OBJECT-ORIENTED MANAGEMENT OF ALGORITHMS AND MODELS

R. Mlekus and S. Selberherr

Institute for Microelectronics, TU Vienna
Gußhausstraße 27–29, A–1040 Vienna, Austria
Phone: +43/1/58801-3692, Fax: +43/1/5059224,
e-mail: mlekus@iue.tuwien.ac.at, URL: http://www.iue.tuwien.ac.at

KEYWORDS

Computer Aided Engineering (CAE), Simulators, Software engineering, Model design, Object-oriented.

ABSTRACT

This paper presents a new C++ library which provides an object-oriented approach to the management of algorithms and models. The Algorithm Library affords a class hierarchy describing arbitrary algorithms, their parameters and documentation. Any program using this library gains an interpreter for the Model Definition Language which allows to define algorithms and their parameters on the input deck. New models can be defined in an object-oriented manner by inheriting features form prepackaged models supplied by the program without needing to edit the source code of the program or to link a new simulator executable. An example for the usage of the Model Definition Language is given to show its basic features.

INTRODUCTION

The continuous development of new processes and devices in combination with the increasing number of devices on a single chip requires to improve process and device simulator programs permanently through implementing new or enhanced models and algorithms. In traditional simulators the integration of new models requires to edit the source code of the simulator and, thus, deep knowledge of implementation details. For that reason a new library based concept was developed, which provides an object-oriented approach to the implementation, parameterization and selection of models[1], without any changes to the source code of the simulator.

The Algorithm Library is designed to support any kind of algorithm using arbitrary user defined data structures as parameters, which are handled in their native C++ representation and forwarded to the models using references. It offers a set of C++ classes and methods to handle these algorithms and parameters directly in C or C++ code and the object-oriented Model Definition Language (MDL). The MDL can be used as an interpreted language to ease the development of new algorithms, or by using a two pass concept as a compiler language to optimize the speed of simulations. Therefore algorithms and data structures used in the innermost simulation loops can be handled using the mechanisms of this library with almost no performance loss compared to traditional function calls.

These features distinguish the Algorithm Library from general purpose extension languages like TCL (Ousterhout 1994) or specialized approaches as presented in (Yergeau and Dutton 1997), (Litsios 1996) or (Radi et al. 1997), where modeling languages are introduced which are specialized to solving PDE's on specific mesh representations and the automatic generation of a Jacobian matrix.

BASIC STRUCTURE

Algorithms and Models defined with the Algorithm Library are represented by C++ classes derived from the base class `Model` or other previously defined model classes (Stroustrup 1986). The thereby defined inheritance tree (Figure 1) is used to classify the various model algorithms and for checking the user supplied definitions on the input deck during the initialization of the Algorithm Library. `Model`-classes encapsulate the algorithm itself, private data values used to evaluate the algorithm, an interface containing the required input and output parameters and the documentation (Figure 2).

The Algorithm Library provides an interface mechanism which separates arbitrary algorithms and/or model instances from the rest of the simulator. These interfaces contain the information about the type of algorithm to be used (requested model type), a specific instance name for the model and all the input and output parameters which are necessary to evaluate any algorithm of the requested type ("fat" interface concept). The actual algorithm used for a certain model instance

[1] In this text no conceptual distinction is made between the nouns "model" and "algorithm".

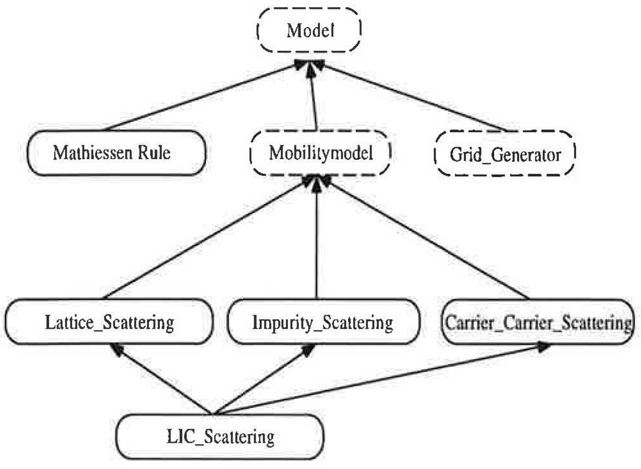

Figure 1: A sample Model Hierarchy

can be selected on the input deck of the program, or by supplying a default type in the interface definition. During the initialization the Algorithm Library checks whether the model instances are either equal to or derived from the requested model type.

Parameter classes contain a reference to the value, a name which has to be unique inside of the given interface, and optionally documentation and default values (Figure 2). Several types of parameters according to the standard C++ variable types are predefined. New parameter types can be instantiated by specializing the template class `Parameter` with arbitrary C++ classes describing the values. For each of these parameters a set of operators and functions can be specified which can be used in calculations defined on the input deck as well as in algorithms defined in C++.

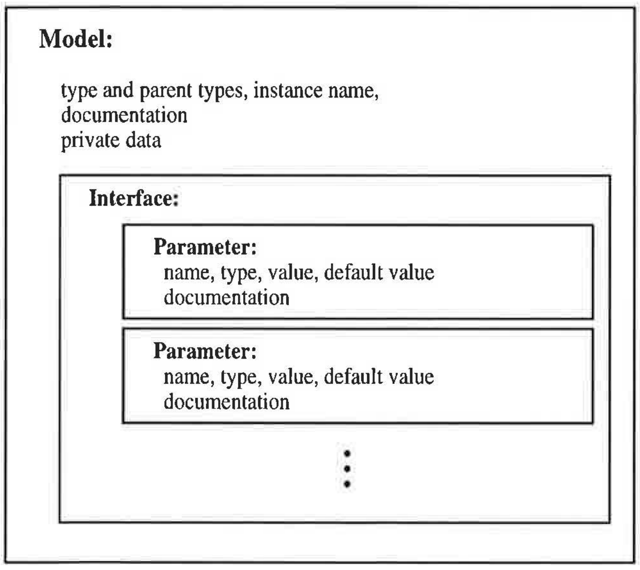
Model:
type and parent types, instance name,
documentation
private data
Interface:
Parameter:
name, type, value, default value
documentation
Parameter:
name, type, value, default value
documentation

Figure 2: The data structure of a model

To evaluate the algorithms, the parameter values are forwarded to the model instances by reference. Therefore the interfaces of the program and the model have to be linked by the Algorithm Library in the initialization phase of the program, so that the value references of affected parameters are set to equal values as shown in Figure 3. To support the optimization of data structures as needed by advanced CPU architectures these references can explicitly be set to specified values.

Parameters with the same name and type are linked automatically; other links can be specified in the C++ code of the program and on the input deck using the Model Definition Language. A run time type check of the parameters ensures the software integrity of the input deck and the program. Since default values for parameters can be specified in the interface definition of a model, in the definition of the program interface and on the input deck, the actual default value of linked parameters is determined by the source with the highest priority as depicted in Table 1.

priority	source of default value
3	input deck definition
2	interface definition
1	model definition

Table 1: Default value priorities

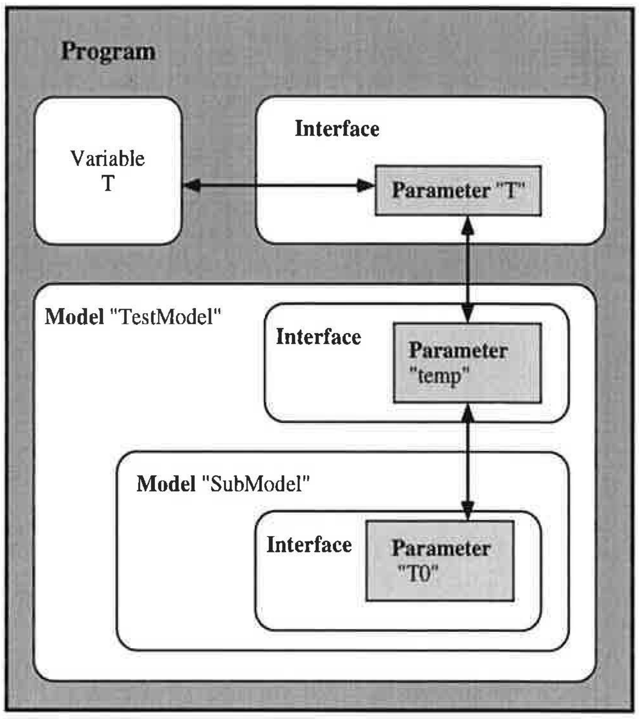

Figure 3: Linking of parameter values

Different sets of algorithms and appropriate parameter definitions can be collected in separate libraries of object code or MDL source files. Rapid prototyping of new algorithms is supported by an interpreter for the object-oriented MDL which is used to parse model definitions on the input deck of the program. Additional models can be implemented and tested during the run time of the program and added later to the model libraries by using the MDL-compiler which translates the definitions on the input deck into C++ code.

An instance of a specific algorithm can be generated by forwarding the model type name to the Algorithm Library or by giving an instance name for the algorithm. In this case the actual class type is determined at run time by parsing the input deck. To evaluate the algorithm, its class instance is connected to an interface providing the necessary parameter values.

MODEL DEFINITION LANGUAGE

The Algorithm Library contains an interpreter and a compiler for the Model Definition Language which allows to:

- Define the actual algorithms (model instances) to be used for a specific task.
- Define the parameter values for model instances and default values for the parameters of certain types of algorithms.
- Define new algorithms by inheriting and combining methods and interfaces from previously defined ones.
- Define global parameters which can be used for communication between model instances where the author of the program didn't anticipate the necessity.
- Request a database record, describing all available algorithms, their interfaces and documentation and the thereby defined model hierarchy.
- Request a debug report describing the actually used algorithms, the values and default values of parameters for specific model instances, and a table showing how these parameters are linked together.

MDL classes contain private and protected interface parameters, private and protected local parameters and sub-models defined on previously scanned MDL source files or object libraries of compiled C++ code. The inheritance rules for protected and local parameters are similar to the C++ inheritance rules and support multiple inheritance of parameters and single inheritance of the evaluation rule.

The evaluation rules of MDL classes can contain calculations with parameters of any type. For the predefined C++ compatible parameter types the standard C++ operators are predefined with C++ compatible precedence rules. Operators for user defined parameter types can be used if they are supported by the classes describing the parameter values. These calculations can be combined with evaluations of sub-models by using conditional and loop expressions and evaluations of sub-models provided from the Algorithm Library.

A minimal program using the Algorithm Library to evaluate a single algorithm may be structured as shown in Figure 4. A corresponding source code example is given in Figure 6:

1. The Algorithm Library is initialized by parsing and analyzing the input deck.
2. The "fat" interface containing all parameters a certain type of algorithms might need, the required model type, and a default model type is created. Optionally the documentation of the interface can be defined in this place, too.
3. The model instance is requested from the Algorithm Library and linked against the parameter interface.
4. Repeat as necessary: Compute the values of the input parameters; evaluate the model; use the resulting parameter values for further computations.
5. To release the acquired resources cleanly, the required model instance has to be deleted after the last evaluation. A shutdown function for the Algorithm Library resets the library into the initial state, so that new definitions can be parsed independently from any previous ones.

Steps 1–3 should take place during the initialization phase of the program because they require the rather time consuming parsing and interpretation of the input deck. Once the internal data structures of the Algorithm Library are assembled, the additional time consumption caused by the usage of the Algorithm Library are typically between 5–30 % depending on the complexity of the models.

EXAMPLE

To give a short example for the usage of the Model Definition Language, Figure 8 shows how a simple carrier mobility model is defined by combining a lattice scattering model (Arora et al. 1982)

$$\mu_{LS} = \mu_0 \cdot \left(\frac{T}{300}\right)^{-\alpha} \tag{1}$$

with a carrier carrier scattering model (Adler 1981)

$$\mu_{CCS} = \frac{1.428 \cdot 10^{20}}{\sqrt{np} \cdot \ln(1 + 4.54 \cdot 10^{11} \cdot (np)^{-1/3})} \tag{2}$$

using Mathiessen's rule.

$$\mu_g = \frac{1}{1/\mu_{LS} + 1/\mu_{CCS}} \tag{3}$$

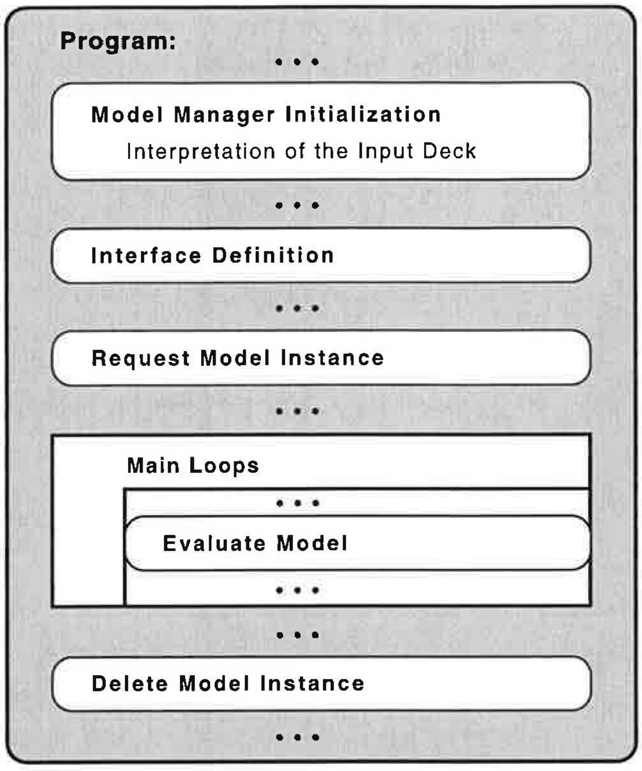

Figure 4: Structure of a program using the Algorithm Library

The source code of a program which uses the Algorithm Library to evaluate a mobility model is depicted in Figure 6. The interface for all carrier mobility models contains among others the parameters `temp` (lattice temperature in K), `mu` (the resulting carrier mobility in $cm^2V^{-1}s^{-1}$) and `np` (the product of the electron and hole densities in cm^{-6}).

The abstract model type `MobilityMdl` — the parent class of all carrier mobility models — can be defined as a part of the C++ source code of the simulator. It has no valid evaluation rule but specifies the names, types and default values of the interface parameters common to all mobility models (Figure 7). These definitions are inherited by the mobility models defined by using the MDL as depicted in Figure 8.

Figure 5 illustrates the structure of the resulting mobility model by showing the equivalences between the various interface- and model parameters which are generated by using the `link` command of the Model Definition Language.

CONCLUSION

By using the model library a clean interface is introduced between modularized algorithms and the rest of the program. These algorithms can easily be replaced by newly defined ones during run time without any additional coding efforts within the simulator. Due to the almost negligible run time performance loss and the great simplifications in introducing new algorithms into simulators, the Algorithm Library is a valuable tool for the developers as well as for the users of a simulator. Developers gain methods to write modularized user extensible programs whereas the users of these programs can customize them to their own needs by using the powerful MDL extension language.

ACKNOWLEDGMENT

The authors would like to thank Siemens AG, Munich, Germany for the support.

REFERENCES

Adler, M. 1981. Accurate Numerical Models for Transistors and Thyristors. In Miller, J., editor, *An Introduction to the Numerical Analysis of Semiconductor Devices and Integrated Circuits* pages 5–8 Dublin. Boole Press.

Arora, N.; Hauser, J., and Roulston, D. 1982. Electron and Hole Mobilities in Silicon as a Function of Concentration and Temperature. *IEEE Trans.Electron Devices* ED-29(2):292–295.

Litsios, J. 1996. *A modeling language for mixed circuit and semiconductor device simulation.* Hartung-Gorre.

Ousterhout, J. 1994. *Tcl and the Tk Toolkit.* Addison Wesley.

Radi, M.; Leitner, E.; Hollensteiner, E., and Selberherr, S. 1997. AMIGOS: Analytical Model Interface & General Object-Oriented Solver. In Riedling, K., editor, *Basics and Technology of Electronic Devices* pages 57–60 Grossarl, Austria. Gesellschaft für Mikroelektronik. Proc. of the Seminar "Grundlagen und Technologie Elektronischer Bauelemente".

Stroustrup, B. 1986. *The C++ Programming Language.* Addison-Wesley.

Yergeau, D. and Dutton, R. 1997. Alamode: A Layered Model Development Environment for Simulation of Impurity Diffusion in Semiconductors. Documentation for release 97.06.18.

BIOGRAPHY

Robert Mlekus was born in Tulln, Austria, in 1968. He studied electrical engineering at the Technical University of Vienna, where he received the degree of 'Diplomingenieur' in 1994. He joined the 'Institut für Mikroelektronik' in December 1994, where he is currently working for his doctoral degree. His work is focused on object-oriented techniques for the integration of physical models into process and device simulators.

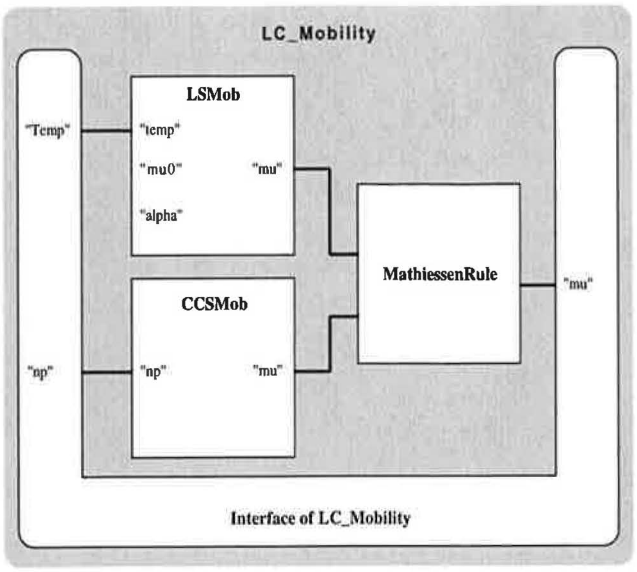

Figure 5: Block diagram of the new mobility model

```
#include "vmodel.hh"

int main()
{
// define parameter mob with the name "mu"
Parameter<double> mob("mu");
// define other parameters (T,np,...)
...

// add the parameters to the interface
Interface MobilityInterface;
MobilityInterface.addParam(&mob);
...

// get the model instance
ModelPtr MobModel = mdlRequestModel(
  "ElectronMobilityMdl",    // instance name
  "MobilityMdl",            // required type
  &MobilityInterface);      // interface

// compute the values of input parameters
// evaluate the model and
// use the resulting parameter values
T = ..; np = ....;
MobModel->evaluate();
...

// release resources
delete MobModel;
mdlShutDownModelServer();

return 0;
}
```

Figure 6: Simulator source code example

```
class MobilityMdl : public Model
{
protected:
   Param<double> mu; // [cm^2 V^-1 s^-1]
   Param<double> T;  // [K] temperature
   ...               // further parameters
public:
   bool evaluate () { return false; }
   bool init () {
      addParam( &mu, "mu" );
      addParam( &T , "temp", 300 );
      return true;
   }
   MODEL_DECLARATION_EXTENSION
};
MODEL_DEFINITION_EXTENSION(MobilityMdl,Model)
```

Figure 7: Abstract mobility model class in C++

```
#include "CC_Mobility.mdl"

// lattice scattering mobility model
Model LatScatMobility : MobilityMdl
{
   calc "mobility" {
     Interface."mu"=Interface."mu0" *
     exp(-Interface."alpha"*
     ln(Interface."temp"/300));}
   EvaluationRule { "mobility"; }
}

// Define the combined mobility model
Model LC_Mobility : MobilityMdl {
  Model LatScatMobility "LSMob";
  Model CC_ScatMobility "CCSMob";

  // new default parameters for sub models
  Parameter " LSMob"."mu0" = 1448;
  Parameter " CCSMob"."alpha"  = 2.33;

  link Interface."temp" to "LSMob"."temp";
  link Interface."np" to "CCSMob"."np";

  EvaluationRule {
    "LSMob";
    "CCSMob";
    calc {
      Interface."mu" =
        1/(1/"LSMob"."mu"+1/"CCSMob"."mu");
    }
  }
}

// Specify the actual Mobility Model Type
Model "Mobility Model" = LC_Mobility;
```

Figure 8: MDL definition of the new mobility model