VISTA Status Report
December 2000

T. Binder, M. Gritsch, C. Heitzinger, V. Palankovski, S. Selberherr

Institute for Microelectronics
Technische Universität Wien
Gusshausstrasse 27-29
A-1040 Vienna, Austria

Contents

1 Analysis of HBT Behavior After Strong Electrothermal Stress

1.1 Introduction

The two-dimensional device simulator MINIMOS-NT [1] deals with different complex materials and structures such as binary and ternary alloys with arbitrary material composition profiles. Various physical effects, like band gap narrowing, surface recombination, and self-heating, are taken into account. The efficiency of the models was proven by hydrodynamic DC-simulations with self-heating of forward, reverse and output characteristics of one-finger AlGaAs/GaAs and InGaP/GaAs-HBTs [2], furthermore, by small-signal RF-simulation [3]. Simulation results are in very good agreement with measured data at several ambient temperatures. For reliability reasons of high practical interest, a study on the particular influence of the InGaP ledge on the device performance of InGaP/GaAs-HBTs is presented.

1.2 Impact of the InGaP Ledge

It is well known that GaAs-HBTs with InGaP emitter material can be improved with respect to reliability if the emitter material covers the complete p-doped base layer [4]. Outside the active emitter area remains the so-called InGaP ledge. Using MINIMOS-NT we investigate the impact of the ledge thickness d and negative surface charges, which are known to exist at the ledge/nitride interface, on the device performance. A schematic drawing of the simulated device structure is shown in Fig. 1. Because of symmetry, the simulation domain covers only a half of the real device in order to save computational effort.

1.3 InGaP Ledge Thickness

In Fig. 2 we show the measured and simulated collector and base currents of a one-finger InGaP/GaAs HBTs with different ledge thickness operating under forward Gummel plot conditions with V_{BC} = 0 V. Measurement refers to a device with a 40 nm thick ledge. Note the strong increase in the base current at low bias with increasing ledge thickness. As can be seen from Fig. 2 simulated and measured base currents differ significantly in the case of a 40 nm thick ledge. The reason is that insulator surface Fermi-level pinning is not accounted for if surface charges are not considered in the simulation. Therefore, a non-physical electron current path occurs in the upper ledge part as shown in Fig. 3. The corresponding electron distribution in the ledge using vertical cross-sections at x = 1.6 μm, 2.0μm, and 2.4μm are shown in Fig. 4. The hole distribution in the middle of the ledge (x = 2.0 μm) is also included. These concentrations shall be compared to the ones in the case of surface charges in the next subsection.

1.4 Negative Surface Charges

The influence of fixed negative surface charges which are homogeneously distributed along the interface between ledge and passivation was investigated. As can be seen from Fig. 5, where simulation refers to a device with 40 nm ledge, the base current is reduced if more negative surface charges are introduced. The upper part of the ledge is also depleted [5] and the leakage is reduced (Fig. 6). In Fig. 7 we present the corresponding electron distribution in the ledge at x = 1.6 μm, 2.0μm, and 2.4μm, and the hole distribution at x = 2.0 μm. Note that even in this case the ledge is not completely depleted. However, the electron concentrations near the InGaP/SiN interface are significantly lower in comparison to the ones shown in Fig. 4. Thus, with a surface charge density of 10^{12} cm^{-2} the measured base current can be simulated

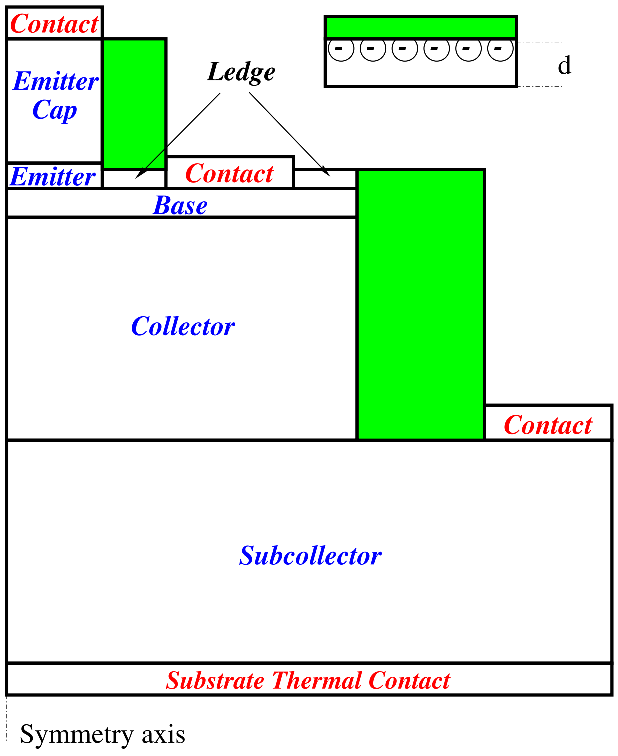


Figure 1: A schematic drawing of the simulated device structure of InGaP/GaAs HBT with an InGaP ledge. Negative surface charges on the ledge/nitride interface exist.

very well. We have to note that in the case of negative surface charges the hole concentration in the ledge increases and at higher values gives the opportunity a hole current path to occur.

1.5 Device Reliability

Based on these investigations it is possible to explain the base current degradation of an InGaP/GaAs HBT which was strongly stressed under conditions far from normal operating conditions. In this case the base current degradation in the middle voltage range can be explained by a decreasing surface charge density along the interface between ledge and passivation from 10^{12} cm^{-2} to 4.10^{11} cm^{-2}. This might be due to compensation of the negative surface charges by H+ ions which are known to be present in the device due to the epitaxial manufacturing processes [6, 7]. In Fig. 8 a comparison of measured and simulated forward Gummel plots at $V_{CB} = 0$ V is shown. Filled and open symbols denote measured characteristics of the non-degraded and degraded device, respectively. The corresponding simulation results are shown with lines. The good agreement also for stressed devices demonstrates the applicability of physics-based device simulation to device reliability issues. In Fig. 9 we present the electron current density corresponding to 4.10^{11} cm^{-2} surface charge density. In Fig. 10 we present the corresponding electron distribution in the ledge at x = 1.6 μm, 2.0μm, and 2.4μm, and the hole distribution at x = 2.0 μm. Note that the upper part of the ledge is now not completely depleted, thus again allowing a base leakage current.

Several other effects supposed to lead to strong increase in the base leakage current, e.g. spreading out of the base contact at the metal/GaAs interface, increased recombination/generation in the InGaP layer, degradation of the SiN/GaAs interface (see e.g. [8], [9] and references therein) were also analyzed. The simulation results show such effects cannot be the dominant reason for beta-degradation. The decrease in the collector current at high level injection is suggested to be due to increased emitter resistance which could occur due to emitter contact detachment, indium segregation in the metal layer, or dislocations at

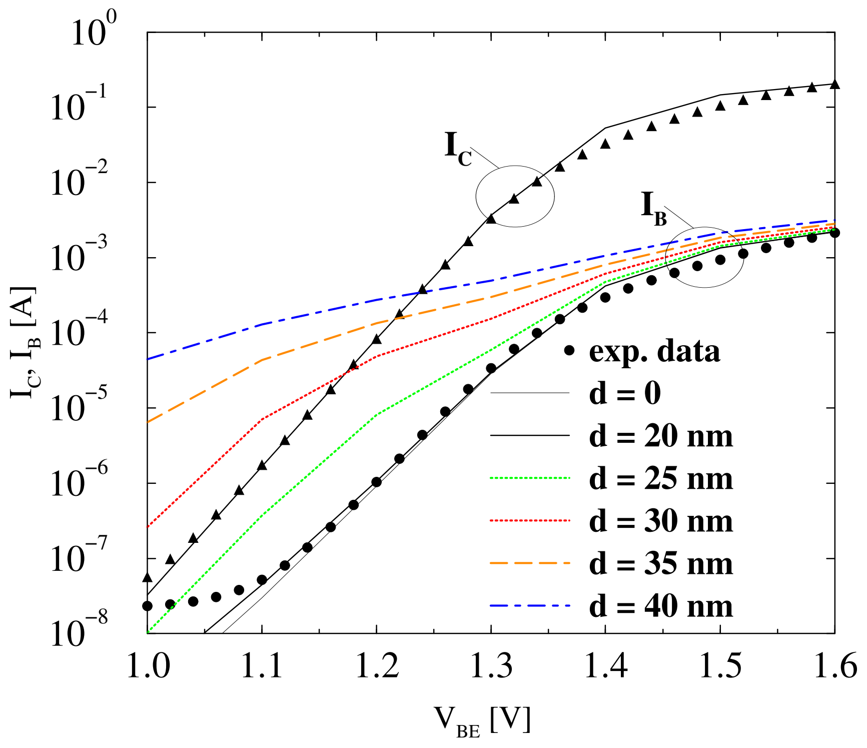


Figure 2: Dependence of I_B on the InGaP ledge thickness compared to measurement.

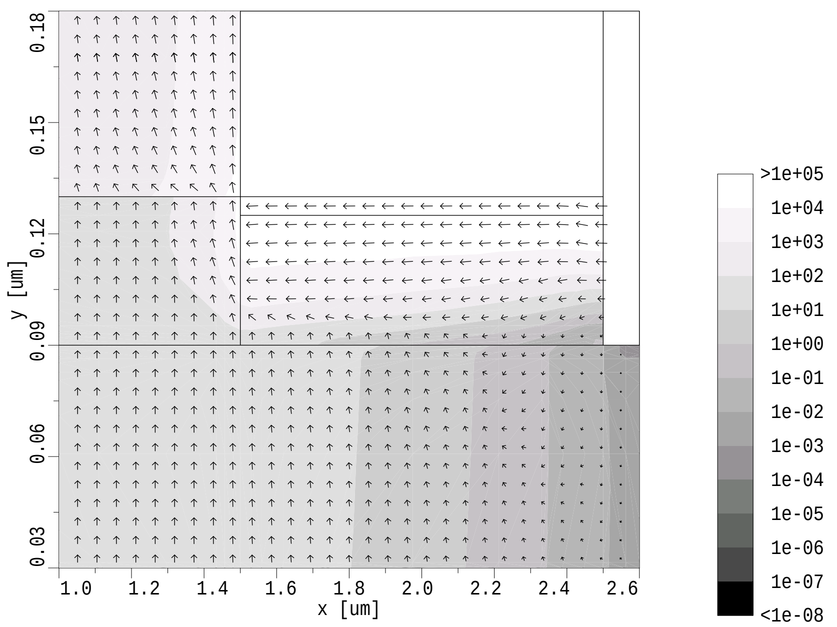


Figure 3: Electron current density [A/cm^2] at V_{BE}=1.2V. Simulation without surface charges.

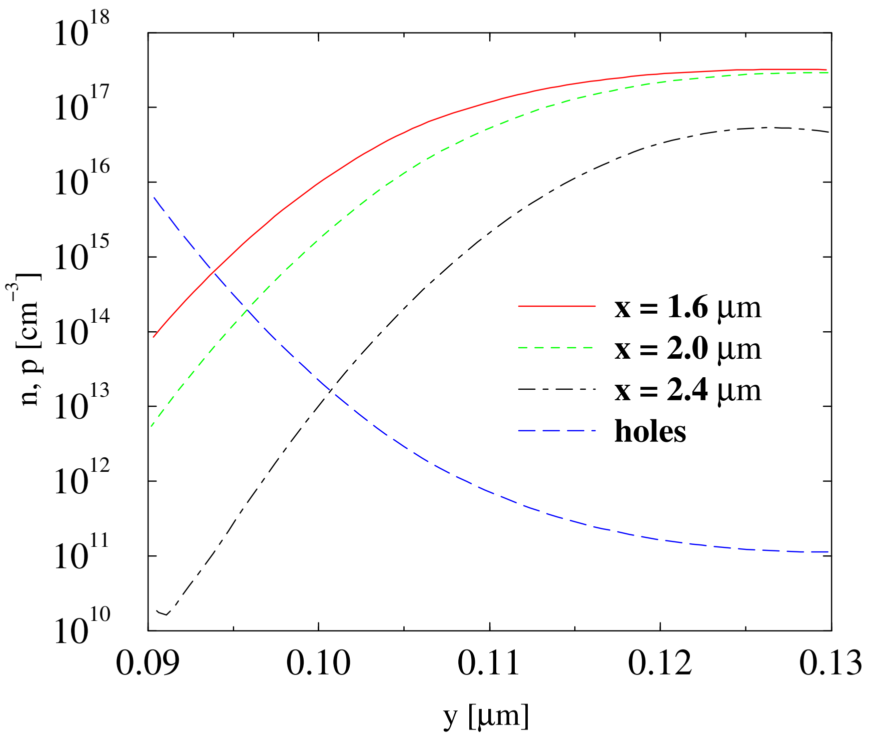


Figure 4: Electron and hole distribution in the ledge. Simulation without surface charges.

the InGaAs/GaAs interface (see e.g. [9]). Our simulations show that in the case of contact detachment there is an electron current crowding in the remaining contact area which leads to insignificant changes. Only a slightly probable emitter contact detachment of more than 80% can explain the measured values (see Fig. 11). We find indium segregation in the metal can be the reason by increasing the emitter contact resistance while the decrease of the indium content in the cap has no significant influence on the emitter resistance.

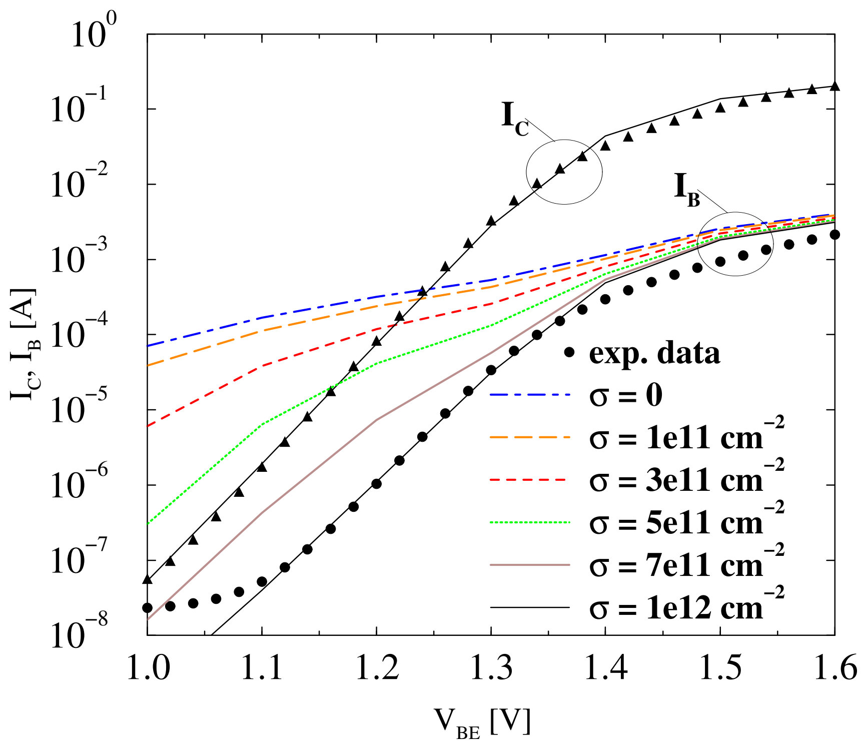


Figure 5: Dependence of I_B on the charge density at the ledge/nitride interface. Charge density of 10^{12} cm^{-2} is sufficient to get agreement with the measurements.

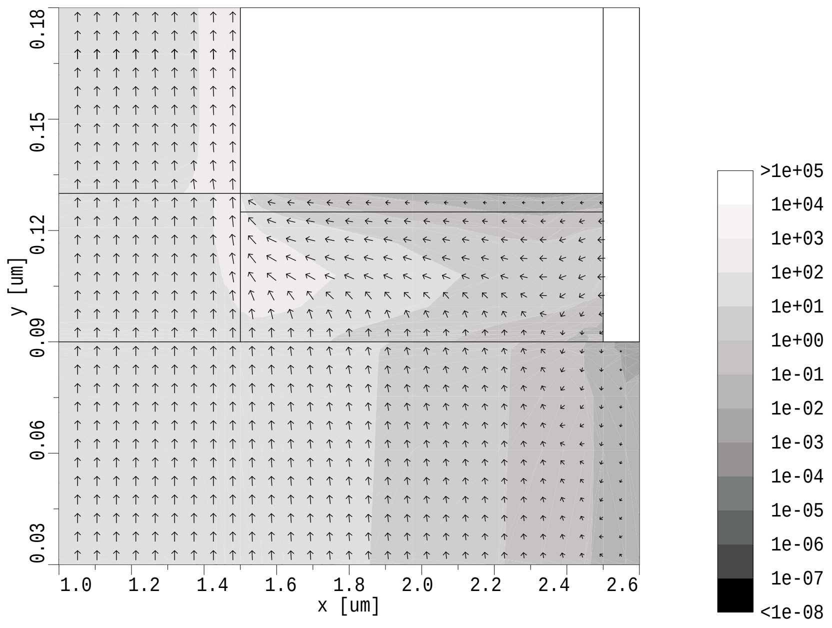


Figure 6: Electron current density [A/cm^2] at V_{BE}=1.2V. Simulation with a surface charge density of 10^{12} cm^{-2}.

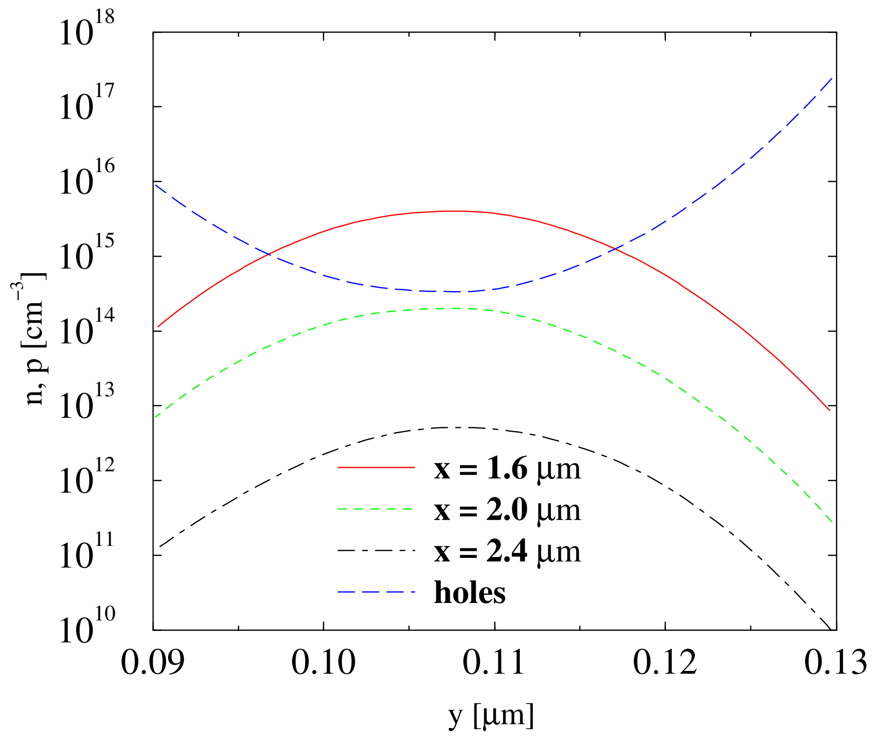


Figure 7: Electron and hole distribution in the ledge. Simulation with a surface charge density of 10^{12} cm^{-2}.

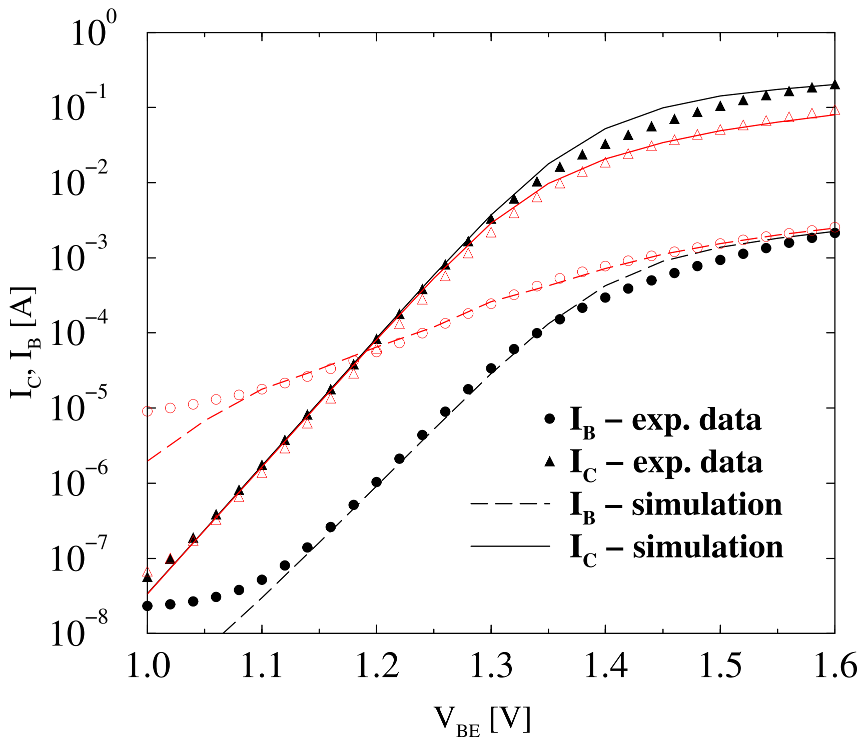


Figure 8: Forward Gummel plots at V_{CB} = 0 V. Comparison of measurement (symbols) and simulation (lines) before (filled) and after (open) HBT aging.

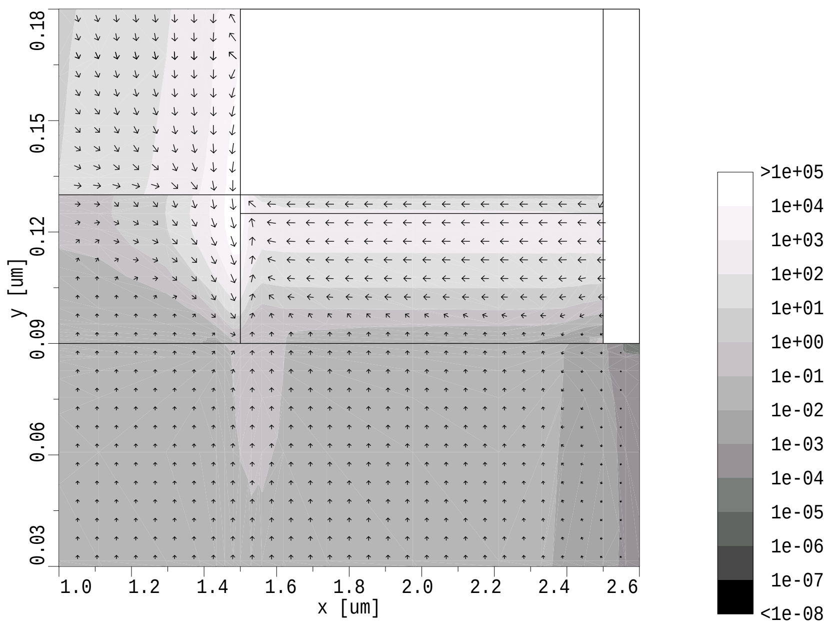


Figure 9: Electron current density [A/cm^2] at V_{BE}=1.2V. Simulation with a surface charge density of 4.10^{11} cm^{-2}.

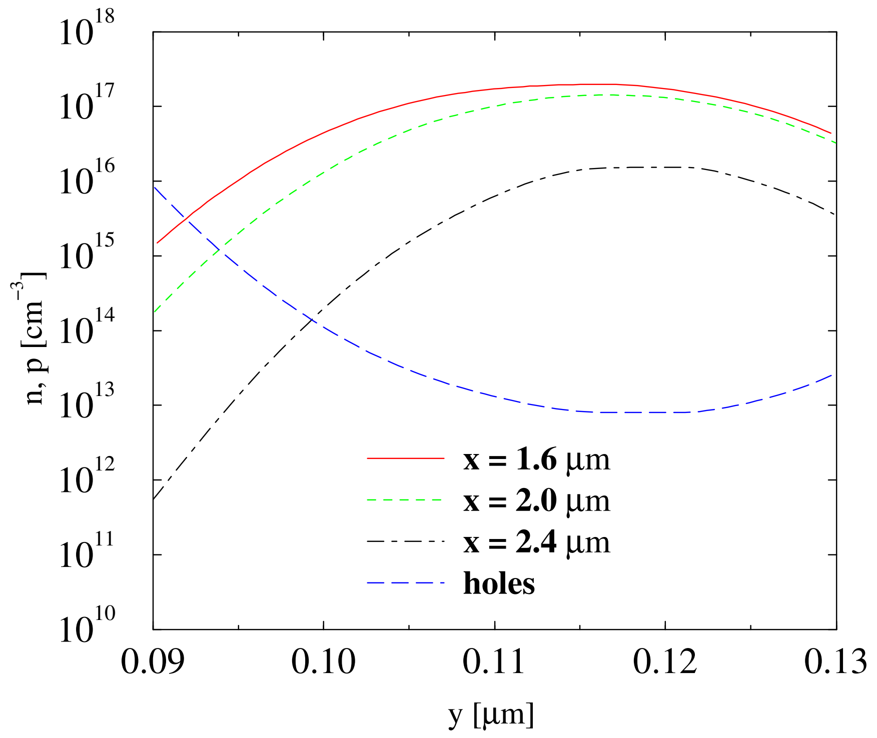


Figure 10: Electron and hole distribution in the ledge. Simulation with a surface charge density of 4.10^{11} cm^{-2}.

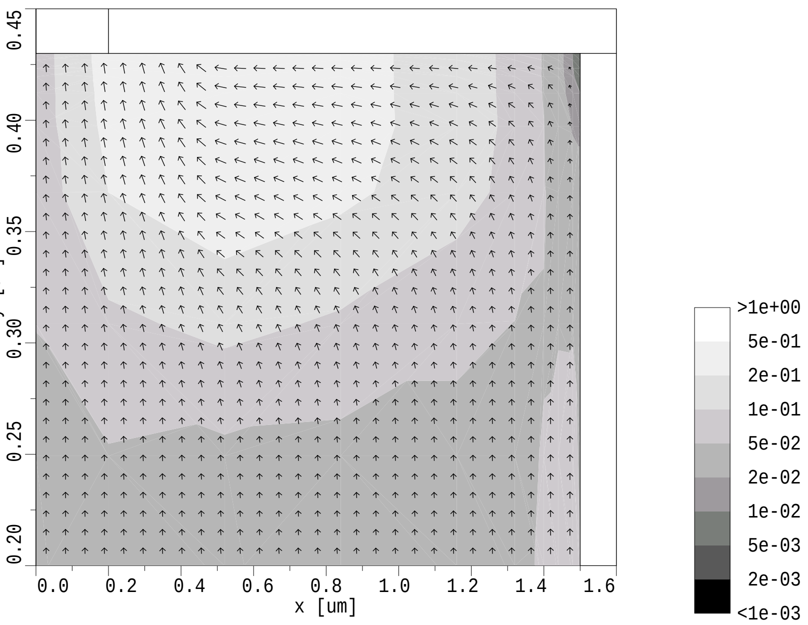


Figure 11: Electron current density [A/cm^2]. Simulation of emitter contact detachment.

2 Influence of Generation/Recombination Effects in Simulations of Partially Depleted SOI MOSFETs

2.1 Introduction

The small minimum feature size of todays devices makes it more and more difficult to get proper simulation results using the widely accepted drift-diffusion (DD) transport model. In particular the lack of accounting for nonlocal effects like carrier heating and velocity overshoot makes it desirable to use more sophisticated transport-models which are obtained by considering the first three or four moments of the BOLTZMANN transport equation. However these so called hydro-dynamic transport models, which are nowadays also quite common in simulations of bulk MOSFETs, lead to interesting problems when used in conjunction with SOI MOSFETs.

2.2 Used Device

The simulated SOI device is depicted in Fig. 12. It has an effective gate-length of $130\,\mathrm{nm}$, a gate-oxide thickness of $3\,\mathrm{nm}$, and a silicon-film thickness of $200\,\mathrm{nm}$. With a p-doping of $\mathrm{N_A} = 7.5 \times 10^{17}\,\mathrm{cm}^{-3}$ the device is partially depleted. The Gaussian n-doping under the electrodes has a maximum of $\mathrm{N_D} = 6 \times 10^{20}\,\mathrm{cm}^{-3}$.

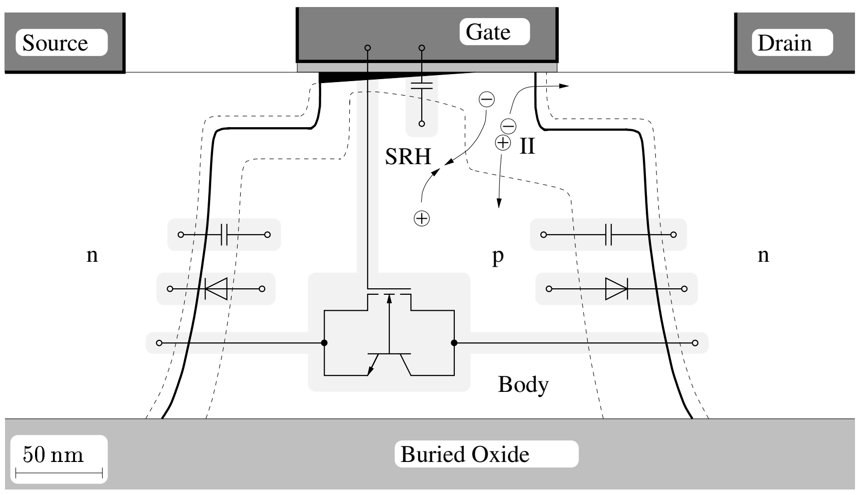


Figure 12: The geometry of the simulated SOI including the symbolic compact devices.

2.3 Simulation Results

2.3.1 Drift-Diffusion

Carrying out DD simulations shows a remarkable difference depending on whether impact-ionization (II) is turned on or off (Fig. 13). The increase of the drain current can be partially explained by the kink-effect [10]: Due to II in the pinch-off region, the holes are drawn into the floating body where they raise the potential (Fig. 14). This increased body potential leads via the body effect to an increased drain current.

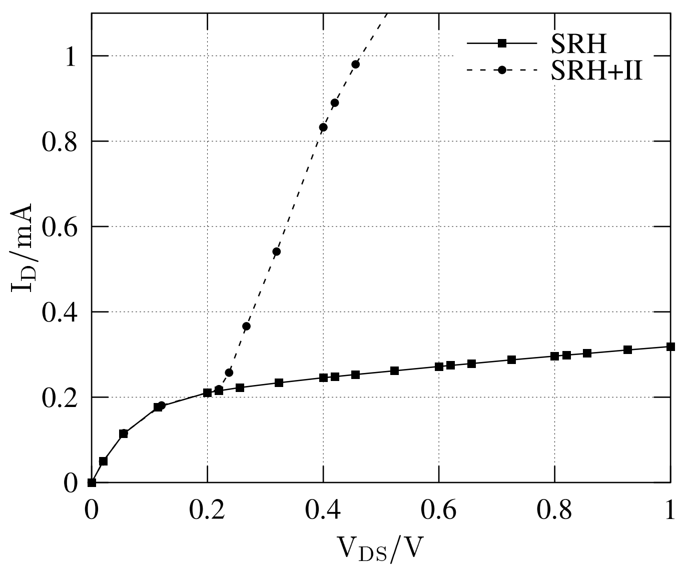


Figure 13: Output characteristics of the SOI obtained by DD simulations.

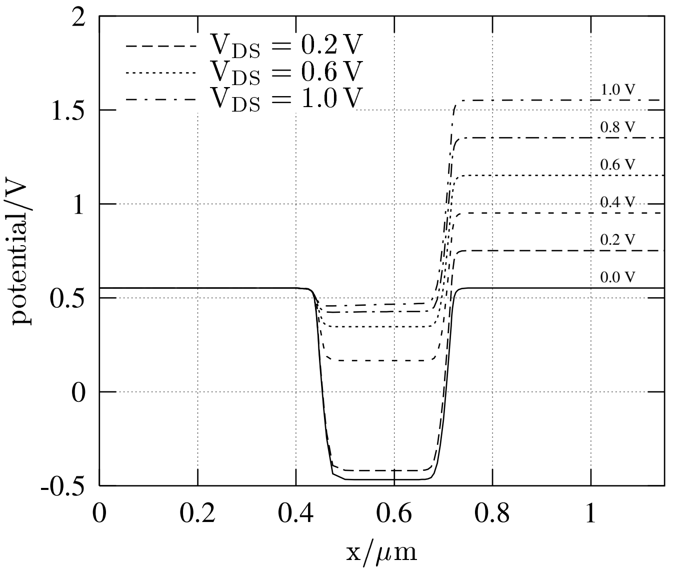


Figure 14: Distributed potential of the SOI obtained by DD simulations with II.

Simulating the device without II leads to a much smaller shift in the body potential (Fig. 15).

The kink effect alone cannot be responsible for such a big increase of the drain current. Another effect happening here is the bipolar effect, which means that the increased body potential causes the source-body diode being biased slightly forward, and thus makes it possible that more electrons can cross the lowered barrier coming from the source (emitter). Because of the small body (base) width, they are able to diffuse towards the drain (collector), where they are sucked off.

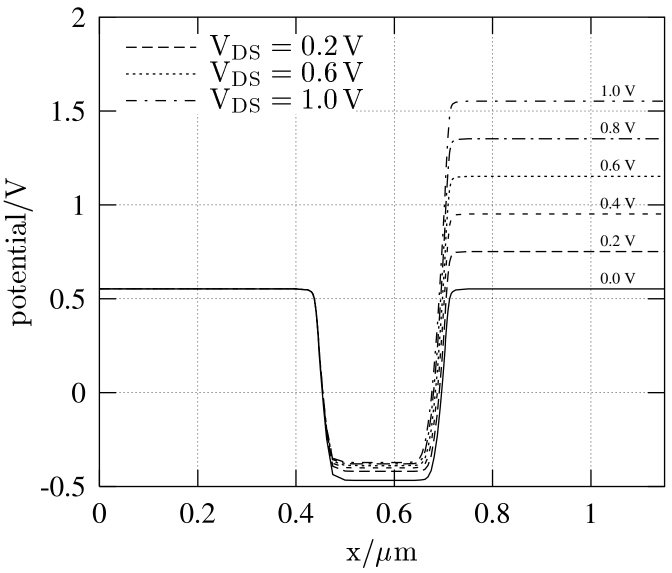


Figure 15: Distributed potential of the SOI obtained by DD simulations without II.

2.3.2 Hydro-Dynamic

Carrying out HD simulations complicates the subject further. As can be seen in Fig. 16 and Fig. 17, the output characteristics behave quite differently from those obtained by DD. Without II the drain current

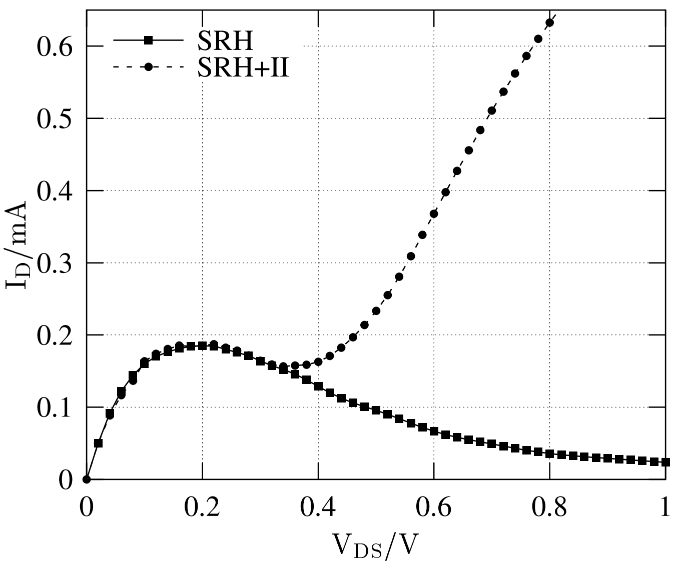


Figure 16: Output characteristics of the SOI obtained by HD simulations using MINIMOS-NT.

shows a negative differential output characteristic after the maximum at about $V_{DS} = 0.2\,\text{V}$. This can be explained by the difference in Shockley-Read-Hall generation/recombination (SRH): Using the DD transport model it makes virtually no difference whether SRH is turned on or not, but with the HD transport model the diffusion of the heated electrons near the pinch-off region is so significant, that they are transferred in the floating body where they recombine. This leads to an experimentally not observed decrease

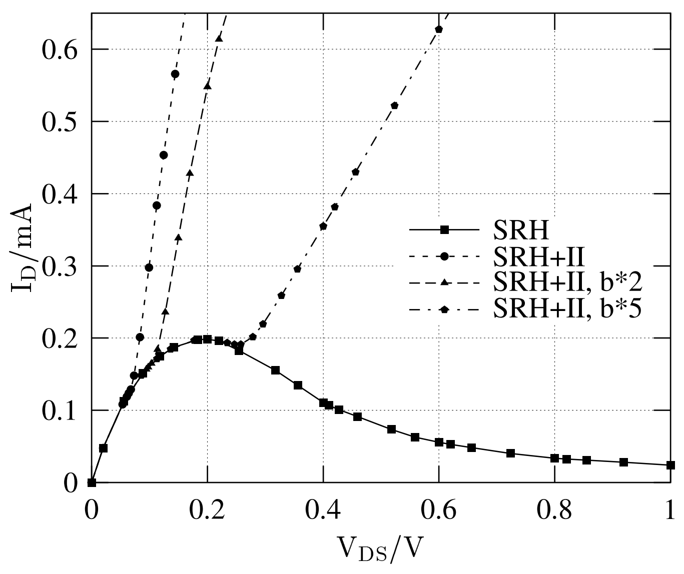


Figure 17: Output characteristics of the SOI obtained by HD simulations using DESSIS.

in the body potential which decreases via the body effect the drain current at a given drain-source voltage. The interesting point is that this behavior is observed with two different device simulators. Fig. 16 shows the results obtained from MINIMOS-NT [11], while Fig. 16 was produced using DESSIS [12]. In the DESSIS simulation the kink is located even before $V_{DS} = 0.1\,\mathrm{V}$, a difference to DD which is not yet understood and needs further analysis. The II model used in MINIMOS-NT is described in [13]. Further analysis of the problem will be given in Section 2.4.

2.3.3 Body Contact

The order of magnitude of the involved currents can be estimated by looking at simulations of a device with a body contact attached. Fig. 18 shows the output characteristics of this device and it is clear, that because of the fixed (pinned) body potential the drain current is not much affected by II. The big difference can be seen in the corresponding bulk (body) currents (Fig. 19): If SRH and II are used, one obtains the expected result that there is a body current which flows out of the transistor and has therefore a negative sign. But if in contrast only SRH without II is used, there is a body current of positive sign, which is several orders of magnitude smaller.

To estimate if the resulting current obtained by simulations with II is really caused by the increased body potential the simulations shown in Fig. 20 and Fig. 21 were made. In this case the source-body diode (and at small V_{DS} even the drain-body diode) is biased in forward direction using a body potential of $V_{BS} = 0.93\,\mathrm{V}$. (This voltage is taken from Fig. 14, where the body potential is raised by this value.) Accounting for the negative current offset at $V_{DS} = 0\,\mathrm{V}$ total agreement with the SRH+II curve taken from Fig. 13 is obtained at $V_{DS} = 1\,\mathrm{V}$.

The mentioned b-parameter depicted in several figures can be found in the formula of the II-coefficient which is known as the Chynoweth law, $\alpha(F) = \gamma\, a\, e^{-\frac{\gamma\, b}{F}}$. The default coefficients are taken from [14]. The factor γ expresses the dependance with respect to the lattice temperature. A variation of b can be interpreted as a scaling of the effective electric field F.

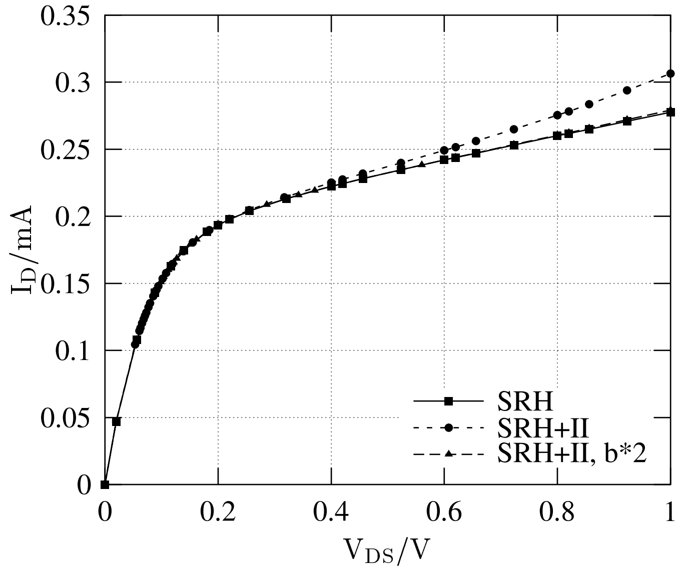


Figure 18: Output characteristics of the SOI with a body contact obtained by HD simulations.

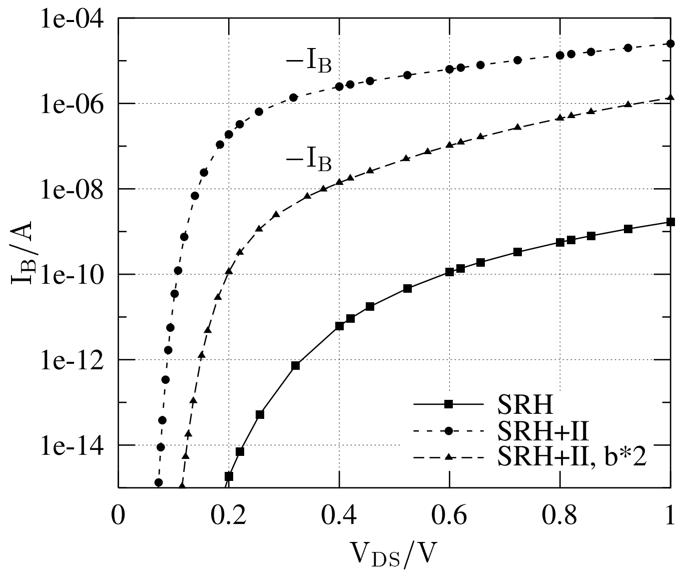


Figure 19: Bulk currents of the SOI with body contact obtained by HD simulations.

2.4 Cause of the Effect

It is believed that the main difference between the DD and the HD transport model responsible for the negative output conductance is the difference in the balance of the drift and diffusion currents:

$$\frac{|\mathbf{J}_{\mathrm{diff}}|}{|\mathbf{J}_{\mathrm{drift}}|} = \frac{k_B\,T_L}{q}\,\frac{|\boldsymbol{\nabla}\,n|}{n\,|\mathbf{E}|}\cdot\begin{cases} 1 & \text{... DD} \\ T_n/T_L & \text{... HD} \end{cases}$$

Apparently, in the HD model carrier diffusion is by a factor T_n/T_L higher than in the DD model.

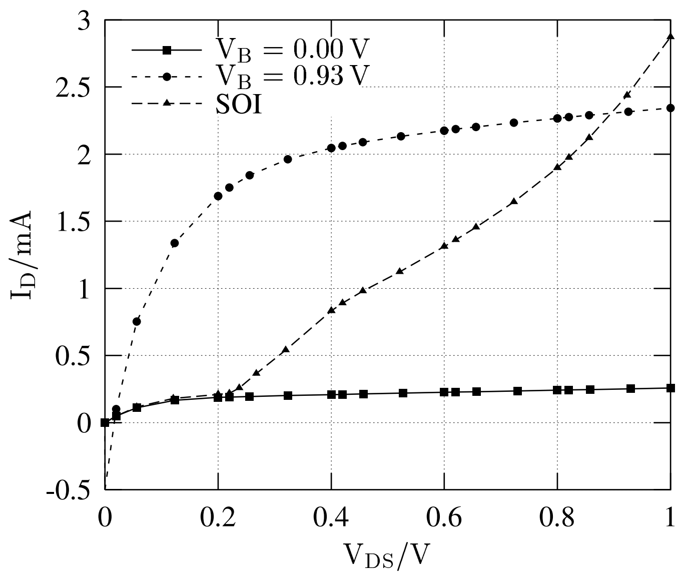


Figure 20: Comparison of the drain currents of the SOI and the device with body contact obtained by DD simulations.

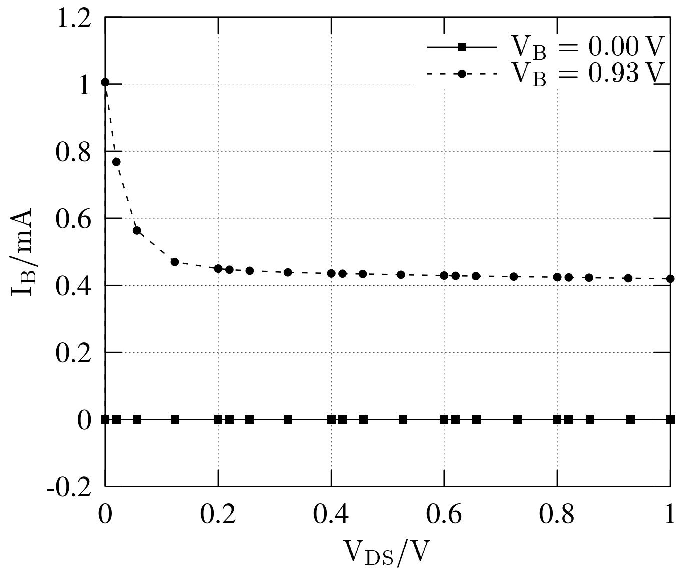


Figure 21: Comparison of the bulk currents at different body potentials obtained by DD simulations.

Due to the high temperature in the pinch-off region, the electrons spread away from the interface and diffuse into the body, where they recombine (Fig. 12). Removing holes there causes the body potential to drop which decreases the drain current via the body effect.

The difference in the carrier concentration between DD and HD can be seen clearly in Fig. 22 and Fig. 23.

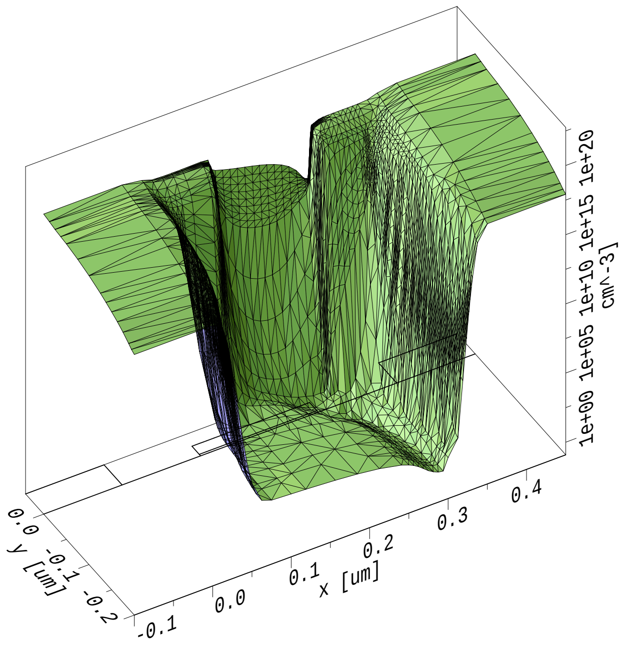


Figure 22: Electron concentration in the SOI obtained by a DD simulation.

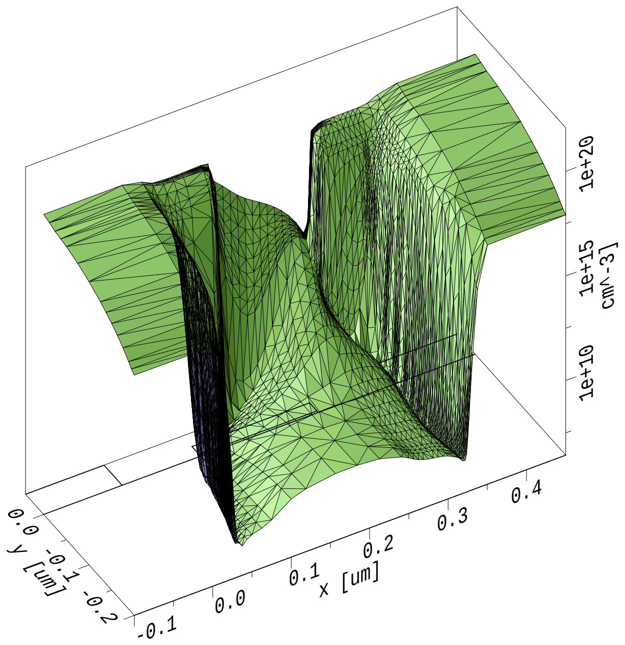


Figure 23: Electron concentration in the SOI obtained by a HD simulation.

3 Object-Oriented Design Patterns for Process Flow Simulations

3.1 Introduction

In a modern Technology CAD (TCAD) simulation environment like SIESTA or VISTA [15] data exchange between different simulators often remains an unsolved challenge. One approach to attack this problem is to use a file format common to all tools involved in the process flow. The PIF [16] (profile interchange format) file format presents an implementation of such an approach. In a PIF file data are stored as a set of lists. A powerful API for C, FORTRAN and LISP accounts for ease-of-use for various simulators and applications like file converters or meshing tools. It turned out however, that the definition of the pure file format is not sufficiently addressing all kinds of problems arising in a process flow. In order to illustrate the difficulties the following section will briefly discuss of what data a wafer-state description actually must consist of.

3.2 Wafer-State Description

Since a real-world wafer may consist of several million devices (transistors, diodes, capacitors) per die only a small number[1] of them can be treated in a process or device simulation. As a consequence a suitable wafer-state description need not contain any circuit or macro-model information as it is used in circuit simulators like SPICE [17]. Instead, the information required consists of:

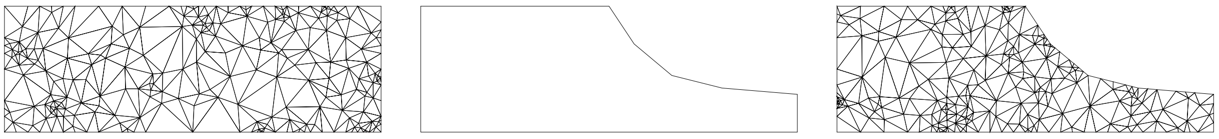
- geometry description
- properties (e.g., material type)
- distributed quantities (e.g., dopants)
- and grids on which the quantities are stored

Depending on the type of simulation carried out only a certain subset of the available data may be requested as input. Given that, one can already imagine the problems arising:

First, there is the need to ensure a consistent input-wafer for each individual simulator. This means that one needs a strong definition of what a certain simulator must read (input) what it must deliver as a result and also what (unhandled) quantities from previous simulation results are invalidated by the simulation and thus need to be removed.

Second, depending on the type of process simulation, it must be possible for a simulator to operate only on a subset of the whole data contained on a wafer. For instance, for a topography tool like an etching simulator only the geometry and material properties are of concern, whereas data like impurity concentrations and meshes are usually ignored. Problems arise once the etching simulation is finished and the tool is storing the results into a file. Since the result of the tool consists only of a pure geometry, the question of what happens to quantities stored on the (input) wafer arises. Such problems can only be solved by merging the newly generated geometry of the etch-step (Fig. 24(b)) with the original input-file (Fig. 24(a)), to create a new consistent wafer-state (Fig. 24(c)). Note that the resulting geometry must be re-gridded (new elements were inserted in this example), and all distributed quantities need to be interpolated onto the new points.

[1]In case of a process simulation it is only one device or maybe even just a part of one device.



(a) Original Wafer - contains several distributed quantities

(b) Geometry as returned from the etch-tool (no attribute information)

(c) Final result of the merge operation - contains all attributes from the original wafer

Figure 24: Merge operation after an etch-step

3.3 Transparent Interpolation

Interpolating values of distributed quantities onto newly generated grid points leads to the need for an automated mechanism. This is especially important since there are different attribute value types (scalar, vector). In order to keep the introduced interpolation error as small as possible different interpolation methods must be applied to the various attribute types. Some attributes (e.g., doping concentrations) require a logarithmic interpolation (the values are logarithmized before a linear interpolation takes place), others (e.g., stress components) need to be interpolated linearly.

3.4 Wafer-State Server

Our WAFER-STATE SERVER addresses this kind of problems and presents a standardized application programming interface (API) common to all simulators and tools. This API defines a strong protocol the simulators must adhere to. Tools must manipulate data in the wafer-state exclusively through this protocol. The WAFER-STATE SERVER also contains gridding and re-gridding capabilities. These are required for repair steps as outlined in the above example and are invoked transparently. The strategy chosen for interpolation allows different interpolation methods for each attribute without any reflection in the API. The user simply requests the value of an attribute at a certain point. The WAFER-STATE SERVER chooses the appropriate (configured) interpolation method for the attribute, and returns the interpolated value. Prior to interpolating, the grid-element in which the point is contained has to be found (point-location). This task is achieved with a binary tree, a finite-quad-tree and a finite-oct-tree [18] based data-structure for one-dimensional, two-dimensional and three-dimensional applications, respectively. These tree based data-structures support the WAFER-STATE SERVER in (a) performing efficient point-locations and (b) in identifying the grid elements in the repair step for which intersections with the geometry have to be computed.

3.5 Modularity

The WAFER-STATE SERVER is organized as a set of independent modules each dedicated to a certain task. These modules are a direct reflection of the problems discussed in the previous section. These modules are:

- READER module
  The READER module takes care of reading data from a certain file-format or database.
- WRITER module
  This module takes care of writing data to a certain file-format or database.

- GRIDDER module
  Handles all gridding problems. This is needed for repair steps.

- FINITE-OCT-TREE module
  This module represents the internal data-structure of the WAFER-STATE SERVER. It is used for point-location and to determine the grid elements which need to be repaired after a topography change.

Since the modules are realized as independent libraries, it is possible to use only some of them in a certain application. It is for example feasible to use the READER and WRITER modules to implement a file converter. There is no need to link an application against all of these modules or against the whole WAFER-STATE SERVER in case the provided services are not needed.

Each module is used by the WAFER-STATE SERVER via an interface class. Fig. 25 gives an overview of all used interface classes.

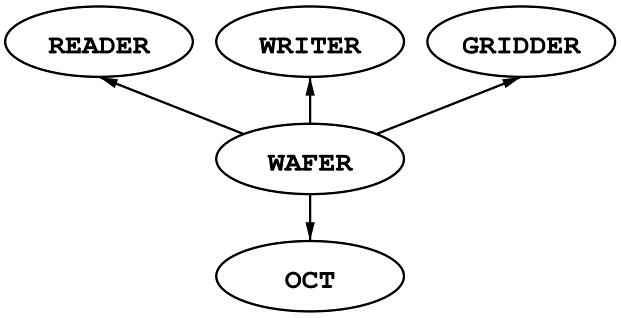


Figure 25: Overview of class interfaces used from WAFER class

Note that the use of abstract interfaces rather than concrete classes keeps the implementation details away from the WAFER-STATE SERVER's core classes in a typical object-oriented way. This design allows future extension of some of the capabilities like, e.g., supported file-formats, gridding algorithms or even of the internal data-structures the elements are stored on.

3.6 Interfaces

The use of several individual modules (READER, GRIDDER, ...) as opposed to using only a single one (WAFER) may seem complicated at first sight, however, it introduces two major advantages: On the one hand details about the underlying file format and about the gridding algorithms are well hidden to the WAFER-STATE SERVER's core functions. As already mentioned above this ensures that reading support for another file-format can be added later by implementing a READER module for this file-format. The same holds true for supporting various WRITER and GRIDDER modules. In general, any class that implements the interface to a certain module qualifies as a wafer-state module. On the other hand, settings specific to a certain implementation of a GRIDDER module (e.g., quality constraints) can directly be accessed by the simulator without the need for extra wafer-state functions.

Currently READER modules for PIF, DFISE [19] and the newly developed WSS file formats as well as support modules for the GRIDDER DELINK [20] (three dimensional) and TRIANGLE [21] (two dimensional) are implemented. Note that the READER and WRITER interfaces are not restricted to file based data access. It is also conceivable to implement READER/WRITER modules that directly connect to a certain database engine.

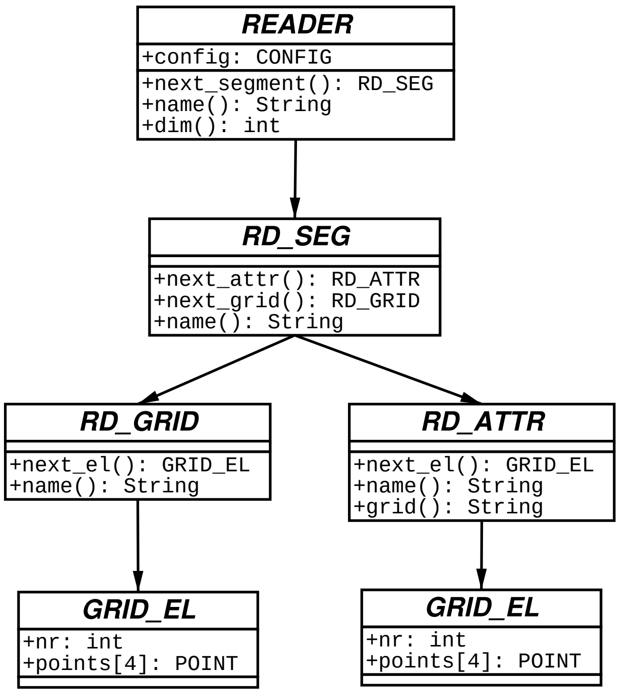


Figure 26: Attributes and methods of READER interface

READER Interface

Fig. 26 shows classes with their attributes and methods of the READER interface. All depicted classes are interface classes and are derived and implemented by a concrete class of the given file-format.

In order to read the file identified by a READER the WAFER-STATE SERVER first invokes the `next_segment` method. This will either return an instance of an implementation of a `RD_SEG` interface or indicate `end-of-file` in case no more segments are available. The methods `next_attr` and `next_grid` of the `RD_SEG` interface allow iteration over all attributes and grids of a segment respectively. Note that grids are allowed both at segment and attribute level. These methods return an implementation of a `RD_GRID` or `RD_ATTR` interface class, respectively. Again `end-of-file` indicates the last grid or attribute of this segment was reached. Finally, to iterate over all grid elements of a grid, the method `next_el` of the `RD_GRID` or `RD_ATTR` interface is invoked. Note that the implementation of the `next_el` method must take care to instantiate the points of the grid element as well as thereon stored quantities.

The READER interface contains a so called `CONFIG` object. This object holds information about which interpolation method must be applied to a certain attribute, about what fundamental data-type (scalar, vector) an attribute has and also what attributes at all are supported. This object is needed to properly instantiate the points of a `GRID_EL` object.

WRITER Interface

In Fig. 27 the classes comprising the WRITER interface are shown. In a similar manner to the READER several interfaces are used to transfer data to a certain file.

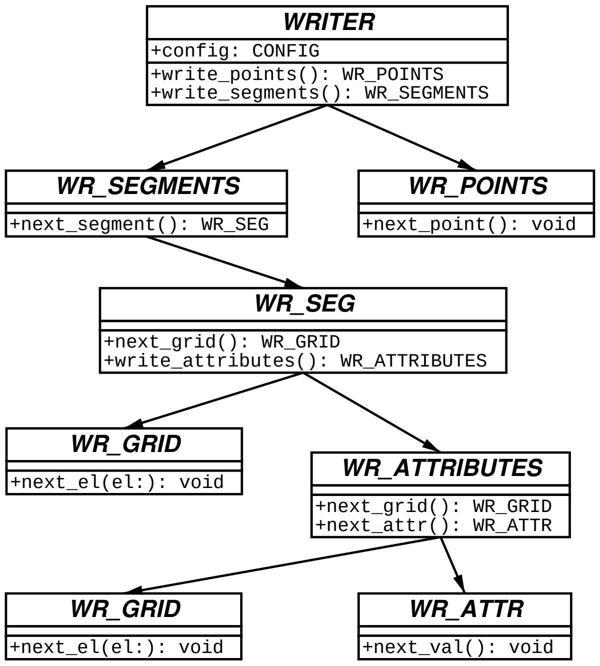


Figure 27: Attributes and methods of WRITER interface

First, the WAFER-STATE SERVER invokes the method `write_points` to indicate that a list of all points is following. The method returns an implementation of a `WR_POINTS` interface class. By using the method `next_point` all points are transferred to the WRITER module.

Next the segments and thereon stored data are created. For this purpose the method `write_segments` is called to retrieve an implementation of a `WR_SEGMENTS` class. This object is henceforth used to transfer grids and attributes to the file. The method `next_segment` of the `WR_SEGMENTS` object introduces a new segment. It returns an object of type `WR_SEG`. This object is used to define (a) stand-alone grids (grids that are not used to store attributes on) and (b) distributed quantities. The stand-alone grids are written using the `next_grid` method. Attributes are stored via the `WR_ATTRIBUTES` object by several invocations of methods `next_grid` and `next_attr`. Note that several attributes can share one grid. The method `next_attr` receives the name of a prior defined grid as one of its arguments. This means that a grid comprising a distributed quantity has to be defined before the definition of the quantity referring to this grid. Each call to `next_grid` or `next_attr` returns an object of type `WR_GRID` and `WR_ATTR` respectively. The methods `next_el` and `next_val` must be used to add a grid element or a value of an attribute.

GRIDDER Interface

In Fig. 28 the classes comprising the GRIDDER interface are shown. A gridding mechanism is invoked by first defining the topography, second starting the actual gridding algorithm and, finally collecting the generated elements.

The first step in the topography definition is to define all points of the geometry. The method `add_point` of the `GRIDDER` interface class must be invoked once for every point. Next, all elements forming the boundaries of the segments must be defined with the `add_boundary` method. Now the elements comprising the segments are defined using `add_segment`. Once the topography has been defined the method `start` triggers the actual meshing process. The elements are now ready to be collected. Method

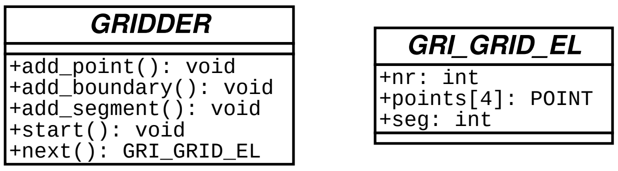


Figure 28: Attributes and methods of GRIDDER interface

`next_el` is used to iterate over all generated elements. The method returns `false` after having delivered the last element.

It is worthwhile mentioning that the above interface is used to access two-dimensional as well as three-dimensional gridder modules.

3.7 Protocol between Application and Wafer-State Server

The first step in the protocol (Fig. 29) from an application's point of view is to instantiate appropriate READER and GRIDDER objects. The choice of what READER module to instantiate depends on the file-format in which the data are stored. Next, the actual wafer object is instantiated by supplying the READER and GRIDDER objects as well as a CONFIG object to the wafer class.

At this point the data in the WAFER-STATE SERVER are ready for the application to be requested.

In the next protocol step the simulator requests the geometry and thereon stored attributes. Geometries and attributes are identified by names. These names are usually stored in a so called input-deck file (or they are passed on the commandline) which is read by the simulator independently from the wafer definition. This input-deck contains several settings for the simulator, among other details, it identifies which regions are to be treated in the simulation.

All relevant data have now been transferred to the simulator. Once the simulator has finished its calculations the results must be merged with the wafer-state. This so called update operation is performed in several individual steps. Each attribute which is configured as simulator output (c.f. Section 3.8) must now be stored back onto the wafer. The WAFER-STATE SERVER checks whether all attributes were received. Next the attributes which are invalidated by this process step are deleted.

Now the repairing mechanism is invoked. The newly added geometry is clipped with the one stored on the wafer-state. Grid points are taken over from the old grid where possible. The regions are then meshed using the supplied GRIDDER module. All attributes which were not treated by the simulator and which are not configured to be invalidated are interpolated onto the new geometries. Attributes which lie on no longer existing regions (e.g., a segment was altered by the simulator) are discarded.

After the repair operation the application instantiates the appropriate WRITER object and invokes the dump method of the wafer class to permanently store the simulation results on a file or database.

3.8 Definition of Process Steps

Each class of process step is configured in a database. The necessity for such a classification is best illustrated in an example. Take, for instance, a diffusion step: If the input wafer contains any stress components, the diffusion simulator either must take them into account (use them in the simulation) and update them when writing back the results, or the WAFER-STATE SERVER has to remove these components right

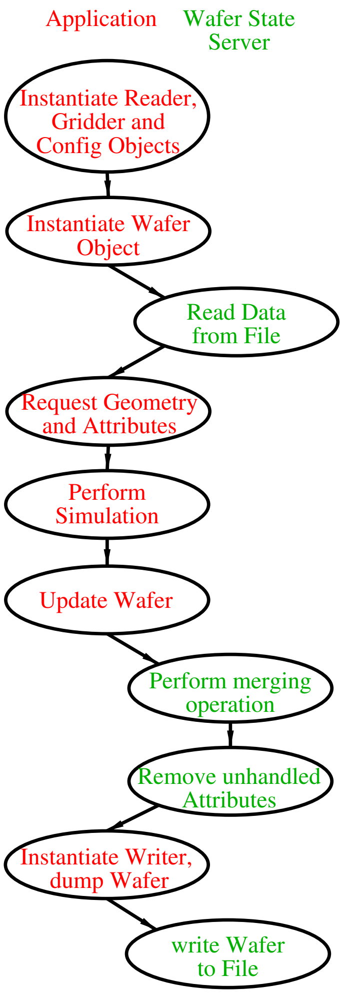

Application
Wafer State Server
Instantiate Reader, Gridder and Config Objects
Instantiate Wafer Object
Read Data from File
Request Geometry and Attributes
Perform Simulation
Update Wafer
Perform merging operation
Remove unhandled Attributes
Instantiate Writer, dump Wafer
write Wafer to File

Figure 29: Basic protocol between wafer-state server and application

after the diffusion step. All attributes which are modified either directly by the simulation or indirectly (e.g., invalidated stress component) must be listed in the database.

```
Diffus
{
    Invalidate
    {
        inv = "Stress"; // quantities to invalidate.
        exc = "";  // qu. to exclude from invalidation
    };

    read  = "Boron, Arsenic";
    write = "Boron, Arsenic";

    topography = false;
};
```

Figure 30: Configuration of a simple diffusion step

Fig. 30 shows a possible configuration of a simple diffusion step. The keywords `inv` and `exc` specify quantities which are to be invalidated and excluded from invalidation respectively. The keyword `read` lists all attributes which must be treated by the simulator (here: `Boron, Arsenic`). Similarly `write` specifies all attributes which must be supplied in the update operation. An asterisk (`*`) can be used with `inv` and `exc` to denote all contained attributes, however, attributes in the `read` and `write` statements will overrule this notion. In this example the attribute named `Stress` will be removed upon update.

The keyword `topography` is used to indicate whether the simulator changes the topography (`true`) in which case the WAFER-STATE SERVER must merge the new geometry with the existing one.

The types of the various attributes are also defined in the database. Fig. 31 depicts the configuration for attributes of type `Concentrations`.

```
Concentrations
{
    interpolation = "log";
    unit          = "1/cm^3";
    members       = "Donors, Acceptors, Boron,
                     Phosphorus, Arsenic, Indium,
                     Antimony, Nitrogen, Oxygen";
    datatype      = "Scalar";
};
```

Figure 31: Configuration of attribute type Concentrations

The definition of an attribute class consists of the unit (cm^{-3}) the data-type (scalar, vector, tensor), the name (Concentrations), a list of all possible instances (Arsenic, Phosphorus, Boron, Donors, Acceptors, . . .), and the interpolation method (linear, logarithmic).

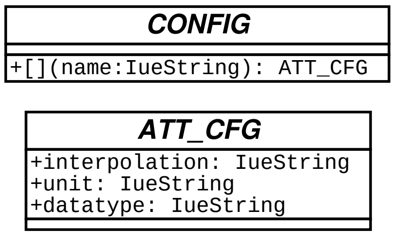


Figure 32: CONFIG object passed to READER and WRITER modules

The attribute type definitions are read into a `CONFIG` class (Fig. 32). An instance of such a class is passed to the READER and WRITER objects. The READER module needs the configuration to (a) identify a certain attribute by name and (b) to be able to properly instantiate the points of the grid elements. Since these points hold the actual attribute values the READER module is the only place where an instantiation can take place. The WRITER module uses the configuration to store attribute type information which is not explicitly contained in the WAFER-STATE SERVER data-structures (interpolation method, data-type) onto the file.

The actual configuration of a certain attribute is obtained via `operator[]` of the `CONFIG` class. The operator returns an `ATT_CFG` object of the named attribute or delivers an error if a configuration for an attribute with the given name does not exist.

3.9 TCAD Analysis

Performing TCAD analysis tasks [22] like optimization of VLSI semiconductor devices [23, 24] or inverse modeling of doping profiles [25] often results in an enormous number of individual simulation runs. Frameworks like SIESTA or VISTA take care of aspects like describing an experiment and queuing jobs on a cluster of workstations. Another aspect of TCAD analysis is how the input-data for the simulation runs are generated. Thus, the need for a tool to generate a wafer based on a textual description arises.

Input Wafer Creation

The wafer-state services contain a tool (MKWAFER) to create three-dimensional wafers suitable for a process or device simulation. This tool uses a file as input which contains commands written in the input-deck language as it is also used in our device simulator MINIMOS-NT [26, 1], and in the configuration of the process steps and attribute types within the WAFER-STATE SERVER. Fig. 33 and Fig. 36 show the input-deck description to generate the three-dimensional heterostructure bipolar transistor device depicted in Fig. 34 and Fig. 35.

```
Cube
{
   // coordinates are X/Y/Z
   points =
   [ [0.0, 0.0, 0.0], [1.0, 0.0, 0.0], [1.0, 1.0, 0.0],
     [0.0, 1.0, 0.0], [0.0, 0.0, 1.0], [1.0, 0.0, 1.0],
     [1.0, 1.0, 1.0], [0.0, 1.0, 1.0]
   ];

   // positive orientation of faces identifies the inner
   // region of the cube
   solid =
   [ [0, 1, 2, 3],    // bottom face
     [4, 7, 6, 5],    // top face
     [5, 6, 2, 1],    // front face
     [4, 0, 3, 7],    // back face
     [4, 5, 1, 0],    // left face
     [7, 3, 2, 6]     // right face
   ];

   Scaling { x = 1.0; y = 1.0; z = 1.0; };
   Offset  { x = 0.0; y = 0.0; z = 0.0; };
};
```

Figure 33: Description of a three-dimensional cube

For each section in the input-deck file (`Bulk, Base, Emitter, Contacts,...`) a corresponding segment is created. After having processed the last section of the input-deck the program computes the boundary representation of the whole geometry. The computation is achieved by first transferring all sets of coplanar faces into a two-dimensional representation. Second, a two-dimensional polygon clipping software [27] based on an algorithm of Kevin Weiler [28] is used to determine the intersections. Finally the resulting two-dimensional faces are transferred back into three-dimensional space and added to the structure. The boundary representation is then passed on to a gridder module (DELINK [20]) in order to generate a mesh of the whole structure. The final device is saved to a file using the WSS WRITER module.

3.10 Visualization

Another important aspect addressed in the WAFER-STATE SERVER is the visualization of attributes and geometries. Due to the abstraction of the file access we only need to support one certain file format (WSS). The chosen visualization environment is the *Visualization Toolkit* (VTK) [29]. To keep the visualization platform independent the JAVA programming language is used for both parsing the WSS file and for the actual visualization (JAVA-VTK binding). The parser generator used (ANTLR [30]) to generate the parser code is capable of producing JAVA and C++ parsers from the same language description, which relieves us from maintaining a separate JAVA and C++ version of the very same parser. The visualization runs on Unix and Windows platforms.

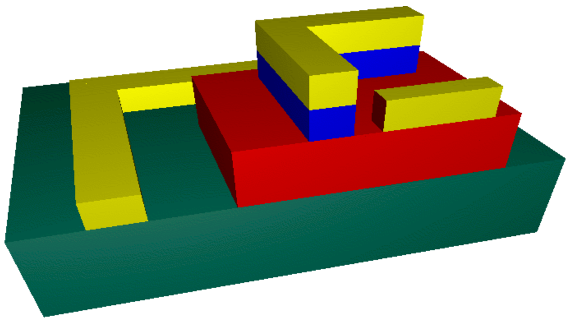


Figure 34: Three-dimensional HBT device

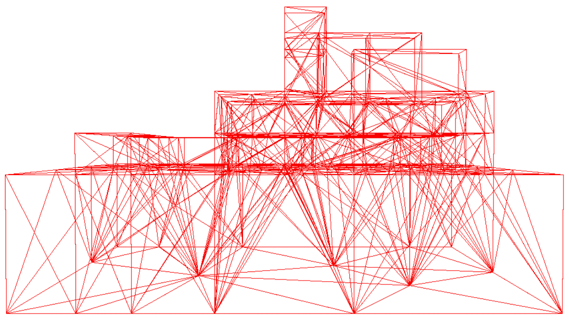


Figure 35: Mesh of the three-dimensional HBT device

```
#include "cube3d.ipd"

Geometry
{
   Bulk:   ~Cube {  Scaling  { x = 4.0; y = 3.0;  }  };
   CollectorContact1: ~Cube
   {  Scaling { x = 3.0; y = 0.4; z = 0.3; }
      Offset  { x = 0.5; y = 2.0; z = 1.0; }
   };
   CollectorContact2: ~Cube
   {  Scaling { x = 0.4; y = 2.0; z = 0.3; }
      Offset  { x = 0.5;          z = 1.0; }
   };

   Base1: ~Cube
   {  Scaling { x = 2.0; y = 1.5; z = 0.3; }
      Offset  { x = 1.5;          z = 1.0; }
   };
   Base2: ~Cube
   {  Scaling { x = 2.0; y = 1.5; z = 0.3; }
      Offset  { x = 1.5;          z = 1.3; }
   };
   BaseContact: ~Cube
   {  Scaling { x = 0.8; y = 0.2; z = 0.3; }
      Offset  { x = 2.5;          z = 1.6; }
   };

   Emitter1: ~Cube
   {  Scaling { x = 1.2; y = 0.3; z = 0.3; }
      Offset  { x = 2.0; y = 1.0; z = 1.6; }
   };
   Emitter2: ~Cube
   {  Scaling { x = 0.3; y = 1.0; z = 0.3; }
      Offset  { x = 2.0;          z = 1.6; }
   };
   EmitterContact1: ~Cube
   {  Scaling { x = 1.2; y = 0.3; z = 0.3; }
      Offset  { x = 2.0; y = 1.0; z = 1.9; }
   };
   EmitterContact2: ~Cube
   {  Scaling { x = 0.3; y = 1.0; z = 0.3; }
      Offset  { x = 2.0;          z = 1.9; }
   };

   epsilon = 1e-5;
};
```

Figure 36: Description of a three-dimensional HBT device

3.11 Implementation

The chosen programming language for the implementation of the WAFER-STATE SERVER is C++. This language facilitates a full object-oriented design as realized in the wafer-state servers' core components as well as an easy integration of existing programs (DELINK, TRIANGLE, DFISE-READER) thus ensuring good overall code re-usability. During the implementation of the wafer-state server care was taken to adhere to the ANSI C++ standard as close as possible. The WAFER-STATE SERVER is running on all platforms that offer an ANSI C++ compiler. The code does not rely on any operating system dependent features.

4 An Extensible TCAD Optimization Framework Combining Gradient Based and Genetic Optimizers

4.1 Introduction

Technology CAD (TCAD) tools like SIESTA[22, 24] have been successfully used for optimizing semiconductor devices[23] and for inverse modeling[25]. Although SIESTA proved to be a valuable tool and several interesting results[31, 32] were achieved using it, it did not, at that time, provide any global optimizer, but two gradient based optimizers[33]. The most recent advances include new, global optimizers, combine these two approaches to optimization, and improve on the flexibility and extensibility.

Over the years, it has been recognized that a successful TCAD optimization framework has to meet the following criteria.

1. The ability to execute simulation tools on a number of computers in the local network, and to schedule the execution of these simulation tools in reaction to changes in this network. For example, in a heterogeneous network that is not solely dedicated to executing simulation tools, the question how the tasks have to be scheduled so that the overall execution time is minimized is not trivial. Furthermore, software and hardware failures have to be taken into account.
2. Stability. This property is crucial for a program that usually runs for several days and has to deal with all kinds of software failure.
3. Extensibility. A TCAD framework has to deal with various programs[34], various data formats[35] and combinations thereof. Evaluating the goal function (i.e., the function to be optimized) often entails several calls to simulation tools. Because of the abundance of possible goal functions, a framework has to provide a flexible extension language which enables the user to succintly describe the desired goal function.
4. Specialized Optimizers. The evaluation of the goal function is usually very expensive: times range from about a minute to one hour or more for process simulations on current hardware. Strategies for finding global extrema of computationally very expensive functions are needed. The respective advantages and disadvantages of gradient based optimizers and evolutionary computation will be discussed in Sect. 4.4.
5. Finding a suitable starting value is often the most difficult and time consuming task when using a gradient based optimizer. Hence global optimizers which do not need a starting value sufficiently near to the global extremum are called for.

It should be noted that the goal function of an optimization in TCAD analysis may not even be a function in the mathematical sense. Simulation tools like MINIMOS [36, 26, 1] provide modes of operation which are not deterministic, i.e., the same input may lead to slightly different results. In the case of MINIMOS this is due to preconditioning which depends on the elapsed simulation time. Furthermore, it happens in practice that simulation tools do not converge for certain inputs, or yield results only after consuming exceptionally long computation time.

4.2 Design and Implementation

SIESTA was originally implemented in a dialect of Lisp[37, 38] called xlisp [39]. Because of new developments in language design and implementation since xlisp was written, the choice of a suitable base language for SIESTA was thought over. In order to meet the requirements described in Sect. 4.1, we posed the following demands on a suitable base language.

1. It has to provide an interface to the underlying operating system and network.
2. There has to be one (preferably more) stable and well supported implementation.
3. It has to support multiprocessing (or multithreading).
4. An extension language is necessary in order to provide the required flexibility.
5. The implementation has to be able to load additional code at run time and to enable the user to execute commands interactively.
6. It should be well designed and preferably standardized.

After evaluating several languages, namely C, C++, Common Lisp, Java, Perl, Python, Scheme, and Tcl, we decided to use Common Lisp [40] with multiprocessing support. In addition to fulfilling all of our requirements, it provides the following features which helped reducing the implementation time.

1. Common Lisp supports the paradigms of functional programming and of object oriented programming.
2. All language constructs are available at run time.
3. Several implementations on all major platforms are available and all of these provide compilers and interactive listeners.
4. A powerful macro system makes Common Lisp a very extensible programming language.
5. Its condition system and operators like `unwind-protect` and `ignore-errors` contribute to stability and robustness.
6. Common Lisp is an ANSI standard.

SIESTA runs on UNIX platforms, since UNIX provides good support for executing commands on remote computers and distributing files in a cluster of computers. Apart from these requirements, SIESTA is platform independent.

The requirements on the software infrastructure installed on the cluster of computers to be used in an experiment have been reduced to a minimum. Early versions of SIESTA required that the user's home directory is visible on all computers of the cluster and relied on NFS (network file system). NFS, however, is a source of problems since it does not provide sufficient synchronization between the state of files on different computers. Files are synchronized only after a pause which leads to problems when the result file of a simulation tool exists on a client, but is not seen on the computer where SIESTA runs. Furthermore, it is not possible to request a synchronization between two computers manually.

Several solutions for that problem were tried, yet none worked satisfactorily. One attempt was to wait a certain amount of time (up to thirty seconds) after a simulation tool had finished. Because of this limitation of NFS, we decided to dispense with it and use `rcp` or `scp` instead.

In order to run SIESTA, the following programs have to be installed on a cluster.

- One of these possiblities for communication has to be chosen:
  - `rsh`, `rcp`, and `rshd`. These are standard on UNIX, but not secure.
  - `ssh`, `scp`, and `sshd`. These are not found on every UNIX system by default, but are secure.

  Of course these programs have to be set up such that no password entry is needed for each individual login, but only once per session.
- Standard UNIX commands like `kill`, `mkdir`, `rm`, `top`, and `uptime`.
- Finally it goes without saying that the simulation tools to be used have to be installed. Licenses are managed as described in Sect. 4.3.6.

4.3 SEAL (Siesta Extension and Application Language)

To outline the capabilities of SIESTA, we give short summaries of some of the most important concepts of its extension language. Some examples of its use are given in order to show what the building blocks of more complex experiments look like.

4.3.1 Parallelization

Because of the need for parallelizing several evalutions of the goal function and thus the simulation tools, we chose to extend the language with a macro called `parallel`. It takes an arbitrary number of expressions as input and returns a list of results after the evalution of all the input forms (which is done in several processes) has finished.

Example.

```
 ?  (parallel (sleep 1) (sleep 1) (sleep 1))
 →  (NIL NIL NIL)
```
This took one second, not three.

```
 ?  (parallel (sleep 1)
              (parallel (sleep 1)
                        (parallel (sleep 1) (sleep 1))))
 →  (NIL (NIL (NIL NIL)))
```
This took one second as well.

The `parallel` macro and its sister function `p-apply` are an important building block of SIESTA and may of course be called by the user.

4.3.2 Persistent Object Storage

In order to facilitate the configuration and exchange of data (e.g., optimization results, populations from evolutionary algorithm optimizers) between different invocations of SIESTA, we implemented a simple persistent object storage. Using two functions for writing objects to and reading objects from a file, all of SIESTA's internal data-structures can be stored and retrieved in a text based format. A human readable format was chosen in order to enable inspection and changes with ordinary text editors.

The question how to define equality for any two given objects is closely related to storing and copying objects. Of course, the equality relation has to be reflexive, symmetric, and transitive. Furthermore, the equality relation should not be affected by storing and retrieving objects. Since objects usually contain references to other objects, there is no general copy operator which does the right thing in every application (deep copy vs. shallow copy). Thus the copy operator and the question whether a copied object and the original object should be considered equal depend on the problem at hand.

In SIESTA objects with id's, named objects, and appropriate equality and copying operators are defined and can be extended in an object oriented way. In addition to Common Lisp's four equality operators, we provide the equality operator `samep` and specialized copy operators. These operators work with the persistant object storage and SIESTA's data-structures as expected.

4.3.3 Setting Up Experiments

In this section we show how SIESTA is typically used. When starting up, it reads its initialization file in which the hosts to be used are defined via `define-all-hosts`. The information about the hosts will be used later by the task manager which schedules the execution of the various simulation tools. Hosts may later be enabled and disabled interactively by the user. Furthermore, for every host a function can be provided which decides if a host is usable right now; this is useful when certain hosts must not be used at certain times of day.

Commands are entered in any number of interactive listeners. After loading a file containing the definition of an experiment, the `run` command starts an optimization run. The value returned by `run` can be stored in a file, although the automatically generated log file contains all the results and information about the progress of the optimization. The result can also be used as a starting value for the next run.

Experiments are defined using `define-experiment`. The definition of an experiment consists of an optional description, the list of the free variables, their interval and their default values, the list of user variables which enable sophisticated customized setups, the goal function, the default value of the goal function to be used when no attempt was successful, the constraint function, and the configuration of one or more optimizers.

States correspond to points in the search space or to individuals of the population in the language of evolutionary computation. A state consists of a list of all (free) variables and their respective values, the experiment it belongs to and—after evaluation—the value of the goal function. States can be manipulated with the `make-state`, `copy-state`, `with-state-vars`, `setq-in-copied-state`, and `show-state` operators.

Setting up the evaluation function is usually the hardest part of defining an experiment. `test-run` can be used to evaluate the goal function on the default values interactively and to see if it works satisfactorily.

The following simple example shows the important steps when setting up new experiments.

Example.
```
  ? (defun sum-of-all-free-vars (state)
      (reduce #'+ (free-vars state) :key #'value))
  → SUM-OF-ALL-FREE-VARS
```

```
   ? (define-experiment :e1 (genetic-experiment)
       :description "A simple test experiment.  Optimizes the sum of
all free variables."
       :vars '((var1 (interval 0 10) :default 5)
               (var2 (interval 0 10) :default 5)
               (var3 (interval 0 10) :default 5 :free nil))
       :evaluation-function 'sum-of-all-free-vars
       :last-resort 0
       :optimizer-configuration (make-genopt-configuration :e1
                                  :minimize-or-maximize :maximize
                                  :algorithm "GASteadyStateGA"
                                  :crossover "TwoPointCrossover"
                                  :population-size 50
                                  :number-of-generations 12))
  → (#<GENETIC-EXPERIMENT :E1>)
```

```
   ? (test-run :e1)
  → 10
```

```
   ? (sum-of-all-free-vars (setq-in-copied-state (make-state :e1) var1
1 var2 2))
  → 3
```

```
   ? (run :e1)
     Several lines deleted.
  → ========== Generation 12 ==========
  → Number of requests evaluated:  426
  → Best:  #<REQUEST (id 637) (score 20.0) (vars ((VAR1 10.0) (VAR2
10.0)))>
  → #<REQUEST (id 637) (score 20.0) (vars ((VAR1 10.0) (VAR2 10.0)))>
```

4.3.4 Calling Simulation Tools

Since several types of software (and hardware) failures may occur when running simulation tools, especially in a networked environment, we extended the base language with a macro called `with-retries`. Its calling signature is `(number-of-tries &body default-forms) &body body`. `with-retries` executes its `body` until no error was raised, but at most `number-of-tries` times. Upon success the result values are those of the last form in `body`, otherwise the values returned by `default-forms`.

The first example tries to execute the body of the call three times and succeeds. The body in the second example is tried three times as well, but call to `error` defeats any success. The same happens in the third example; after three tries the default forms, here a call to `p-norm`, are executed and the resulting values are returned.

Example.
```
  ? (with-retries (3)
      (p-norm (vector 1 2 3)))
 → 3.7416575


  ? (with-retries (3)
      (error "Foo!"))
 → NIL


  ? (with-retries (3
                   (p-norm (vector 1 2 3)))
      (error "Foo!"))
 → 3.7416575
```

Among the inputs to the call of a simulation tool there is always one state. Calling simulation tools through the preferred interface entails the construction of an instance of class `task`, or a subclass thereof. Tasks contain all the information about calling a simulation tool and returning the required data. While subclasses and methods specialized to certain simulation tools are provided, the class hierarchy starting at `task` can also be specialized by the user, as well as the methods acting on tasks, e.g., the `execute` method.

The task manager schedules the execution of the tasks. In order to execute a task, it looks for reachable hosts (i.e., hosts whose load average can be retrieved) that are not disabled and that are not too busy. If such a host is available, the task is run on the host which currently provides the most computational resources. Otherwise, it waits until a host becomes available.

The following is an example of calling MINIMOS and extracting a certain vector from a result file. Here `tm-exec` constructs an instance of `minimos-task` and executes it using the task manager. MINIMOS is run at nice level 10 and it will consume at most 120 seconds of cpu time. The input state is constructed from the default values for a certain experiment (we assume this experiment has already been set up). Variables will be substituted in the given input deck (cf. Sect. 4.3.7) file. The input pif file[16] (containing the device structure) is given as well, and the last option means we request a curve file as result. After the task has finished, we parse the resulting curve file with `parse-curve-file` and return one of its column vectors. Other simulation tools are called similarly.

Example.
```
  ? (with-scratch-directory (dir)
      (find-column-vector-named
       (parse-curve-file
        (output-crv
         (tm-exec 'minimos-task
                  :nice 10
                  :max-cpu-time 120
                  :input-state (make-state :e12)
                  :input-deck #f"~/work/experiment-12/nmos.ipd"
                  :input-pif #f"~/work/experiment-12/nmos.pif"
                  :output-crv (make-unique-file "crv" :directory dir
                                                      :pre "crv-"))))
        "Id"))
→  #(-9.203809e-9)
```

4.3.5 Inverse Modeling

Many models in TCAD applications contain free parameters which depend on properties of the device material and have to be calibrated using measurements. Usually vectors of measured values are fit to characteric curves of the device in question.

It is not obvious which goal function should be used in an inverse modeling experiment where the distance between two vectors (where one is constant) is to be minimized. To facilitate experimentation and allow users to construct a suitable goal function, we provide the `log10`, `euclidean-norm`, `p-metric`, `p-norm`, `append-vectors` and `relative-error` functions.

Example.

```
 ?  (mapcar #'log10 (list 1e10 1e20 1e30))
→  (10.0 20.0 30.0)


 ?  (p-norm (list 1 2 3))
→  3.7416575


 ?  (mapcar (lambda (p)
              (p-norm (vector 1 2 3) p))
            (list 1/4 1/3 1/2 1 2 3 4))
→  (150.97025 50.7422 17.19151 6 3.7416575 3.3019273 3.1463463)


 ?  (append-vectors (vector) (vector 1) (vector 2 3) (vector 4 5 6))
→  #(1 2 3 4 5 6)
```

4.3.6 License Management

When using commercial simulation tools, the number of available licenses for a certain program is often limited. Thus we have to provide a way to ensure that at any point in time only a certain user prescribed amount of licenses of such programs are in use. Users can call simulation tools not only by using the predefined functions of the framework, but also from self written programs like shell scripts which are in turn called from within the framework. A suitable license management scheme has to take this into account.

In order to make our license management scheme meet these needs it works independently from the predefined functions of the framework. The following steps are necessary to use it. Whenever the number of requested licenses exceeds the number of available ones, certain processes have to wait until the required number becomes available. In order to show how many licenses are currently in use, the command `show-licenses` can be used.

1. Define the names of the licenses and how many of each may be used simultaneously. This is accomplished with `define-licenses` and is usually done in your SIESTA configuration file.
2. When using a license, wrap the code into `with-locked-licenses`.

Example.
```
 ?  (define-licenses
      (:dios :total-number 5)
      (:foo :total-number 7))
 →  (#<LICENSE :Dios> #<LICENSE :Foo>)
```

The following locks one license.
```
 ?  (with-locked-licenses (:dios)
      (show-licenses))
 →  Dios    Total:  5   In use now:  1
    Foo     Total:  7   In use now:  0
```

The following locks four licenses total.
```
 ?  (with-locked-licenses (:dios)
      (with-locked-licenses (:foo 3)
        (show-licenses)))
 →  Dios    Total:  5   In use now:  1
    Foo     Total:  7   In use now:  3
```

Users expressed interest in varying the number of available licenses while an optimization is running. This need frequently arises in a setting where people want to reserve one or two licenses for interactive work at certain times, but want all of them to be used, e.g., at night time. For simply changing the number of totally available licenses the function `set-number-of-licenses` can be used in an idle listener.

Example.
```
 ?  (set-number-of-licenses :dios 3)
 →  3
```

Later, increase the number of licenses to use again.
```
 ?  (set-number-of-licenses :dios 5)
 →  5
```

4.3.7 Input Deck Handling

Nearly all simulation tools use a text file for configuration. The configuration files of MINIMOS are called input deck[1] files, and we will use this term for all simulation tools. SIESTA generates these files by substituting the values of the variables of a state in template files. If a template file contains a string `<(foo)>` and the value of `foo` in the current state is, e.g., `1.23`, the string will be replaced with `1.23`. This applies to free and user variables of a state.

In case the use of `<(` and `)>` leads to collisions in the input deck file of some simulation tool, the begin and end marker can be changed.

4.4 Gradient Based Optimizers and Evolutionary Computation

During the last three or four decades there has been increasing interest in optimization algorithms which work similar to processes found in nature. The methods of genetic algorithms[41, 42, 43] and evolutionary strategies[44], although having different roots, have converged and are now commonly known under the name of evolutionary computation. A journal of the same name[45] has been published since 1993.

A brief outline of evolutionary algorithms is as follows. They work with sets (or populations) of potential solutions. Starting from a random population, each individual is assigned a score via a goal function. In the selection step, certain individuals are chosen to proliferate and form a new population. Operators (e.g., mutation) are applied to the individuals of the new population with prescribed probability, and the population is evaluated again. New generations are formed in this way until a termination condition is fulfilled. Obviously, many alternatives for every step in this algorithm exist and have been described and discussed in many publications[43]. Although the Schema Theorem[43, page 53] and similar theorems explain how evolutionary algorithms work in a quantifiable way, and many special algorithms have been intensively studied, no consistent theory of evolutionary computation exists to date and the question of which evolutionary algorithm to choose for a given problem can only be answered by experimentation and experience. Nevertheless evolutionary algorithms proved to be very general and valuable tools. While domain specific optimizers typically perform better than their general evolutionary algorithm counterpart, evolutionary algorithms can easily be adapted to the problem at hand and are usable whenever the lack of detailed knowledge about the goal function prohibits developing a domain specific optimization algorithm.

Although evolutionary computation is a well established optimization technique today, its application to TCAD analysis has been limited. Reasons are certainly the need for lots of computational resources and the requirements outlined in Sect. 4.1. While most research in evolutionary computation has been done on relatively cheap to evaluate goal functions, the optimization of semiconductor devices has to cope with a relatively limited number of evaluations.

The most important difference from the usual practice of evolutionary algorithm optimizers is that runs are usually finished before a common termination condition like "95% of the population are identical" is fulfilled.

In the following we discuss the advantages and disadvantages of gradient based and evolutionary algorithm optimizers and show why the combination of both is worthwhile. An optimization run with an evolutionary algorithm optimizer usually yields a set, or population, of nearly optimal solutions. The best of these is used as the starting point for a run with a gradient based optimizer, thus bringing together global and local optimization methods. Other combinations are also possible, for example: starting populations can be constructed manually; other states than the best in a population may yield better final results because they lie closer to the global optimum; the configuration of an optimizer can be changed and the computation restarted with the latest population or starting point.

4.4.1 Advantages and Disadvantages of Gradient Based Optimizers

Most importantly, gradient based optimizers are hill climbing algorithms and therefore local optimization techniques. Although very sophisticated algorithms[46] have been developed, they all depend on a suitable starting point. In practice, finding this starting point has been found to be the major hurdle when trying to do unattended, automatic optimizations. Typically finding such a point and shortening the parameter intervals so that the goal function can actually be evaluted requires several tries and can easily take several days.

When increasing the number of variables, the number of evaluations increases as well. While goal functions with few variables are feasible, optimizations with about 20 variables are usually impractical. Evolutionary algorithms do not suffer as much from this effect.

4.4.2 Advantages and Disadvantages of Evolutionary Algorithm Optimizers

Evolutionary algorithm optimizers are global optimization methods and scale well to higher dimensional problems. They are robust with respect to noisy evaluation functions, and the handling of evalution functions which do not yield a sensible result in given period of time is straightforward.

The algorithms can easily be adjusted to the problem at hand. Almost any aspect of the algorithm may be changed and customized. On the other hand, although lots of research has been done on which evolutionary algorithm is best suited for a given problem, this question has not been answered satisfactorily. Although the standard values usually provide reasonably good performance, different configurations may give better results. Furthermore, premature convergence to a local extremum may result from adverse configuration and not yield (a point near) the global extremum.

4.5 Available Optimizers

The following brief overview lists all optimizers currently available in SIESTA, namely two gradient based[46] and two stochastic global ones.

4.5.1 Genopt

The interface to GAlib[47], a C++ library for genetic optimization, is called genopt. It provides standard selection, crossover, mutation, scaling, and termination methods[43].

For our experiments we mainly use the following setup, because it provides good results in an acceptable amount of computation time. Since all parameters are reals chosen from intervals, we represent them as floating point numbers, and not as binary vectors as favoured in early genetic optimization. We use a mutation operator which adds a random number from a normal distribution, more precisely, $x \in [a, b]$ is changed to $\min(\max(N(x, \sigma), a), b)$, where σ depends on the length of the interval.

As crossover operators we use two point and uniform crossover. Most populations consist of about 40 to 50 individuals. Some optimization tasks allow us to evaluate about 20 generations per hour, which amounts to roughly 500 generations per day. Typical runs last for two or three days.

Constraints handling is done using the popular penalty method, i.e., the scores of states which do not fulfill given constraints, which are defined as an arbitrary function, are increased by prescribed amounts.

4.5.2 Siman

Simulated annealing[48, 49] was invented by Kirkpatrick in 1982 and is a modified version of hill climbing. Starting from a random point in the search space, a random move is made. If this move yields a better point, it is accepted. If it yields a worse point, it is accepted only with a certain probability $p(t)$ which depends on the time t. The function $p(t)$ is initially close to 1, but gradually reduces towards 0 in analogy to the cooling of a solid. Hence initially any moves are accepted, but as the temperature reduces, the probability of accepting a negative move is lowered. Negative moves are essential sometimes if local maxima are to be escaped, but obviously too many negative moves will simply lead away from an extremum. Versions like fast re-annealing, adaptive annealing and parallel annealing have been developed. In our framework we provide an interface to an implementation[50] by Lester Ingber.

4.5.3 Donopt

This gradient based optimizer[22, 33] minimizes a scalar value and supports equality and inequality constraints. It is based on donlp2[51, 52] by Peter Spellucci.

4.5.4 Lmmin

The Levenberg-Marquardt algorithm[53] is an efficient method to solve nonlinear least squares problems, and is therefore well suited for inverse modeling tasks. SIESTA provides an interface to the implementation found in the MINPACK [54, 55] project.

The parameter values are chosen from prescribed intervals. However, arbitrary constraints are not supported by this optimizer. The step size used for the gradient computation and a tolerance value acting as termination criterion can be adjusted.

4.6 Summary

SIESTA has been used for several optimizations of real world devices on a cluster of a dozen workstations with 18 cpus and has proven to be very robust and to yield good results. The combination of gradient based optimizers and evolutionary computation allows to take advantage of the benefits of both approaches. While the default configurations of the optimizers provide reasonably good performance, lots of aspects of an optimization run can be customized. The output of one or more optimization runs can be combined and used as input for the next run. This interoperability allows for interesting combinations of optimizers, comparisons of their performance, and specialization to the problem at hand.

References

[1] T. Binder, K. Dragosits, T. Grasser, R. Klima, M. Knaipp, H. Kosina, R. Mlekus, V. Palankovski, M. Rottinger, G. Schrom, S. Selberherr, and M. Stockinger. *MINIMOS-NT User's Guide*. Institut für Mikroelektronik, 1998.

[2] V. Palankovski, S. Selberherr, and R. Schultheis. Simulation of Heterojunction Bipolar Transistors on Gallium-Arsenide. In SISPAD'99 [57], pp 227–230.

[3] V. Palankovski, R. Quay, S. Selberherr, and R. Schultheis. S-Parameter Simulation of HBTs on Gallium-Arsenide. In *Proc. High Performance Electron Devices for Microwave and Optoelectronic Applications EDMO*, pp 15–19, London, 1999. King's College.

[4] T. Low, C. Hutchison, P. Canfield, T. Shirley, R. Yeats, J. Chang, G. Essilfie, W. Whiteley, D.D'Avanzo, N. Pan, J. Elliot, and C. Lutz. Migration from an AlGaAs to an InGaP emitter HBT IC process for improved reliability. In Digest GaAs IC Symp. *Atlanta, Georgia*, pp 153–157, November 1998.

[5] P. Ma, P. Zampardi, L. Zhang, and M.F. Chang. Determining the Effectiveness of HBT Emitter Ledge Passivation by Using an On-Ledge Schottky Diode Potentiometer. *IEEE Electron Device Lett.*, 20(9):460–462, 1999.

[6] N. Bovolon, R. Schultheis, J.-E. Müller, P. Zwicknagl, and E. Zanoni. A short-term high-current-density reliability investigation of AlGaAs/GaAs heterojunction bipolar transistors. In Electron.Lett., *vol. 19, no. 12*, pp 469–471, 1998.

[7] M. Borgarino, R. Plana, S.L. Delage, F. Fantini, and J. Graffeuil. Influence of Surface Recombination on the Burn-In Effect in Microwave GaInP/GaAs HBT's. *IEEE Trans.Electron Devices*, 46(1):10–16, 1999.

[8] K.A. Christianson. Reliability of III–V Based Heterojunction Bipolar Transistors. *Microelectron.Reliab.*, 38(1):153–161, 1997.

[9] K. Mochizuki, T. Oka, K. Ouchi, and T. Tanoue. Reliability Investigation of Heavily C-doped InGaP/GaAs HBTs Operated under a Very High Current-Density Condition. *Solid-State Electron.*, 43:1425–1428, 1999.

[10] J. Tihanyi and H. Schlötterer. Influence of the Floating Substrate Potential on the Characteristics of ESFI MOS Transistors. *Solid-State Electron.*, 18:309–314, 1975.

[11] T. Simlinger, H. Brech, T. Grave, and S. Selberherr. Simulation of Submicron Double-Heterojunction High Electron Mobility Transistors with MINIMOS-NT. *IEEE Trans.Electron Devices*, 44(5):700–707, 1997.

[12] DESSIS-ISE Users Manual, Release. 6.

[13] M. Knaipp, W. Kanert, and S. Selberherr. Hydrodynamic Modeling of Avalanche Breakdown in a Gate Overvoltage Protection Structure. *Solid-State Electron.*, 44:1135–1143, 2000.

[14] R. VanOverstraeten and H.J. DeMan. Measurement of the Ionization Rates in Diffused Silicon p-n Junctions. *Solid-State Electron.*, 13:583–608, 1970.

[15] R. Strasser, Ch. Pichler, and S. Selberherr. VISTA - A Framework for Technology CAD Purposes. In Hahn and Lehmann [56], pp 450–454.

[16] F. Fasching, C. Fischer, S. Selberherr, H. Stippel, W. Tuppa, and H. Read. A PIF Implementation for TCAD Purposes. In W. Fichtner and D. Aemmer, editors, *Simulation of Semiconductor Devices and Processes*, volume 4, pp 477–482, Konstanz, 1991. Hartung-Gorre.

[17] G. Massobrio and P. Antognetti. *Semiconductor Device Modeling with Spice*. McGraw-Hill, New York, second edition, 1993.

[18] T. Binder and S. Selberherr. A Parallel Finite Oct-Tree for Multi-Threaded Insert, Delete, and Search Operations. In *Intl. Conf. Applied Modeling and Simulation*, pp 613–616, Cairns, Australia, 1999.

[19] ISE. *ISE TCAD Manuals vol. 6, release 4*. ISE Integrated Systems Engineering, 1997.

[20] P. Fleischmann and S. Selberherr. A New Approach to Fully Unstructured Three-dimensional Delaunay Mesh Generation with Improved Element Quality. In *Simulation of Semiconductor Processes and Devices*, pp 129–130, Tokyo, Japan, 1996. Business Center for Academic Societies Japan.

[21] J. R. Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *First Workshop on Applied Computational Geometry*, pp 124–133. Association for Computing Machinery, 1996.

[22] R. Strasser. *Rigorous TCAD Investigations on Semiconductor Fabrication Technology*. Dissertation, Technische Universität Wien, 1999. `http://www.iue.tuwien.ac.at/phd/strasser`.

[23] R. Plasun, Ch. Pichler, T. Simlinger, and S. Selberherr. Optimization Tasks in Technology CAD. In Hahn and Lehmann [56], pp 445–449.

[24] R. Plasun, M. Stockinger, R. Strasser, and S. Selberherr. Simulation Based Optimization Environment and its Application to Semiconductor Devices. In *Intl. Conf. on Applied Modelling and Simulation*, pp 313–316, Honolulu, Hawaii, USA, 1998.

[25] R. Strasser, R. Plasun, and S. Selberherr. Practical Inverse Modeling with SIESTA. In SISPAD'99 [57], pp 91–94.

[26] T. Simlinger, H. Kosina, M. Rottinger, and S. Selberherr. MINIMOS-NT: A Generic Simulator for Complex Semiconductor Devices. In H.C. de Graaff and H. van Kranenburg, editors, *25th European Solid State Device Research Conference*, pp 83–86, Gif-sur-Yvette Cedex, France, 1995. Editions Frontieres.

[27] Klamer Schutte. An edge labeling approach to concave polygon clipping. *Submitted to ACM*, 1995. `http://www.ph.tn.tudelft.nl/People/klamer/clip.ps.gz`.

[28] Kevin Weiler. Polygon Comparision using a Graph Representation. In *Computer Graphics, 14*, pp 10–18. SIGGRAPH 80, 1980.

[29] Will Schroeder, Ken Martin, and Bill Lorensen. *An Object-Oriented Approach To 3D Graphics*. Prentice Hall, 1999.

[30] Gary L. Schaps. Compiler Construction with ANTLR and Java. *Dr. Dobb's Journal*, 1999. `http://www.ddj.com/articles/1999/9903/9903h/9903h.htm`, `http://www.antlr.org`.

[31] M. Stockinger. *Optimization of Ultra-Low-Power CMOS Transistors*. Dissertation, Technische Universität Wien, 2000.

[32] M. Stockinger, A. Wild, and S. Selberherr. Closed-Loop MOSFET Doping Profile Optimization for Portable Systems. In *Proc. 2nd Intl. Conf. on Modeling and Simulation of Microsystems*, pp 411–414, San Juan, Puerto Rico, USA, 1999.

[33] R. Plasun. *Optimization of VLSI Semiconductor Devices*. Dissertation, Technische Universität Wien, 1999. `http://www.iue.tuwien.ac.at/phd/plasun`.

[34] J. Daniell and S.W. Director. An Object-Oriented Approach to CAD Tool Control. *IEEE Trans.Computer-Aided Design*, 10(6):698–713, 1991.

[35] T. Binder and S. Selberherr. Object-Oriented Design Patterns for Process Flow Simulations. In Proc. 4th Annual IASTED International Conference on Software Engineering and Applications, Las Vegas, 2000.

[36] S. Selberherr, W. Fichtner, and H. Pötzl. MINIMOS – A Program Package to Facilitate MOS Device Design and Analysis. In B.T. Browne and J.J. Miller, editors, *Numerical Analysis of Semiconductor Devices and Integrated Circuits*, volume I, pp 275–279, Dublin, 1979. Boole Press.

[37] J. McCarthy. Recursive Functions of Symbolic Expressions and their Computation by Machine (Part I). *Communications ACM*, 3:184–195, 1960.

[38] J. McCarthy. *Lisp 1.5 Programmer's Manual*. MIT Press, 1962.

[39] D.M. Betz. *XLISP: An Object-Oriented Lisp, Version 2.1*. Apple, Peterborough, New Hampshire, USA, 1989.

[40] P. Graham. *ANSI Common Lisp*. Prentice Hall, New Jersey, 1996.

[41] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[42] J.H. Holland. *Adaption in Natural and Artificial Systems*. MIT Press, 1975.

[43] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, Berlin, 1996.

[44] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog Verlag, Stuttgart, 1973.

[45] K.A. De Jong, editor. *Evolutionary Computation*. journal published by MIT Press, 1993.

[46] C.T. Kelley. *Iterative Methods for Optimization*. SIAM, Philadelphia, 1999.

[47] Matthew Wall. GAlib: A C++ Genetic Algorithm Library. `http://lancet.mit.edu/ga/`, 1994.

[48] D. Beasley, D.R. Bull, and R.R. Martin. An Overview of Genetic Algorithms: Part 2, Research Topics. *University Computing*, 15:170–181, 1993.

[49] R.A. Rutenbar. Simulated Annealing Algorithms: an Overview. *IEEE Circuits & Devices*, pp 19–26, 1989.

[50] L. Ingber. Adaptive Simulated Annealing. `http://www.ingber.com/#ASA-CODE`, 1993.

[51] P. Spellucci. donlp2 Users Guide. part of the netlib project, 1995.

[52] P. Spellucci. Solving General Convex QP Problems via an Exact Quadratic Augmented Lagrangian with Bound Constraints. `http://www.mathematik.th-darmstadt.de/ags/ag8/spellucci`, 1996.

[53] D.W. Marquardt. An Algorithm for the Estimation of Nonlinear Parameters. *Soc. Ind. Appl. Maths. J.*, 11:431–441, 1963.

[54] J.J. Moré, B.S. Garbow, and K.E. Hillstrom. Users Guide for MINPACK-1, 1980. Argonne National Laboratory Report ANL-80-74, Argone, IL.

[55] J.J. Moré, D.C. Sorensen, K.E. Hillstrom, and B.S. Garbow. *The MINPACK Project*. Sources and Development of Mathematical Software. Prentice-Hall, Englewood Clifs, NJ, 1984.

[56] W. Hahn and A. Lehmann, editors. *Proc. 9th European Simulation Symposium*, Passau, Germany, 1997. Society for Computer Simulation International.

[57] *Proc. Simulation of Semiconductor Processes and Devices*, Kyoto, Japan, 1999.