newline (or return) character. Any other character that immediately follows the backslash is converted to the character itself alone.

The secondary meta-characters are those processed and interpreted by the individual commands called. These characters can mean differently to different commands. The most common of these is the double-quote character (") that is used to enclosed a string that contains blanks. Another one is the % character that is used to obtain the literal meaning of the double-quote (usage : %") and the literal meaning of the % character itself (usage : %%). The " and % are secondary meta-characters in commands that expect to read strings.

Other Special Characters

The # sign is a comment-indicator if it appears as the first non-blank character in a statement. Everything after the # sign till the end of the line is simply a comment and thus ignored by the shell.

The ! (exclamation) is a shell-escape character if it appears as the first non-blank character in a statement. It tells the shell that everything after the ! sign till the end of the line are to be executed by the system shell, without first performing the usual input-processing (such as macro-substitution). The **shell** command should be used if input-processing is required before the shell-escape.

The . (dot) tells the shell to exit the current interactive-session, if it is entered as the first non-blank character in a line.

Conditionals

See IF-ELSE(1) for more detail

Loop Control

See LOOPS(1) for more detail

Predefined String and Variables

There are several predefined strings and variables. They are :

prompt	This is a string which is used for the interactive prompt. Defaulted to contain "creep > ".
on	This is an **int** variable that is mainly used as a flag. It is set to 1 and is read-only.
off	This is an **int** variable that is mainly used as a flag. It is set to 0 and is read-only.
true	Same as **on** .
false	Same as **off** .

Other Commands in the Root Module

path *module_1 . . . module_N*
Search for the input key-word, starting from the current module, then the root module, *module_1* , *module_2 . . . , and module_N* in sequence until the key-word is found.

exitm
Exit from the current module to the previous module.

print *arithmetic expression*
Print the value evaluated from the arithmetic expression.

p *arithmetic expression*
same as **print** .

shell *shell escape expression*
This is another way of performing a shell escape. Input processing is first performed on the arguments of this command before escaping to the operating system shell.

stdout *[output_file]*

Redirect the standard output to a file called *output_file* (using append-mode). With no argument, this command sends standard output back to the system's standard output. Note that some diagnostic messages from certain commands are directly sent to the system's standard output and thus cannot be redirected.

stderr *[error_file]*

Redirect the standard error to a file called *error_file* (using append-mode). With no argument, this command sends standard error back to the system's standard error. Note that some diagnostic messages from certain commands are directly sent to the system's standard error and thus cannot be redirected.

echo *[–n] anything*

echo *anything* to the standard output. Carriage return is suppressed with the *–n* argument.

interactive Enter an interactive input mode.

source *file_name* execute the commands in *file_name* .

exit exit one level from a source file or interactive mode.

kill_p kill the program unconditionally.

rename *key-word1 key-word2*

Rename *key-word1* to *key-word2*. Many key-words pre-defined by the program cannot be renamed (find them out for yourself). We do not recommend the use of this command as it may create great confusion.

tset *term-type* This is used for the graphics screen. Currently supported term-types are "hp2648" , "xterm" (X-window system) and the "null" device.

AUTHOR

Pantas Sutardja

DIAGNOSTICS

Error messages are generally send to stderr, and interactive prompts to stdout.

BUGS

Currently, the shell does not recognize the following arithmetic construction :

var1 = var2 binary-op – var3

Please use the following construction instead :

var1 = var2 binary-op (– var3)

See the manual pages for other bugs in the available commands.
Please report undocumented bugs to the author.

EXAMPLE

```
# This is an example file.
# Anything after the # sign is a comment string
# Blank input lines are allowed :

int i ;         # create an integer called i
float x ;       # create an integer called x

# create a string and assign something to it :
set st = "Hello there" ;

i = 12 ;        # assign value 12 to i
x = 1.4 ;       # assign value 1.4 to x
print i * 4 ;   # print the value of the expression ( = 48 )
print exp( x ) ; # print the value of exp( 1.4 )

# The following prints "The value of i is 12 "
echo The value of i is $i ;

# The following prints "x = 1.4 "
echo x = $x ;

# The following prints "Hello there "
echo $st ;

# The following calculates the approximate value of e
x = 1.001
repeat 999
   x = x * 1.001
end
# The above take a while to complete

print x ;       # print the value of x ( = 2.71692 )

# The following executes the echo command 3 times:
foreach string Hello.  "Welcome here."  "Thanks for learning this."
   echo $string
end
# and the results are :
# Hello.
# Welcome here.
# Thanks for learning this.

# The following will print numbers from 0 through 9 twice
i = 0
while ( i < 20 )
   if ( i < 10 )
      print i
   else
      print i - 10
   end
   i = i + 1
end
```

```
# The following repeats the above commands, but now the output goes
# to the file "outputfile"
stdout outputfile ; # send standard output to "outputfile"
i = 0
while ( i < 20 )
   if ( i < 10 )
      print i
   else
      print i - 10
   end
   i = i + 1
end
stdout ;  # send standard output back to the system stdout.


# The following prints the pattern :
# A
# AA
# AAA
# AAAA
# AAAAA
set st = A
repeat 5
   echo $st
   set st = A$st
end
```

NAME

if , elseif, end – conditional control-statement

SYNOPSIS

if *arith. expr.*
    command-group 1
elseif *arith. expr.*
    command-group 2
end

DESCRIPTION

The **if** and **elseif** are statements for conditional controls. The arguments are arithmetic-expressions. If the argument of the **if** evaluates to a non-zero value, the first group of commands are executed. Thereafter, execution continues after the **end** statement. The **elseif** commands provide alternative conditionals. There can be any number of **elseif** in the if-else conditional structure. The conditional structure is terminated by the **end** command. The effect of an error-condition in the evaluation of an arithmetic-expression is the same as a false or zero condition.

AUTHOR

Pantas Sutardja

BUGS

For every source file, the total level of loop-nesting and the **if** statements cannot exceed a specified number which is pre-determined during compilation time (currently set to 16). Unpredictable results will occur if this restriction is violated.

NAME

foreach , repeat , while , end , break , continue --- loop-control facilities

SYNOPSIS

foreach strname *string1 string2 ... stringN*
    commands
end

repeat *arith. expr.*
    commands
end

while *arith. expr.*
    commands
end

DESCRIPTION

The **foreach** command allows the user to substitute a list of strings in a loop. The body of the loop will be executed once for each of the strings in the list. A string is defined as a series of characters enclosed between two double-quotes ("). The quotes can be omited if the there is no blank character in the string. To get literal double-quote, use \"
Strname is any string variable. It is created, if it doesn't exist prior to the foreach command. Strname is set to the strings, one at a time, with each pass of the loop.

Examples:

```
foreach stringname "Guten Tag"   CREEP   "Das Beste Programm"
   echo $stringname
end
```

The command echo string will be executed 3 times. This will produce an output that looks like:

```
Guten Tag
CREEP
Das Beste Programm
```

The **repeat** command is the simplest of the looping facilities. The argument it takes is an arithmetic expression which evaluates to a number. In case this number is not an integer, it is truncated to an integer n. If n is less than one, the loop is not executed. Otherwise, the loop is executed n times. In case of an error in the evaluation of the arithmetic expression, the loop will not be executed.

The **while** command executes the loop as long as the arithmetic-expression does not evaluate to zero (real-number). In case of an error in the evaluation of the arithmetic-expression, the loop will not be executed.

The **end** statement is used to terminate the structure of all loop commands

The **break** statement is used to force a break from the inner-most loop that contains it.

The **continue** statement is used to jump to the top of the inner-most loop to force the next loop-iteration.

RESTRICTIONS

General : All loop-control commands cannot share its input-line with other commands. Further, they (the key-words) should not be generated by macro-substitution.

Specifics :
foreach : macro-substitution that appears in the arguments of **foreach** is expanded only once at the time the loop is entered.
repeat : the arithmetic-expression is evaluated once at the time the loop is entered, and so also is any macro-substitution that appears in the arithmetic-expression.
while : the arithmetic-expression is evaluated as many times as it takes before the loop breaks. So, if there is any macro-substitution in the arithmetic-expression, removal of the macro-variable within the loop will cause an error condition.

AUTHOR

Pantas Sutardja

BUGS

For every source file, the total level of loop-nesting and the **if** statements cannot exceed a specified number which is pre-determined during compilation time (currently set to 16). Unpredictable results will occur if this restriction is violated.

NAME

plotter – data-plot module

DESCRIPTION

The built-in plotter module in **creep** provides the capability to do simple x-y plot. In addition to generating data-plot on the graphics screen of the terminals that **creep** supports, it also generates postscript output for hard-copy printing on postscript printers such as the Apple Laserwriter. Since the plotter is a module within **creep**, the commands belonging to this module can be accessed by either entering the **plotter** module or setting the **path** command to access the plotter module.

Commands and Variables

window *xsize ysize*
: Set the size of the plotting window. The default size is 550 by 275 points or pixels.

origin *x_offset y_offset*
: Set the location of the lower-left corner of the plotting window relative to the lower-left corner of the graphics screen (or paper in case of a postscript plot). The default location is (80 , 80) in points or pixels.

x_range *x_low x_high*
: Set the range of the x-axis. There is no default (ie, auto-ranging).

y_range *y_low y_high*
: Set the range of the y-axis. There is no default (ie, auto-ranging).

read *data_file*
: Read x-y paired data from the file *data_file* . There is a limit of 1000 data pairs. Unpredictable result may occur if this limit is not observed.

r *data_file*
: An abbreviation of **read** .

ra *col# xcol ycol array_file*
: Read x-y paired data from the file *array_file* . The data in the file is assumed to be organized in an array format, with *col#* of columns. The x-axis is read from the *xcol*-th column, and the y-axis is read from the *ycol*-th column. Columns are numbered from 1 through *col#*. The limit that applies to the **read** command applies to this command as well.

get_range
: Perform **x_range** and **y_range** from the data read by **read** or **ra** .

plot *[–rg]*
: Plot a graph of the x-y data on the graphics screen. Without any argument, the x-y ranges are automatically determined from the data. With *–r* , the ranges are determined by **x_range** and **y_range** or **get_range**. The *–g* option plots the graph without grids. The *–rg* option turns off both auto-ranging and grids.

clear	Clear the graphics screen.
cl	A synonym of **clear** .
xlog	Set the x-axis to logarithmic scale.
ylog	Set the y-axis to logarithmic scale.
xlin	Set the x-axis to linear scale. This is the default.
ylin	Set the y-axis to linear scale. This is the default.
log_drange	An integer variable. This is used to set the dynamic range (in number of decades) for the fine grids of log-plot. If the range of the data in number of decades is larger than **log_drange**, the fine grids will not be drawn. However, the fine grids will always be drawn if the range of the data is within 5 decades, regardless of the value of **log_drange** .
pl	An integer variable (defaulted to 1). This is the line pattern for the data-line. Currently defined 11 line-patterns. The line-patterns are similar or identical to the line-patterns on the hp2648 terminals (1 = solid-line , 7 = dotted-line , 11 = point-plot).
gl	An integer variable (defaulted to 7). This is the line pattern for the grids.
uxlabel	This is a string variable that is defaulted to be null. It is used to to write a label on top of plotting window (upper x-label).
lxlabel	Initially a null-string. This is the lower x-label .
lylabel	Initially a null-string. This is the left y-label .
rylabel	Initially a null-string. This is the right y-label .
uxframe	This is an integer variable used as a flag to instruct the plotter to draw the upper-x window frame. It is initially set to 1 (draw uxframe).
lxframe	This is an integer variable used as a flag to instruct the plotter to draw the lower-x window frame. It is initially set to 1 (draw lxframe).
lyframe	This is an integer variable used as a flag to instruct the plotter to draw the left-y window frame. It is initially set to 1 (draw lyframe).
ryframe	This is an integer variable used as a flag to instruct the plotter to draw the right-y window frame. It is initially set to 1 (draw ryframe).

uxscale	This is an integer variable used as a flag to instruct the plotter to print scale-coordinates on the upper-x window frame. It is initially set to 0 (no uxscale).
lxscale	This is an integer variable used as a flag to instruct the plotter to print scale-coordinates on the lower-x window frame. It is initially set to 1 (draw lxscale).
lyscale	This is an integer variable used as a flag to instruct the plotter to print scale-coordinates on the left-y window frame. It is initially set to 1 (draw lyscale).
ryscale	This is an integer variable used as a flag to instruct the plotter to print scale-coordinates on the right-y window frame. It is initially set to 0 (no ryscale).
uxtick	This is an integer variable used as a flag to instruct the plotter to draw scale-ticks on the upper-x window frame. It is initially set to 1 (draw ticks).
lxtick	This is an integer variable used as a flag to instruct the plotter to draw scale-ticks on the lower-x window frame. It is initially set to 1 (draw ticks).
lytick	This is an integer variable used as a flag to instruct the plotter to draw scale-ticks on the left-y window frame. It is initially set to 1 (draw ticks).
rytick	This is an integer variable used as a flag to instruct the plotter to draw scale-ticks on the right-y window frame. It is initially set to 1 (draw ticks).
lmark	Initially a null-string. If it is set to any string, the string will be printed out to the right of the last data-point plotted.

The following are only for postscript plot :

fw	A floating point variable (defaulted to 1.0). This is the thickness of the pen for drawing the plotting window frame (in points).
pw	A floating point variable (defaulted to 0.5). This is the thickness of the pen for drawing the data-line (in points).
pt	An integer variable (defaulted to 0). It is used to define the point type when **pl** = 11 (0 : dot , 1 : cross , 2 : delta , 3 : del , 4 : square , 5 : diamond/rhombus , 6 : circle , 7 : filled circle , 8 : filled diamond , 9 : filled square , 10 : filled del ; 11 : filled delta).

psplot *[output_file]*

A postscript version of the **plot** command. Write output to the *output_file* if it is specified. Otherwise write output to a file called "GX.ps.out" . Before sending the output file to a postscript printer, you need to concatenate the file with a header file ("creep.ps.pro") and a trailer file ("creep.ps.trail"). On UNIX, the usage is :

```
cat creep.ps.pro output_file creep.ps.trail | lpr -Ppostscript_printer
```

append An integer variable used as a flag. Initially set to 0 (false). This is used to determine if the output of **psplot** will be written to a new file or merely appended to an existing file.

autor An integer variable used as a flag. Initially set to 1 (true). This is used to determine if **psplot** will use auto-ranging on the data (default) or simply take the ranges set by **x_range** and **y_range** .

grid An integer variable used as a flag. Initially set to 1 (on). This is used to determine if the grids will be drawn when doing **psplot** .

AUTHOR

Pantas Sutardja

FILES

You should have these two files if you want to use **psplot** :

creep.ps.pro
creep.ps.trail

DIAGNOSTICS

Error messages are generally send to stderr, and interactive prompts to stdout.

BUGS

Vertical labels (**lylabel** and **rylabel**) cannot be printed on an X-window. Otherwise there seems to be no serious bugs. Just impossible to please everyone.

NAME

ds_mod – data-structure module

DESCRIPTION

The data-structure module is used to handle the 2D geometric information. It is currently being used by the oxidation simulator. It is separated from the oxidation simulator (**annealer** module) because it is designed to be rather general purpose so that it may support other simulation modules in the future. As with any other (sub-) module, it is invoked by entering the name of the module (**ds_mod**). Access of the commands belonging to this module can also be made from other modules by appropriately setting the **path** command.

<u>Commands and Variables</u>

struct *structure_file*
: Read the geometric information stored in *structure_file*. The format of the input file is explained under <u>File Format</u>.

save_struct *[structure_file]*
: Save the geometric information stored in the data-structure module to *structure_file*. If a file is not given, the information will be written to the file ".creep.struct" .

draw *[region_number]*
: Generate a plot of the structure. The *region_number* is (currently) an integer value for specifying the material-region that needs to be plotted (1 : ambient region , 2 : rigid substrate , 3 : silicon , 4 : oxide , 5 : nitride). Without the argument, all the material regions within the structure will be plotted.

draw_top *output_file*
: Print the x and y coordinates of the top profile of the structure. Output is overwritten to *output_file* .

psdraw *[+p]*
: Generate a postscript plot of the whole structure. With the *+p* argument, the nodes in the structure will also be shown. The output is appended to the file "GX.ps.out". As with any postscript file generated by **creep**, the output file must be concatenated with a header file ("creep.ps.pro") and a trailer file ("creep.ps.trail") before it can be sent to a postscript printer.

plot_window *x_off y_off xsize ysize*
: Set the offset and the size of the plotting window for **draw** and **psdraw**. Note that the plotting window will not be drawn by **draw** or **psdraw**. The offset is the relative offset of the lower-left corner of the plotting window from the lower-left corner of the graphic-screen or paper. Default values for the arguments are 80 80 550 275 , all in points (1/72 inch) or pixels.

plot_range *x_low y_low x_high y_high*
: Set the range values of the plotting window. The x-coordinates of the left-edge and right-edge of the window are set to *x_low* and *x_high* respectively.

The y-coordinates of the lower-edge and upper-edge of the window are set to *y_low* and *y_high* respectively.

File Format

The structure file should look like the following:

```
nodes
0  x0  y0
1  x1  y1
2  x2  y2
3  x3  y3
4  x4  y4
5  x5  y5
   .
   .
   .
N  xN  yN
segments
0  n1  n2  m1  m2
1  n1  n2  m1  m2
3  n1  n2  m1  m2
   .
   .
   .
   .
S  n1  n2  m1  m2
```

In the above, there are N nodes and S segments in the structure. Under every node-entry, the first (integer) number is the node-number. The second and third (real) numbers are the x and y coordinates of the node. The first four nodes (node 0 through 3) are reserved for the corner nodes of the simulation window (the whole structure must be enclosed in a rectangular simulation window). Node 0 , 1 , 2 and 3 are respectively located at the lower-left , lower-right , upper-right and upper-left corners of the simulation window. The other nodes are located any where else within the simulation window.

Under every segment-entry, the first (integer) number is the segment-number, the second and third (integer) numbers are the node-numbers of the end points of the segments, and the fourth and fifth (integer) numbers are the identification numbers for the materials on the two sides of the segment. The first material entry (m1) spans the region counter-clockwise from n1 to n2. The other region is of material type m2 .

For segments that lie on the boundary of the simulation window, the material type outside the window must be set to type 0 . Other legal material types are numbered from 1 through 127. The current version of the data-structure module does not associate any material type with any material identification number. However, the **annealer** module assigns the numbers from 1 through 5 for ambient regions, rigid substrate regions, silicon regions, oxide regions, and nitride regions. It is recommended that this convention be consistently followed in the future.

AUTHOR

Pantas Sutardja

FILES

You should have these two files if you want to use **psdraw** :

creep.ps.pro
creep.ps.trail

SEE ALSO

PLOTTER(1)

DIAGNOSTICS

Error messages are generally send to stderr, and interactive prompts to stdout.

NAME

annealer – oxidation/reflow/shrinkage module

DESCRIPTION

The **annealer** module reads the information stored in the data-structure module and perform a combination of wet oxidation , reflow and film shrinkage on the structure, according to the input instruction. The module uses a two-dimensional extension of the Deal-Grove models. The oxide is modeled as a viscous incompressible fluid with shear-stress dependent viscosity. The surface reaction rate parameter and the oxidant diffusivity are also stress-dependent. The models are described in reference [1].

As with any other (sub-) module, this module is invoked by entering its name **ds_mod**). Access of the commands that belong to this module can also be made from other modules by appropriately setting the **path** command.

Commands and Variables

oxidize This is the command to actually perform the combination of oxidation , reflow and shrinkage , depending on what the environment parameters are.

temp A **float** variable (initially set to 1273.15). This is used to set the anneal/oxidation temperature (in Kelvin).

time A **float** variable (initially set to 1.0). This is used to set the anneal/oxidation time (in min) for the **oxidize** command.

step An **int** variable (initially set to 1). This is used to set the number of time-steps used by the **oxidize** command. Note that the user need to specify as large a number for this variable to obtain the desired time stepping accuracy in the simulation.

pressure A **float** variable (initially set to 1.0). This is used to set the oxidant (water) partial pressure (in number of atmospheric pressure). A value of zero or less is interpreted by the **oxidize** command as the presence of no oxidant.

surf_tension A **float** variable (initially set to -20.0). This is used to set the surface tension at the oxide/ambient interface (in dyne/cm). Negative value means that the surface energy is higher than the bulk energy (ie, contraction force).

div_v A **float** variable (initially set to 0.0). This is used to set the rate of divergence of the velocity for shrinkage simulation (in per second unit). Set a negative number for shrinkage, and positive value for expansion.

surf_orient A **int** variable (initially set to 2). This is tell the crystal orientation of the silicon substrate. A value of 1 means that the surface shown on the graphic screen is the (100) plane, with the x and y axis in the [110] direction. A value of 2 means that the surface shown on the graphic screen is the (110) plane, with the y axis pointed in the [100] direction and the x axis pointed in

	the [110] direction. Other orientations are not yet defined.
mesh_density	A **float** variable (initially set to 10). The automatic mesh generator generates a quasi-uniform mesh. The nominal mesh-length will be approximately 1 / **mesh_density** (in micron).
density	A synonym of **mesh_density** .
stress_effect	An **int** variable used as a flag to determine if the **oxidize** command will use the stress-effect models. Initially set to 1 (on).
print_stress	An **int** variable used as a flag to tell the **oxidize** command to print the interfacial traction at the oxide/silicon interface after every time-step of the computation. Initially set to 0 (off). If set to 1 (on) , the x-component , y-component , normal component and tangential component of the surface traction will be respectively (over-) written to the files "force.x" , "force.y" , "force.n" and "force.t". The files contain 3-column data array, with the first two columns locating the x and y coordinates of the interfacial points, and the third column storing the value of the stress at the points.
adjust_nodes	An **int** variable used as a flag to instruct the **oxidize** command to automatically move/remove nodes which are too close together. Initially set to 0 (no adjust). We do not recommend the use of this command unless it is absolutely needed (eg, when oxidizing a sharp convex silicon corner), because the algorithms used are not very robust.
Dmax	A **float** variable used to set the limiting (maximum) value of the normalized oxidant diffusivity. If set to a number less than 1.0 , The **oxidize** command will use an internal model to determine the value of the maximum limit on the normalized oxidant diffusivity. If set to a value greater than 1.0 , the **oxidize** command will be forced to use its value instead. Initially set to -1.0 (use internal model) .
Dmin	A **float** variable used to set the minimum value of the normalized oxidant diffusivity. This parameter is not described in reference 1. It is mainly used to increase the stability of the program, and is initially set to 0.01 . It only has an effect on the execution of the **oxidize** command when its value lies in [0.0 , 1.0) .
eta0	A **float** variable used to set the linear-viscosity of oxide (in poise) . If set to a non positive number, the **oxidize** command will be forced to used its own internal model to compute the linear-viscosity of oxide. Initially set to -1.0 (use internal model).
eta_vol	A **float** variable used to set the volume parameter for the shear-stress-dependent oxide viscosity model used in the **annealer** module (in cubic angstroms). If set to a non positive number, the **oxidize** command will be forced to used its own internal model. Initially set to -1.0 (use internal model).

Dox0_h	A float variable used to set the pre-exponential term of the oxidant diffusivity in the high temperature range (in square micron per second).
EDox_h	A float variable used to set the activation energy of the oxidant diffusivity in the high temperature range (in eV).
Dox0_l	A float variable used to set the pre-exponential term of the oxidant diffusivity in the low temperature range (in square micron per second).
EDox_l	A float variable used to set the activation energy of the oxidant diffusivity in the low temperature range (in eV).
Dox_brkpoint	A float variable used to set the break-point temperature (in Kelvin) for the oxidant diffusivity, separating the low and high temperature range.
Dox_vol	A float variable used to set the volume parameter for the hydrostatic-pressure-dependent oxidant diffusivity model (in cubic angstroms). Initially set to 75.0 .
ks0_100	A float variable used to set the pre-exponential term of the surface rate parameter on (100) silicon surface (in micron / second).
Eks_100	A float variable used to set the activation energy of the surface rate parameter on (100) silicon surface (in eV).
ks0_110	A float variable used to set the pre-exponential term of the surface rate parameter on (110) silicon surface (in micron / second).
Eks_110	A float variable used to set the activation energy of the surface rate parameter on (110) silicon surface (in eV).
ks0_111	A float variable used to set the pre-exponential term of the surface rate parameter on (111) silicon surface (in micron / second).
Eks_111	A float variable used to set the activation energy of the surface rate parameter on (111) silicon surface (in eV).
ks_vol	A float variable used to set the volume parameter for the normal-stress-dependent surface rate model (in cubic angstroms). Initially set to 12.5 .
K_Nitride	A float variable used to set the Elastic Bulk Modulus of Nitride.
G_Nitride	A float variable used to set the Elastic Shear Modulus of Nitride.
damp	A float variable used to set the damping (less than 1.0) or acceleration (greater than 1.0) factor in the newton-raphson algorithm. Initially set to 1.0 .

monitor_mesh An **int** variable used as a flag to tell the **oxidize** command to show the finite-element mesh generated at every time-step of the computation. It is initially set to 1 (true).

psdraw_mode An **int** variable that is initially set to 0. If set to 1, the oxidize command will generate a postscript plot of the whole structure together with the 3-node triangular mesh used. The output is appended to the file "GX.ps.out" . If set to 2, the nodes in the system are also shown. If set to 3, it will draw 6-node triangular mesh instead of the 3-node triangular mesh. Nothing is done if any other value is used.

EXAMPLE

The following is the command input file for a LOCOS simulation

```
path ds_mod          ; # set alternate command path

annealer             ; # enter the annealer module

# The following 3 commands belongs to the data-structure
#  ( ds_mod ) module.

plot_window : 80 80 550 275   ;  # Set the plotting
plot_range  : 0 0 2 1         ;  #     window information.

struct locos.st   ; # read the initial structure from
                 # the file "locos.st"

pressure = 0.84   ; # oxidant partial pressure = 0.84 atm
temp = 1000 + 273 ; # temperature = 1273 Kelvin
time = 4          ; # 4 min. oxidation time per oxidize command
step = 1          ; # 1 time-step per oxidize command
mesh_density = 10 ; # set the mesh-density to 10 µm.

draw              ; # Draw the initial structure.  This
                    #    command belongs to the ds_mod module.

# Repeat the oxidize and draw command 10 times ( ie, total
#    oxidation time of 40 minutes ) :

repeat 10
   oxidize ;
   draw    ;
end

save_struct locos.st.final ;  # Save the final structure.
```

The following is the structure file "locos.st" for the locos simulation :

```
nodes
0  0   0
1  2   0
2  2   1
3  0   1
4  0   0.56
5  2   0.56
6  2   0.565
7  1   0.6
8  0   0.6
9  0   0.65
10 1   0.65
11 0.95 0.6
12 0.96 0.565
segments
0   0 1 0 2
1   1 5 0 2
2   5 4 4 2
3   4 0 0 2
4   5 6 0 4
5   6 12 1 4
6   7 11 5 1
7   8 4 0 4
8 7 10 1 5
9 10 9 1 5
10 9 8 0 5
11 6 2 0 1
12 2 3 0 1
13 3 9 0 1
14 11 8 5 4
15 11 12 4 1
```

SEE ALSO

DS_MOD(1)

DIAGNOSTICS

Error messages are generally send to stderr, and interactive prompts to stdout.

REFERENCE

[1] P. Sutardja , W.G. Oldham and D.B. Kao, *"Modeling of Stress-Effects in Silicon Oxidation Including the Non-linear Viscosity of Oxide,"* IEDM 1988 .

APPENDIX B

Examples of Input Files

Example 1. LOCOS with thin nitride

Input Command File :

```
# ++++++++++++++++++++++++++++++++++++++
#    LOCOS simulation example file    +
# ++++++++++++++++++++++++++++++++++++++

path ds_mod plotter
annealer

# Set plotting window information :
plot_window : 80 80 550 275 ;
plot_range  : 0 0 2 1 ;

shell rm GX.ps.out ; # remove the file GX.ps.out if it exists.
shell rm nfile1.ps  ; # remove the file nfile1.ps  if it exists.

struct locos.st ;  # Read the initial structure file.
draw            ;  # draw the structure on the screen.
psdraw          ;  # add the profile into the file GX.ps.out
                   #     in postscript format

# Make another copy of the initial profile :
shell cp  GX.ps.out  locos.st.ps

pressure = 0.84 ;  # oxidant partial pressure = 0.84 atm.

# Set 6 minutes oxidation per oxidize command , and
#  use 1 time-step computation :
step = 1 ;
time = 6 ;

mesh_density = 10 ;  # mesh density = 10 per micron .
monitor_mesh = on ;  # draw the mesh on the graphics screen.
print_stress = on ;  # print the interface stress.

int count  ;  # create an integer called count and
count = 0  ;  #   set it to 0 .

# Perform oxidation :

repeat 15

   oxidize ;

   clear   ;  # clear the screen
   draw    ;  # draw the new profile
   psdraw  ;  # add the profile into the file GX.ps.out
   # save the normal traction data in the files f.n0 .... f.n15 :
   shell mv force.n f.n$count ; # move to a new file

   count = count + 1
end


# Read the x-coordinates and the values of the
#   first set of normal traction data :
ra 3 1 3 f.n0 ;

# Set the range of the plotter using the first set of data :
get_range     ;

append = on   ;
autor  = off  ;
grid   = off  ;

# Set data-plot window information
origin 200 200 ;
window 400 250 ;

# Set the left-y and lower-x labels for the data plot.
#   Because strings are automatically created even if not
#   found in the current module, we are forced to set these
#   strings inside the plotter module ( the path command doesn't
#   help here ) :

plotter  ;  # Enter the plotter module.
set lylabel = "Normal traction ( dyne/cm^2 )"
set lxlabel = "X-coordinate of Si/SiO2 interface (um) "
exitm    ;  # Exit the plotter module.

# We exit the plotter module because the variable count is
#   defined in the annealer module :
count = 0  ;

repeat 15
   ra 3 1 3 f.n$count ;
   psplot nfile1.ps    ; # generate postscript output
   count = count + 1  ;
end

# Send outputs to the postscript printer named "psprinter" for
#    hard copies of the outputs :

shell cat creep.ps.pro  locos.st.ps  creep.ps.trail  | lpr -Ppsprinter
shell cat creep.ps.pro  GX.ps.out  creep.ps.trail  | lpr -Ppsprinter
shell cat creep.ps.pro  nfile1.ps  creep.ps.trail  | lpr -Ppsprinter


# Note : 1. cat is a UNIX command for concatenating files
#        2. lpr is a UNIX printer command
#        3. rm  is a UNIX command for removing file
#        4. mv  is a UNIX command for renaming file

# Obviously you need to have the postscript header and trailer
#   files  "creep.ps.pro" and creep.ps.trail to print the
#   postscript files.
```

<u>Input Structure File ("locos.st") :</u>

```
nodes
0  0   0
1  2   0
2  2   1
3  0   1
4  0   0.56
5  2   0.56
6  2   0.565
7  1   0.6
8  0   0.6
9  0   0.65
10 1   0.65
11 0.95 0.6
12 0.96 0.565
segments
0   0 1 0 2
1   1 5 0 2
2   5 4 4 2
3   4 0 0 2
4   5 6 0 4
5   6 12 1 4
6   7 11 5 1
7   8 4 0 4
8 7 10 1 5
9 10 9 1 5
10 9 8 0 5
11 6 2 0 1
12 2 3 0 1
13 3 9 0 1
14 11 8 5 4
15 11 12 4 1
```

The hard-copies generated in this example are shown in Fig. 7.1.1 (initial profile) , Fig. 7.1.2 (profile evolution) and Fig. 7.1.3 (evolution of the normal traction at the Si/SiO_2 interface) of chapter 7.

Example 2. LOCOS with thick nitride

<u>Input Command File :</u>

```
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
#          LOCOS simulation with stiff nitride.                +
# ____________________________________________________________+
#                                                              +
#  Because CREEP currently does not integrate the stress       +
#     history of nitride, the nitride appears to be softer     +
#     than it should be.  Here, we attempt to simulate a       +
#     thick nitride LOCOS process by giving nitride a larger   +
#     elastic moduli instead of thickening the nitride         +
#     ( in order to save computation time ).                   +
#                                                              +
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

path ds_mod plotter
annealer

# Set plotting window information :
plot_window : 80 80 550 275 ;
plot_range  : 0 0 2 1 ;

# Whatever the elastic moduli of nitride are, increase them
#   by 100 times :
G_Nitride = G_Nitride * 100
K_Nitride = K_Nitride * 100

shell rm GX.ps.out  ; # remove the file GX.ps.out if it exists.
shell rm nfile2.ps  ; # remove the file nfile2.ps  if it exists.

struct tlocos.st ;  # Read the initial structure file.
draw             ;  # draw the structure on the screen.
psdraw           ;  # add the profile into the file GX.ps.out
                    #     in postscript format

# Make another copy of the initial profile :
shell cp  GX.ps.out  tlocos.st.ps

pressure = 0.84 ;  # oxidant partial pressure = 0.84 atm.

# Set 6 minutes oxidation per oxidize command , and
#  use 1 time-step computation :
step = 1 ;
time = 6 ;

mesh_density = 10 ;  # mesh density = 10 per micron .
```

```
monitor_mesh = on ;  # draw the mesh on the graphics screen.
print_stress = on ;  # print the interface stress.

int count  ;  # create an integer called count and
count = 0  ;  #   set it to 0 .

# Perform oxidation :

repeat 15

   oxidize ;

   clear   ;  # clear the screen
   draw    ;  # draw the new profile
   psdraw  ;  # add the profile into the file GX.ps.out

   # save the normal traction data in the files t.n0 .... t.n15 :
   shell mv force.n t.n$count ; # move to a new file

   count = count + 1
end


# Read the x-coordinates and the values of the
#   first set of normal traction data :
ra 3 1 3 t.n0 ;

# Set the range of the plotter using the first set of data :
get_range     ;

append = on   ;
autor  = off  ;
grid   = off  ;

# Set data-plot window information
origin 200 200 ;
window 400 250 ;

# Set the left-y and lower-x labels for the data plot.
#   Because strings are automatically created even if not
#   found in the current module, we are forced to set these
#   strings inside the plotter module ( the path command doesn't
#   help here ) :

plotter  ;  # Enter the plotter module.
set lylabel = "Normal traction ( dyne/cm^2 )"
set lxlabel = "X-coordinate of Si/SiO2 interface (um) "
exitm    ;  # Exit the plotter module.

# We exit the plotter module because the variable count is
#   defined in the annealer module :
count = 0  ;

repeat 15
```

```
   ra 3 1 3 t.n$count ;
   psplot nfile2.ps    ; # generate postscript output
   count = count + 1  ;
end

# Send outputs to the postscript printer named "psprinter" for
#   hard copies of the outputs :

shell cat creep.ps.pro  tlocos.st.ps  creep.ps.trail  | lpr -Ppsprinter
shell cat creep.ps.pro  GX.ps.out  creep.ps.trail  | lpr -Ppsprinter
shell cat creep.ps.pro  nfile2.ps  creep.ps.trail  | lpr -Ppsprinter


# Note : 1. cat is a UNIX command for concatenating files
#        2. lpr is a UNIX printer command
#        3. rm  is a UNIX command for removing file
#        4. mv  is a UNIX command for renaming file

# Obviously you need to have the postscript header and trailer
#   files  "creep.ps.pro" and creep.ps.trail to print the
#   postscript files.
```

<u>Input Structure File ("tlocos.st") :</u>

```
nodes
0  0   0
1  2   0
2  2   1
3  0   1
4  0   0.56
5  2   0.56
6  2   0.6
7  1   0.6
8  0   0.6
9  0   0.65
10 1   0.65
segments
0   0 1 0 2
1   1 5 0 2
2   5 4 4 2
3   4 0 0 2
4   5 6 0 4
5   6 7 1 4
6   7 8 5 4
7   8 4 0 4
8   7 10 1 5
9  10 9 1 5
```

```
10  9  8  0  5
11  6  2  0  1
12  2  3  0  1
13  3  9  0  1
```

The hard-copies generated in this example are shown in Fig. 7.2.1 (initial profile) , Fig. 7.2.2 (profile evolution) and Fig. 7.2.3 (evolution of the normal traction at the Si/SiO_2 interface) of chapter 7.

Example 3. Oxidation around a Silicon Gate

Input Command File :

```
# +++++++++++++++++++++++++++++++++++++++
#      Oxidation of a silicon gate      +
# +++++++++++++++++++++++++++++++++++++++

path ds_mod plotter
annealer

# Set plotting window information :
plot_window : 80 80 550 275 ;
plot_range  : 0 0 2 1 ;

shell rm GX.ps.out ; # remove the file GX.ps.out if it exists.
shell rm nfile3.ps  ; # remove the file nfile3.ps  if it exists.

struct poly.st  ;  # Read the initial structure file.
draw            ;  # draw the structure on the screen.
psdraw          ;  # add the profile into the file GX.ps.out

# Rename the file GX.ps.out to another file
shell mv  GX.ps.out  poly.st.ps

pressure = 0.84 ;  # oxidant partial pressure = 0.84 atm.

# Set 6 minutes oxidation per oxidize command , and
#  use 1 time-step computation :
step = 1 ;
time = 4 ;

mesh_density = 10 ;  # mesh density = 10 per micron .
monitor_mesh = on ;  # draw the mesh on the graphics screen.
print_stress = on ;  # print the interface stress.

int count  ;  # create an integer called count and
count = 0  ;  #   set it to 0 .

# Perform oxidation :

while count < 6

   oxidize ;

   clear   ;  # clear the screen
   draw    ;  # draw the new profile
   psdraw  ;  # add the profile into the file GX.ps.out
   # save the normal traction data in the files p.n0 .... p.n15 :
   shell mv force.n p.n$count ; # move to a new file

   count = count + 1
end


# Send output to the postscript printer named "psprinter" for a
# hard copy of outputs :
shell cat creep.ps.pro  poly.st.ps  creep.ps.trail  | lpr -Ppsprinter
shell cat creep.ps.pro  GX.ps.out  creep.ps.trail  | lpr -Ppsprinter


# It is not obvious how to meaningfully plot the normal traction
#   at the Si/SiO2 interface for this example.  So we will not
#   attempt to generate the normal traction plot as in the previous
#   two examples.


# Note : 1. cat is a UNIX command for concatenating files
#        2. lpr is a UNIX printer command
#        3. rm  is a UNIX command for removing file
#        4. mv  is a UNIX command for renaming file

# Obviously you need to have the postscript header and trailer
#   files  "creep.ps.pro" and creep.ps.trail to print the
#   postscript files.
```

<u>Input Structure File ("poly.st") :</u>

```
nodes
0  0    0.3
1  2    0.3
2  2    1.0
3  0    1.0
4  0    0.5
5  2    0.5
6  0    0.6
7  0.98 0.6
8  0.98 0.78
9  0    0.78
10 0    0.8
11 1    0.8
12 0.94 0.52
13 2    0.52
14 1    0.58
15 0.92 0.58
```

```
segments
0  0   1 0 2
1  1   5 0 2
2  5  13 0 4
3 13   2 0 1
4  2   3 0 1
5  3  10 0 1
6 10   9 0 4
7  9   6 0 3
8  6   4 0 4
9  4   0 0 2
10 6   7 4 3
11 7   8 4 3
12 8   9 4 3
13 13  12 1  4
14 14  11 1  4
15 11  10 1  4
16 4   5   2 4
17 12  15 1  4
18 15  14 1  4
```

The hard-copies generated in this example are shown in Fig. 7.3.1 (initial profile) and Fig. 7.2.2 (profile evolution) of chapter 7.

Example 4. Flow-anneal of PSG

Input Command File :

```
#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
#     Reflow of a PSG film deposited over a silicon step      +
#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

path ds_mod plotter   ; # Set alternate command path

annealer              ; # Enter the annealer module

plot_window : 80 80 550 275 ;  # Set the plotting window
plot_range :  0 0.1 2 1.1   ;  #   information

shell rm GX.ps.out    ; # Remove the file GX.ps.out if it exist

struct psg.st         ; # Read the initial structure
draw                  ; # Show the initial structure
psdraw                ; # draw the initial structure in the file
              #   GX.ps.out in postscript format

# Move the file to a new file :
shell mv GX.ps.out psg.st.ps

pressure = 0          ; # Set no oxidant

mesh_density = 6      ; # Set mesh-density to 6

surf_tension = -20    ; # Set surface tension value to -20 dyne/cm

eta0 = 1e9            ; # Set viscosity of oxide to 1e9 poise

step = 1              ; # Set the anneal time to 2 minutes per
time = 2              ; #   time-step of the computation
temp = 1400           ; # Set the anneal temperature ( in Kelvin )


repeat 3
   oxidize            ; # Perform the anneal step
   cl                 ; # Clear the graphics screen
   draw               ; # draw the updated profile
   psdraw             ; # add the updated profile to the file
              #   GX.ps.out in postscript format
end

time = 1.5              ; # Set anneal time to 1.5 minute per time-step
repeat 10
```

```
   oxidize            ; # Perform the anneal step
   cl                 ; # Clear the graphics screen
   draw               ; # draw the updated profile
   psdraw             ; # add the updated profile to the file
end

# Send outputs to the postscript printer named "psprinter" for
#    hard copies of the outputs :

shell cat creep.ps.pro  psg.st.ps  creep.ps.trail  | lpr -Ppsprinter
shell cat creep.ps.pro  GX.ps.out  creep.ps.trail  | lpr -Ppsprinter
```

<u>Input Structure File ("psg.st") :</u>

```
nodes
     0  0      0.1
     1   1.5    0.1
     2   1.5    1.1
     3 . 0      1.1
     4   0.00  0.25
     5   0.75  0.25
     6   0.83  0.27
     7   0.87  0.36
     8   0.85  0.53
     9   1.5   0.52
    10   1.5   0.68
    11   1.5   0.875
    12   1.25  0.875
    13   1.07  0.89
    14   0.98  0.92
    15   0.86  0.941
    16   0.75  0.94
    17   0.7   0.915
    18   0.63  0.859
    19   0.58  0.79
    20   0.55  0.67
    21   0.568 0.59
    22   0.6   0.515
    23   0.5   0.55
    24   0.25  0.59
    25   0     0.6
    26   0     0.42
segments
 0  4  5  2 4
 1  5  6  2 4
 2  6  7  2 4
```

```
3  7  8  2 4
4  8  9  2 4
5  9  10  0 4
6  10  11  0 4
7  11  12  1 4
8  12  13  1 4
9  13  14  1 4
10  14  15  1 4
11  15  16  1 4
12  16  17  1 4
13  17  18  1 4
14  18  19  1 4
15  19  20  1 4
16  20  21  1 4
17  21  22  1 4
18  22  23  1 4
19  23  24  1 4
20  24  25  1 4
21  25  26  0 4
22  26  4   0 4
23   0  1   0 2
24   1  9   0 2
25  11  2   0 1
26   2  3   0 1
27   3  25  0 1
28   4  0   0 2
```

The hard-copies generated in this example are shown in Fig. 7.4.1 (initial profile) and Fig. 7.4.2 (profile evolution) of chapter 7.

Example 5. Shrinkage of SOG

Input Command File :

```
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
#    Shrinkage of an SOG film coated over a silicon step      +
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

path ds_mod plotter   ; # Set alternate command path

annealer              ; # Enter the annealer module

stress_effect = off   ; # Do not use stress dependent models

plot_window : 80 80 400 200   ; # Set the plotting window
plot_range  : 0 0 2 1         ; #   information

shell rm GX.ps.out    ; # Remove the file GX.ps.out if it exist

struct sog.st         ; # Read the initial structure
draw                  ; # Show the initial structure
psdraw                ; # draw the initial structure in the file
              #   GX.ps.out in postscript format

# Move the file to a new file :
shell mv GX.ps.out sog.st.ps

pressure = 0          ; # Set no oxidation

surf_tension = -20    ; # Set surface tension value to -20 dyne/cm

div_v = -0.001        ; # Set the diverges of the velocity to
              #   -0.001 everywhere in the oxide

step = 1              ;
time = 1.5            ; # Set 1.5 minute time-step

mesh_density = 7      ; # Set mesh-density to 7

repeat 5
   oxidize            ; # Perform the anneal step
   cl                 ; # Clear the graphics screen
   draw               ; # draw the updated profile
   psdraw             ; # add the updated profile to the file
              #   GX.ps.out in postscript format
end

mesh_density = 10     ; # Increase the mesh density to 10
```

```
repeat 10
   oxidize              ; # Perform the anneal step
   cl                   ; # Clear the graphics screen
   draw                 ; # draw the updated profile
   psdraw               ; # add the updated profile to the file
end

# Send outputs to the postscript printer named "psprinter" for
#    hard copies of the outputs :

shell cat creep.ps.pro  psg.st.ps  creep.ps.trail  | lpr -Ppsprinter
shell cat creep.ps.pro  GX.ps.out  creep.ps.trail  | lpr -Ppsprinter
```

Input Structure File ("sog.st") :

```
nodes
0  0   0
1  1.5 0
2  1.5 1
3  0   1
4  0   0.4
5  0.5 0.4
6  0.5 0.1
7  1.5 0.1
8  0   0.7
9  1.5 0.7
segments
0  0  1  0 2
1  1  7  0 2
2  7  9  0 4
3  9  2  0 1
4  2  3  0 1
5  3  8  0 1
6  8  4  0 4
7  4  0  0 2
8  8  9  4 1
9  4  5  2 4
10 5  6  2 4
11 6  7  2 4
```

The hard-copies generated in this example are shown in Fig. 7.5.1 (initial profile) and Fig. 7.5.2 (profile evolution) of chapter 7.