VISTA Status Report December 1998

R. Klima, V. Palankovski, M. Radi, R. Strasser, S. Selberherr

Institute for Microelectronics Technical University Vienna Gusshausstrasse 27-29 A-1040 Vienna, Austria

This work was partly supported by the special research project SFB 011 AURORA

Contents

1 TCAD Optimization with SIESTA

The SIESTA simulation environment offers powerful features for TCAD optimization problems. For the first, it offers modeling mechanisms that enable an engineer to describe the simulation problem in a clearly structured and encapsulated manner through so called *TCAD Models* (see Fig. 1). Secondly SIESTA's integrated optimizer allows for global optimization with bound constraints, and minimization of vector quantities, which can be used for calibration and inverse modeling tasks, respectively.

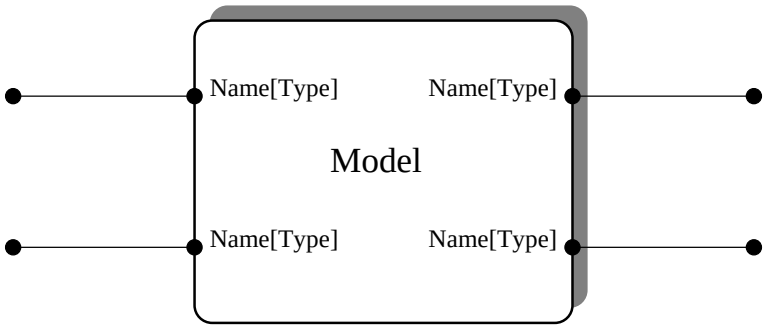

Figure 1: A model transforms its data at input ports into data at the output ports

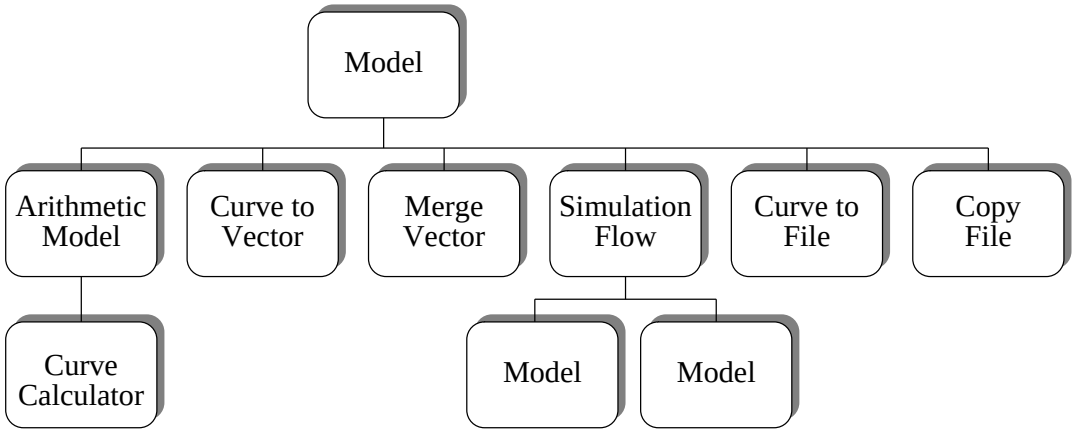

Figure 2: The hierarchy of models available in SIESTA

Several kinds of TCAD models exist for various tasks. Fig. 2 shows the models that are more or less sufficient to describe state of the art simulation problems. Aside from auxiliary models, the Process Model, the Device Model, and the Network Model are the core components that are used to encapsulate simulation tasks.

1.1 Process Simulation Abstractions

The Process Model offers a highly flexible interface to control sequences of arbitrary simulation tools (see Fig. 3) that are driven by some sort of input deck or controlled through their command line. Additionally, it allows for the control of dedicated parts of these input decks through the *ports* of the model's (see Fig. 5).

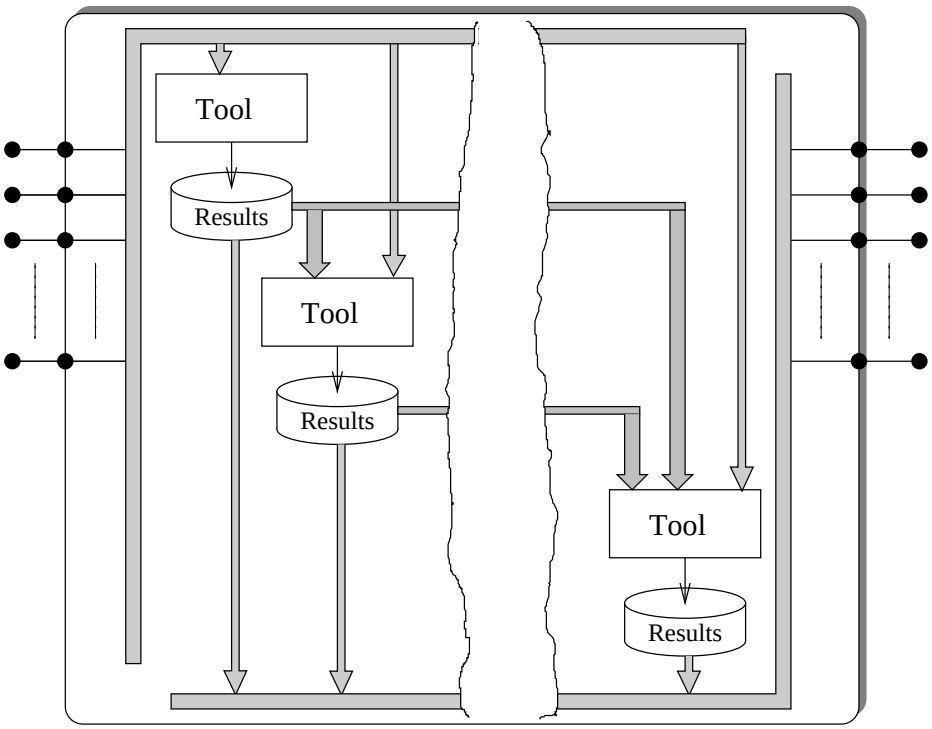

Figure 3: The Process Model encapsulates a sequence of simulation tools.

1.2 Device Simulation Abstractions

Similar to the Process Model, the Device Model is used to encapsulate the device simulation task. Therefore, a device simulator (MMNT, MEDICI, DESSIS) is controlled by the Device Model. All necessary control parameters of the device simulator (terminal potentials, various simulator settings) are available through the ports of the Device Model and so it serves as an abstract device simulator which hides as many details as possible behind its model ports.

1.3 Modeling Networks of Simulation Tools

The Network Model combines several models in order to create a new model. Fig. 4 gives a schematic view of a Network Model encapsulating two models.

The Network Model offers features, that keep even extensive simulation tasks maintainable to a TCAD engineer. Due to its flexible structure, it enables a wide range of simulation applications, since it implies virtually no restrictions to a users creativity. The Network Model serves as a perfect basis for sophisticated TCAD applications like optimization, calibration, or statistical analysis.

1.4 Model Reuse

By means of the Network Model already existing models can be utilized to perform different investigations based on the same subject. That means that a library of models can be com-

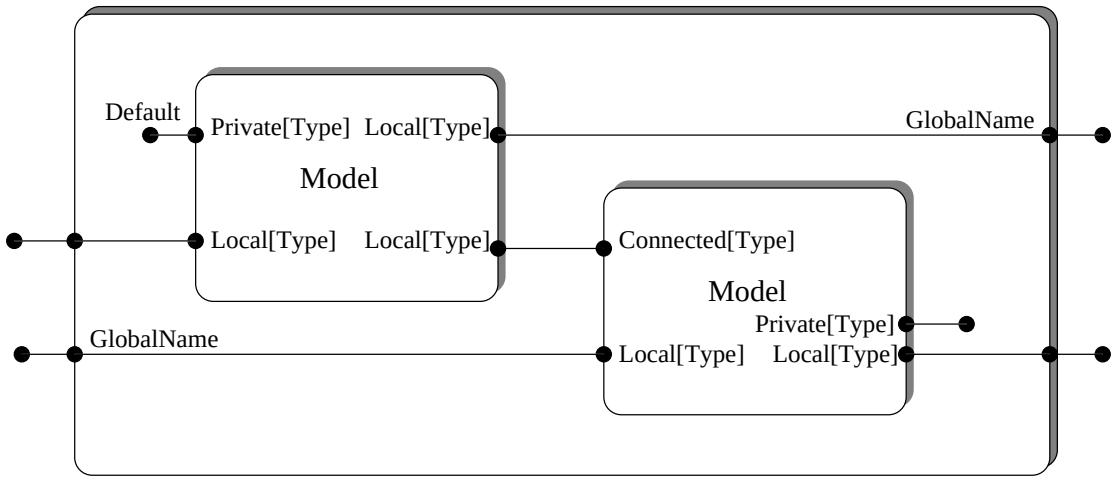

Figure 4: The Network Model combines several models to create a new one. Its models are able to connect their input ports to input ports of the Network Model, to output ports of other models, or they can leave it unconnected and assign it a default value. Output ports of models can be defined as output ports of the Network Model, or they can remain unconnected.

piled and used later-on for various tasks in multiple projects. For example one could imagine a Process Model producing a CMOS device. Multiple instances of this Process Model might be necessary for the evaluation of a certain technology under various conditions. It would be an overwhelming effort for a designer if he had to maintain separate Process Models for all these technology snapshots. Instead, model reuse together with *default overloading* of models reduces this effort dramatically.

1.5 Default Overloading

Since many TCAD applications require several similar variants of a Process Model, the possibility to overload a port's default setting is desirable. Let us assume that the gate length of a device that is produced by a Process Model is available via a model port of this Process Model. This Process Model could then be used multiply in a Network Model, as depicted in Fig. 4, using different values for the gate length by means of default overloading. As a result, the model is only defined once but there exist several similar instances of it in the simulation. Given that an investigation requires more than one instance of this Process Model, all of these models share common properties except the gate length. Therefore, if a change of the Process Model has to be done, this change has only to be done once at the definition of the Process Model. As a rule of thumb, typical TCAD investigations might require up to ten snapshots of a process (including several wafer areas under various process conditions). This illustrates how useful model reuse in conjunction with parameter overloading is, and the drastic decrease of administrative efforts, respectively, that are necessary to maintain simulation descriptions.

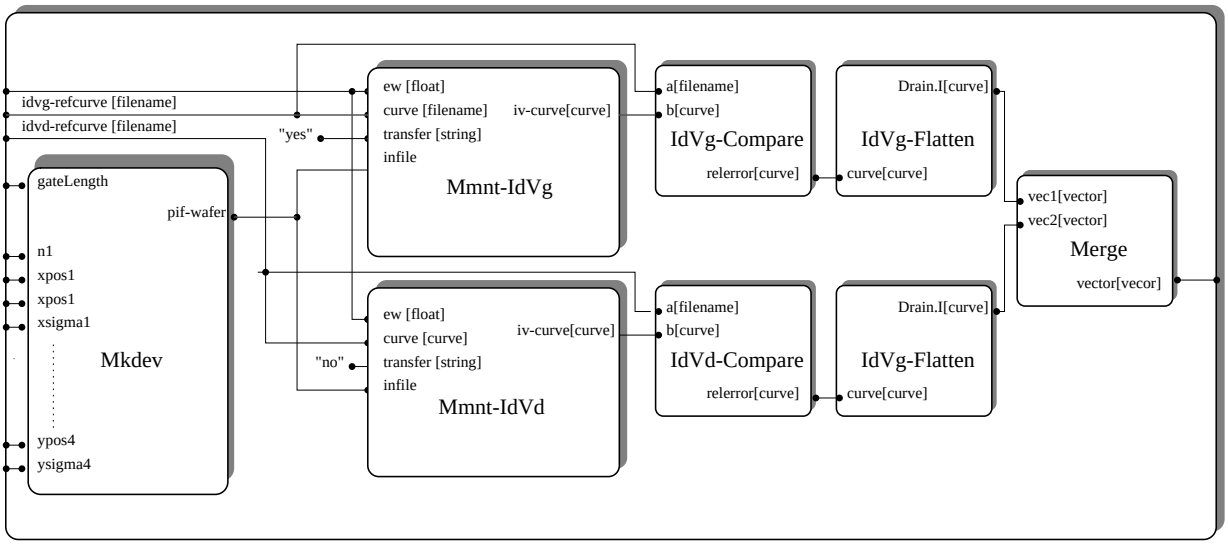

Figure 5: This model (`subnet.mod`) generates a synthetic device of a specific gate length and compares the electrical characteristics of this device with corresponding measurements.

1.6 An Inverse Modeling Example

The features described above are demonstrated by an inverse modeling application as described in the following. A Process Model (`subnet.mod`) is used to generate a synthetic semiconductor device. A Device Model (`device.mod`) performs an electrical characterization (transfer characteristics, output characteristics) of this device and afterwards these characteristics are compared to measurements, which is done in a Network Model as depicted in Fig. 4 and Fig. 5. A network model creates three instances of `subnet.mod` in order to perform the described procedure for different values of the gate-length of the synthetic device (see Fig. 6). Here the default overloading is used to create devices with gate-lengths of 0.18μm, 0.48μm, and 9.88μm and compare their electrical characteristics with measurements for these device geometries.

1.6.1 The Optimizer

Fig. 7 shows the optimization that is performed based on the models described above. The optimizer is searching for parameters of the synthetic model which deliver an optimum fit between simulated electrical device characteristics and the measurements. Since SIESTA offers a powerful job farming mechanism which performs dynamic load balancing, these experiments are computing extremely fast. The whole system does as much work in parallel as possible and therefore the required simulation time for such an experiment is kept at acceptable values. Fig. 8 shows SIESTA's queueing system that allows a user to track running and queued system jobs, and the status of the computation hosts.

```
(network-model
  ;;~~~~~~~~~~~~~~~~~~~
  (submodel dev-018 "subnet.mod"
    ;;##########
    (inputs (private (gateLength 0.18)
     (idvd-refcurve    "meas/018um/Id-Vd.crv")
     (idvg-refcurve    "meas/018um/Id-Vg.crv")
     (idvd-archive     "simu/018um/Id-Vd.crv")
     (idvg-archive     "simu/018um/Id-Vg.crv")
     (wafer-archive    "simu/018um/device.pbf"))))

  ;;~~~~~~~~~~~~~~~~~~~
  (submodel dev-048 "subnet.mod"
      ;;##########
      (inputs (private
     (gateLength     0.48)
     (idvd-refcurve  "meas/048um/Id-Vd.crv")
     (idvg-refcurve  "meas/048um/Id-Vg.crv")
     (idvd-archive   "simu/048um/Id-Vd.crv")
     (idvg-archive   "simu/048um/Id-Vg.crv")
     (wafer-archive  "simu/048um/device.pbf"))))


  ;;~~~~~~~~~~~~~~~~~~~
  (submodel dev-988 "subnet.mod"
    ;;##########
    (inputs
    (private  (gateLength 9.88)
      (idvd-refcurve "meas/988um/Id-Vd.crv")
      (idvg-refcurve "meas/988um/Id-Vg.crv")
      (idvd-archive  "simu/988um/Id-Vd.crv")
      (idvg-archive  "simu/988um/Id-Vg.crv")
      (wafer-archive "simu/988um/device.pbf"))))


  ;;~~~~~~~~~~~~~~~~~~~
  (submodel merge1 (merge-vector-model
           (inputs     (vec1 vector)
              (vec2 vector)
              (vec3 vector)))
  ;;##########
  (inputs (connect (vec1 dev-018 error)
   (vec2 dev-048 error)
   (vec3 dev-988 error)))
  ;;##########
  (outputs (public (vector error))))
)
```

Figure 6: The outer Network Model for inverse modeling

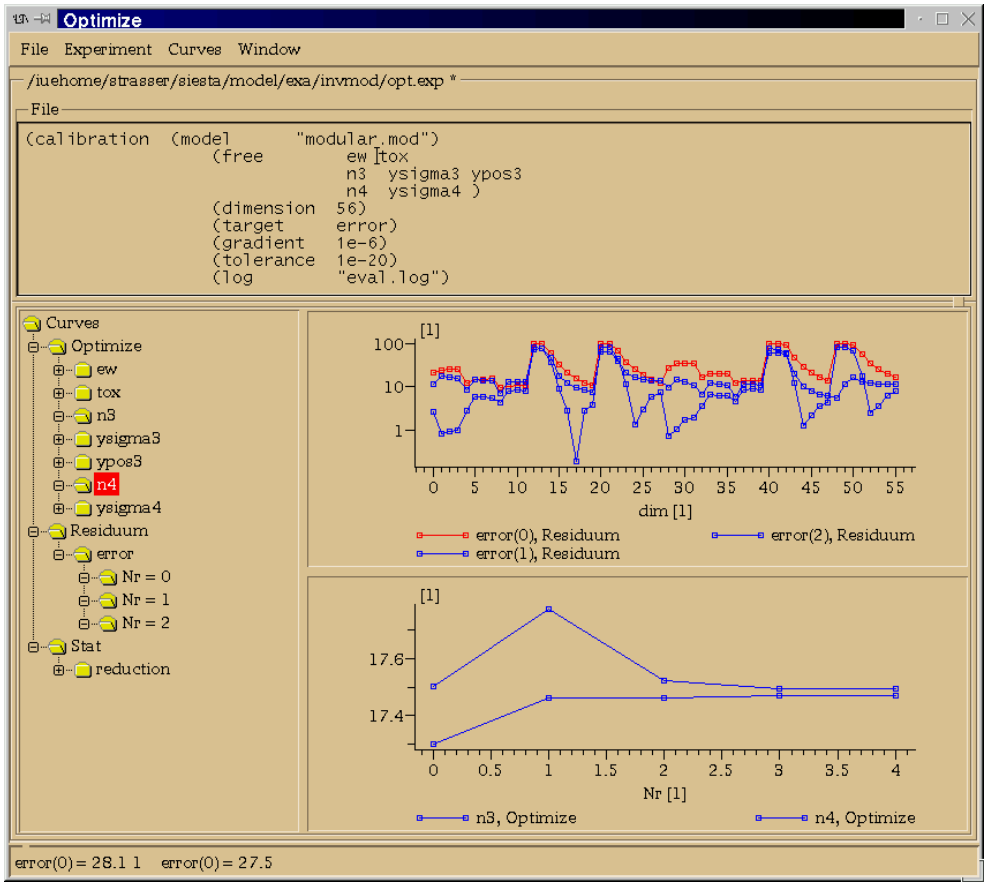

Figure 7: SIESTA's integrated optimizer allows a user to track the optimization progress in a comprehensive way.

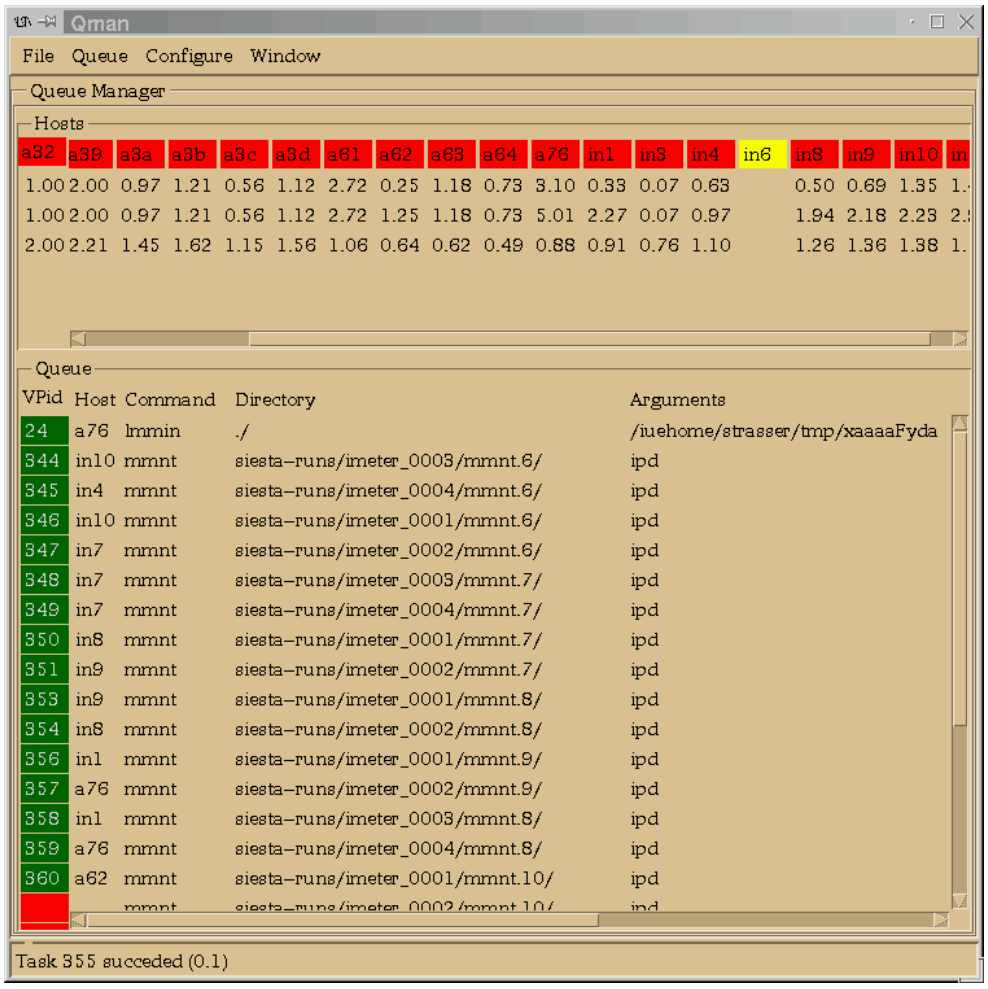

Figure 8: The Queue Manager shows the status of SIESTA's queueing system.

2 Input Deck Programming Language

2.1 Introduction

A growing complex, general purpose device and circuit simulator like MINIMOS-NT demands a highly developed controlling interface. Default values, control parameters and their mathematical dependencies must be set and decisions must be given. To keep the simulator compact and maintainable it is necessary to supply a separate new module, the input deck, which eases the burden of the simulator and administers all user settings.

The simulator can be controlled with so-called *keywords*. In MINIMOS-NT hundreds of keywords are introduced to control all possible events. All keywords are grouped in so-called *sections* and are allowed to be time-dependent functions, tables, expressions, and the like. To accomplish this, the input deck is handled as a database which is only queried when the desired keyword is required. This way, the simulator stores internal values which should be externally accessible in a special keyword of a specific section. These external variables (e.g., iteration counter, different update norms) can then be used to form keyword expressions in the input deck. When the simulator inquires a special keyword from the input deck, its current value is returned.

The entire input deck functionality has been implemented as a library which can be customized by adding user-defined functions. These external functions can be used like any other function for any keyword. Due to this powerful feature, the functionality of the simulator increases without changing one line of the code.

Another case where these features are utilized is the control of block iteration. Especially, hydrodynamic simulations require such a great flexibility because convergence is usually hard to achieve. Each iteration block is defined by a separate section in the input deck file. For each block several keywords such as the list of quantities to be solved, the damping scheme to use, parameters for this damping scheme, and a stop expression are supplied to control the iteration. These sections can be arbitrarily nested, forming complex sequences of iteration blocks.

All data are stored in ASCII text files following a well-defined syntax described below. The input deck programming language (IPL) is a programming language of its own which has been designed with simulator control in mind. The syntax of IPL is similar to that of the C programming language. Several typical features of programming languages are available, e.g., `if()` statements, function definitions, etc.

2.2 Keywords

In the input deck an arbitrary number of variables can be defined. Moreover, variables can be grouped in sections as shown in the next chapter. A variable may be defined only once. All variable names are case sensitive.

Variables which are inquired by the simulator are called *keywords*. Most of the variables specified in an input deck file are such *keywords*.

Variables can be classified by their different kind of access by MINIMOS-NT: Variables which can be read or written, variables which can only be read and variables which are hidden to

the simulator. The type of a variable must be explicitly specified before the name. If no type is given when defining a variable, MINIMOS-NT is only permitted to read it, e.g.,

```
// This is as simple comment
ext x = 32;                     // variable the simulator may read and write
ext y = 4;                      // variable the simulator may read and write
aux a = sqrt(x * x + y * y);    // auxiliary variable, hidden to the simulator
ex    = x / a;                  // read-only variable
ey    = y / a;                  // read-only variable
```

No data-type declaration of variables is necessary. Since the expressions stored in a variable will be evaluated at runtime the data-type will be determined during calculation.

Data-types available for variables are shown in Table 1.

Type	Example
`Boolean`	`a = true;`
`Integer`	`a = 3;`
`Real`	`a = 3.1415;`
`Complex`	`a = 4.3 + 3.1 j;`
`Quantity`	`a = 3.1415 m;`
	`a = 3.1415 "m/s";`
`String`	`a = "This is a string";`
`Array`	`a = [ 1, 2, 3 ];`
	`a = [ 1, "pi", 3 A];`

Table 1: IPL data-types

Several common operators are defined on these data-types. The precedence of all operators is shown in Table 2. All operators within the same box have the same precedence. Operators in higher boxes have higher precedence than operators in lower ones. For instance the expression `a + b * c` means `a + (b * c)` rather than `(a + b) * c`.

The results of the given rules are expressions denoted as *`expr`* where an expression *`expr`* again may contain other expressions described by these rules.

Unary operators are right-associative, binary operators are left-associative. For instance the expression `a - b + c` means `(a - b) + c` but not `a - (b + c)`.

All operators are not defined for all data-types. Furthermore, the result of an operator depends on the data-type of the operands given. For example the addition operator (+) is defined for `Integers`, `Real`, and `Complex` numbers, `Quantities`, `Strings` and `Arrays` as well but not for `Boolean` values.

When using variables the input deck combines various functionalities of a database and a calculation tool. Variables may be set and queried by MINIMOS-NT where expressions are evaluated at runtime. When defining variables, expressions can be assigned by equations. Such expressions will be simplified to increase the performance of calculation. The input deck distinguishes between constant and non-constant expressions where the latter contains references to other variables. Each time a value of a variable is inquired by the simulator, the input deck recalculates its value using its stored expression, but only if recalculation is necessary.

Operator	Rule
array subscripting	*name* [*expr*]
function call	*name* (*expr_sequence*)
not	! *expr*
unary minus	- *expr*
unary plus	+ *expr*
multiply	*expr* * *expr*
divide	*expr* / *expr*
modulo (remainder)	*expr* % *expr*
add	*expr* + *expr*
subtract	*expr* - *expr*
less than	*expr* < *expr*
less than or equal	*expr* <= *expr*
greater than	*expr* > *expr*
greater than or equal	*expr* >= *expr*
equal	*expr* == *expr*
not equal	*expr* != *expr*
logical AND	*expr* && *expr*
logical inclusive OR	*expr* || *expr*

Table 2: Operator precedence

The advantage of this proceeding is, that when a value of a variable changes, all values of influenced variables (which refer to it) would have to change too, which may, again, cause many others referring to them to change. To keep calculation times very short, recalculations take only place when the variable is inquired.

2.3 Sections

To manage large input deck files it is necessary to introduce clearly arranged structures, and to support classification of keywords. Therefore, the IPL makes it possible to group variables into so-called sections. Each section has a name which is handled case sensitive. The section body is enclosed in braces ({}), e.g.,

```
global_var = 7;
MySection
{
   decision     = no;
   value1       = 2.3 + ~global_var;
   text         = "short Text";
   value2       = value1 * 2;
   MySubSection
   {
      value3    = 4.5;
      array     = [1, 2, 4, 8, 16];
   }
}
```

Sections can contain variables and subsections, thus sections can be nested to an arbitrary depth. To prevent from splitting up a single section definition across several blocks reopening of sections defined once is forbidden.

The absolute name of a variable which is its real name consists of its location (the *path* which may consist of an arbitrary number of section names separated by a dot) starting at the *root section* and its defined name. The *root section* is denoted by the tilde symbol (~). Variables defined outside of any section are global variables. For instance the absolute name of the variable `value1` in the section `MySection` is `~MySection.value1` whereas the absolute name of the global variable `global_var` is `~global_var`.

An input deck file may consist of several sections. In order to refer to a variable which is defined within the same section only its name must be given. To refer to a variable located in another section the *path* of the respective variable must be specified. This *path* can be given relatively or absolutely. If the variable is defined outside of the current section, the circumflex (^) can be used to step up one section, on the other hand the absolute name can be explicitly specified by means of the tilde symbol (~) which denotes the *root section*.

The IPL uses an inheritance mechanism similar to object-oriented programming languages. In this connection, the equivalent to object classes are IPL-sections. A structure of a section can be passed on to other sections. This can be done using the inheritance operator (`:`) followed by the name of the section (the base-section) which may be given by a relative or absolute name, e.g.,

```
A
{
   x = 3;
   y = x + 2;
}
B : A;
```

First a section named A is defined. Subsequently the section B is inherited from section A to contain all elements (variables and all subsections) of section A.

A section may have several parent sections. In this way, to define a multiply inherited section, a list of (relative or absolute) names of the base-sections must be given in decreasing order of priority. If an element with the same name occurs in several base-sections its first occurrence will be used, e.g.,

```
C1
{
   a = 1;
   c = 17;
   X
   {
      m = 100;
      n = 101;
   }
}

C2
{
   b = 2;
   c = 18;
   X
   {
      o = 200;
      p = 201;
   }
}

D : C1, C2;
```

The section D now consists of the three variables a, b, and c inherited from C1 and C2 respectively, where the variable c is inherited from section C1. The subsection ~D.X contains the two variables m and n. Note that the variables o and p are not elements of this subsection.

Inherited sections may be locally modified. This can be done by specifying a section body for the inherited section, e.g.,

```
A
{
   x = 3;
   y = x + 2;
}
B : A
{
   x = 4;
}
```

The variables ~B.y and ~A.y store the same formula containing a relative addressed variable reference x. Evaluating both variables will deliver ~A.y=3+2 and ~B.y=4+2.

When using inherited sections, only those elements can be locally modified that are passed on to this section. By default an inherited section is protected from appending new entries. This allows to detect wrong typed keywords in inherited sections and prevents from creating new ones.

This behavior can be explicitly disabled for a whole section by enclosing the section-name in angle brackets or by explicitly preceding the new variable name with a plus (+).

2.4 Functions

The use of functions can clearly enlarge the field of application. Functions can be used in expressions which are stored in variables. In the input deck, the standard mathematical functions and functions for conversion of values are implemented. Additional functions can easily be defined by the user. Parameters are enclosed in parentheses and can be complex expressions as well as simple constants. Parameters must be given in a defined order, separated by commas, and possibly have to be of a specific data-type. Some functions might have optional parameters which must be given with their parameter-names and must be the last parameters in the list.

```
a1 = func1();                  // Calling function func1(). No Parameter.
a2 = func2(1 + 2);             // Calling function func2(). 1  Parameter.
a3 = func3(10, optPara = true); // Calling function func3(). 2  Parameters.
                               // The second parameter is optional.
```

Furthermore application specific functions may be added to the input deck by the simulator itself (for instance the step() function defined by MINIMOS-NT). Those functions can be used in combination with this application only.

2.4.1 Built-in Functions

The standard mathematical functions are implemented which are defined for `Complex` numbers in general. Many-valued mathematical functions (for instance $asin$) are implemented too (for instance `asin()`) which return their principal value. All available functions are listed in Table 3. The types of the arguments are written within the parentheses. The first box shows the mathematical functions defined, the second shows all available cast-functions, and the third shows additional functions supported which are useful for operations on `Complex` or `Quantity` numbers respectively.

A simple example calculates the diagonal of a cube with its volume given:

```
v = 27 "m*m*m";
a = cbrt(v);                    // -> 3 m
r = sqrt(3 * squ(a));           // -> 5.2 m
```

An example for cast-functions is to cast the value of an environment variable, which is usually given as a `String`, to a `Real` number, e.g.,

```
a   = "3.56";
b   = real(a);                  // -> 3.56
```

The next example shows how the additional functions shown in the last box of Table 3 can be used:

```
a  = (3.1 + 4.2 j) * 1 A;
// Remove the unit:
b  = value(a);                  // -> 3.1 + 4.2 j
// Get the real part of a quantity:
c  = realpart(a);               // -> 3.1 A
// Get the imaginary part of a complex number:
d  = imagpart(b);               // -> 4.2
// Get the unit:
f  = unit(a);                   // -> "A"
```

Another built-in function is the `if()` statement which expects three arguments. The first argument is a `Boolean` value. The second and the third one can be of any data-type. If the first argument is equal to logical `true`, the second argument will be evaluated and its result will be returned. If not, the third argument will be evaluated and its result will be returned. An example calculating the minimum value of two variables is given below:

```
a   = 2;
b   = 4;
min = if (a < b, a, b);
```

To extract the absolute name of a specified variable the function `fullname()` can be used, e.g.,

```
a     = 1;
aName = fullname(a);            // -> "~a"
E
```

```
{
   b     = 2;
   bName = fullname(b);         // -> "~E.b"
   ESub
   {
      c     = 3;
      cName = fullname(c);      // -> "~E.ESub.c"
   }
}
```

Function	Description
`abs(Complex Quantity)`	absolute value
`arg(Complex Quantity)`	argument value
`sign(Real Quantity)`	sign
`sigma(Real Quantity)`	step function
`sin(Complex)`	sine
`cos(Complex)`	cosine
`tan(Complex)`	tangent
`cot(Complex)`	cotangent
`asin(Complex)`	inverse sine
`acos(Complex)`	inverse cosine
`atan(Complex)`	inverse tangent
`acot(Complex)`	inverse cotangent
`sinh(Complex)`	hyperbolic sine
`cosh(Complex)`	hyperbolic cosine
`tanh(Complex)`	hyperbolic tangent
`coth(Complex)`	hyperbolic cotangent
`asinh(Complex)`	inverse hyperbolic sine
`acosh(Complex)`	inverse hyperbolic cosine
`atanh(Complex)`	inverse hyperbolic tangent
`acoth(Complex)`	inverse hyperbolic cotangent
`exp(Complex)`	exponential function
`pow(Complex, Complex)`	first argument to the power of the second argument
`pow2(Complex)`	2 to the power of the given argument
`pow10(Complex)`	10 to the power of the given argument
`log(Complex)`	natural logarithm
`log2(Complex)`	logarithm to the base 2
`log10(Complex)`	common logarithm (to the base 10)
`squ(Complex Quantity)`	square
`sqrt(Complex Quantity)`	square root
`cub(Complex Quantity)`	power of three
`cbrt(Complex Quantity)`	cubic root
`boolean(Complex)`	cast to `Boolean`
`integer(Real)`	cast to `Integer`
`real(Integer)`	cast to `Real`
`string(Any value)`	cast to `String`
`value(Complex Quantity)`	value without unit
`realpart(Complex Quantity)`	real part of a `Complex` number
`imagpart(Complex Quantity)`	imaginary part of a `Complex` number
`unit(Complex Quantity)`	unit of a `Quantity` as a string

Table 3: Built-in functions

Environment variables can be queried using the function `getenv()`. The data type of the value returned is `String`, e.g.,

```
home = getenv("HOME");
  // -> "/home/user9"
path = getenv("HOME") + "/work/defaults.ipd";
  // -> "/home/user9/work/defaults.ipd"
```

2.4.2 User-Defined Functions

With the IPL new functions can be defined very simply and flexibly. So the functionality of the programming language can be easily enlarged and customized.

Functions can be defined similar to variables. The function name is followed by an argument list enclosed in parentheses, e.g.,

```
add(x,y) = x + y;
sub(x,y) = x - y;
inc(x)   = x + 1;
dec(x)   = x - 1;
```

In this example four functions named `add()`, `sub()`, `inc()`, and `dec()` are defined. Parameters like `x` and `y` are not input deck variables – they are parameters, valid within the function definition only. Defining a variable named `x` anywhere in the input deck file would not influence the behavior of a function using a parameter `x`.

User-defined functions can be used in the same way as built-in functions, for example:

```
a = 11;
b = 23;
sum = add(a,b);
sum = inc(sum);
```

The value of the variable `sum` will evaluate to 35.

2.5 Files

The default input deck files are part of the MINIMOS-NT distribution. These files contain default values and should never be changed.

With the `#include` command the input deck can be told to insert a file at the position the command is called. This command allows including of files like its counterpart in the C programming language. The file name must be enclosed in double quotes, if a path is specified, or in angle brackets, if the default paths of MINIMOS-NT should be used. After including the default files the keywords can be changed by locally modifying them.

The default extension of input deck files is ipd which may be omitted. File names used may contain environment variables too. Some examples:

```
#include <defaults>             // Include the file "defaults.ipd".
#include "~/work/mydefs"        // Include the file "mydefs.ipd".
file1 = "$HOME/work/myfile1";   // variable containing a file name
file2 = "~/work/myfile2";       // variable containing a file name
```

2.6 Special Features

2.6.1 Tracing of Variables

Sometimes it might be useful to know when a variable is set by the simulator or if a specific variable is used for calculation or not. To be able to observe access on specific variables, the `trace` statement can be used. The `trace` statement expect a list of variables to be traced, e.g.,

```
a = 1;
b = 2;
c = 3;
trace a, c;
```

`trace` reports each time a variable is

- used while calculation,
- overwritten by the simulator,
- deleted by the simulator.

2.6.2 eval()

The input deck can be told to evaluate an expression immediately while parsing. This can be done using the `eval()` function, e.g.,

```
ext a = 10;
ext b = 2;
c     = (a * b);
d     = eval(c);               // evaluated to 20
```

In this example the variables `a` and `b` are declared as extern and can, therefore, be changed by the simulator. Both variables are initialized. The variable `c` contains a formula calculating the product of the variables `a` and `b`. Variable `d` immediately calculates the result of variable `c` and, from now on, stores a constant value. In contrast to variable `c` it does no longer depend on any other variable. Each time variable `a` or `b` is changed by the simulator variable `c` will change automatically while variable `d` remains unchanged and keeps its initialization value calculated once while parsing.

2.7 Constants

The input deck supplies the following constants:

Constant	Value	Description
`j`	$\sqrt{-1}$	indicates the imaginary unit
`e`	2.71828182845904523536	the Euler constant `e`
`pi`	3.14159265358979323846	π

Table 4: Predefined constants

3 The Physical Models in MINIMOS-NT

A lot was done recently for our two-dimensional device simulator MINIMOS-NT in order to have it ready for its first release. In order to enable simulation of devices with high complexity and specifity in respect to materials, geometries, etc. many of the existing physical models had to be refined, some of them were replaced by finer ones, many new models were added. This will be the topic of the following section, more details can fe found in the documentation of the simulator.

3.1 Sets of Partial Differential Equations

In MINIMOS-NT carrier transport can be treated by the drift-diffusion (DD) and the hydrodynamic (HD) transport models. For either carrier type the transport model can be chosen independently, or transport can even be neglected by assuming a constant quasi-Fermi level for one carrier type. In addition, the lattice temperature can be treated either as a constant or as an unknown governed by the lattice heat flow equation.

3.2 Carrier Mobility

MINIMOS-NT provides mobility models for various materials. Their particular transport properties lend themselves to divide the materials into three main groups: IV group semiconductors, III-V compound semiconductors and their alloys, and non-ideal dielectrics. To each group a separate section is devoted in the following.

3.2.1 Silicon and Other Basic Materials

In MINIMOS-NT, the well-tried mobility model of MINIMOS 6 [1] is implemented. In addition, we have a new model, which distinguishes between the majority and minority electrons in Si, as well as between dopant species [2] (see Fig. 9). The expression

$$\mu_n^{\mathrm{LI}} = \frac{\mu_n^{\mathrm{L}} - \mu_1 - \mu_2}{1 + \left(\frac{C_I}{C_1}\right)^{\alpha}} + \frac{\mu_1}{1 + \left(\frac{C_I}{C_2}\right)^{\beta}} + \mu_2 \tag{1}$$

can be used to express the mobility in any semiconductor of interest, GaAs for instance (see Fig. 10).

3.2.2 Alloys

In the case of III-V semiconductor alloys or SiGe we use a model which employs the low-field mobilities of the basic materials (A and B) and combines them by a harmonic mean.

$$\frac{1}{\mu^{\mathrm{AB}}} = \frac{1-x}{\mu^{\mathrm{A}}} + \frac{x}{\mu^{\mathrm{B}}} + \frac{(1-x)\cdot x}{C} \tag{2}$$

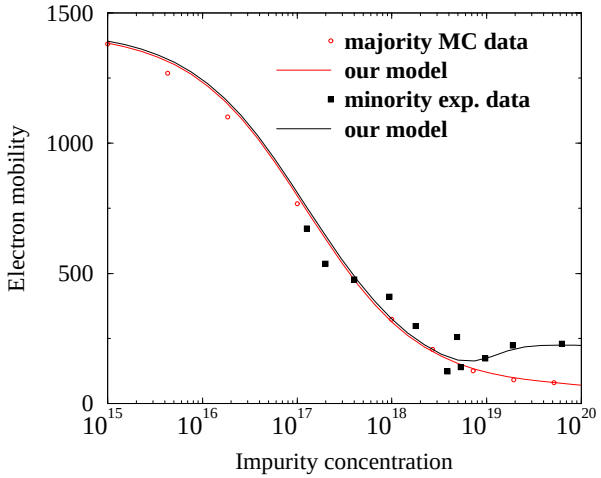

Figure 9: Electron mobility in Si

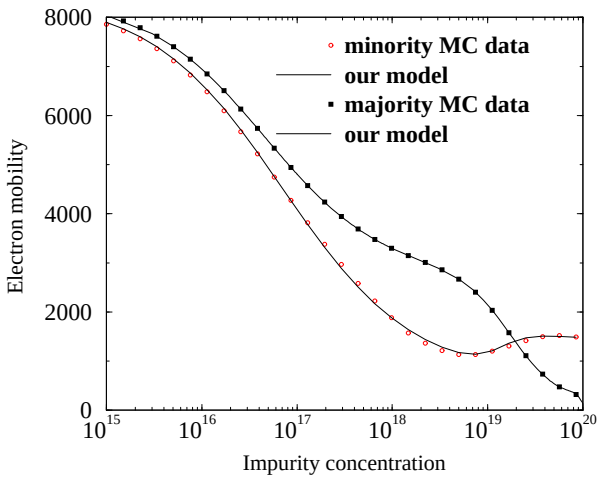

Figure 10: Electron mobility in GaAs

C is referred to as nonlinear or bowing parameter. For the high-field mobility the model suggests a quadratic interpolation between the saturation velocities of the basic materials (A and B).

$$v_{\nu,300}^{\mathrm{AB}} = v_{\nu,300}^{\mathrm{A}} \cdot (1-x) + v_{\nu,300}^{\mathrm{B}} \cdot x + C \cdot (1-x) \cdot x \tag{3}$$

3.3 Energy Relaxation Time

The energy relaxation times are used in the HD mobility models, in the energy balance equations of the hydrodynamic transport model, and in the lattice heat flow equation. For elementary and binary semiconductors the energy relaxation time for electrons is calculated by

$$\tau_{\epsilon,n} = \tau_{\epsilon,0} + \tau_{\epsilon,1} \cdot \exp\left(C_1 \cdot \left(\frac{T_n}{300\ \mathrm{K}} + C_0 \right)^2 + C_2 \cdot \left(\frac{T_n}{300\ \mathrm{K}} + C_0 \right) + C_3 \cdot \left(\frac{T_L}{300\ \mathrm{K}} \right) \right) \tag{4}$$

where T_L is the lattice temperature and T_n is the carrier temperature. This model is applicable to Si, Ge, GaAs, AlAs, InAs, InP, GaP (Fig. 11, Fig. 12).

In case of SiGe, AlGaAs, InAlAs, InGaAs, GaAsP, or InGaP the model for alloys is used. The parameters $\tau_{\epsilon,0}$ and C_0 depend on the material composition x. For an alloy $\mathrm{A}_{1-x}\mathrm{B}_x$ the they are calculated by

$$\tau_{\epsilon,0}^{\mathrm{AB}} = \tau_{\epsilon,0}^{\mathrm{A}} \cdot (1-x) + \tau_{\epsilon,0}^{\mathrm{B}} \cdot x + \tau_C \cdot (1-x) \cdot x \tag{5}$$

$$C_0^{\mathrm{AB}} = C_0^{\mathrm{A}} \cdot (1-x) + C_0^{\mathrm{B}} \cdot x + C \cdot (1-x) \cdot x \tag{6}$$

τ_C and C are referred to as nonlinear or bowing parameters (Fig. 13, Fig. 14).

3.4 Bandgap Energy

The semiconductor bandgap is well-studied. The new solution of interest here is the bandgap alignment. An energy offset Eoffs is used to align the band edge energies of

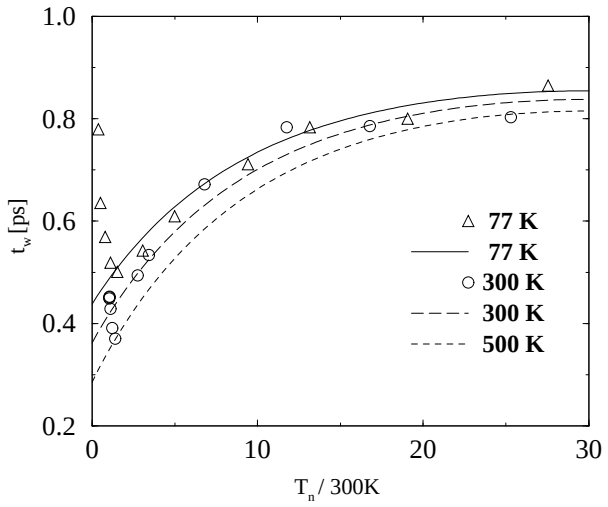

Figure 11: Energy relaxation time in Si

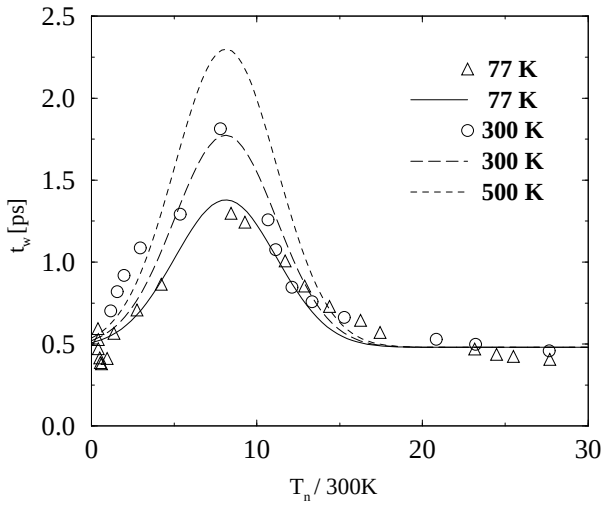

Figure 12: Energy relaxation time in GaAs

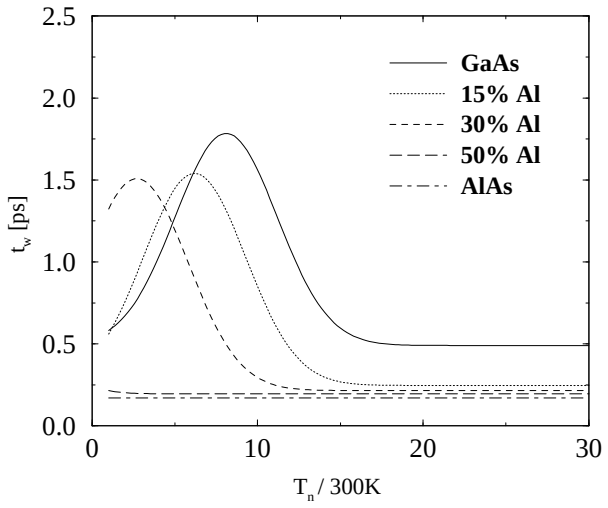

Figure 13: Energy relaxation time in AlGaAs

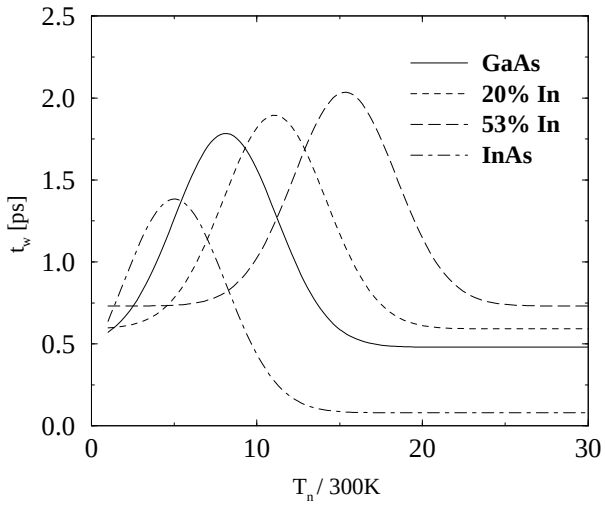

Figure 14: Energy relaxation time in InGaAs

different materials. In the case of semiconductor alloy it is expressed as

$$E_{\text{off}} = \frac{E_{\text{off}}^{\text{A}} \cdot \left(E_{\text{g}}^{\text{AB}} - E_{\text{g}}^{\text{B}}\right) - E_{\text{off}}^{\text{B}} \cdot \left(E_{\text{g}}^{\text{AB}} - E_{\text{g}}^{\text{A}}\right)}{E_{\text{g}}^{\text{A}} - E_{\text{g}}^{\text{B}}} \tag{7}$$

3.5 Bandgap Narrowing

We use a band gap narrowing model which delivers better results in comparison with other models (see Fig. 15) and is valid for various semiconductors, also alloys (see Fig. 16).

$$\Delta E_{\text{g}} = -\frac{\text{q}^2 \cdot \beta}{4 \cdot \pi \cdot \varepsilon_0 \cdot \varepsilon_{\text{r}}} \tag{8}$$

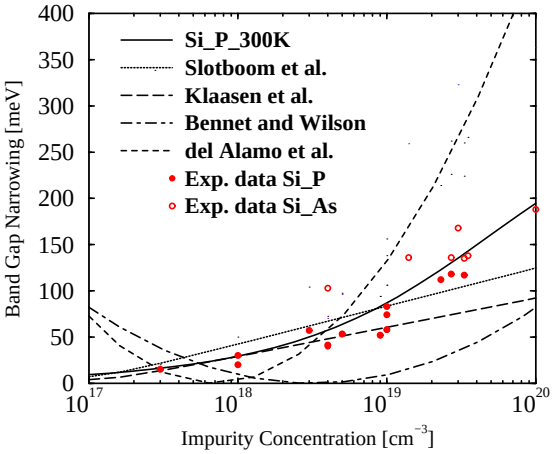

Figure 15: Models for BGN in n-Si

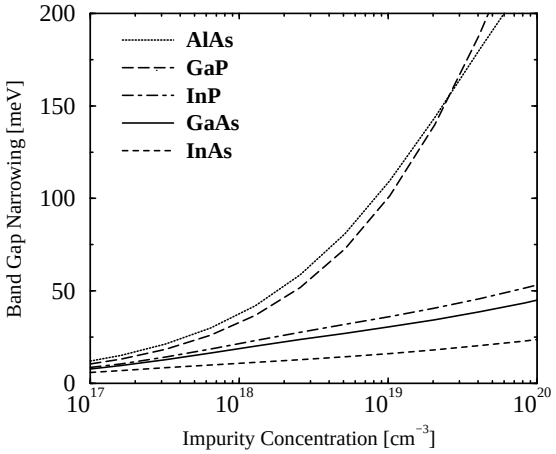

Figure 16: BGN in III-V materials

3.6 Generation and Recombination

Several models are available in MINIMOS-NT in order to account for processes like SRH, Auger, and direct recombination, band-to-band tunelling, impact ionization, etc.

3.7 Lattice Heat Flow Models

To account for self heating effects, the lattice heat flow equation has to be solved. For different materials, MINIMOS-NT uses proper models for the mass density, specific heat, and thermal conductivity.

3.7.1 Mass Density Models

The mass density in the case of SiGe and ternary III-V compounds is expressed by a linear change between the values of the basic materials.

$$\rho^{\mathrm{AB}} \quad = \quad (1-x)\cdot\rho^{\mathrm{A}} + x\cdot\rho^{\mathrm{B}} \tag{9}$$

3.7.2 Heat Flux Models

In this subsection the expressions representing the MINIMOS-NT thermal heat flux models for pure and compound materials are summarized. The model calculates the lattice thermal flux density between two boxes and its derivatives to the input quantities, which are the temperatures T_1 and T_2. The thermal conductivity equation is given by

$$\kappa(T_L) \quad = \quad \kappa_{300}\cdot\left(\frac{T_L}{300\ \mathrm{K}}\right)^{\alpha}, \tag{10}$$

where κ_{300} is the exact value for the thermal conductivity at 300 K. The thermal heat flux is expressed as

$$\int_{T_1}^{T_2} \kappa(T_L)\, dT_L = \frac{\kappa_{300}}{\alpha+1} \cdot \left(\left(\frac{T_2}{300\ \mathrm{K}} \right)^{\alpha+1} - \left(\frac{T_1}{300\ \mathrm{K}} \right)^{\alpha+1} \right). \tag{11}$$

The model is used for the basic materials Si, Ge, GaAs, AlAs, InAs, InP, or GaP. The thermal conductivity for SiGe, AlGaAs, InAlAs, InGaAs, GaAsP, or InGaP is varied between the values of the basic materials (A and B).

$$\kappa_{300}^{\mathrm{AB}} = \frac{1}{\left(\frac{1-x}{\kappa_{300}^{\mathrm{A}}} + \frac{x}{\kappa_{300}^{\mathrm{B}}} + \frac{(1-x)\cdot x}{C} \right)} \tag{12}$$

$$\alpha^{\mathrm{AB}} = (1-x)\cdot \alpha^{\mathrm{A}} + x \cdot \alpha^{\mathrm{B}} \tag{13}$$

The thermal heat flux is again expressed by (11).

3.7.3 Specific Heat Capacity Models

The model evaluates the specific heat capacity for the transient simulation with self-heating. The input parameters are the temperatures $T_{L,n}$ and $T_{L,n-1}$. The parameter n denotes the corresponding time step. The function also requires the coefficients for the specific heat capacity of the considered material.

$$\overline{c}_L(\overline{T}) = c_0 + c_1 \cdot \frac{\left(\frac{\overline{T}}{300\ \mathrm{K}} \right)^{\alpha} - 1}{\left(\frac{\overline{T}}{300\ \mathrm{K}} \right)^{\alpha} + \frac{c_1}{c_0}} \tag{14}$$

$$\overline{T} = \frac{T_{L,n} + T_{L,n-1}}{2} \tag{15}$$

C_0 is the exact value for the specific heat capacity at 300 K. The model is used for the basic materials Si, Ge, GaAs, AlAs, InAs, InP, or GaP. The specific heat capacity coefficients in the case of SiGe and ternary III-V compounds are expressed by a linear change between the values of the basic materials (A and B). The specific heat capacity is then expressed by (14).

4 AMIGOS : Analytical Model Interface & General Object Oriented Solver

AMIGOS is a problem independent simulation system which can handle a wide range of nonlinear partial differential equation systems in time and space in either one, two or three dimension(s). It is designed to automatically generate optimized numerical models from a simple mathematical input language, so that no significant speed loss in comparison to 'hand coded' standard simulation tools occurs.

In difference to similar algorithms based on the so called 'operator on demand' concept [3], AMIGOS is completely independent of the kind of discretization since the model developer can formulate any discretization of choice. There are no restrictions whether to use scalar, field or tensor quantities within a model, and, if desired, any derived field quantity can be calculated, too. Furthermore, the user can influence the numerical behavior of the differential equation system by complete control of the residual vector and its derivative (e.g. punishing terms, damping terms, etc.). Even interpolation and grid-adaptation formulations can be used within a developed model and can thus be very well fitted to a special problem.

AMIGOS is equipped with three layers of access to serve the needs of the variety of users (Fig. 17).

- *user-mode:* users, who are just interested in simulating a special process step (e.g. oxidation, diffusion, etc.), can select a model appropriate for the necessary calculation and can modify several process parameters (e.g. duration, temperature, material characteristics, etc.) as well as the boundary and grid definitions using the Input & Control Interface.
- *model developer-mode:* users, who are interested in developing new models or extending existing ones with new physical characteristics, have access to all models via the developed Analytical Model Interface (AMI). They may modify existing equations by simply adding parameters, mathematical terms or equations or even develop a complete new model.
- *platform developer:* users, interested in developing new features for AMIGOS itself concerning e.g. special damping paradigms, using other solver libraries, time integration algorithms or just supporting a wider range of input grid formats than the tool does at the moment, need not concern with physical modeling.

In contrast to previous generations of software only the *platform developer* requires access to and modification of the source code. Even during model development the analytical user input will be interpreted, optimized, transformed and solved on any complex simulation domain at once without the necessity of time consuming recompilations (*one-pass concept*) supporting a variety of several testing and debugging features. After finishing the test and calibration phase the user can switch to the *two-pass* concept where all modifications are translated to C-code and are linked to a model library. This allows for high performance calculations on large simulation domains in the standard user-mode.

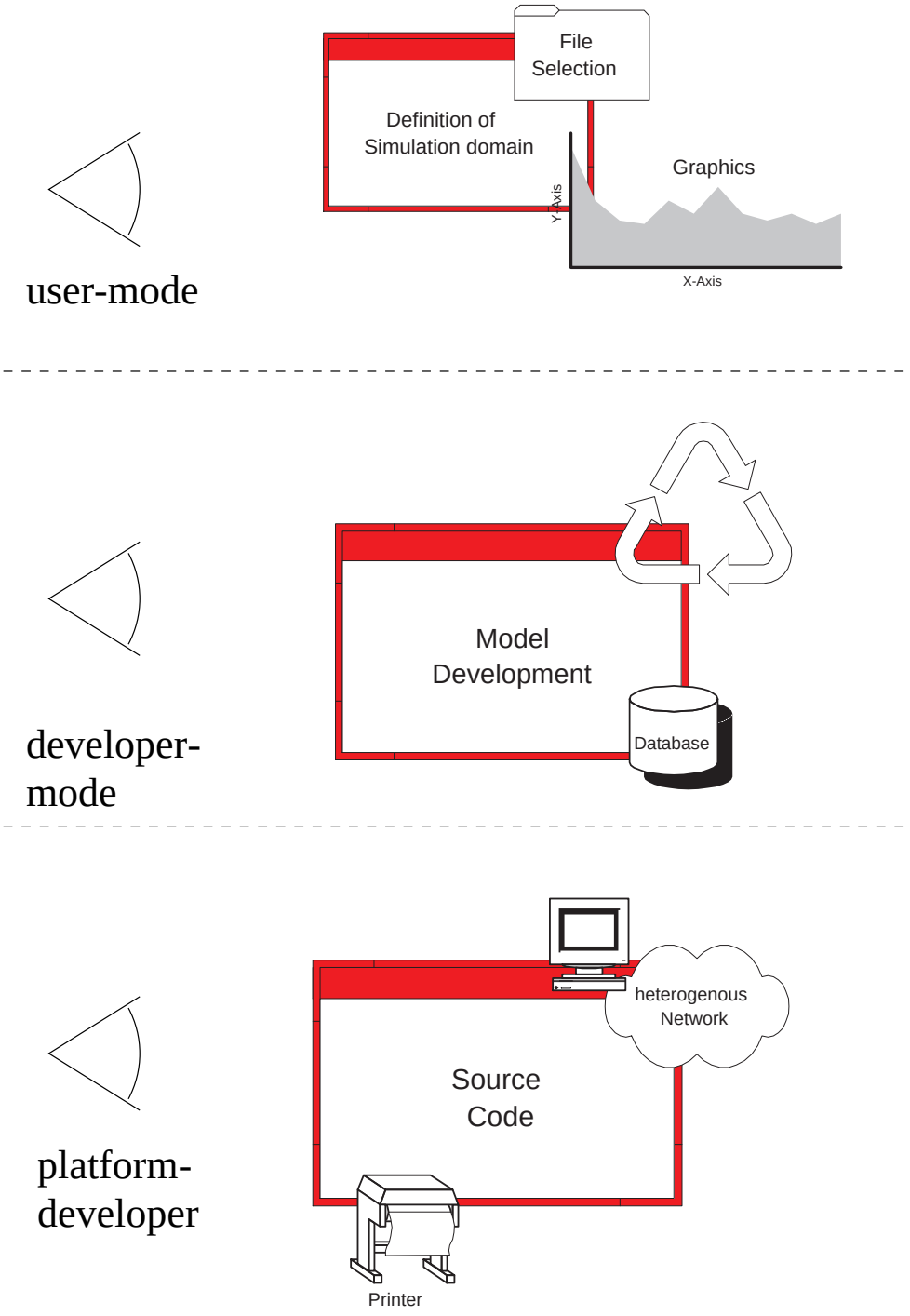

Figure 17: Several different user perspectives

4.1 Example 1: A Pair-Diffusion Model in One Dimension

A five species phosphorus diffusion model introduced by Richardson and Mulvaney [4] has been developed. The species behavior can be formulated by the following system of coupled reaction diffusion equations

$$\frac{\partial V}{\partial t} = D_V \nabla^2 V - k^E_{for} P V + k^E_{rev} E - k_{bi} (V I - V^{eq} I^{eq}) \tag{16}$$

$$\frac{\partial I}{\partial t} = D_I \nabla^2 I - k^E_{for} P I + k^E_{rev} F - k_{bi} (V I - V^{eq} I^{eq}) \tag{17}$$

$$\frac{\partial P}{\partial t} = -k^E_{for} P V + k^E_{rev} E - k^F_{for} P I + k^F_{rev} F \tag{18}$$

$$\frac{\partial E}{\partial t} = D_E \nabla^2 E + k^E_{for} P V - k^E_{rev} E \tag{19}$$

$$\frac{\partial F}{\partial t} = D_F \nabla^2 F + k^F_{for} P I - k^E_{rev} F \tag{20}$$

where the solution vector $u = [V, I, P, E, F]^T$ represents, respectively, the concentration of vacancies, interstitials, substitutional phosphorus as well as phosphorus–vacancy pairs (E–centers) and phosphorus–interstitial pairs (F–centers).

Furthermore, zero flux boundary conditions for all species are enforced everywhere except at the exposed surfaces where V, I, P, E, and F are specified with

$$\begin{aligned} V &= V^{eq} \text{ and} I = I^{eq} \\ P &= C^* \text{ (ambient gas concentration)} \\ E &= \frac{k^E_{for}}{k^E_{rev}} P V \\ F &= \frac{k^F_{for}}{k^F_{rev}} P I \end{aligned}$$

Using a finite element discretization within the analytical model interface the complete formulation of the discretized differential equation system is written in the code segment as shown below. The predefined variables like X, Y, Z (coordinates of the element points) and t (time) are initialized during runtime and can be used like any other variable within the model definition language.

```
MODEL PairDiff = [V,I,E,F,P];
{

############## Discretization ###################################

N(xsi) = [1-xsi,xsi];        # the element shape function defined as
                             # a function of xsi
dNdxsi = [ -1  , 1 ];        # derivative of the shape function

dxdxsi = dNdxsi * X;         # X is a predefined vector [X1,X2,...,Xn]
                             # getting real size and coordinates
                             # during runtime

detJ = |dxdxsi|;
dxsidx = Inv(dxdxsi);        # calculates the inverse matrix

L = dNdxsi*dxsidx;           # gradient operator
K = L^T*L;                   # Laplace operator where (^T) is a
                             # transpose operator
C = N(1/2)*N(1/2)^T;         # time operator using lumping
                             # N(1/2) -> calculates the value
                             #           of the shape function
                             #           for xsi=0.5


############## The Parameters ###################################

Param k_for_e = 1.0E-14;     # in cm^3/s
Param k_for_f = 1.0E-14;     # initializing default values
Param k_bi = 1.0E-10;        # for parameters

Param k_rev_e = 10;          # in 1/s
Param k_rev_f = 12;

Param Dv  = 1.0E-10;         # in cm^2/s
Param Di  = 1.0E-09;
Param De  = 1.0E-13;
Param Df  = 2.0E-13;

Param Veq  = 1.0E14;         # in 1/cm^3
Param Ieq = 1.0E14;


############## The Differential Equations ####################
```

```
dt = t.t0-t.t1;              # access to several time steps
dV = V.t0-V.t1;              # of unknown and predefined
dI = I.t0-I.t1;              # variables
dE = E.t0-E.t1;
dF = F.t0-F.t1;
dP = P.t0-P.t1;

i = 1..2;                    # running variable from 1 to 2

PV[i]  = P[i] * V[i];
PI[i]  = P[i] * I[i];
VI[i]  = I[i] * V[i];
VIeq[i] = Veq * Ieq;

resV = detJ * (Dv*K*V + C * (dV/dt-k_for_e*PV+k_rev_e*E-k_bi*(VI-VIeq)));
resI = detJ * (Di*K*I + C * (dI/dt-k_for_f*PI+k_rev_f*F-k_bi*(VI-VIeq)));
resE = detJ * (De*K*E + C * (dE/dt+k_for_e*PV-k_rev_e*E));
resF = detJ * (Df*K*F + C * (dF/dt+k_for_f*PI-k_rev_f*F));
resP = detJ * C * (dP/dt-k_for_e*PV+k_rev_e*E-k_for_f*PI+k_rev_f*F);


############## The Residual and its Derivative ######################


residuum = [[resV][resI][resE][resF][resP]];     # the residual vector
jacobian = D([[V][I][E][F][P]],residuumˆT)ˆT;     # auto derivative
                                                 # of residual
}
```

To define the necessary simulation parameters and to map the developed analytical pair-diffusion model onto a definite simulation domain including its boundaries the specifications of the Input & Control Interface have to look like:

```
# Where to read the grid from
Source=Pif
{
    Physical = lin2.pbf    # the input filename
    Logical  = BareWafer   # the logical name of the mesh
}
# Where to write the grid and the result to
Output = Pif
{
    Physical = lin2res.pbf # the output filename
    Logical  = BareWafer   # the logical name of the new generated
                           # output mesh
}
```

```
Time = 60                  # the first 60 seconds
{
   Step    = 1.0E-3        # initial timestep
   Epsilon = 1.0E-3        # maximum error
   RejectionRatio = 0.5    # reduce step-size in case of non
                           # converging or epsilon error
   StretchRatio   = 1.5    # increase step-size in case of success
   MaxNewton      = 15     # stop after 15 iterations and reduce step
                           # size
}



Grid = Si_Region  # the name of the grid in the default input file
{
    Model = PairDiff(V,I,E,F,P)   # use the model PairDiff
    {                             # and map the quantities
        Value(V) = 1.0E14         # initialize the quantity V
        Value(I) = 1.0E14         # initialize the quantity I
        Value(E) = 0.0E0          # initialize the quantity E
        Value(F) = 0.0E0          # initialize the quantity F
        Value(P) = 0.0E0          # initialize the quantity P
    }
}

Boundary = Backside         # any name because auto boundary detection
                            # is used
{
   Interface(Si_Region,Top) # Choose the top boundary of the grid with
                            # name Si_Region

   # Use Dirichlet boundaries for the defined quantities

   Model = Dirichlet(V){Value(V) = 1.0E14}
   Model = Dirichlet(I){Value(I) = 1.0E14}
   Model = Dirichlet(P){Value(P) = 5.0E20}
   Model = Dirichlet(E){Value(E) = 5.0E19}
   Model = Dirichlet(F){Value(F) = 4.16666E19}
}
```

Finally the simulation results that show the distribution of all five species after 60 seconds (Fig. 18) are written to the file defined in the output section. Fig. 19 shows the result for the same model calculating the distribution after 600 seconds.

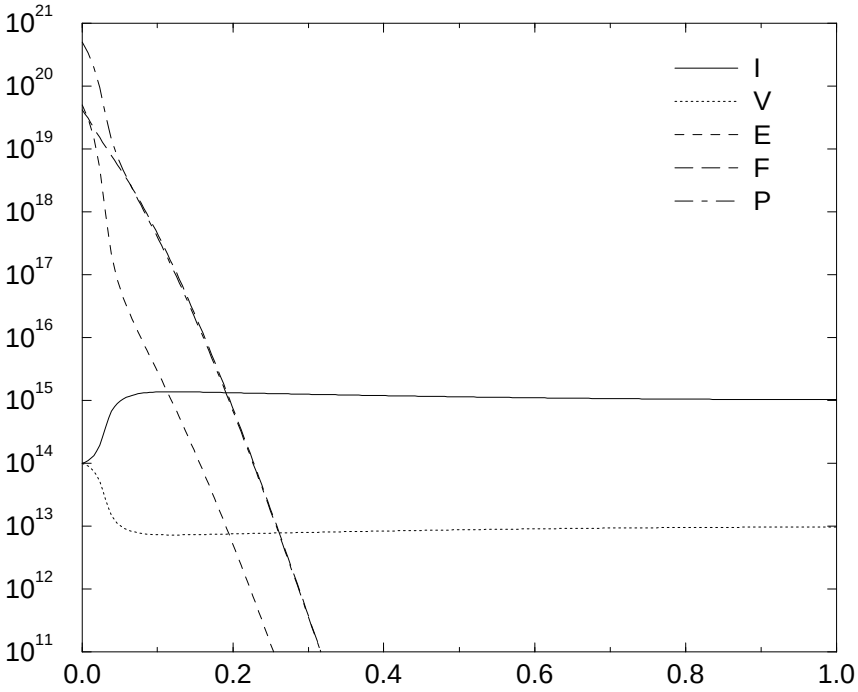

Figure 18: Dopant distribution after 60 seconds

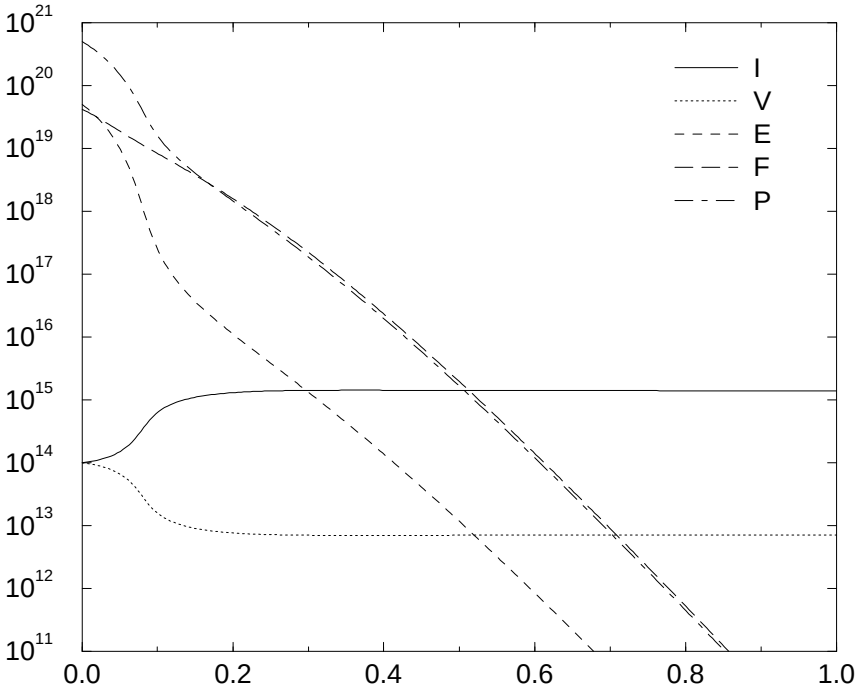

Figure 19: Dopant distribution after 600 seconds

4.2 Example 2: Local Oxidation with a Floating Nitride Mask

Thermal oxidation of silicon is one of the most important steps in fabrication of highly integrated electronic circuits, being mainly used for efficient insulation of adjacent devices. Those parts of silicon which shall not be oxidized are masked by a layer structure of silicon nitride that is not effected by an oxidation. Exposed parts of already existing silicon oxide are penetrated by the oxidant (oxygen diffusion), which finally reacts at the interface of silicon and silicon dioxide to form new dioxide. This chemical reaction consumes silicon, and due to the increased volume of the newly generated dioxide the old dioxide layer is lifted up.

From the mathematical point of view the problem can be described by a coupled system of partial differential equations:

- *diffusion equation* describing the penetration of the oxidant through the existing silicon dioxide
- *chemical reaction* describing the transformation of material due to the storage of oxygen molecules into the silicon layer reacting to silicon dioxide
- *displacement of the oxide layer* usually modeled as an elastic, visco-elastic or viscous flow. Due to the unknown motion of the interface between silicon and silicon dioxide this leads to a free boundary problem.

Using AMIGOS a new model has been developed based on the model of E. Rank [5]. The key idea is the description of the local oxidation as a three component thermodynamic process involving silicon, silicon dioxide and oxidant molecules. This results in a reactive layer of finite width in contrast to the sharp interface between silicon and silicon dioxide in the conventional formulation. The numerical approximation takes advantage of this description in a finite element approach which models silicon, silicon dioxide and the reactive layer together, thus avoiding the necessity to track the interface with element edges. The smooth transition zone is a means to regularize the mathematical free boundary problem and can be selected in a way, that numerically seen, the same results appear as in case of sharp interface formulations. To distinguish between different materials a method similar to the level set method was chosen to keep the transition zone as small as possible (usually the transition zone is limited to a single element).

Fig. 20 shows the three-dimensional geometry where an oxidation process has been calculated. To distinguish between the different materials a *level set function* has been chosen that provides an immediate jump from zero to one at the interface between silicon and silicon dioxide. The deformation of the nitride mask was calculated with an elasto-mechanical model whereas the oxide itself was modeled with a visco-elastic deformation behavior. Due to the enormous size of the model itself (eight coupled quantities) no more than 17376 node points putting up 41421 elements have been used to calculate the oxidation process. Nevertheless, due to the grid adaption algorithm it was possible to get quite acceptable results. The post-processed cuts through the layer shown in Fig. 21 and Fig. 22 depict the familiar oxidation results firstly at the longer side of the mask and secondly at the shorter one.

Fig. 23 shows the same calculation but with a four times thicker nitride mask. The fact that now the mask can hardly be deformed leads to high stresses beneath the nitride mask, and because of the viscous material behavior of oxide the deformation of the formed oxide layer differs in form and size (Fig. 24 and Fig. 25) from the previous examples.

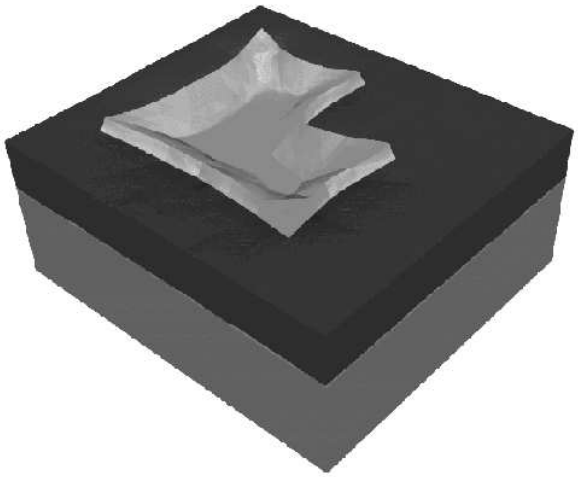

Figure 20: Three-dimensional oxide growth around a thin floating nitride mask

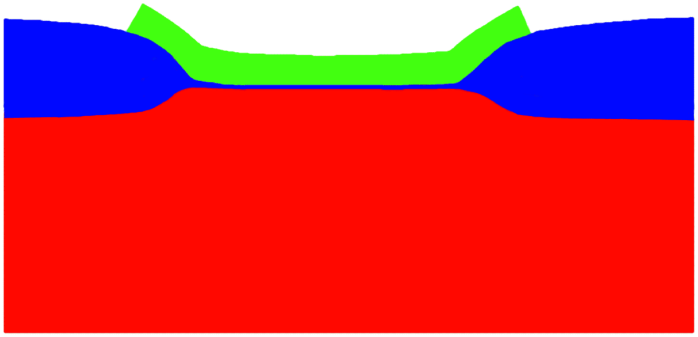

Figure 21: Two-dimensional cut through the simulation result showing the materials deformation along the longer side of the mask

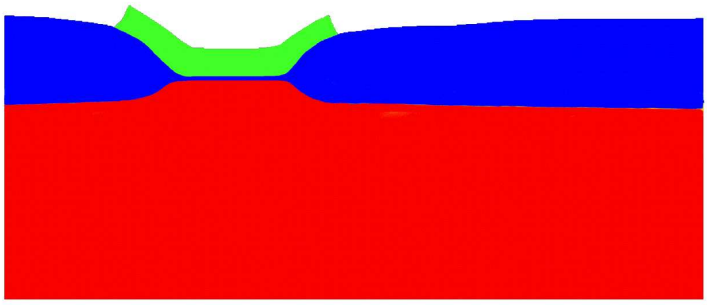

Figure 22: Two-dimensional cut through the simulation result showing the materials deformation along the shorter side of the mask

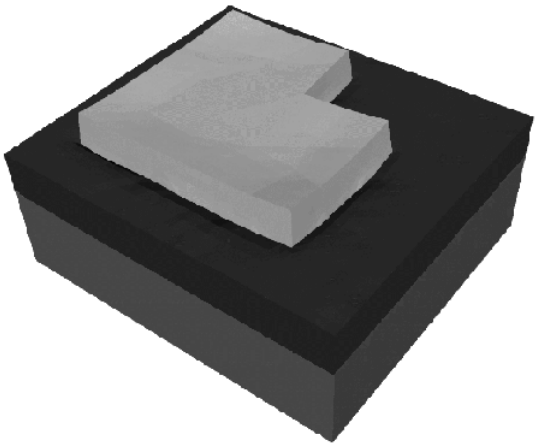

Figure 23: Three-dimensional oxide growth around a fat floating nitride mask

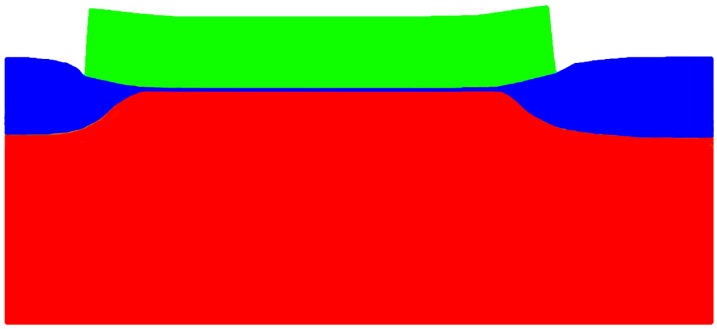

Figure 24: Two-dimensional cut through the simulation result showing the materials deformation along the longer side of the mask

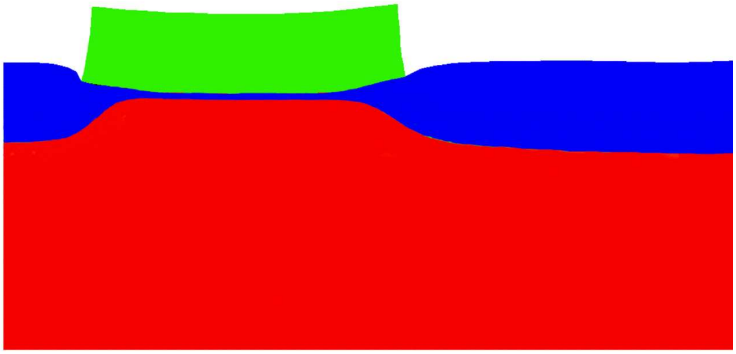

Figure 25: Two-dimensional cut through the simulation result showing the materials deformation along the shorter side of the mask

References

[1] C. Fischer, P. Habaš, O. Heinreichsberger, H. Kosina, Ph. Lindorfer, P. Pichler, H. Pötzl, C. Sala, A. Schütz, S. Selberherr, M. Stiftinger, and M. Thurner. *MINIMOS 6 User's Guide*. Institut für Mikroelektronik, Technische Universität Wien, Austria, 1994.

[2] G. Kaiblinger-Grujin, T. Grasser, and S. Selberherr. A Physically-Based Electron Mobility Model for Silicon Device Simulation. In K. De Meyer and S. Biesemans, editors, *Simulation of Semiconductor Processes and Devices*, pp 312–215. Springer, Leuven, Belgium, 1998.

[3] D.W. Yergeau, E.C. Kan, M.J. Gander, and R.W. Dutton. ALAMODE: A Layered Model Developement Environment. In H. Ryssel and P. Pichler, editors, *Simulation of Semiconductor Devices and Processes*, volume 6, pp 66–69, Wien, 1995. Springer.

[4] W.B. Richardson, G.F. Carey, and B.J. Mulvaney. Modeling Phosphorus Diffusion in Three Dimensions. *IEEE Trans.Computer-Aided Design*, 11(4):487–496, 1992.

[5] E. Rank and U. Weinert. A Simulation System for Diffusive Oxidation of Silicon: A Two-Dimensional Finite Element Approach. *IEEE Trans.Computer-Aided Design*, 9(5):543–550, 1990.