PhoeniX Software

FLOWDESIGNER MODULE

Design of Experiment (DOE)
+
Monte-Carlo (MC)

Application Note 032008

Remco Stoffer, Senior Designer
PhoeniX B.V, The Netherlands
# Table of Contents

1. Introduction .......................................................................................................................... 3  
2. Basic Process Flow .............................................................................................................. 4  
   2.1. Graphical Flow Description ........................................................................................ 4  
   2.2. Process Flow Script .................................................................................................... 6  
3. DOE Analysis and Quadratic Fit .......................................................................................... 10  
4. Predicting Parameter Values ............................................................................................... 12  
5. Monte-Carlo Analysis ......................................................................................................... 13
1. Introduction

A Design of Experiment (DOE) is used to determine the influence of variation of fabrication parameters on measurable quantities. Due to the expense of fabrication, usually only a very limited amount of experiments is performed, from which a maximum amount of information must be gained. In general, a fit of the data is performed using the measured data points. This fit can then be used to find optimal working points or to determine the expected yield of a process. One way to determine this yield is by doing so-called Monta-Carlo analysis: a large number of parameter sets is generated, using a normal distribution with the (presumed to be known) process standard deviations, and the number of samples within specifications is determined.

One can also use simulations to perform these 'experiments'. This application note shows one way to do this in FlowDesigner. A structure consisting of two channels, one on both sides of a wafer, is created. Three parameters are varied, and a fit is made into a full quadratic description. This fit is used to determine what fraction of samples will pass a yield criterion (we rather arbitrarily choose a certain minimum desired wall thickness between the two channels) using a 10,000 sample Monte-Carlo analysis. Additionally, the fit is used to determine the expected parameter value when the two channels will merge. It turns out that this expectation is not exact (due to being relatively far from the original parameter values, so the quadratic fit is not expected to be perfect), but still close.
2. Basic Process Flow

Note that the flow presented here is by no means a realistic one; it was chosen because it gives interesting results and we can do some DOE analysis on it.

The flow is first visualized by means of a series of illustrations; next, the script that describes this process flow is presented.

2.1. Graphical Flow Description

The process flow starts with a glass wafer:

This wafer is covered on both sides by silicon nitride:

and subsequently with a layer of sacrificial material:
Next, photoresist is applied to both sides, and a hole is developed into it:

An etchant is applied to this structure. It etches all materials except the photoresist isotropically, but the etch rates are different for the different materials; the sacrificial layer etches three times as fast as the silicon nitride, while the glass etch rate is close to that of the nitride. Due to this difference in etch rates, a tapered structure appears in the nitride layer:

After removal of the photoresist layers, a glass etchant is applied to the wafer that etches the glass and the sacrificial layer isotropically:
2.2. Process Flow Script

First, the materials and etchants, and their interactions, are defined:

```cpp
/**
 * Glass material. This will be the bulk of the 'wafer',
 * into which channels will be etched.
**/
material Glass{
}
sim::AddMaterial(Glass(),RGB(255,255,255));

/**
 * Silicon Nitride. Will be used as a masking material
 * for the glass.
**/
material Si3N4{
}
sim::AddMaterial(Si3N4(),RGB(0,255,255));

/**
 * Photoresist material.
**/
material Resist{
}
sim::AddMaterial(Resist(),RGB(255,0,0));

/**
 * Sacrificial material; this will be placed between the
 * photoresist and the silicon nitride. Its relatively
 * high etch rate will cause a tapering in the nitride
 * layer.
**/
material Sacrificial{
}
sim::AddMaterial(Sacrificial(),RGB(0,255,0));

/**
 * Three etchants are defined:
 * - A developer that only attacks the photoresist
 * - An isotropic etchant that attacks all materials except
 *   the resist with different etch speeds
 * - An isotropic etchant that only attacks the glass
**/
etchant 'Developer';
etchant 'Glass Etchant';
etchant 'Glass Etchant';
```
etchant 'Isotropic Etchant';

/**
 * The glass has no response to the developer, and
 * different isotropic etch rates for the other two
 * etchants.
 **/
sim::SetResponse(Glass, Developer, NONE, {1.0}, 54.74, 20);
sim::SetResponse(Glass, 'Glass Etchant', ISOTROPIC, {1.0}, 54.74, 20);
sim::SetResponse(Glass, 'Isotropic Etchant', ISOTROPIC, {2.5}, 54.74, 20);
/**
 * The silicon nitride only reacts with the
 * Isotropic Etchant.
 **/
sim::SetResponse(Si3N4, Developer, NONE, {1.0}, 54.74, 20);
sim::SetResponse(Si3N4, 'Glass Etchant', NONE, {1.0}, 54.74, 20);
sim::SetResponse(Si3N4, 'Isotropic Etchant', ISOTROPIC, {1.0}, 54.74, 20);
/**
 * The photoresist only reacts (directionally) with the
 * developer.
 **/
sim::SetResponse(Resist, Developer, DIRECTIONAL, {1.0}, 54.74, 20);
sim::SetResponse(Resist, 'Glass Etchant', NONE, {1.0}, 54.74, 20);
sim::SetResponse(Resist, 'Isotropic Etchant', NONE, {1.0}, 54.74, 20);
/**
 * The sacrificial layer is etched isotropically by
 * both the glass etchant and the isotropic etchant,
 * but with significantly different rates.
 **/
sim::SetResponse(Sacrificial, Developer, NONE, {1.0}, 54.74, 20);
sim::SetResponse(Sacrificial, 'Glass Etchant', ISOTROPIC, {10.0}, 54.74, 20);
sim::SetResponse(Sacrificial, 'Isotropic Etchant', ISOTROPIC, {3}, 54.74, 20);

The process flow itself is contained in a function, which performs the flow, measures the
measurables, and returns them in a single variable (which is an array of length three).

/**
 * This function performs the flow, and returns an
 * array of doubles; the first one is the wall width
 * between the channels, the second one the depth of
 * the channels, and the third their width.
 **/
function PerformFlow (i, j, k)
{
    double result[3];
    double sacrificial_t = sacrificial_thickness+i*sacrificial_thickness_sigma;
    double bottom_mask_p = bottom_mask_pos+j*bottom_mask_pos_sigma;
double Glass_Etch_s = Glass_Etch_Speed+k*Glass_Etch_Speed_sigma;
// Set the response of the glass to the isotropic etchant.
sim::SetResponse(Glass,'Isotropic Etchant',ISOTROPIC,(Glass_Etch_s),54.74,20);
// Remove all structure from the previous run.
sim::ClearStructure();
// Take a 'wafer' of glass
sim::GetWafer(Glass(),{-8,-1,8,1});
// Deposit silicon nitride on both sides
sim::IsotropicDeposit(Si3N4(),1.0,0,BOTH,0.2,0.1);
// Followed by the Sacrificial layer
sim::IsotropicDeposit(Sacrificial(),sacrificial_t,0,BOTH,0.2,0.1);
// and photoresist
sim::IsotropicDeposit(Resist(),1,0,BOTH,0.2,0.1);
// We simulate exposing / developing the photoresist by
// directionally etching the resist through a mask.
  sim::SetMask({-2.2,-1.8},0);
  sim::Etch(Developer,1,0,FRONT,0.2,0.1);
  sim::SetMask({bottom_mask_p-0.2, bottom_mask_p+0.2},0);
  sim::Etch(Developer,1,0,BACK,0.2,0.1);
// Now that the etch holes in the photoresist have been
// created, we can etch using the Isotropic Etchant.
  sim::Etch('Isotropic Etchant',1.4,0,BOTH,0.02,0.01);
// The photoresist and sacrificial layer are stripped from
// the structure
  sim::RemoveMaterial(Resist());
  sim::RemoveMaterial(Sacrificial());
// And the glass is etched.
  sim::Etch('Glass Etchant',1.2,0,BOTH,0.02,0.06);
  sim::AddToGraph ("+i+", "+j+", "+k");
// Get the air-material interfaces along the line between the
// centres of the holes in the top and bottom of the glass
// wafer; the distance between them is a reasonable approximation
// of the wall thickness between the two channels.
  double holes[] = sim::GetHolesInSlice({-2.0,1.0},{bottom_mask_p,-1.0});
  if (len(holes)==2)
    result[0] = holes[1] - holes[0];
  else result[0] = 0.0;
  printf (result[0], "	");
// The depth of the channels is determined in a similar way; by
// calculating the interfaces from the top of the wafer to its
// bottom along the centre of the hole.
  holes = sim::GetHolesInSlice ({-2.0, 1.0}, {-2.0, -1.0});
  if (len(holes)>0)
    result[1] = holes[0];
  else result[1] = 0.0;
printf (result[1], "\t");
// and the channel width is easily calculated from the interfaces
// near the top of the wafer.
holes = sim::GetHolesInSlice ({-7, 0.99}, {7, 0.99});
if (len(holes)==2)
    result[2] = holes[1] - holes[0];
else result[2] = 0.0;
printf (result[2], "\n");
return result;
}

The results shown in Section 2.1 are generated by calling this function with 0.0 for all three parameters:

PerformFlow (0.0, 0.0, 0.0);
3. DOE Analysis and Quadratic Fit

This application note performs the most simple quadratic DOE analysis with three parameters. These three parameters are:

1. The thickness of the sacrificial layer (the green layer in the pictures in 2.1). Its base value is 1.0 micrometer; its standard deviation is assumed to be 0.1 micrometers.
2. The position of the bottom mask. The designed value is 2.3 microns, with a standard deviation of 0.5 microns.
3. The etch rate of the glass when exposed to the isotropic etchant. The base value is 0.7 microns per minute; the standard deviation is 0.3 microns per minute.

All three parameters are set to three values: Centre, Centre+stdev, and Centre-stdev. Simulations are performed for all permutations of the three values of the three parameters, yielding a total of 27 simulations. The meausurables that are determined at each simulation run are fitted to a full quadratic function of the three parameters. This function has the following form:

\[ v = c_0 + c_1 z + c_2 y + c_{3y} z^2 + c_{3y} y^2 + c_{4y} y z + c_{4y} z^2 + c_{5y} y^2 z + c_{5y} z^2 + c_{6y} y^3 + c_{7} y^4 + c_{7} y^2 z^2 + c_{8} y^2 z^2 + c_9 x + c_{10} x z + c_{11} x^2 z^2 + c_{11} x y + c_{12} x y z + c_{13} x y z^2 + c_{14} x y^2 + c_{16} x y^2 z + c_{17} x y^2 z^2 + c_18 x^2 + c_{19} x^2 z + c_{20} x^2 z^2 + c_{21} x^2 y + c_{22} x^2 y z + c_{23} x^2 y z^2 + c_{24} x^2 y^2 + c_{25} x^2 y^2 z + c_{26} x^2 y^2 z^2 \]

where \( x, y \) and \( z \) are the (normalized) values of the deviations of the parameters from their centre points.

From the 27 measurements, a system of linear equations can be set up by creating a 27x27 matrix \( M \), and filling its elements such that the matrix-vector equation \( M c = v \) describes the 27 equations. For example, the first equation (that corresponds with the first row of the matrix) has -1 for each of the normalized parameters and is thus:

\[ v_0 = c_0 - c_1 + c_2 - c_3 + c_{4y} - c_5 + c_{6y} - c_7 + c_8 - c_9 + c_{10} - c_{11} + c_{12} - c_{13} + c_{14} - c_{15} + c_{16} + c_{17} + c_{18} - c_{19} + c_{20} - c_{21} + c_{22} - c_{23} + c_{24} - c_{25} + c_{26} \]

so the first line of the matrix is

\[
[1 -1 1 -1 1 -1 1 -1 1 -1 1 -1 1 -1 1 -1 1 -1 1 -1 1 -1 1 -1 1 -1 1]
\]

The 27 simulations are performed, the measured values are stored, and the matrix is created and filled in the following script snippet:

```c
double wall_thickness[27];
double depth[27];
double width[27];
double coefficient_matrix[27][27];

int i,j,k;

sim::StartProcessSimulation("DOE Sims");
```
for (i=-1; i<2; i++)
  for (j=-1; j<2; j++)
    for (k=-1; k<2; k++)
    {
      var result[] = PerformFlow (i,j,k);

      wall_thickness[(i+1)*9+(j+1)*3+k+1] = result[0];
      depth[(i+1)*9+(j+1)*3+k+1] = result[1];
      width[(i+1)*9+(j+1)*3+k+1] = result[2];
    }

for (int _i=0; _i<3; _i++)
  for (int _j=0; _j<3; _j++)
    for (int _k=0; _k<3; _k++)
    {
      double tmp = 1.0;
      int ctr;
      for (ctr=0; ctr<_i; ctr++) tmp *= i;
      for (ctr=0; ctr<_j; ctr++) tmp *= j;
      for (ctr=0; ctr<_k; ctr++) tmp *= k;
      coefficient_matrix[(i+1)*9+(j+1)*3+k+1][_i*9+_j*3+_k] = tmp;
    }

The quadratic fit coefficients can then be obtained via:

var wall_thickness_fit = num::LinSolve (coefficient_matrix, wall_thickness);

The following script function uses the quadratic fit coefficients to calculate the interpolated or extrapolated value:

function QuadraticFunction (p1, p2, p3, coeffs[])
{
  double ret;
  for (int _i=0; _i<3; _i++)
    for (int _j=0; _j<3; _j++)
      for (int _k=0; _k<3; _k++)
      {
        double tmp = 1.0;
        int ctr;
        for (ctr=0; ctr<_i; ctr++) tmp *= p1;
        for (ctr=0; ctr<_j; ctr++) tmp *= p2;
        for (ctr=0; ctr<_k; ctr++) tmp *= p3;
        ret += coeffs[_i*9+_j*3+_k]*tmp;
      }
  return ret;
}
4. Predicting Parameter Values

Using the quadratic fit, we can attempt to predict the parameter values for which a certain condition occurs. As an example, we will attempt to find the bottom mask position for which the two channels merge. Only the bottom mask position will be evaluated, the other two parameters will be assumed to be at their central values.

In order to find this value, a root finding algorithm script function is used:

```javascript
var zero_wall_thickness_bottom_pos = find (j in [-3,3] root QuadraticFunction (0, j, 0, wall_thickness_fit)) {};
```

This script statement searches for the zero of QuadraticFunction (0, j, 0, wall_thickness_fit) for j between -3 and 3. The normalized value that is predicted by this is -2.09, which corresponds to a location of 1.255 micrometers.

Of course, the quadratic fit is imperfect, especially this far outside of the original range. If the full flow is calculated for this value, the following picture is generated:

As can be seen, the wall thickness between the two channels is not exactly zero, but close; the calculations show a thickness of about 15 nm.
5. Monte-Carlo Analysis

Using the fit calculated above and the standard deviations of the parameters, we will use a Monte-Carlo analysis to examine what fraction of samples will pass a yield criterion. The yield criterion that we will use is chosen rather arbitrarily: The wall thickness between the channels has to be greater than 200 nm.

A Monte-Carlo analysis operates by picking a large number of parameter sets at random, using a certain probability distribution function, performing a simulation (or using a fit of the simulation data) for all these sets, and determining how many of them pass the yield criterion. In this application note, we assume a normal distribution of the parameters.

Since the random number generator in FlowDesigner is only capable of generating random numbers with a constant probability function, these random numbers have to be converted to a normal distribution. This can be done by generating a random number between -1 and 1, and computing the inverse error function (erf⁻¹) of this number. Unfortunately, the inverse error function is not available, so we use a root finding algorithm to find it. The script to generate a random number with a normal probability distribution with a standard deviation of 1 is:

```plaintext
function get_random_normal ()
{
    double rnd = random(2)-1.0;
    return find (x in [-100,100] root erf(x)-rnd){};
}
```

So, to perform a Monte-Carlo analysis with 1000 data points, the following script is used:

```plaintext
int num_OK = 0;
int num_MC_samples = 1000;

for (i=0; i<num_MC_samples; i++)
{
    double _i = get_random_normal();
    double _j = get_random_normal();
    double _k = get_random_normal();
    double val = QuadraticFunction(_i, _j, _k, wall_thickness_fit);
    if (val > 0.2) num_OK++;
}
printf("Percentage of samples with wall thickness greater than 0.2 = ",num_OK*100.0/num_MC_samples,\".\n");
```

Each run of this script will yield a different result, since the random number generator is differently initialized each time. The yield that should come out is around 97%.