Monte-Carlo Simulation of GaAs Devices Using High Generality Object-Oriented Code and Encapsulated Scattering Tables*

Jason Harris and Dragica Vasileska

Department of Electrical Engineering, Arizona State University
Tempe, AZ 85287-5706, USA
smeger@geekspiff.com, vasilesk@imap2.asu.edu
http://www.geekspiff.com

ABSTRACT

Many factors are responsible for the continued shift in industry prototyping preference from device fabrication to device simulation. One of the incentives for this transition is the generality and flexibility gained by avoiding a physical prototype. Unfortunately, much of the existing simulation software was written to solve a specific problem and therefore lacks flexibility. To combat this problem, we have developed an ensemble Monte-Carlo particle-based simulator for performing device simulations with a high degree of generality. Much of the generality comes simply from using a dedicated object-oriented coding philosophy in which implementation is separated from interface as much as possible. The use of 'smart' objects also increases the flexibility of this device simulator. Finally, the simulation software uses recursively encapsulated scattering tables, allowing a reduction in memory requirements and/or processing speed when simulating devices with complicated doping profiles.

Keywords: Monte Carlo, device simulation, object-oriented programming, non-uniform doping

1 INTRODUCTION

In this paper, we report our progress in the development of an ensemble Monte Carlo particle-based simulator for modeling semiconductor devices fabricated in both silicon and GaAs technology. The code was written entirely in ANSII C++ and should be portable across all computer architectures. It has been tested on computers running x386 Debian Linux, Windows NT and MacOS.

A brief functional overview of a generic particle-based simulator is given in section 2. In section 3, we give an extended structural overview of the simulator internals and a description of the various code objects that we use to enhance its flexibility. In this section, we also describe a new method of using encapsulated scattering tables to speed computation and reduce memory requirements in devices with non-uniform doping profiles. Finally, in section 4, we show simulation results for the electric field dependence of the electron drift velocity and the doping dependence of the electron mobility in bulk GaAs materials.

2 FUNCTIONAL OVERVIEW

A generic flow-chart of a particle-based simulator is shown in Figure 1. The transport kernel of the simulator uses a standard Monte Carlo algorithm to obtain the solution of the Boltzmann transport equation. More precisely, carriers are treated as classical particles until a statistically determined time has elapsed, at which point they undergo an instantaneous quantum mechanical scattering event. The mean intervals between these scattering events are stored in a table indexed by the type of scattering event and the carrier's energy [1,2].

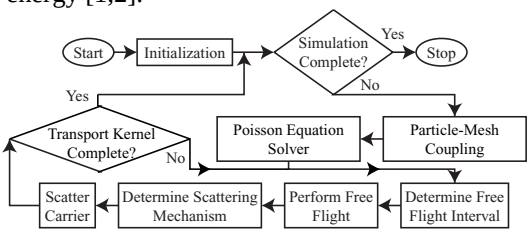

Figure 1: Device Simulation Flowchart

Initialization consists of specifying the device geometry and material parameters, computing the scattering tables, placing the carriers in random locations throughout the device (based on the doping density or using equilibrium Poisson equation results), and randomizing their energies so that the mean energy is thermal. The execution loop consists of determining a statistically likely free-flight interval for each carrier, treating the carrier classically during that interval, determining a statistically likely scattering mechanism for that carrier, and scattering the carrier. During this process, measurements are performed periodically on the system. Once some specified criterion has been met, simulation terminates.

3 STRUCTURAL OVERVIEW

The structure of our code provides a great degree of the simulator's flexibility and distinguishes it from similar device simulators. Throughout, a strict object-oriented philosophy was used to separate interface from implementation. In addition, reference-counted 'smart' pointers are used extensively for automated trash collection and to simplify both memory management and code legibility. These smart pointers are similar to the `autoptr` type in the C++ specification, but they handle polymorphism better and are generally more advanced.

A variety of generic objects are used to perform the particle-based simulation. These consist of encapsulated scattering tables, meshes, carriers and carrier pools, measurement methods, and a Poisson solver. In keeping with the object-oriented design approach, the device simulator has no knowledge of the inner workings, or indeed, the contents of any of these objects.

3.1 Encapsulated Scattering Tables

Encapsulated scattering tables consist of a series of infinitely-nestable conventional scattering tables which can be used to speed simulation and reduce the memory requirements of modeling devices with complicated doping profiles.

Three C++ classes are used to implement the encapsulated scattering tables (see Figure 2). The `ScatterTable` class is used to store an entire encapsulation. A `ScatterTable` object provides a high level interface to the rest of the simulator and contains attributes common to the entire scattering table, such as the effective mass and non-parabolicity factor. The `ScatterTable` object also contains a single `MetaMechanism`.

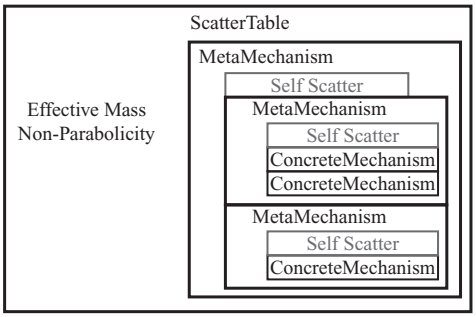

Figure 2: Encapsulated Scattering Table Class Diagram.

A `MetaMechanism` object provides a high-level interface to a collection of scattering mechanisms. These embedded scattering mechanisms can be either additional `MetaMechanisms` or `ConcreteMechanisms`. The `MetaMechanism` provides methods for inserting and removing embedded scattering mechanisms, as well as methods for renormalizing the embedded tables and for performing a scattering operation on a carrier.

A `ConcreteMechanism` is an abstract base class that defines the interface to an actual scattering mechanism. The actual scattering mechanisms descend from `ConcreteMechanism` objects and provide a method to compute the scattering rate for a given energy and a method to scatter an individual carrier.

For the most part, encapsulated scattering tables work identically to conventional scattering tables. A fictitious self-scattering mechanism is added to a collection of scattering mechanisms so that the total scattering rate is a constant [1]. This facilitates the computation of the mean time between scattering events.

Since the `MetaMechanism` object was designed to operate efficiently, insertion and removal of fictitious self-scattering mechanisms was made automatic. A self-scattering mechanism is placed at the beginning of any `MetaMechanism` that contains one or more `ConcreteMechanism` objects. If embedded tables are inserted or removed from a `ConcreteMechanism` object during the simulation, a self-scattering mechanism will be inserted or removed if necessary (the gray regions in Figure 2).

The `MetaMechanism` object also keeps track of the portions of the embedded table that need normalizing so that if embedded tables are inserted or removed during simulation, only the necessary portions of the encapsulation are renormalized. For example, Figure 3 shows the `ScatterTable` of Figure 2 after the insertion of a third `MetaMechanism`. The indicated self-scattering mechanism is the only portion of the `ScatterTable` that needs to be renormalized after this insertion.

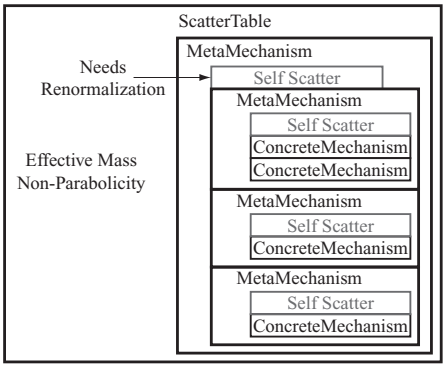

Figure 3: Self-Scatter Renormalization After Inserting a MetaMechanism

This is accomplished by always keeping the contents of a `MetaMechanism` normalized. When a container `MetaMechanism` needs to be renormalized, it changes the advertised scattering rate of an embedded `MetaMechanism`, but internally, the embedded `MetaMechanism` does not change.

In a device simulator that uses conventional scattering tables, a device with many doping regions would require computation of an identical number of explicit scattering tables during initialization. When a carrier transits from one doping region to another, its associated scattering table changes. This approach works well provided that the number of individual doping regions is not excessive. In devices with many doping regions, or with non-uniform doping profiles, the memory used by the scattering tables becomes prohibitive.

Alternately, one could compute the position-independent portion of the scattering tables during initialization and compute the position-dependent portion 'on-the-fly' as the carriers enter the various doping regions. If this approach is taken, the memory requirements are no longer excessive, but the computation required to renormalize the scattering tables when doping transitions occur can become prohibitive.

Our approach combines both of the above techniques. The position-independent and the position-dependent portions of the scattering tables are computed and normalized individually during initialization. During execution, the appropriate subtables are swapped in and out of the carriers' scattering tables as needed.

The memory requirements are identical to the second conventional case given above, but the periodic renormalization of all scattering mechanisms in the conventional approach is replaced with a single renormalization of a single self-scattering mechanism.

3.2 Mesh

The `Mesh` object is used to store and manipulate information about the spatial structure of the device. It converts the positional representation of the device into a mesh of node points used to update the electrostatic potential and local electric field via the solution of Poisson's equation. The `Mesh` object is also used to modify carriers' scattering tables as they move from one doping region to another.

The `Mesh` object contains material parameters for each node. These parameters consist of the intrinsic carrier concentration, the doping profile (N_A - N_D), and the dielectric permittivity. These parameters are stored using smart pointers so that simple devices do not require excessive memory allocation, but complicated devices can still be constructed.

In addition, one or more boundary conditions can be specified at each node. Dirichlet boundary conditions are used to model ohmic contacts and directional Neumann boundary conditions are used to model artificial device boundaries.

The `Mesh` object abstracts this information into `MeshRegions`. These `MeshRegions` are spatial volumes in which the material parameters and boundary conditions remain constant. For example, they can represent heterostructure layers, doping regions, ohmic contacts, or artificial boundaries.

`MeshRegions` are generally created during initialization. The `Mesh` object implements a method used to create rectangular solid regions. Either these regions are inserted into the existing device at a specified location, or the device is expanded to include the new region. Regions that are left in an undefined state after an expansion are assigned a set of default material parameters. These parameters are specified when the `Mesh` is constructed, and could describe a substrate material.

Region sizes can be specified using node counts, actual distances, or a combination of node counts and symbolic notation describing the current edges of the device. For example, the following code snippet will add a *p*-type region to an existing *n*-type block of semiconductor material:

```
mesh.SetRange(
      MeshPoint(
            kMeshEndAxis,
            kMeshStartAxis,
            kMeshStartAxis,
            mesh),
      MeshPoint(
            32,
            32,
            kMeshEndAxis,
            mesh),
      MeshElement(
            MaterialParameter(1e15)));
```

This snippet will insert a block of *p*-type material with a doping profile of N_A - N_D = 10^{15} m^{-3} onto the end of an existing device. The block will consist of 32 node points along the *x*- and *y*-axis. The *z*-axis will be the size of the existing device. A similar method is provided to insert boundary conditions at a node or along a range of nodes. These two methods make creating devices with complex geometries relatively simple.

Finally, the `MeshRegion` contains a list of its nearest neighbors. This list is used to accelerate searches for spatial carrier transitions.

3.3 Carriers

A `Carrier` object is also provided to represent the information associated with a single carrier. This information consists of the carrier's kinetic and potential energy, the carrier's wave vector and current position within the device, the `MeshRegion` in which the carrier is located, and the carrier's current `ScatterTable`. The ensemble of `Carriers` is stored in a `CarrierPool`, which provides high-level access to the `Carriers`.

3.4 MeasurementList

The code that performs measurements on the ensemble of carriers is a common source of 'brittleness' in existing device simulators. Our solution to this problem is a smart `MeasurementList` object that localizes all measurement-related code. The `MeasurementList` contains two sets of `MeasurementMethod` objects; a standard set and a termination set.

A `MeasurementMethod` is an abstract base class that defines the interface to an actual measurement method (i.e., kinetic energy, valley occupancy, etc.). The `MeasurementMethod` is notified whenever it is inserted into a `MeasurementList`, so it has an opportunity to pre-insert any `MeasurementMethod`s upon which it pre-depends. For example, a valley-dependent kinetic energy measurement and a valley-dependent velocity measurement would both need to compute valley occupancy. Insertion notification allows them to both pre-insert a valley occupation `MeasurementMethod`. The second insertion is ignored since a valley occupation `MeasurementMethod` is already present in the `MeasurementList`.

Whenever a set of measurements is requested from the `MeasurementList`, both the standard and the termination sets are executed. All `MeasurementMethods` in the termination set have the opportunity to end the simulation.

3.5 Poisson and Monte Carlo

A `Poisson` object provides an interface to a Poisson equation solver. Currently, we use the successive over-relaxation method to obtain the solution of the linear/non-linear set of equations that arise from the finite-difference discretization of the two-dimensional Poisson equation. However, it is set up so that a more efficient solver can descend from the base `Poisson` class.

A `MonteCarlo` object is used to tie all of the above objects together (see Figure 4). The `MonteCarlo` object is constructed with a `CarrierPool`, a `Mesh`, a `MeasurementList`, and a `Poisson` object. The `MonteCarlo` object has no knowledge of the contents or implementation of any of these objects. This separation of implementation from interface makes the code extremely robust.

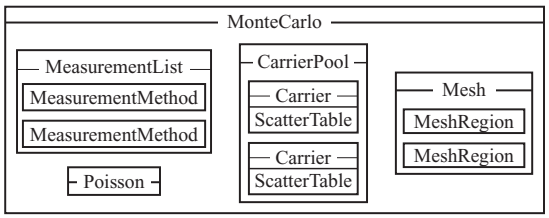

Figure 4: The MonteCarlo Object

4 EXPERIMENTAL VERIFICATION

In order to assure that the simulator is working properly, we performed several test cases using bulk GaAs samples. In our theoretical model, we included all relevant scattering mechanisms for this material (intravalley acoustic and polar optical phonon scattering, and intervalley scattering between the three conduction band valleys Γ, L and X). Figure 5 shows the time evolution of the average drift velocity when an electric field of 7 kV/cm is suddenly applied to the sample. One can clearly see the velocity overshoot effect. The electric field dependence of the steady-state drift velocity is shown in Figure 6. Figure 7 shows the dependence of the velocity on the doping concentration.

5 CONCLUSIONS

This paper has illustrated the object-oriented framework we have developed to perform generic, flexible device simulation. It has also shown that the code we have implemented produces results that match well with experimental data.

There are several steps necessary to complete the device simulator. First, we need to complete the device simulation portion of the code. Next, we need to simulate devices with complicated doping regions and test the efficiency of the simulator against existing device simulators such as DAMOCLES [4]. This will allow us to gauge the effectiveness of the encapsulated scattering tables. We need to modify the Poisson solver to handle non-uniform mesh spacing and spatially-dependent dielectric permitivity. Finally, we need to add the ability to create `MeshRegions` with non-uniform device parameters so that we can simulate devices with non-uniform doping profiles.

REFERENCES

* The authors would like to acknowledge the financial support from NSF under contract No. ECS98750S1-001 and ONR under contract No. N00014-99-1-03-18.

[1] M. Lundstrom, *Fundamentals of Carrier Transport*. Boston: Cambridge University Press, 1998.

[2] C. Jacoboni, and P. Lugli, *The Monte Carlo Method for Semiconductor Device Simulation*. New York: Springer-Verlag, 1989.

[3] S. M. Sze, *Physics of Semiconductor Devices*, New York: Wiley-Interscience, 1981.

[4] S. Laux and M. Fischetti, in *Monte Carlo Device Simulation, Full Band and Beyond*, Ed. by K. Hess. Boston: Kluwer, 1991.

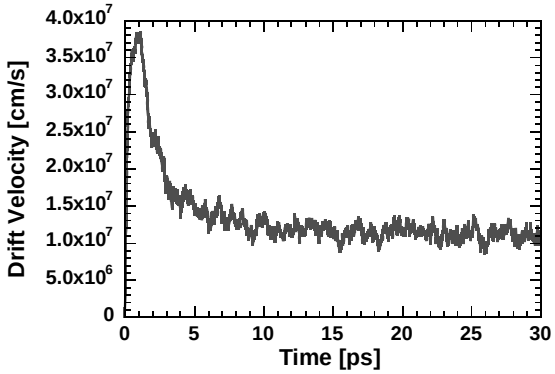

Figure 5: Drift Velocity Time Evolution

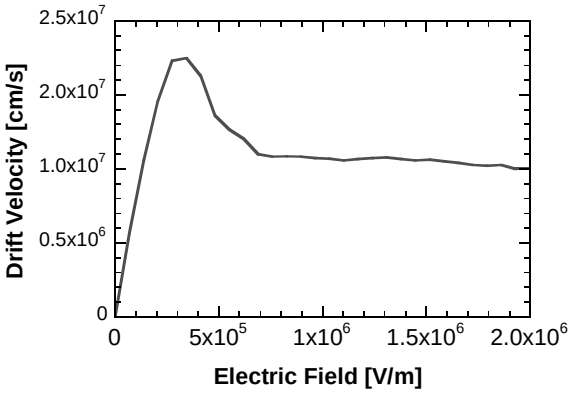

Figure 6: Velocity vs. Electric Field

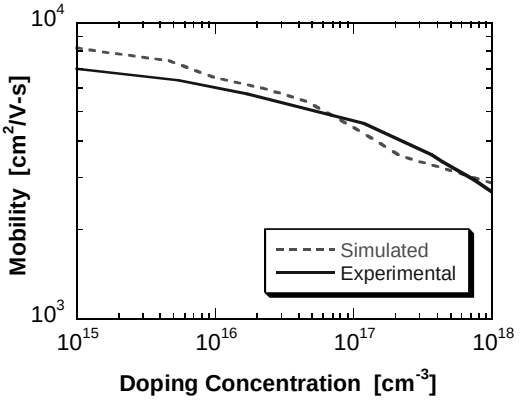

Figure 7: Velocity vs. Doping Concentration [2]