# VISTA Status Report
# June 1995

R. Mlekus, Ch. Pichler, H. Puchner,
S. Selberherr, W. Tuppa

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

# Contents

# 1 The VISTA Simulation Flow Control Module

## 1.1 Overview

The integration of an external optimizer lead to defining a collection of task-level framework services that take care of all aspects of tool control and simulation data management and allow for an easy implementation of a variety of task-level applications such as sensitivity analysis and design of experiments, optimization, RSM extraction, and tool calibration.

The submission of process simulation tasks in the background or via telephone connections is now possible as the flow control module has been made completely independent from the visual user interface. This batch capability is used to perform automatic self tests of the TCAD shell together with all simulators and auxiliary tools.

## 1.2 Framework Service Layer

The service layer provides access to a set of high-level framework services based on VISTA's simulation flow representation [1]. It allows for the creation, modification, and execution of process flow instances, the submission of tasks for execution, the retrieval of responses, and the persistent storage of results. Automatic split generation and scheduling minimize the number of simulator runs required for iterative as well as parallel optimization techniques. Independent split branches are executed simultaneously over the network to quickly obtain results.

The service layer is implemented in VISTA's extension language VLISP, a superset of XLISP. An instance of a process flow together with its run-time data is called an *experiment* and is represented by a VLISP object. Table 1 gives examples of available services to create and manipulate experiments.

| Service | Description |
|---|---|
| `Define Experiment` | Defines experiment attributes, e.g., process flow, initial wafer, etc. |
| `New Experiment` | Creates new instance of existing experiment. |
| `Edit Step Parameter` | Modifies parameter values at step in process flow. |
| `Submit Experiment` | Requests execution of process flow or retrieves previously computed results. |
| `Inquire Step Data` | Returns responses, current wafer data, etc. |

Table 1: Examples of framework services to create and access experiments.

## 1.3 Framework – Optimizer Interface

When an optimization task is initiated by the framework, an *agent* is assigned to the optimizer tool, which establishes a connection between the task-level services and the optimizer. The agent is realized as a VLISP object. It takes care of passing messages between the optimizer and the framework by means of a callback-based, asynchronous connection, allowing for the execution of multiple optimization tasks at the same time. Figure 1 shows the interaction between the optimizer agent and the service layer on the one hand, and between the agent and the external optimizer on the other hand. The framework passes a description of the model to

the optimizer, defining the model's type and its control and response variables. During the course of the optimization, the optimizer requests the evaluation of the model for a certain set of control values by sending a message to the framework. Messages between the optimizer and the framework rely on VISTA's operating-system independent standard-input/standard-output redirection capabilities. Depending on the internal operation of the optimizer, evaluation requests may be sent synchronously, or a number of requests may be sent at a time. Upon termination of the optimization, the result found and diagnostic information are passed back to the framework.

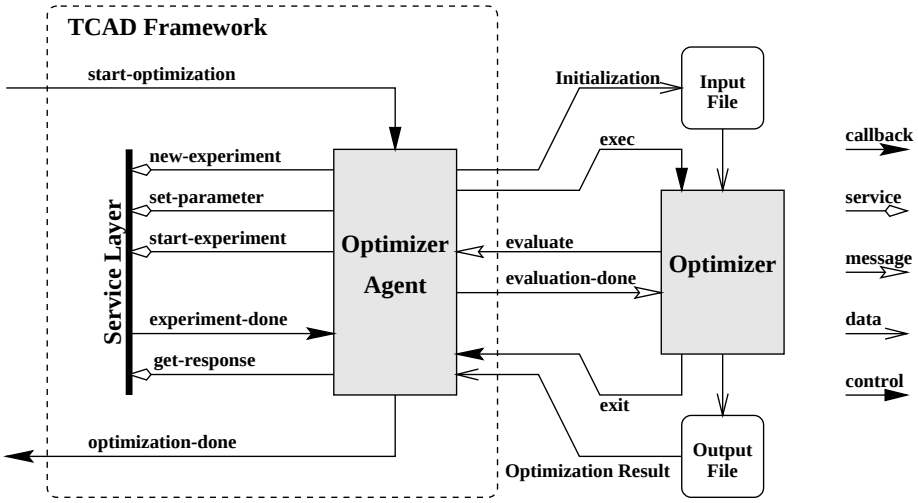



Figure 1: Communication between the service layer, the optimizer agent, and the external optimizer.

## 1.4 Self Tests

Any simulation flow can be executed in the background. The command `vsfc` is used to start the flow controller without any visual user interface connections. The following VLISP commands start the simulation of a flow `cmos035.sfe`:

```
(sfc::set-setting 'sfd-file #"cmos035.sfe")
(sfc::ui-start-task)
```

Callbacks can be registered at the termination of a simulation task and after every step of the simulation. For the self test, the output of a process flow simulation, i.e., the resulting PIF wafer model, is compared against a precomputed result, using `vmake`'s test capabilities. As the computation of a complete process flow involves complex interactions between a variety of tools, the successful termination of the self test is a good indicator for the real-life behavior of the TCAD framework.

# 2 Polygonal Geometry Reconstruction after Cellular Etching or Deposition Simulation

A new algorithm for the recalculation of a polygonal geometry representation after the computation of etching and deposition simulations based on a cellular geometry representation was developed. The purpose of that algorithm is to totally avoid any discretization errors in those parts of the geometry which were not affected by the surface movements resulting from the simulation.

## 2.1 Motivation

In two-dimensional process simulation, etching and deposition simulations are central steps. The thereby required surface advancement algorithms are often performed on a cellular geometry representation, e.g. [2]. During the simulation each of the cells contains one material type. Etching and deposition is modeled by changing the material type of some cells, leaving their geometric extensions unchanged.

Therefore it is necessary for each etching or deposition simulation step during the process simulation, to discretize the original polygonal geometry (OPG), run the simulation and recalculate a final polygonal geometry (FPG) representation. Former algorithms, e.g. [3], use only the final discrete geometry description to compute the FPG. Discretization errors occur all over the geometry, which demand regriding of every geometry conformal grid defined on the original geometry. In addition discretization errors of subsequent etching or deposition simulation steps might accumulate and under certain circumstances endanger the accuracy of the whole process simulation.

To minimize these problems a cellular algorithm was developed which generates the FPG by combining informations from the OPG, the original discrete geometry and the final discrete geometry. Fig. 2 and Fig. 3 show the differences in the flow of data during an etching or deposition simulation with the PROMIS – etch module between the new algorithm and the Marching Squares Algorithm.

The new algorithm totally avoids any discretization errors in those parts of the geometry which were not affected by the surface movements resulting from the simulation. Therefore the extensions of the cells giving the accuracy of the discretization must only be adjusted to the minimum extensions of the affected parts of the geometry. Structures much smaller than the resolution of the discretization will keep their original shape when they were not affected by the etching or deposition simulation.

## 2.2 The Algorithm

The computation of the FPG starts with a copy of the OPG: Firstly, in the main part of the algorithm, a provisional polygonal geometry is assembled by the following three steps which are performed on the cells of the discrete geometry:

1. **Classification:** Every cell of the discrete geometry is classified depending on the original and final materials of the cell itself and all it's neighboring cells. These have to be taken into

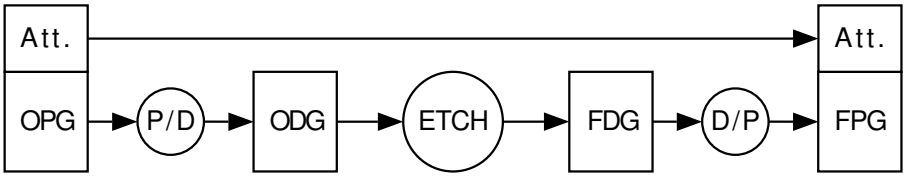



Figure 2: Flow of data in Marching Squares Algorithm

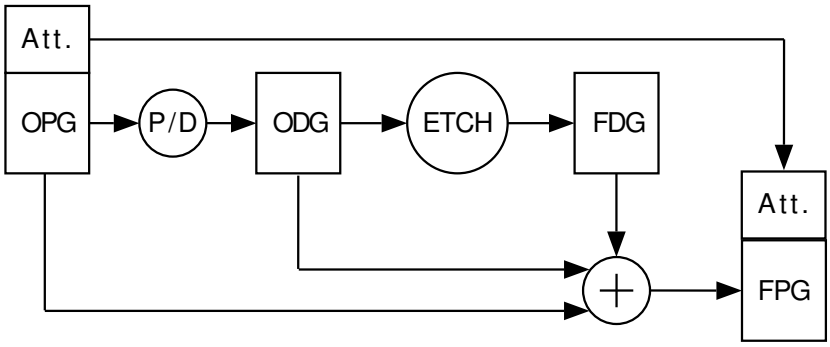



Figure 3: Flow of data for an improved algorithm

account because the regions of the OPG usually do not correspond to the borders of the cells. Five different types of cells are distinguished:

**Etched Cells:** a cell is classified as etched if the material changed to vacuum due to the simulation, or if it is originally vacuum and one of its neighboring cells changed to vacuum.

**Partially Etched Cells:** the original and final material of the cell is not vacuum and at least one of the neighboring cells changed to vacuum in the final discrete geometry. This category is necessary to describe accurately etching at etch stops.

**Deposited Cells:** the material changed from vacuum to the deposited material, or the cell was originally containing some material and one of its neighboring cells changed to the deposited material.

**Original Vacuum Cells:** the original and final material is vacuum and the cell is not classified as etched, partially etched or deposited before.

**Original Material Cells:** the original and final material is not vacuum and the cell is not classified as etched, partially etched or deposited before.

2. **Geometry-Extraction:** For etched, partially etched and deposited cells a polygonal description of the original geometry is computed. This description contains every part of the OPG which is located inside of the cell, informations about the material types inside and outside of the borders of the cell and the classifications of the cell and all its neighboring cells. (Fig. 2.2 – Fig. 2.2 show examples for this geometry extraction using quadratic cells.)

3. **Geometry-Correction:** the FPG inside of the recent cell is computed by modifying the polygonal description of the cell dependent on the classification of the cell:

   Fig. 4 shows a two-dimensional example for the construction of the FPG after an etching and deposition simulation. In Fig. 4a the discretization of th OPG is demonstrated. In this example the material of a cell is determined by the material type of the OPG at the center

of the cell. The resulting discrete and polygonal geometries after removing some cells by an etching simulation are presented in Fig. 4b. Fig. 4c shows the resulting discrete and polygonal geometries after the redeposition of material 1 on top of the geometry of Fig. 4b.

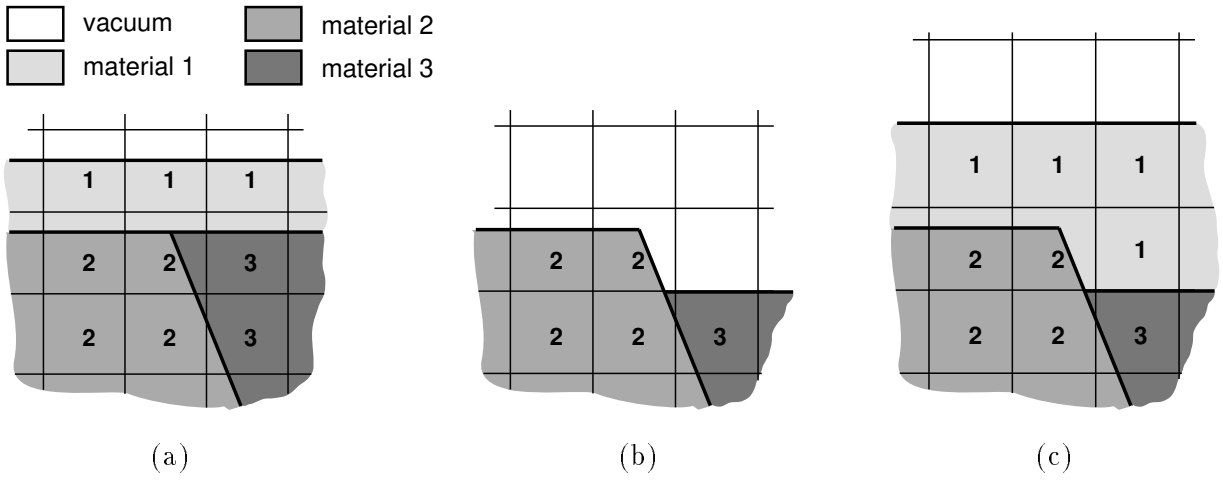


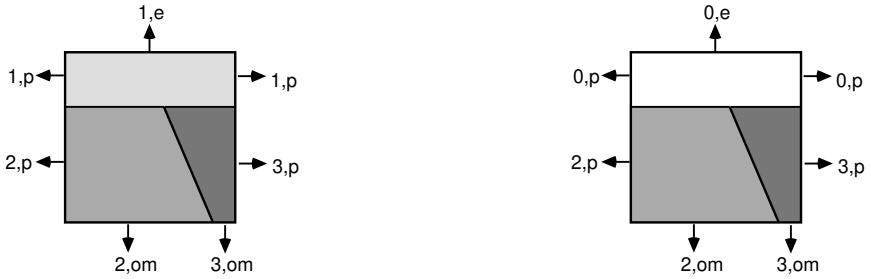

Figure 4: Discrete and polygonal geometries: (a) original, (b) etched, (c) redeposited

**Etched Cells:** Any region of the geometry which does not contain vacuum is removed from the provisional polygonal geometry.


(a) (b)

Extracted (a) and Corrected (b) Local Cell-Geometry for an etched cell

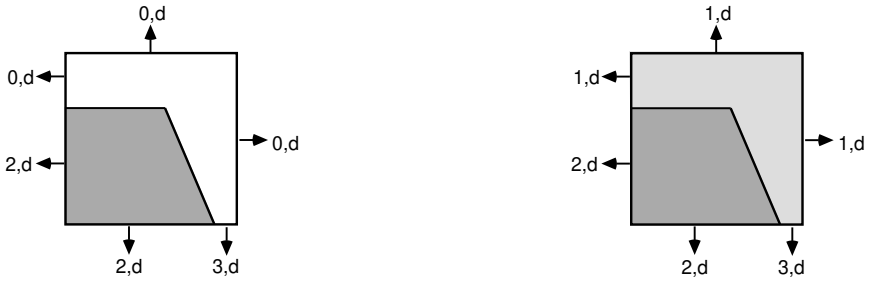
**Partially Etched Cells:** Every region of such a cell which is not containing the material itself and is bordering to an etched cell is removed from the provisional polygonal geometry. Remaining parts of the borders to neighboring cells which are classified as *Original Material Cell* or *Etched Cell* are added to the provisional polygonal geometry to ensure a consistent description of the geometry.




(a) (b)

Extracted (a) and Corrected (b) Local Cell-Geometry for a partially etched cell

**Deposited Cells:** Regions of the geometry containing vacuum are replaced by regions containing the deposited material and added to the provisional polygonal geometry like newly created borders to cells which were classified as *Original Material Cell* or *Original Vacuum Cell*.




(a) (b)

Extracted (a) and Corrected (b) Local Cell-Geometry for an deposited cell

The structure which is thereby created contains a high number of segments. Therefore in a second step the face structure is simplified and locally smoothed as far as it is not defined by parts of the OPG. The extent of reduction can be controlled, and the number of segments is often drastically reduced.

## 2.3 Examples

The performance of the new algorithm can be shown by comparing Fig. 5 to Fig. 7. Fig. 5 shows an example for an original polygonal geometry with some grid defined on a single segment. Fig. 6 shows the final polygonal geometry resulting from the Marching Squares Algorithm after an etching simulation. Due to discretization errors the extensions of the segment with the grid defined on it changed and regriding would be necessary to ensure correct representation of the grid in this segment. In Fig. 7 the final polygonal geometry obtained from the new algorithm after the same etching simulation is shown. The validity of the grid was preserved because that part of the geometry was not affected by the etching simulation.

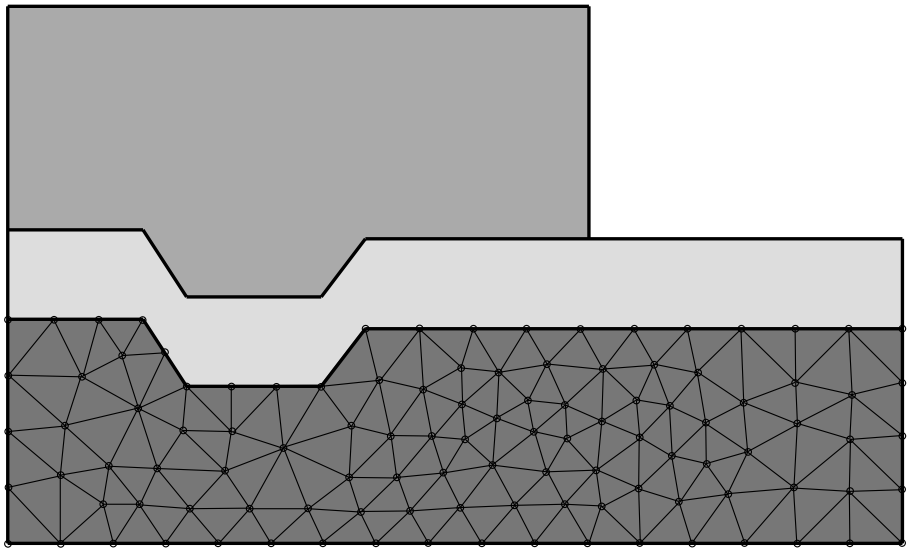


Figure 5: The Original Polygonal Geometry

## 2.4 Conclusions

The algorithm is highly independent of the dimension and shape of the discretization cells. Possible restrictions arise only out of numerical and algorithmic problems during the computation of the inner geometry of the cells. Therefore it is applicable to a large group of problems which require temporary conversions from polygonal to discrete geometry representations.

The increased computational effort of this new algorithm can be justified by considerable savings of calculation time in following regriding algorithms, because these have only to be applied in those parts of the geometry which actually changed during the etching or deposition simulation.

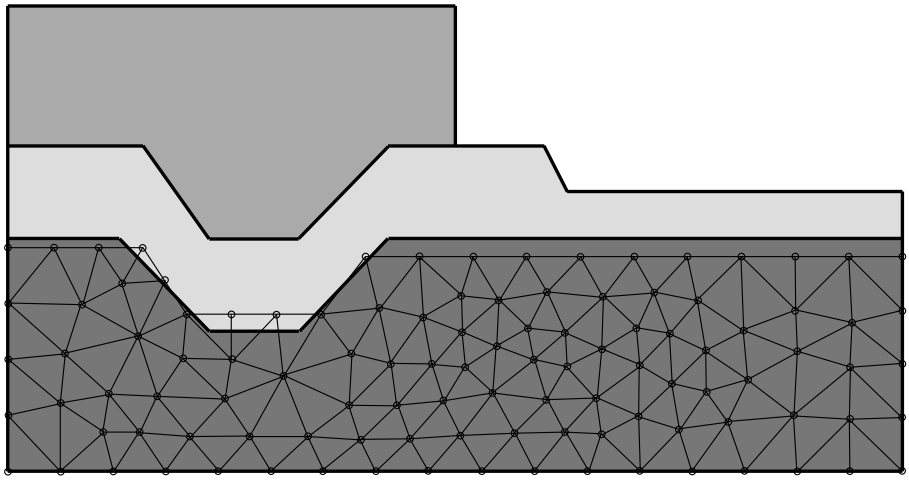


Figure 6: The Final Polygonal Geometry generated with MSQ - algorithm

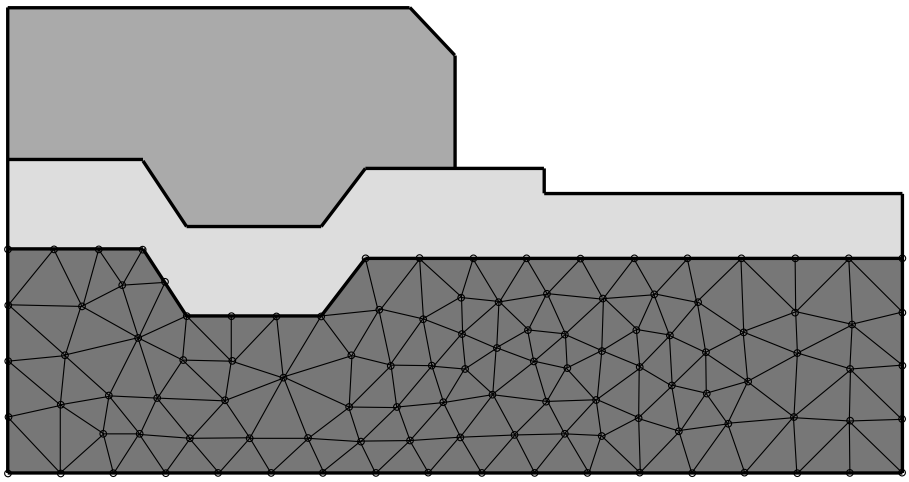


Figure 7: The Final Polygonal Geometry resulting from the new algorithm

# 3 Analytical Ion Implantation Model using the Four-Parameter Kappa Distribution Function

The already existing two-dimensional model for the analytical simulation of ion implantation into arbitrary geometries has been extended by a new distribution function. The four-parameter kappa distribution function was introduced the first time in semiconductor technology to describe the vertical dopant profile of implanted ions. Owing to the low computational effort and the short simulation time the given method is an alternative to modern Monte Carlo simulations for ion implantation processes. To handle any ion - any target implantations new range statistic data have been calculated from amorphous Monte Carlo simulations and stored in VISTA's material server data base.

## 3.1 The Analytical Ion Implantation Method

To describe the ion implantation profiles, a method based on distribution functions and their spatial moments is applied. For multilayer, non-planar structures we obtain the two-dimensional dopant profile by lateral convolution of a given vertical distribution function. Therefore the given simulation geometry is cut into slices which are arranged in the direction of the incoming ions (see Fig. 8). In each slice of the discretized geometry the vertical distribution function is initialized using the numerical range scaling method according to the different target materials [4]. To get the final concentration $C(x, y)$ at the spatial coordinates we add up the lateral and vertical distribution functions by

$$C(x,y) = N_d \cdot \int_{-\infty}^{+\infty} f_{vert}(x,\eta) \cdot f_{lat}(y-\eta, x) d\eta, \tag{1}$$

where $\eta$ is the lateral position and $N_d$ is the implantation dose. Due to our convolution method we would loose dopants at the given geometry boundaries, so the simulation geometry is extended to avoid this loss of dopants and to satisfy the Neumann boundary conditions at the geometry boundaries. Arbitrary tilt angles for the incoming ions can be handled with this slab method for the initialization of the vertical distribution function [5].

## 3.2 The Four-Parameter Kappa Distribution Function

There are several distribution functions to describe the vertical dopant profile. The Gaussian distribution or distributions using higher moments, such as the family of the Pearson distributions, are used to give an accurate fit to the implantation profiles. All these distributions are based on conventional central moments. In our approach described here we introduce the so-called "L-moments" the first time in semiconductor technology to specify statistical distributions.These L-moments are analogous to conventional central moments but can be estimated by linear combinations of order statistics. L-moments are able to characterize a wider range of distribution functions and are more robust to out-liners of the given data set than central moments. These L-moments can be defined in terms of probability weighted moments $\beta_r$ by a linear combination. The probability weighted moments of a given distribution are defined by

$$\beta_r = \int_0^1 x(F) \cdot F(x)^r dF, \tag{2}$$

where $x(F)$ is the quantile function and $F(x)$ is the cumulative distribution function. We use the four-parameter kappa distribution function as vertical distribution to fit the dopant profile,

because analytical formulations exist for $x(F)$ and $F(x)$ [6]. The four-parameter kappa distribution is a combination of the generalized logistic, generalized extreme-value and generalized Pareto distribution and is given by

$$f(x) = \alpha \cdot (1 - k \cdot (x - \xi)/\alpha)^{(1/k)-1} \cdot (F(x))^{1-h}, \tag{3}$$

where $\xi$ is a location parameter, $\alpha$ is a scaling parameter and $h, k$ are shape parameters. The cumulative distribution function $F(x)$ is given by

$$F(x) = \left(1 - h \cdot (1 - k \cdot (x - \xi)/\alpha)^{1/k}\right)^{1/h}. \tag{4}$$

The estimation of the parameters of the kappa distribution using L-moments requires a Newton-Raphson iteration method, because no explicit solution of the probability weighted moments for the kappa parameters is possible.

To demonstrate the accuracy of our analytical implantation model Fig. 9 shows the one-dimensional comparison of a Boron ion implantation profile with several distribution functions and the dopant profile computed by modern Monte Carlo simulation. Fig. 10 shows the two-dimensional result of a Phosphorus implantation at 70keV and 30 degrees tilt. Comparing our analytical results with Monte Carlo simulations (Fig. 11) we found good agreement. Due to the neglection of the reflected particles in the mask sidewall region, we obtain a lower concentration in the silicon substrate. But we get a more realistic dopant profile over the whole distribution range, where Monte Carlo simulation can only give accurate results within two or three orders of magnitude. Also the computational effort is very low compared to Monte Carlo simulations; the simulation time was reduced with the analytical method by a factor of 10 on a DEC-3000/400.

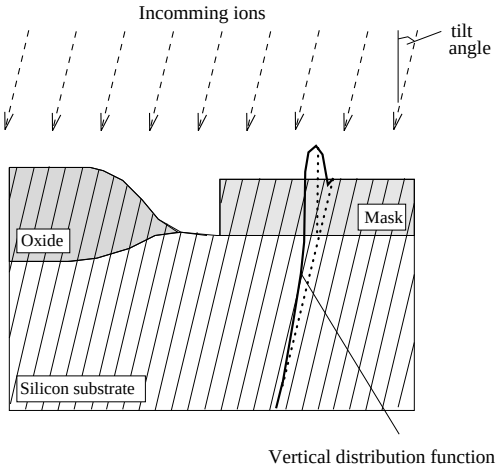



Figure 8: Discretization of the geometry using the slab method and the initialization of the vertical distribution function

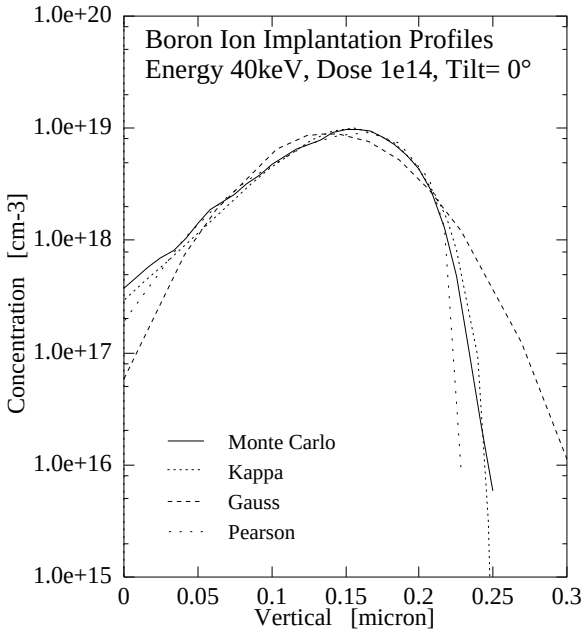



Figure 9: Comparison between analytical implantation profiles and Monte Carlo simulation

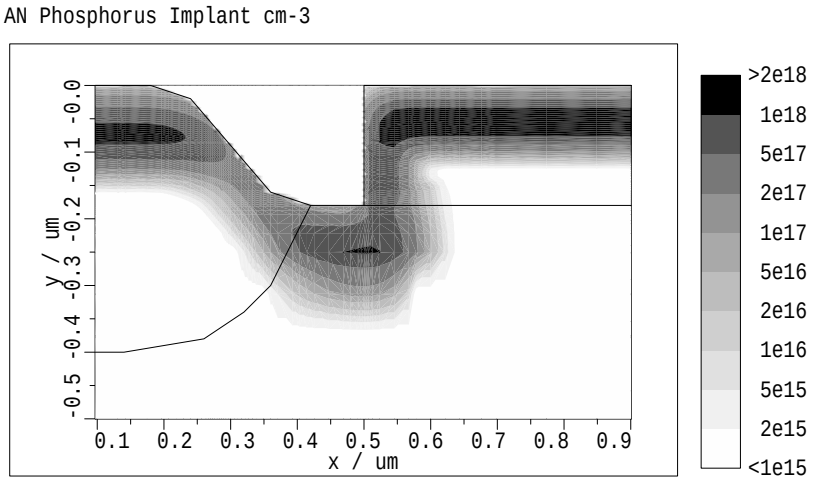



Figure 10: Two dimensional Phosphorus implantation profile with the analytical method (70 $keV$, $10^{13}$ cm$^{-2}$, +30° tilt)

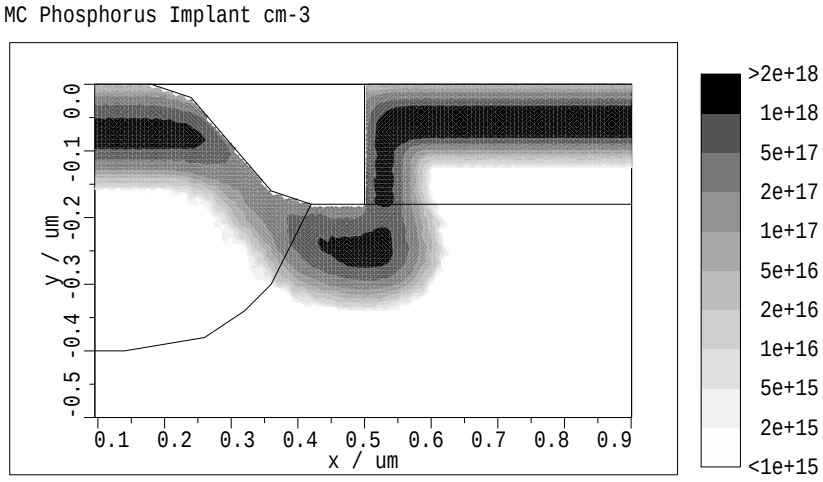



Figure 11: Two dimensional Phosphorus implantation profile with the Monte Carlo method (70 $keV$, $10^{13}$ cm$^{-2}$, +30° tilt)

# 4 VMAKE – A CASE-Oriented Configuration Management Utility

## 4.1 Introduction

The Viennese Make Utility (**VMake**) is based on a publicly available `LISP` interpreter written in `C`[7]. The interpreter is entirely platform-independent and runs currently on a number of *UNIX* systems and on *VMS*. **VMake** itself is written in `LISP` and supports, in addition to common make features, a number of CASE tasks like automatic code generation, version management (using RCS and CVS from a common repository), and automated high-level source code processing features, like language bindings between `C`, `LISP` and `FORTRAN` and extraction of reference manuals. **VMake** maintains automatically a private project file which contains up-to-date symbolic definitions of source code files, modules, libraries, language binding mechanisms, application executables, and all build targets. Dependencies between these objects are extracted from local description files or generated automatically from source code files. This enforces compact description files and allows for efficient management of large scale software projects.

Platform independence has been one of the major design requirements for **VMake**. Fig. 12 shows a list of currently supported computer architectures and operating systems. Only an `ANSI C` compiler to compile the `LISP` interpreter is required for porting **VMake** to another platform.

| computer | operating system |
|---|---|
| DEC AXP 3000,7600 | OpenVMS/AXP 6.1 |
| DEC AXP 3000,7600 | OSF 2.1, OSF 3.0 |
| Apollo DN1000 | Apollo DOMAIN 10.3 |
| Decstation 3000,5000 | Ultrix 4.2, Ultrix 4.3 |
| HP/Apollo 9000/700 | HP/UX 8.05, HP/UX 9.0 |
| IBM RS6000 | AIX 3.1, AIX 3.2 |
| PC 386, PC 486 | Interactive Unix 4.0 |
| PC 386, PC 486 | Linux 1.1.54 + AT&T f2c |
| Sparc Station | SunOS 4.1 |
| Sparc Station | Solaris 2.3 |
| Sony RISC | Sony NEWS 4.2 |
| VAX, VAXstation | OpenVMS/VAX 6.1 |

Figure 12: Platforms currently supported by **VMake**

Similar to imake, **VMake** employs a small number of standardized higher–level rules to reduce the complexity of local module description files, but overcomes the aforementioned insufficiencies by maintaining all global project information automatically. In a **VMake**–internal global context file, the time of the last modification of every local description file is stored (in addition to the timestamps of all managed source code files) and updated automatically. Changes to the local description files are recognized and the (partial) regeneration of the dependency information is started automatically. Fig. 13 shows a typical **VMake** description file. Since **VMake** is based on `LISP`, the syntax chosen is a subset of `LISP` so that the `LISP` reader can be used for parsing.

```
; this defines a name for the directory
(Module-Directory My-dir)

;; compile main source file
(CC-Target My-C-main
           :source "mymain.c")
;; compile library objects
(CC-Target My-C-objects
           :source "my1.c" "my2.c" "my3.c")
;; build library
(Library-Target My-C-library
                :libname "my"
                :objects My-C-objects
                :archive)
;; generate program
(Program-Target My-C-program
                :progname "myprog"
                :objects My-C-main
                :libraries My-C-library
                :language C)
```

Figure 13: Example of a **VMake** description file

## 4.2 Software Installation, Release/Patch Generation and Version Management

To build a binary software release, all modules of a project must be installed under an installation directory. As **VMake** knows all global include files, public libraries, and executable programs it can automatically put them into respective installation directories. Only for additional installable, otherwise unmanaged files (like `README` files and `LISP` sources) a dedicated installation directory must be specified in the module description file.

**VMake** supports source code level releases and patches between releases. The basic process is similar to the software installation, but a full second instance of the managed source code is created, which can then be packed for shipping.

**VMake** supports the *Concurrent Version System*[8], a public–domain version management system based on RCS. **VMake** reads CVS' special files and upon request, prints lists of all source files modified with respect to the repository, of all files not currently checked in, and of all files under control of CVS but not known to **VMake**. This automatism helps to detect and avoid version/configuration management inconsistencies in an early stage of the production process (i.e. before the test phase). **VMake** also checks whether project modules make proper use of the global error system which accesses the version management information to track down errors occurring during module and project test and use.

## 4.3 CASE operations – The Tool Abstraction Concept

**VMake** uses a *Tool Abstraction Concept* (**TAC**) for generating language bindings of functional modules and constants for different programming languages than the implementation language of

the functions. Currently, bindings can be generated from `C` to `FORTRAN`, from `FORTRAN` to `C`, and from `C` to `LISP`. The automatic support of multi–language programming has proven valuable for two reasons. First, writing the required stub code manually is a tedious and error–prone task and secondly, multi–language interfaces between compiled languages are highly system–dependent.

The **TAC** module of **VMake** scans the source code file and extracts information from the function definitions and special formal comments, as depicted in Fig. 14. The comment `/***TF` starts the definition of a **T**AC–able **F**unction.

```
/***TF counts the number of occurences of a character
       within a string. The start and end of the search range can
       be specified to simplify substring operations. */
/***R myStrChar myStrReverseChar */
int /* [:not-ok 0] */
  myStrCount(  char *str,       /*  [IN] input string to search */
               char ch,         /*  [I] character to search for */
               int start,       /*  [I :opt :key :default 0]
                                    start index for search */
               int end)         /*  [I :opt :key :default strlen(str)]
                                    end index for search */

  /* implementation of function */
```

Figure 14: **TAC** documented function

The comments after the function arguments consist of a formal description of the argument characteristics and a textual documentation part. (The comments are also used by the documentation extraction facility.) In the example in Fig. 14, all parameters are used as input `[I]` and the string may be given as `NULL` pointer `[IN]`. The rule `Module-Directory` defines the required module prefix for all functions of the current module. This module prefix must be unique within the project, it is used to identify the module affiliation of a given function. To bind the function `myStrCount` to another language, the definition

```
(Module-Directory MyModule
                  :prefix "my")
(Define-TAC-Target TAC-module
                   :files "mysrc.c"  ; source file of function
                   :source-domain C)
```

is used in the description file of the module implementation for exporting the function. To generate `LISP` bindings for the `C` function `myStrCount` (which is part of the module `"my"`) somewhere else in the project tree the rule

```
(Create-TAC-Interface TAC-LISP-Interface
                      :modules "my"
                      :target-domain LISP)
```

has to be used in the description file where the language bindings shall be generated. All the **TAC** information that has been extracted by **VMake** is tied to the symbolic name and prefix of

the module. Once defined, this information can be used for the generation of multiple language binding interfaces.

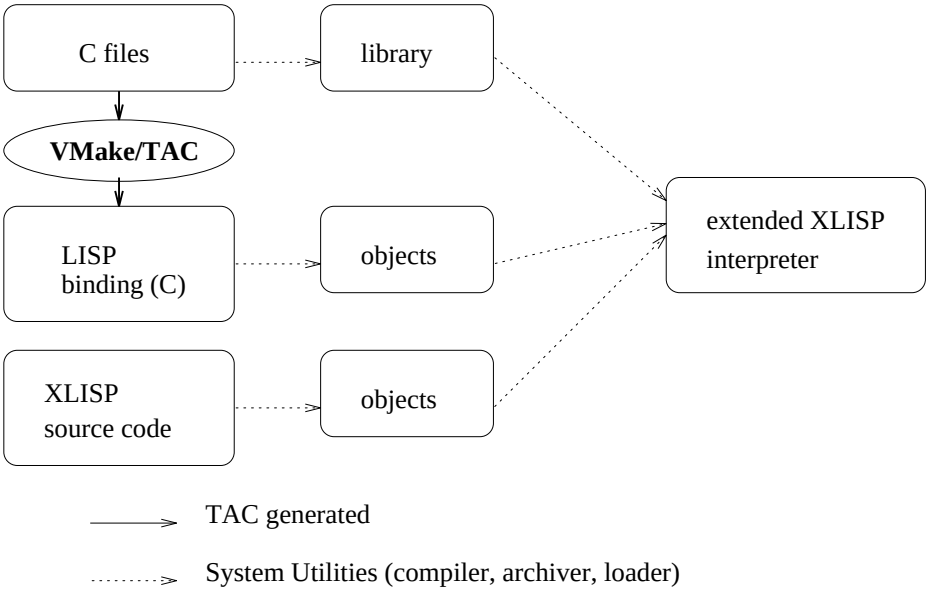



Figure 15: **TAC** used for `LISP` binding

The **TAC** is also used for the extraction of reference manuals from the source code. A function documented with a `/***F` comment is parsed by **VMake** and a LaTeX reference manual entry for that function is generated (shown in Fig. 16).

---

`myStrCount` C-Function

---

**Usage:**

```
int myStrCount(str, ch, start, end);
```

| | | |
|---|---|---|
| input | `char *str` | input string to search |
| input | `char ch` | character to search for |
| input | `int start` | start index for search |
| input | `int end` | end index for search |

`myStrCount` counts the number of occurences of a character within a string. The start and end of the search range can be specified to simplify substring operations.

**References:**

| | |
|---|---|
| `myStrChar` | 6 |
| `myStrReverseChar` | 12 |

Figure 16: **TAC** extracted documentation

## References

[1] Ch. Pichler and S. Selberherr. Process Flow Representation within the VISTA Framework. In S. Selberherr, H. Stippel, and E. Strasser, editors, *Simulation of Semiconductor Devices and Processes*, volume 5, pp 25–28. Springer, 1993.

[2] E. Strasser, G. Schrom, K. Wimmer, and S. Selberherr. Accurate Simulation of Pattern Transfer Processes Using Minkowski Operations. *IEICE Trans.Electronics*, E77-C:92–97, 1994.

[3] W.E. Lorenson and H.E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics*, 21:163–169, 1987.

[4] H. Ryssel. Implantation and Diffusion Models for Process Simulation. In K.M. DeMeyer, editor, *VLSI Process and Device Modeling*, pp 1–41. Katholieke Universiteit Leuven, 1983.

[5] K. Wimmer. *Two-Dimensional Nonplanar Process Simulation.* Dissertation, Technische Universität Wien, 1993.

[6] J.R.M. Hosking. The Four-Parameter Kappa Distribution. *IBM Journal of Research and Development*, 38(3):251–258, 1994.

[7] D.M. Betz. *XLISP: An Object-Oriented Lisp, Version 2.1*, 1989.

[8] D. Grune, B. Berliner, and J. Polk. *CVS — Concurrent Versions System.* Free Software Foundation, Cambridge, Mass., 1992. Manual Page.