ICEE2006 14th Iranian Conference on Electrical Engineering

High Performance Process and Device Simulation with a Generic Environment

René Heinzl[△], Michael Spevak[°], Philipp Schwaha[△], Tibor Grasser[△]

[△]Christian Doppler Laboratory for TCAD in Microelectronics
at the Institute for Microelectronics

[°]Institute for Microelectronics, Technical University Vienna,
Gußhausstraße 27-29/E360, A-1040 Vienna, Austria
E-mail: {heinzl|spevak|schwaha|grasser}@iue.tuwien.ac.at

Abstract

A high performance generic environment for TCAD and general scientific computing is presented that imposes no restrictions on geometry, topology or discretization schemes, so that equations and even complete models can easily be implemented. The generic programming approach with the corresponding base concepts, and the applicability in the area of TCAD with the performance behavior is presented.

1. Introduction

The scientific computing approach is used to gain an understanding of scientific and engineering problems by the analysis of mathematical models implemented in computer programs and solved by numerical solution techniques. Numerical methods employed in discretization, interpolation, or optimization make use of highly nonlinear functions $f_x(\mathbf{x})$ which can consist of several coupled differential equations. Due to the great diversity of physical phenomena, which can be described by differential equations of considerable complexity, present in different areas of scientific computing, the development of several discretization schemes [1] has been necessary in order to best model the underlying physics and to accommodate the mathematical peculiarities of each of these equations while transferring them to the discrete world of digital computing. As a consequence the development of simulation software is quite challenging, especially with high performance requirements.

In the field of TCAD, as in many others, the numerical simulation results are based on different discretization schemes such as finite differences, finite elements, and finite volumes. Each of these schemes has its merits and shortcomings and is therefore more or less suited for different classes of equations. All of these methods have in common that they require a proper tessellation and adaption of the simulation domain [2, 3], so-called (unstructured) meshes or (structured) grids.

To deal with all of these issues, our institute has developed different simulation environments, libraries, and applications in the last decade such as the Wafer-State-Server [4], STAP [5], Minimos-NT [6], FEDOS [7]. A more detailed overview of these simulators is given in [8].

2. Generic Programming

Generic programming extends the interface capabilities of the object oriented programming approach in the direction to concept based programming. With its parametric polymorphism, the data type [9] must support the required concepts only. The main focus of programming with concepts does not lie in the actual source code. This programming paradigm focuses on the development of concepts, which means that the user can easily add new data types or classes, which must only support the required concepts. In C++ parametric polymorphism is provided by the template mechanism which offers an overall high performance due to separate optimization possibilities. Therewith, the generic programming paradigm establishes homogeneous interfaces between algorithms and data structures without subtyping polymorphism (class hierarchies).

Within our approach, three different basic concept have been separated. The **topology**, which means the study of manifolds and their embeddings like structured or unstructured topologies whereas the **geometry** means the properties of configurations of geometric objects, e.g. cylinder coordinates for a structured topology. Finally, the **quantity** extends the geometry concept to all kinds of attributes or properties, which can be attached to objects (topological or geometrical).

To reflect these concepts, the implementation is heavily based on the type parameterization, which means the static or parametric polymorphism of generic programming. In our environment, there are different levels of type parameterization:

- Objects (vertex, cell)
  All different kinds of vertices (structured, unstructured) are implemented with only one generic vertex class. With the specialization mechanism, the compiler generates the suitable vertex class based on the base type of topology. Therewith the code maintenance is reduced to a minimum.
- Iterators (derived iterators)
  We have developed a complete iterator hierarchy. On the one hand side, there are different iterator categories (forward, bidirectional, random access). On the other side there are only two important iterators for each base topology. The cell_on_vertex and the vertex_on_cell iterator. All other iterators can be derived from these two by the compiler.
- Handles, geometrical coordinates, quantity storage
  The handles are parameterized with the counter data type, whereas the geometry is parameterized with the numeric type of coordinate storage. The quantity storage is parameterized with the numeric type for quantity element storage.

Next to the parametrization approach, functional programming techniques are available in C++ such as Lambda (unnamed functions [10]) and Currying (partial function evaluation [11]). To highlight the behavior of the functional programming paradigm, we present an example based on the C++ STL: the usage of generic algorithms on arbitrary data structures. The next sample demonstrates the output from a generic container:

```
std::vector<int> data;
copy(data.begin(), data.end(),
     ostream_iterator<int>(std::cout, "\n"));
```

The complexity is hidden in the behavior of the third argument to the generic copy algorithm. With the functional approach we can write things more intuitively:

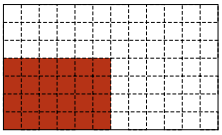
```
for_each(data.begin(), data.end(),
                std::cout << arg1 << '\n');
```

In the next section we present the applicability of the generic and functional programming in the field of scientific computing.

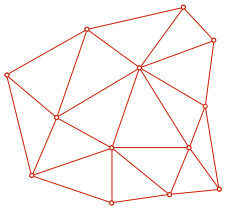
2.1. Concepts for Topology

Most of the used topologies in TCAD and in scientific computing are based on two basic types of topology:

Structured topology: A widespread approach to spatial discretization is to divide the simulation domain into a structured assembly of quadrilateral cells, with the topological information being apparent from the fact that each interior vertex has exactly the same number of neighboring cells. This kind of discretization is called *structured grid* or simply *grid*. The next figure represents an example of this type of mesh.



Unstructured topology: The alternative approach is to divide the computational domain into an unstructured assembly of more or less arbitrarily formed cells. The topological information can not be deduced implicitly from the elements and has to be stored explicitly. This kind of discretization is called *unstructured mesh* or simply *mesh* and can be seen in the next figure for the two dimensional case.



2.2. Concepts for Quantities

Due to the handling of a large number of different quantities, we have developed different concepts for all different kinds of quantities. These quantities need to be stored on various objects (vertices, edges, facets, cells). On the one side, we have developed a completely generic quantity library which is capable of storing various mathematical structures in a dense and a sparse format: scalar values, vector values, matrices, and tensors. On the other side, we have developed specializations of all of these mathematical structures to provide high performance calculations, for instance for small fixed size vectors and matrices.

3. Applicability of our Approach

To demonstrate the applicability of our approach we briefly present some examples. We want to emphasize the coupling of different concepts used in our approach to support simple and robust software development in the area of scientific computing.

Support for several spatial dimensions is inherently included in a way that programs can be written independently of the space dimension without unreasonable penalties on run-time and memory consumption. The key feature to achieve this abstraction is the iterator/cursor concept, which can be used like this:

```
vertex_cursor vcs = segment.vertex_begin();
while (vcs.valid())
{
  segment.store_quantity(*vcs,
          quantity_name, value);
  ++vcs;
}
```

The property or quantity treatment is accessible in an abstract way through function objects. The next few code lines print all quantities found on vertices. Note the dimensional and topological independence:

```
std::string key_pot = "BuiltInPotential";
quan_t pot_quan =
       scalar_quan(*segit, key_pot);

for_each((*segit).vertex_begin(),
       (*segit).vertex_end(),
       std::cout << pot_quan << std::endl );
```

With this approach, a very high level of abstraction from all different kinds of data structures and functionality can be provided. Algorithms can be specified in a dimension independent and data structure neutral way. This enables the opportunity for a mathematical layer where functions can be specified in C++ directly. To show the applicability of this approach we investigate the Laplace equation:

$$\sum_{\text{edge vertex}} (\Psi_j - \Psi_i) \frac{A_{ij}}{d_{ij}} = 0$$

where i stands for the i-th vertex and j for the adjacent vertex traversed over the corresponding edge. The equation can be specified in the following way in C++ with the support or our approach:

Laplace finite volume discretization

```
for (vit = container.vertex_begin();
     vit != container.vertex_end(); ++vit)
{
 equation =
  ( sum<vertex_edge>
   [
     diff<edge_vertex>[pot]
   ] * A / d
  ) (*vit);
}
```

The complete equation is examined at compile-time to construct an optimized means to obtain the needed data. The complex resulting from this mapping is completed by specifying the currently iterated vertex at run-time. In other words, the compiler will always generate the most suitable code for each topology, dimension, and equation. Therewith an overall high performance can be achieved. This algorithm can be used for 1D, 2D, and 3D structured and unstructured meshes.

4. Examples and Results

In this section two different examples are analyzed. First, an interconnect simulation is demonstrated where the current distribution within an interconnect line creates a selfheating effect. This simulation is based on a finite element discretization of the Laplace equation. Secondly, we analyze automatic mesh adaptation by an a posteriori error estimators, which is a major issue in process and device simulation. The error estimation techniques are presented in detail in [12].

4.1.Interconnect Simulation

The temperature distribution caused by self-heating effects in an interconnect line is depicted in Figure 1. The equations used are given in [13, 14]. To illustrate the influence of the problem size on the overall system matrix assembly step, we investigate our test structure with different refined meshes. The run-time performance is compared to one of our fastest simulation engines available [5]. As a result, no relevant run-time (Figure 2) as well as solution differences were obtained.

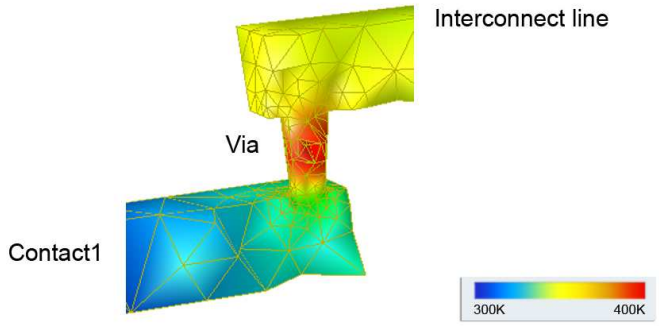


Figure 1. Temperature distribution due to self-heating in a tapered interconnect line with cylindrical vias.

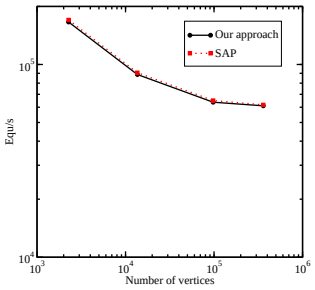


Figure 2. Comparison of our approach with a highly specialized finite element package on uniformly refined meshes.

4.2.Adaptive Mesh Generation

We use an simple structure (Figure 3) to demonstrate the error estimation and adaptive meshing procedure. Therefore the Laplace equation is solved in the SiO_2 layer around the contacts. The main aim of this simulation is to determine the capacitance between the two electrodes both very precisely and efficiently. To calculate the complete example a solver interface [15], as well as a realtime visualization interface [16] is implemented in our generic environment. We start with an initial mesh and increase the point density in regions where an error estimator returns high values. Then we solve the Laplace equation again and re-estimate the error on the new structure. We do this iteratively until we have only elements with an error below a certain error bound.

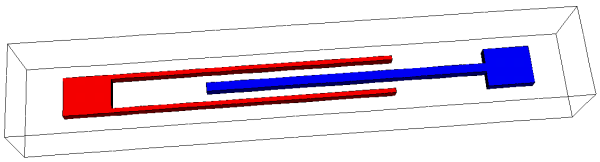


Figure 3. The initial structure with two contacts (blue: 1 V, red: 0 V) and surrounding oxide (transparent)

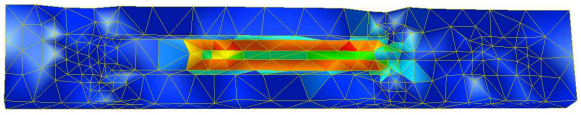


Figure 4. Initial calculation with high error values (red) and low error values (blue) for the example structure with 1000 points.

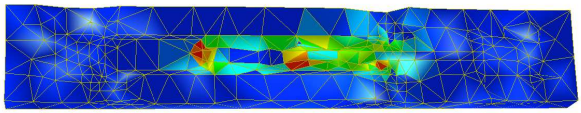
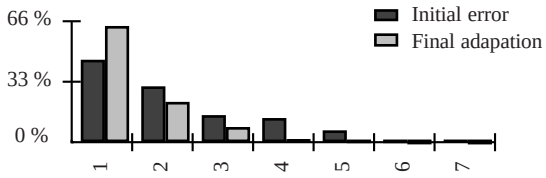


Figure 5. Error distribution after two steps calculation for the example structure with 1500 points.

The next diagram presents the number of elements in each error class, whereas error class 1 stands for a small error and error class 7 for a big error. As can be clearly observed, the number of elements in error class 1 increases and the number of elements in the other error classes drops due to our mesh adaptation technique.



5. Runtime Efficiency

To investigate the abstraction penalty of our generic code we analyze a simple C implementation without any generic overhead to our generic environment. The abstraction penalty means the ratio of the execution time resulting from code which takes advantage of certain abstractions, and the execution time obtained from a lower-level code which has equivalent functionality but does not use the abstractions. The next code snippet presents the C source code for a three dimensional object traversion:

C approach for a simple traversal over all vertices

```
for (i3 = 0; i3 < sized3; i3++)
  for (i2 = 0; i2 < sized2; i2++)
    for (i1 = 0; i1 < sized1; i1++)
    {
       //operations
       // use i1, i2, i3
    }
```

Our approach for the traversal over all vertices

```
vit1 = container.vertex_begin();
vit2 = container.vertex_end();
for (;vit1 != vit2; ++vit1) {
  //operations
  // use *vit1
}
```

The following figures present the run-time behavior of the compared traversion operations on different currently used computer systems. As can be seen, there is no abstraction penalty of generic programming in C++ at all.

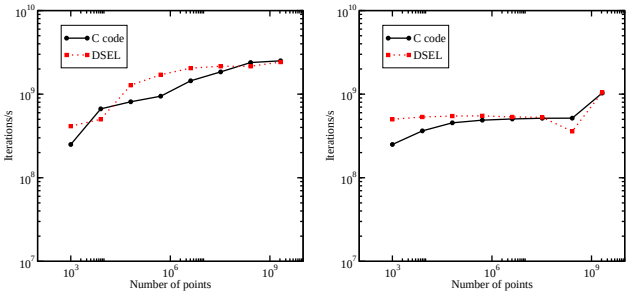


Figure 6. Vertex traversal executed on a P4 (left) and AMD (right).

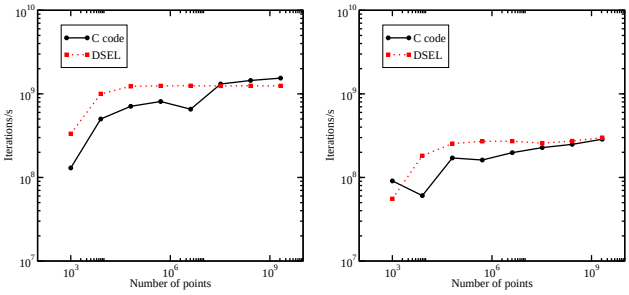


Figure 7. Vertex traversal executed on a G5 (left) and IBM (right).

6. Conclusion

We have presented a generic environment suitable for all kinds of simulations, especially for the numerical solution of PDEs. We have presented the main paradigms, the generic programming as well the functional programming approach, the applicability and the runtime efficiency. Due to the shift of most of the calculations to compile time, runtime performance is excellent. With our approach it is possible to build scientific applications with few lines of code.

7.References

[1] S. Selberherr, *Analysis and Simulation of Semiconductor Devices*. Wien–New York: Springer, 1984.

[2] R. Heinzl and T. Grasser, “Generalized Comprehensive Approach for Robust Three-Dimensional Mesh Generation for TCAD,” in *Proc. SISPAD*, (Tokyo, Japan), pp. 211–214, Sept. 2005.

[3] P. Schwaha, R. Heinzl, M. Spevak, and T. Grasser, “Coupling Three-Dimensional Mesh Adaptation with an A Posteriori Error Estimator,” in *Proc. SISPAD*, (Tokyo, Japan), pp. 235 – 238, Sept. 2005.

[4] A. Hössinger, R. Minixhofer, and S. Selberherr, “Full Three-Dimensional Analysis of a Non-Volatile Memory Cell,” in *Proc. SISPAD*, (Munich, Germany), pp. 129–132, Sept. 2004.

[5] R. Sabelka and S. Selberherr, “A Finite Element Simulator for Three-Dimensional Analysis of Interconnect Structures,” *Microelectronics Journal*, vol. 32, no. 2, pp. 163–171, 2001.

[6] IμE, *MINIMOS-NT 2.1 User’s Guide*. Institut für Mikroelektronik, Technische Universität Wien, Austria, 2004. http://www.iue.tuwien.ac.at/software/minimos-nt.

[7] H. Ceric, *Numerical Techniques in Interconnect and Process Simulation*. Dissertation, Technische Universität Wien, 2004.

[8] R. Heinzl, P. Schwaha, M. Spevak, and T. Grasser, “Multi-dimensional Process and Device Simulation with a Generic Scientific Simulation Environment,” in *6th IEEE International Caribbean Conference on Devices, Circuits and Systems Proceedings",(2006)*, (Playa del Carmen, Mexico), April, 26-28 2006.

[9] J. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[10] C. Böhm, ed., *Lambda-Calculus and Computer Science Theory, Proceedings of the Symposium Held in Rome, March 25-27, 1975*, vol. 37 of *Lecture Notes in Computer Science*, Springer, 1975.

[11] M. Sperber and P. Thiemann, “Realistic Compilation by Partial Evaluation.,” in *PLDI*, pp. 206–214, 1996.

[12] R. Heinzl, P. Schwaha, M. Spevak, and T. Grasser, “"Adaptive Mesh Generation for TCAD with Guaranteed Error Bounds",” in *The 2005 European Simulation and Modelling Conference Proceedings",(2005)*, (Porto, Portugal), pp. 425 – 429, October 24-26 2005.

[13] R. Bauer, *Numerische Berechnung von Kapazitäten in dreidimensionalen Verdrahtungsstrukturen*. Dissertation, Technische Universität Wien, 1994.

[14] R. Sabelka, *Dreidimensionale Finite Elemente Simulation von Verdrahtungsstrukturen auf Integrierten Schaltungen*. Dissertation, Technische Universität Wien, 2001.

[15] S. Wagner, T. Grasser, and S. Selberherr, “Performance Evaluation of Linear Solvers Employed for Semiconductor Device Simulation,” in *Proc. SISPAD*, (Munich, Germany), pp. 351–354, Sept. 2004.

[16] IBM Corporation, Yorktown Heights, NY, USA, *IBM Visualization Data Explorer*, third ed., Feb. 1993.