The Gnucap Model Compiler

Albert Davis
Idaho State University
Pocatello, Idaho 83209 USA
davialbe@isu.edu

Abstract

This paper describes a modeling language and its compiler, for adding models to a mixed-signal simulator "Gnucap". This language is supplies more simulator details than the popular "AMS" languages do, and provides a similar high level interface.

1 Overview of the model compiler

For many years, adding models to simulators has been considered to be a nuisance. Model code is usually very simulator specific, requiring intimate knowledge of the details of the simulator you are installing the model into. Every simulator requires some "overhead" code. This is the code to parse the input, fill in default values and the like. For most simulators, most of the code for a typical model is this overhead.

Hard coded models restrict the ability to change the simulator for new algorithms. It is often necessary to change all of the models to make a trivial change to an algorithm.

The goal of this project is to express a model in modeler's terms, to minimize simulator dependencies, and produce an end result that is as efficient as a manually coded model. An expedient implementation is also a factor. This language supports more simulator details than the popular "AMS" languages do.

The language allows a model developer to describe the model in a mixed behavioral and circuit form. It also provides a mechanism to describe in an intuitive form the overhead associated with modeling, such as parsing, probes, and binning. It also handles the details associated with a mixed-mode queue driven simulator.

The description language is divided into two sections, matching the Spice device and model statements. For the model, inheritance is supported, so similar devices or models can share core or parameters by inheriting from a common base. A device can link to any model inherited from the same base. This is used for the many "levels" of MOSFET models. The language is C-like, and is designed for efficiency. All aspects of efficiency are important. Generated models are as efficient as hand-coded models.

The language specifies a subcircuit, which can be composed of any device types known by the simulator, parameter lists, and equations. The simulator has some additional device types, poly-conductances and poly-capacitors, to help with purely mathematical models. The description is hierarchical, in the sense that devices defined by the model compiler can also be used as primitives.

The compiler uses a data-flow oriented design, and is coded in C++. This design style groups code by stages in the data flow. This style was chosen over an object-oriented design because it is easier to change front ends and back ends. Making it easier to change the front end will enable it to support other input languages, such as Verilog-A, Verilog-AMS, and VHDL-AMS. Changing the back end could enable it to be ported to other simulators.

The input stage is a recursive descent parser using the public domain "argparse" parsing class, which is also used in the Gnucap simulator. It builds a data structure that mimics the input file. A simulator-dependent back end reads the data and builds the output file.

All of the advanced device models (diodes, BJT's, MOSFETS) in Gnucap are implemented using the model compiler. The primitives are still hand coded. It has been shown to save a significant amount of time in implement-

ing a model, with no sacrifice in performance.

2 Overview of Gnucap

Gnucap is a mixed-signal simulator, with a Spice-like circuit description format. It is not a Spice derivative. It is written in C++ with an object oriented design. It makes extensive use of advanced C++ features, such as inheritance, templates, and STL.

The transient analysis is Spice-like in the sense that it is based on the same math. It deviates from Spice in that it uses a partial solution scheme, with incremental update and partial solutions. It uses queues to manage the simulation, much like the event queues found in digital simulators. It allows different portions of the circuit to be simulated with different algorithms, with decisions made on the fly about which algorithms to apply where.

Gnucap uses special sparse matrix package that allows incremental update and partial solutions. By allowing incremental update, model evaluation can be controlled by an event queue.

To support the mixed-mode simulation, some steps that are traditionally combined into a single function must be separated. In Spice, the "gather", "precondition", "evaluate", and "load" steps are combined into a single function. In Gnucap, they must be separate functions, that in some cases are not called sequentially.

To support mixed-mode simulation, there are other functions not found in traditional simulators. These include functions to determine whether or not to queue a device for evaluation and functions to pre-calculate some values. Since it is queue driven, the order of evaluation and order of loading are not consistent. Some devices may not be loaded at all if there state does not change.

Complex devices use a subcircuit internal representation. This results in a small speed penalty, which is overcome by the mixed-mode logic. One benefit of this is that the models for other types of analysis, such as AC and pole-zero, are generated implicitly by the subcircuit. They require no effort from the person making the models. A second benefit is that it helps with mixed-mode tracking.

All devices use a 3-level representation internally, but it is presented to the user as 2 levels. For comparison, Spice uses a 2-level representation, the device and a model. Gnucap presents to the user a 2-level representation compatible with Spice, but a third level, a "common" is used internally. The "common" stores information that is specified per device but is shared between identical devices. Parameters such as dimensions, expressions, and component values are stored in the common.

Run times for very large circuits are often considerably faster than Spice-type simulators. It is able to exploit latency, like digital simulators. If a portion of the circuit is linear, or nearly linear, it precomputes that portion. If a circuit is at a stable operating point, it may be considered to be linear. It can also support hierarchy by using block decomposition.

For the future, further enhancements are planned. One approach that shows promise is "cached model evaluation", where the results of several model evaluations are cached for future use. There are also plans for full modeling of self-heating and true multi-rate support.

One of the early goals in Gnucap was to simplify model installation. It is easier than Spice, provided you are not starting with a Spice model, but the emphasis on mixed signal circuits has resulted in model installation being not as easy as was hoped. The modeler needs to contend with the mixed signal aspects of the simulator. It is possible to ignore this, but the simpler approach results in poor performance.

A separate program, the model compiler being described in this paper, provides a higher level interface for modeling. The modeling language is C-like for easy translation, and generally hides the implementation details.

3 History of model compilers

Good programmers have always used scripts to automate repetitive tasks. There are programs that go beyond simple scripting to do many tasks. For developing traditional languages, the utilities "lex" and "yacc" are well known, as tools to generate lexical analyzers and parsers.

The use of model compilers by simulators is also not new. Most commercial simulators had some kind of tools for generating models. One commercial example is the PSPICE "Parts" program. The XSPICE "code models" are another example. The XSPICE language is based on simple processing that generates "C". It is mostly copied through. It does not have the capability or flexibility of the Gnucap modeling language.

CAzM is a table driven simulator. It uses a model compiler to generate tables. The model equations are

evaluated at an array of points, and tables are generated.

The high level circuit description languages VHDL and Verilog are often implemented as compilers. In some, the circuit description is compiled to C then compiled and linked to the simulator. Scripts are used to improve the user interface.

Some in-house simulators use model compilers. The proprietary simulators used by Lucent are examples of this.

Finally, it was often a topic of debate in graduate school.

4 The language

The description language is divided into two sections, matching the Spice device and model statements. For the model, inheritance is supported, so similar devices or models can share core or parameters by inheriting from a common base. A device can link to any model inherited from the same base. This is used for the many "levels" of MOSFET models. The language is C-like, and is designed for efficiency. All aspects of efficiency are important.

4.1 The "device" section

4.1.1 Outline

The "device" section describes the portion of the code relating to each device, as presented to the user in the device or instance statement in a Spice format file. The Gnucap "common" is also included here. It consists of several sections:

Overhead Several lines deal with overhead such as how to identify the device.

Circuit This section describes the internal circuit topology in a netlist form, similar but not identical to the Spice format.

Probes This section lists the ways you can probe a device, for printing and plotting, and how the values are calculated.

Device This section lists the parameters that apply to each device. This includes all of the "state variables". It also lists temperature dependent parameters.

Common This section lists the parameters that apply to all identical devices. In general, the parameters such as length and width are included here.

Functions There are several "function" sections where calculations are done.

```
device BJT { // what it is called in code
  parse_name bjt; // ...  when parsing
  model_type BJT; // model compatible with
  id_letter Q; // Spice format label letter
  circuit {....} // subckt form as netlist
  tr_probe {....} // list of probes
  device {....} // device specific data
  common {....} // ... can be shared
  function zzzz {....} // helper functions
  tr_eval {....} // evaluation function core
}
```

4.1.2 The "circuit" section

The circuit section is used to describe a circuit toploogy for the model. It is made of a list of components, similar to a netlist, except that some extra parameters are added to interface to the model. All components already defined in the simulator are available, including types that are dynamically loaded, logic devices, and types defined by the model compiler. The simulator includes some primitives such as poly conductances and poly capacitors to help with modeling.

A circuit can have components that may or may not be used depending on parameters. There can also be internal nodes, and they too can exist or not exist based on parameters.

Internal components can get their data in several ways. The simplest is a value determined from the model data. A nonlinear sub-device can get its state information directly from the state table of the device being defined. Another approach is to use a separate evaluation function.

```
circuit {
  ports {col base emit sub};
  local_nodes {
    ic short_to=col short_if="m->rc == 0.";
    ie short_to=emit short_if="m->re == 0.";
  }
  args sb DIODE {
```

```
    area = b->as;
    perim = ps;
    is_raw = b->issat;
    cj_raw = m->cbs;
    cjsw_raw = NA;
    off = true;
  }
  cpoly_g Ice {ic ie ib ie} state=cce_cpoly;
  resistance Rc {col i} value="m->rc/c->area"
    omit="m->rc == 0.";
  capacitance Cj {anode cathode} eval=Cj;
  poly_cap Cbe {iemit ibase icol ibase}
    state=qbe;
  diode Dsb {sub ic} args="sb"
    reverse="m->polarity==pP"
    omit="_n[n_bulk] == _n[n_isource]";
}
```

4.1.3 The "probes" section

The "probes" section provides a way to list parameters than can be output as the simulation runs. Each statement consists of a keyword and an expression that is used to calculate it. The expression is C-like, with some extensions. All device state variables are available. All node voltages are available. In addition, anything that can be probed on any internal element is available. You can make expressions based on these data.

```
tr_probe {
  Vd = "@n_anode[V] - @n_cathode[V]";
  Id = "@Yj[I] + @Cj[I]";
  CAPCUR = "@Cj[I]";
  QBE = qbe;
  CQBE = cqbe;
  GM = "(reversed) ? gmr : gmf";
  P ="@Rs[P] +@Rd[P] +@Ids[P]";
}
```

Once defined, the data can be accessed with an ordinary plot or print statement to the simulator.

```
.print tran vd(M*) capcur(M12) gm(M11)
.plot tran vd(M*) capcur(M12) gm(M11)
.print tran + I(Yj.M25) Charge(Cj.Ddb.M12)
```

4.1.4 The device section

This section contains a list of parameters that are specific to each device. In general, these are the state variables, that are calculated every time the model is evaluated.

```
device {
  calculated_parameters {
    double vbe "B-E voltage";
    double cmu "collector-base current";
    double gmu "Small signal conductance";
    double qbe "Charge storage B-E junction";
  }
}
```

4.1.5 The common section

The common section is used to list parameters that define the device. Identical devices can share this information. User defined parameters that are specified for each device are specified here. No instance specific state information is here, but calculated parameters that can be calculated based only on other parameters are specified here. You can specify the names to be used for parsing, if and how it is calculated or defaulted, and limits to the values.

```
common {
  unnamed area;
  raw_parameters {
    double area "area factor" name=Area
      default=1.0 positive;
    double perim "perimeter factor"
      name=Perim default=0.0
      positive print_test="perim != 0.";
  }
  calculated_parameters {
    double is_adjusted "" name = IS
      calculate="(m->js * area)";
  }
}
```

4.1.6 Evaluators and the "tr_eval" function

A function "tr_eval" is the main evaluator function for transient analysis. It gathers data and checks it then dispatches it to the appropriate model evaluate function.

Elements that are used to build the model can also have evaluator functions. They assume a simple input - output relationship. They calculate an output and its derivative based on its input. The actual meaning of input and output depends on the type of device. It is a function, with a C-like syntax.

4.2 The "model" section

4.2.1 Outline

The "model" section describes the portion of the code relating to the ".model" statement in a Spice format file. It consists of several sections:

Overhead Several lines deal with overhead such as how to identify the model.

Inherit One line states a model to inherit from. Unlike C++, it is not necessary for the model compiler to know anything about it. The base is may be either hand coded or coded through the model compiler.

Independent This section lists the user specified parameters, with their defaults and limits.

Size_dependent This section lists parameters that are calculated based on size. It also lists user specified parameters that will be automatically scaled. The model compiler automatically generates the scaling code, and a set of parameters relating to how it is scaled.

Temperature_dependent This section lists temperature dependent parameters, and how the actual value is computed based on temperature.

Validate This section is a code block that checks model parameters against size data in each device. It returns true or false, depending on whether the data is valid for that size. It is used for binning.

Functions There are several "function" sections where calculations are done. The most important is "tr_eval" which is the actual model equations.

```
model MOS6 { // what it is called in the code
  level 6; // selecting this one by level
  dev_type MOS; // what device it matches
  inherit MOS123; // what to inherit from
  independent {....} // parameters and fixup
  size_dependent {....}
  temperature_dependent {....}
  validate {....} // how to validate
  tr_eval {....} // model evaluation code
}
```

4.2.2 A base class

```
model MOS_BASE { // in the code
  dev_type MOS; // what device it matches
  base; // a device refers to this one
  inherit DIODE; // even a base can inherit
  keys {
    NMOS polarity=pN; // id keys for parse
    PMOS polarity=pP; // with adjustments
  } // not restricted to base
  independent {....}
  size_dependent {....}
  temperature_dependent {....}
  validate {....}
  tr_eval {....}
}
```

4.2.3 The "independent" section

These are the classic "model card" parameters or parameters based on them. "Raw" parameters are used directly. "Calculated" parameters are calculated based on other data, but are not entered directly by the user. The "override" section allows a derived model to override parameters defined in a base model, with different defaults or different methods of calculating.

```
independent {
  override {....} // override from base
  raw {....} // the classic "model card"
  calculated {....} // calculate these
  code_pre {....}
  code_post {....}
}
```

4.2.4 The "size_dependent" section

These parameters are adjusted based on size (length and width) The adjusted parameters are then used like the raw parameters. Each parameter implies "L", "W", and "P" variants.

```
size_dependent {
  override {....} // override from base
  raw {....} // classic "model card", almost
  calculated {....}
  code_pre {....}
  code_post {....}
}
```

4.2.5 The "temperature_dependent" section

These parameters are adjusted based on temperature The adjusted parameters are then used like the raw parameters. The adjustment may occur during simulation, for self-heating. Temperature is local and time-variant.

```
temperature_dependent {
  calculated {....}
  code_pre {....}
  code_post {....}
}
```

4.2.6 The "validate" section

A "validate" function checks the validity of the combined model parameters and device parameters. It returns a truth value, indicating whether this model is valid for this device, given the parameters. If there are multiple models in a group, it can be used to automatically select a good match, for automatic binning.

```
validate {
  return (c->length <= m->lmin);
}
```

4.2.7 tr_eval

This function does the real work. Has read access to model and common. Has write access to device (passed in). Computes the state variables. Does not do gather or scatter.

5 Architecture of the compiler

The compiler uses a data-flow oriented design, and is coded in C++. This design style groups code by stages in the data flow. This style was chosen over an object-oriented design because it is easier to change front ends and back ends. Making it easier to change the front end will enable it to support other input languages, such as Verilog-A, Verilog-AMS, and VHDL-AMS. Changing the back end could enable it to be ported to other simulators.

The input stage is a recursive descent parser using the public domain "argparse" parsing class, which is also used in the Gnucap simulator. It builds a data structure that mimics the input file. A simulator-dependent back end reads the data and builds the output file.

6 Results

Using the model compiler saves a significant amount of time in developing a model, while maintaining the high performance of hand coded models.

Porting Spice models is still somewhat of a nuisance, because they are already coded at a low level. Since Spice does not require the modularity that Gnucap does, it requires a significant effort to untangle the models. In particular, separating temperature effects from precalculations is a significant nuisance. This is necessary for Gnucap because of planned, support for self-heating effects.

Still, the biggest portion of the effort is in testing.