Proceedings of the IASTED International Conference
Applied Modelling and Simulation
August 12-14, 1998, Honolulu, Hawaii - USA

OBJECT-ORIENTED ALGORITHM AND MODEL MANAGEMENT

R. MLEKUS and S. SELBERHERR

Institute for Microelectronics, TU Vienna
Gusshausstr. 27–29, A-1040 Vienna, Austria
Phone +43-1-58801-3692, FAX +43-1-5059224
e-mail: mlekus@iue.tuwien.ac.at

ABSTRACT

We present a new C++ library , the ALGORITHM LIBRARY, which provides an object-oriented approach to the management of algorithms and models. The ALGORITHM LIBRARY offers a class hierarchy describing arbitrary algorithms, their parameters and documentation. Any program using this library gains a byte code compiler 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 from prepackaged models supplied by the program without the need to edit the source code of the program or to link a new simulator executable. An implementation of the ALGORITHM LIBRARY in the process simulator PROMIS-NT [1] is described to give an example for the basic features of the MODEL DEFINITION LANGUAGE.

KEYWORDS

Simulators, Software Engineering, Model Design, Object-Oriented.

INTRODUCTION

The increasing complexity of simulation tools combined with the requirement for short development cycles for the implementation of new models and algorithms raises a strong demand for object-oriented development tools and languages supporting the separation of the simulator into modules which can be maintained without interfering other modules.

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 are 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 (using a "just in time" byte code compiler) 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 in relation to traditional implementations based on function calls.

Sets of models, appropriate parameter types, their operators and functions are packaged into dynamic link libraries which can be loaded during run time to be extended by additional models defined on the input deck by using MDL. Thus binary distributions of simulators can be compiled which are extendible by additional user defined algorithms and models for certain purposes.

These features distinguish the ALGORITHM LIBRARY from general purpose extension languages like TCL [2] or specialized approaches as presented in [3], [4] or [5], where modeling languages are introduced which are specialized to solving PDEs 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 [6]. 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

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

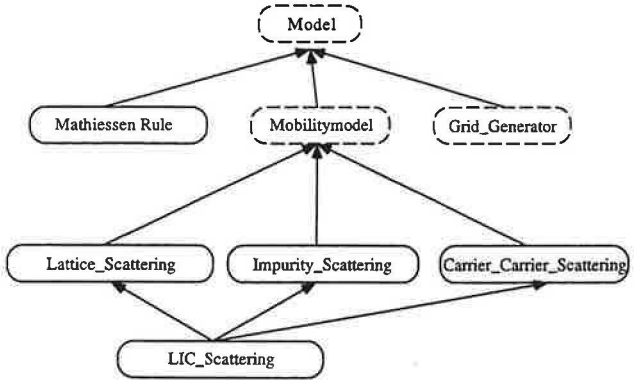

FIGURE 1: A SAMPLE MODEL HIERARCHY

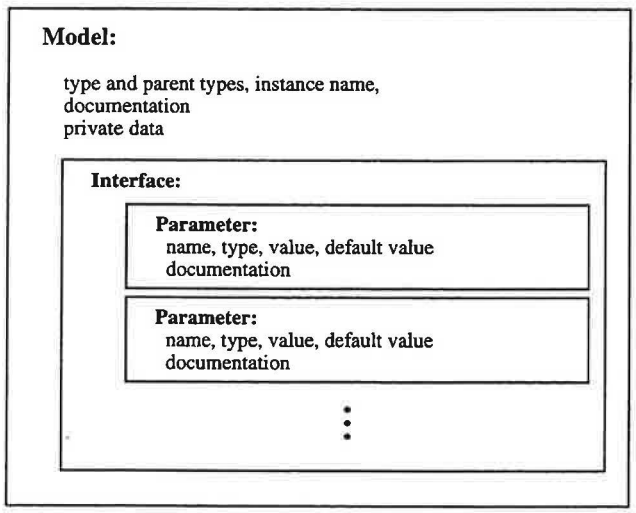

FIGURE 2: THE DATA STRUCTURE OF A MODEL

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 an algorithm of the requested type ("fat" interface concept). The actual algorithm used for a certain model instance 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++.

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 optimization of data structures as needed by advanced CPU architectures these references can explicitly be set to specified values.

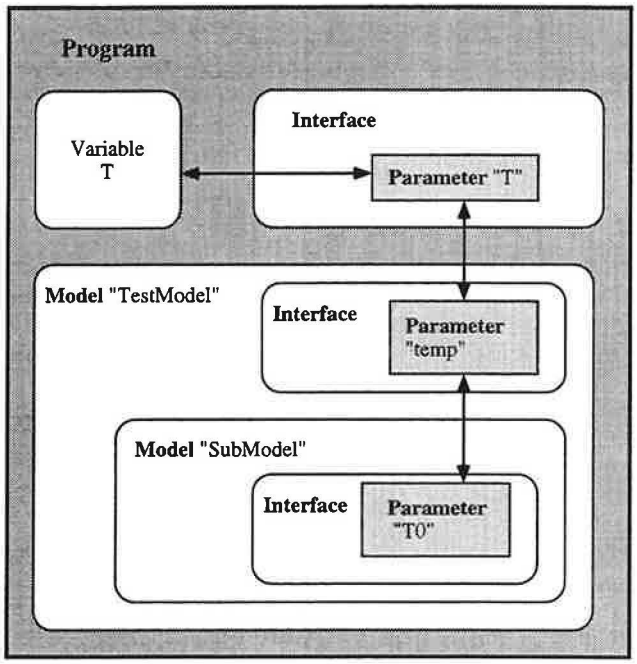

FIGURE 3: LINKING OF PARAMETERS

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

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 im-

plemented 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.

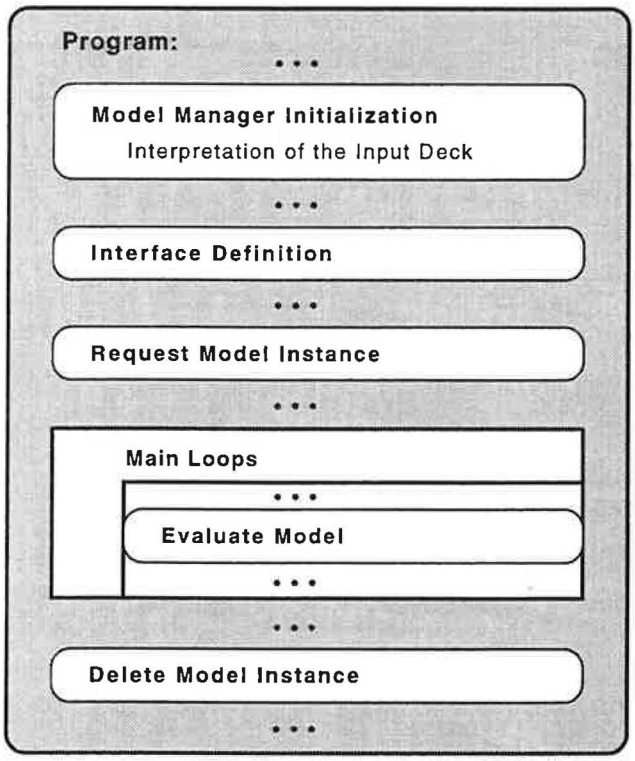

FIGURE 4: STRUCTURE OF A PROGRAM

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 MDL algorithms by inheriting and combining methods and interfaces from previously defined ones.
- Define global parameters which can be used to exchange parameter values between model instances where the author of the program didn't anticipate the necessity for such communication.
- 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 and initialization 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:

1. The ALGORITHM LIBRARY is initialized by parsing and analyzing the input deck.
2. The "fat" interface containing all parameters a certain type of algorithm 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 to 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.

EXAMPLES

In the process simulator PROMIS-NT [1] the ALGORITHM LIBRARY is used for the complete control of the simulation flow. Several interfaces and parameter and model types are implemented and prepackaged to be used in the MDL input deck files:

- All information needed to execute a specific simulation (e.g. input files, simulation time,...) is given to the simulator by using an initialization algorithm which overwrites a number of predefined default values. (Figure 5)
- The process temperature progression can be specified as a function of the actual time and the start and end time of the simulation.
- For every impurity or stress distribution considered an initialization function depending on the values of all other distributions and the position values can be defined which is applied after reading the distributions from the input file.
- The coefficients $\alpha_{ij}, \gamma_i, a_{ij}, b_{ij}, c_{ij}, d_i, \beta_{ij}$ and Φ_i of the transport equations 1, 2 and 3 can be specified as functions of process temperature, time and position and all but β_{ij} can be functions of all distributions.

$$0 = \sum_{j=1}^{N} \alpha_{ij} \cdot \frac{\partial W_j}{\partial t} + \mathrm{div}\vec{J}_i + \gamma_i \quad (1)$$

$$\vec{J}_i = \sum_{j=1}^{N} (a_{ij} \cdot \mathrm{grad}W_j + b_{ij} \cdot W_j \cdot \mathrm{grad}\psi + \vec{c}_{ij} \cdot W_j) + \vec{d}_i \quad (2)$$

$$0 = \sum_{j=1}^{N} \beta_{ij} \cdot J_j^n + \Phi_i \quad (3)$$

W_j denotes the dependent variables giving values of the affected distributions and N is the number of equations. ψ is one of the dependent variables W_j. J_j^n is the current of the j-th distribution perpendicular to the surface.

- Filter algorithms can be applied to the distributions before writing them into the result file.

Figure 5 shows an excerpt of a PROMIS-NT input deck. The respective distributions are denominated with their material names which is equivalent to certain values of i and j in the equations 1 to 3. The model developer needs no further information but the names of the required Model instances (e.g. `ProcessTemperature` for the algorithm specifying the process temperature) and the names of the appropriate input and output parameters. These informations are documented in the manual of the specific simulator or can be requested using MDL commands like `listModels` which would provide a complete list of all available `Model` classes;

CONCLUSION

With the ALGORITHM LIBRARY simulators can easily be built in a modularized manner by specifying interfaces between distinct user definable algorithms and the simulator kernel. As shown in the PROMIS-NT example, simulators can be adapted to specific tasks by instantiating different prepackaged or user defined models without coding efforts within the simulator kernel or knowledge about the internal data or the process flow itself.

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 with the powerful extension language MDL.

ACKNOWLEDGMENT

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

REFERENCES

[1] H. Puchner, *Advanced Process Modeling for VLSI Technology.* Dissertation, Technische Universität Wien, 1996.

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

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

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

```
// load prepackaged  mdl-definitions
// from objeckt-libraries and MDL-files
set $VMODELPATH=".:$HOME/promis-nt/models";
#include "promis-defaults.mdl"
LoadObjectLibrary "promis";

// general simulator setup
NewModel PromisNT_Init : DiffInitModel {
  evaluate {
    :inputPBF             = "example.pbf";
    :endTime              = "15._min";
    :quantityList         = "Quantities";
    :deviceInitialization = "DeviceInit";
    :processTemperature   = "RampUpTemp";
  }
}

// list of quantities to be used
NewModel Quantities : QuantityListModel
{
  evaluate {
    :quantity["B_active" ]={{ B  active }};
    :quantity["As_active"]={{ As active }};
    :quantity["Sxx"]      ={{ Sxx       }};
    // code snipped ...
  }
}

// the process temperature model
NewModel
  RampUpTemp : ProcessTemperatureModel
{
  evaluate {
    if ( :time < 60.0 ) {
      :T = 300.0 + :time * 973.0 / 60.0;
    } else {
      :T = 1273.0;
  }}
}

// Setup:
//   segment initialization of quantities
//   coefficient table for each segment
NewModel ExampleInit : SegmentInitModel {
  evaluate {
    :quantityInit["Si"]  = "SiInit";
    :quantityInit["SiO2"]= "DefaultInit";

    :coefficients["Si"]  = "SiCoeffs";
    :coefficients["SiO2"]= "SiO2Coeffs";
  }
}

// specify the quantity initialization
// models for the Si Segment
NewModel SiInit : QuantityListModel {
  evaluate {
    :qList["B_active"] = "B_Init";
    :qList["Sxx"]      =
        "DefaultQuantityInitModel";
}}
```

```
// code snipped ...

// Initialization check for neg. values
NewModel B_Init : QuantitiyInitModel {
  evaluate {
    if ( :B_active < 0.0 ) {
      ::cout << "Error:...";
      :result = ::false;
    }
  }
}

// Select Coefficient Models for Si
NewModel SiCoeffs : CoefficientInitModel
{
  evaluate {
   :alpha["B_active"]["B_active"]=alpha1;
   :alpha["B_active"]["Sxx"]     =alpha2;
   // code snipped ...
  }
}

// code snipped ...

NewModel alpha1 : CoeffModel {
  Instance D = DiffusionCoefficentBoron;
  evaluate {
    :Coeff = 1.0 / D.Coeff;
    :dCoeff_dB_active = -
    D.dCoeff_dB_active / pow(D.Coeff,2.);
  }
}

// code snipped ...
```

FIGURE 5: EXAMPLE INPUT DECK FOR PROMIS-NT

[5] M. Radi, E. Leitner, E. Hollensteiner, and S. Selberherr, "AMIGOS: Analytical Model Interface & General Object-Oriented Solver," in *International Conference on Simulation of Semiconductor Processes and Devices*, pp. 331–334, 1997.

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

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.