Chapter 10

COMPACT MODELING IN VERILOG-A

Boris Troyanovsky, Patrick O'Halloran, and Marek Mierzwinski
Tiburon Design Automation, Inc.
E-mail: boris@tiburon-da.com

Abstract: Historically, compact transistor models have been developed using general-purpose programming languages such as C or Fortran, with the resulting source code specifically targeted to a given circuit simulator's proprietary model interface. Although this approach has allowed for the creation of robust and efficient compact models, it has nevertheless resulted in a situation where the model development process is lengthy, the models are not portable across the various simulation environments, and where the model development facilities are often not open to independent model developers. The advent of analog hardware description languages (AHDLs) over the last several years promises to address the aforementioned issues by providing a portable, robust, and efficient platform for analog model development. In this chapter, we describe the Verilog-A language and explore the numerous benefits it provides in the area of compact modeling.

Key words: Verilog-A; AHDL; compact device model; transistor model; analog model.

1. Introduction and Overview

The availability of accurate, robust, and efficient compact models is critical to the successful utilization of any circuit simulation tool. As new physical effects manifest themselves due to shrinking geometries, and as an increasingly wide variety of highly specialized device technologies (e.g., RF CMOS, SiGe, III–V) become available to analog circuit designers, the need for rapid development and distribution of advanced semiconductor device models becomes more acute than ever. Traditionally, circuit simulators have relied largely on "built-in" semiconductor device models. Such built-in devices – typically implemented using general-purpose programming languages like C, C++, or Fortran – are targeted specifically to the interface and internal data structures of their host simulator, and are thus inherently non-portable. Facilities for adding custom models (or "user-defined devices") have been made available in some simulation environments, but such interfaces have typically been non-standard, non-portable, and inefficient. New model creation under these conditions was thus a time-consuming and error-prone endeavor.

The rapidly increasing availability and adoption of analog HDLs such as Verilog-A [1, 2] offers the promise of a comprehensive solution to the aforementioned analog model development and deployment problem. Initially conceived as a general-purpose analog modeling language, Verilog-A has over the past several years become increasingly viewed as a leading candidate for new compact model development [3–6]. Although the language has always been applicable across the full range of analog modeling tasks – from behavioral event-driven models all the way down to the transistor level – early Verilog-A implementations were interpreted solutions, and were not viewed as being viable alternatives to hand-coded built-in device models. The recent rise in interest for Verilog-A based compact model development has resulted in compiled solutions becoming available, with an ongoing emphasis on improved simulation performance.

The use of standardized, special-purpose analog HDLs such as Verilog-A allows device modeling experts to focus on their area of expertise, rather than on the underlying simulator-specific implementation details. The increased level of abstraction means that the model developer can focus on model behavior, and let the underlying implementation automatically take care of mundane (and often simulator-specific) details such as matrix stamping and loading, analysis-specific data structures, symbolic derivative computation, and so forth. The device modeling engineer is thus shielded from the idiosyncrasies and complexities of the various device interfaces in existence today.

2. Verilog-A Language Fundamentals

For model developers accustomed to working in a standard programming language such as C or Fortran, the switch to Verilog-A syntax should be straightforward and painless. The language is relatively succinct and compact, and is well-suited to analog model development. Several academic and industrial model development groups now use Verilog-A as a key part of their development methodology.

2.1. Introduction by Example

To illustrate the straightforward and intuitive nature of Verilog-A source code, we consider the following simple example.

```
module simple_diode(pos, neg);
inout pos, neg;
electrical pos, neg;

parameter real Area = 1.0 from (0:inf);
parameter real Is=1e-14 from [0:inf);
parameter real n = 2 from (0:inf);
parameter real Cjo=0 from [0:inf);
parameter real Phi = 0.7, m = 0.5, tt = 1p;

real Id, Qd;

analog begin
   Id = Area*Is*(limexp(V(pos, neg)/(n*$vt))-1);
   Qd = tt*Id + Area*V(pos, neg)*Cjo/
        pow((1-V(pos, neg)/Phi), m);
   I(pos, neg) <+ Id + ddt(Qd);
end
endmodule
```

The fundamental structural unit within Verilog-A is the module. In the first line of the code fragment above, we see that the module is named "simple_diode", and that it has two terminal connections (or ports, in Verilog-A parlance). The language supports the presence of non-electrical domains, such as electro-mechanical or thermal; for this simple diode example, the terminals (*pos* and *neg*) are labeled as being "electrical". (Some compact models incorporate thermal effects, and would thus use a "thermal" discipline for the thermal node.) Internal nodes are declared using the same syntax: if a discipline declaration is present for a node whose name does not match the module port list, that node becomes an internal node within the enclosing module.

The diode's parameters, including default values and allowable ranges, are specified after the terminal disciplines, and two local real variables (*Id* and *Qd*) are then declared. Both real and integer-valued quantities are allowed, and arrays of variables (or parameters) are allowed as well. In a subsequent Section (2.7) we will also encounter the Verilog-A specific variable type known as a "genvar".

Following the variable and parameter declarations, we come to the heart of the model's numerical description – the analog block. Each module can contain at most one analog block, where the module's analog behavior is specified. Because Verilog-A allows hierarchical constructs (Section 2.8), some modules can merely instantiate other modules as child instances and connect them electrically. In these cases, the analog block need not be present.

Our simple diode example has no hierarchical constructs within it, and the diode description resides solely within the analog section. Straightforward mathematical expressions are used to assign physically meaningful values to *Id* (the diode current) and *Qd* (the charge). The resulting current is then directed to the output terminals via the contribution statement:

```
I(pos, neg) <+ Id + ddt(Qd);
```

So long as the target of the contribution statement does not switch from current to voltage (or vice versa), the contributions are all additive. The following two statements would be equivalent to the previous one:

```
I(pos, neg) <+ Id;
I(pos, neg) <+ ddt(Qd);
```

In the next several sections, we present a more detailed overview of the Verilog-A language structure, with particular emphasis on constructs important to the compact model developer.

2.2. Contributions and Branches

Verilog-A uses the so-called "source/probe" formulation for describing the behavior of electrical networks. Consider a pair of electrical nodes, named $n1$ and $n2$. As we saw in the previous section, we can "probe" the voltage between them via the expression $V(n1, n2)$. To insert a current source (a "flow-branch" or "current branch" in Verilog-A parlance) between the two nodes, we would use the contribution statement:

```
I(n1, n2) <+ Idc;
```

Similarly, to insert a voltage source (also called a "voltage branch" or a "potential source") between nodes $n1$ and $n2$, the contribution statement:

```
V(n1, n2) <+ Vdc;
```

would be used. The presence of either of these two contribution statements introduces an "unnamed branch" between the two nodes. Explicit named branches can also be introduced via declarations of the form:

```
branch (n1, n2) br_res;
branch (n1, n2) br_cap;
branch (n1, n2) br_ind;
```

and contributed to by statements such as:

```
I(br_res) <+ V(br_res)/R;
I(br_cap) <+ C*ddt(V(br_cap));
```

```
V(br_ind) <+ L*ddt(I(br_ind)) + RL*I(br_ind);
```

Explicitly named branches can be useful in those cases where the user is interested in current flow through the named branch only, either for direct output or for use in another expression:

```
Pdiss_L  = I(br_ind)*I(br_ind)*RL;
Pdiss_R1 = I(br_res)*I(br_res)*R;
```

As we explain in more detail later, most compact modeling applications should attempt to probe *voltage* (i.e., use $V(\ldots)$ only on the right hand side) and contribute to *current* (i.e., use $I(\ldots)$ only on the left hand side) whenever possible. Failure to do so may result in the introduction of additional state variables, causing the simulation to be slower and more memory-intensive than would otherwise be the case. For example, a nonlinear capacitor should be implemented as:

```
I(p, n) <+ ddt(cap(V(p, n));
```

rather than the alternate (and usually less efficient) choice:

```
V(p, n) <+ idt(f(I(p, n));
```

In most implementations, the second choice will result in the introduction of additional state variables into the system.

For some components, of course, the preceding rule of thumb is not applicable. A truly voltage-controlled component such as an inductor should be implemented as:

```
V(p, n) <+ ddt(phi(I(p, n));
```

where (for the sake of generality) we have used the analog function "phi" to refer to the potentially nonlinear inductance characteristic. Although this formulation will introduce an extra state variable into the system, the voltage-controlled nature of the component makes this intrinsically necessary. The alternate integral-based implementation:

```
I(p, n) <+ idt(g(V(p, n));
```

does not use any fewer state variables than the *ddt*-based implementation.

Before concluding this section, we briefly discuss the topic of current probes. As we have seen, probing voltage in Verilog-A is simple and straightforward. Probing current – although syntactically just as simple – requires a bit more care. Using the expression $I(n1, n2)$ on the right-hand side of a contribution statement yields the current flowing in the unnamed branch between nodes $n1$ and $n2$. Similarly, to probe the current through a named branch $br1$, the syntax $I(br1)$ would be utilized. If the branch being probed is not contributed to, the two terminals of the branch are effectively shorted together. Most Verilog-A implementations will insert an extra state variable to probe the current through the branch, and thus it is desirable to avoid using current probes when possible. For example, the code fragment:

```
I(n1, n2) <+ V(n1, n2)/R;
x = f(I(n1, n2));
```

would typically be less efficient than the analogous:

```
I(n1, n2) <+ V(n1, n2)/R;
x = f(V(n1, n2)/R));
```

Current flow into module ports (terminals) can be probed with the expression *I*(<port_name>), where *port_name* is the name of the port. Note that it is usually an error to write *I*(port_name) instead, as the use of this expression on the right-hand side will create an unnamed shorted branch from *port_name* to ground, and thus almost certainly cause the model to behave in an undesirable way.

2.3. Analog Operators

As a general-purpose analog modeling language, Verilog-A includes a large number of "analog operators" that can be applied to signal waveforms. In addition to conventional operations such as differentiation, integration, and delay, the language also provides the transition, slew, circular integration, laplace transform, and Z-transform operators (see table below).

For compact model development, it is seldom necessary to use analog operators other than ddt. Occasionally, short delays may be used for some device applications, and laplace operators can sometimes prove useful for modeling passive circuitry or packaging outside the device. It is generally best to avoid the use of the integrator with initial conditions, the slew and transition filters, and the Z-transform operator because usage of these facilities restricts the range of analysis types that the model is suitable for [10]. Fortunately, there is almost never a need for such operations in compact modeling work.

Analog operators/ Waveform filters	ddt(x [,abs_tol])	Differentiate 'x' with respect to time.
	idt(x, [ic [, assert [, abs_tol]]])	Integrate 'x' with respect to time with initial condition 'ic.'
	idtmod(x, [ic [, modulus [, ffset]]])	Circular integration of 'x' with respect to time with initial condition 'ic' using modulus and offset.

transition(x [, delay [, rise_time [, fall_time]]])	Control details of signal transition expression ‘x.’
slew(x [, max_pos [, max_neg]])	Control slew rate behavior of expression ‘x.’
absdelay(x, time_delay, max_delay)	Output(t) = x(time − time_delay).
zi_nd(x, num, denom, period, [transition_time [,sample offset time])	z-domain filter function using numerator-denominator form.
zi_zd(x, zeros, denom, period, [transition_time [,sample offset time])	z-domain filter function using zero-denominator form.
zi_np(x, num, poles, period, [transition_time [,sample offset time])	z-domain filter function using numerator-pole form.
zi_zp(x, zeros, poles, period, [transition_time [,sample offset time])	z-domain filter function using zero-pole form.
laplace_nd(x, num, denom, [, abs_tol])	s-domain filter function using numerator-denominator form
laplace_zd(x, zeros, denom, [, abs_tol])	s-domain filter function using zero-denominator form
laplace_np(x, num, poles, [, abs_tol])	s-domain filter function using numerator-pole form
laplace_zp(x, zeros, poles, [, abs_tol])	s-domain filter function using zero-pole form

2.4. Noise

Noise analysis is an area of key importance for many analog applications, and thus comprehensive noise support is a requirement for compact model development. To this end, Verilog-A provides the *white_noise*, *flicker_noise*, and *noise_table* functions.[1] For example, a noisy resistor would be modeled as:

```
I(p, n) <+ V(p, n)/R + white_noise (4*`P_K*$temperature*R);
```

whereas shot noise could be added via the expression:

```
I(b, c) <+ white_noise (2*`P_Q*Ic);
```

As we see above, the noise power arguments can be a function bias. A list of the available noise sources is given in the table below.

Noise functions	white_noise(power [, label])	Generate white noise of power 'power.' Contributions with the same label 'label' are combined for a module by the simulator.
	flicker_noise(power, exp [, label])	Generate pink noise of power 'power' at 1 Hz that varies in proportion to 1/f^exp. Contributions with the same label 'label' are combined for a module by the simulator.
	noise_table(vector [, label])	Generate noise where power is described by linear interpolation from vector 'vector' of frequency-power pairs. Contributions with the same label 'label' are combined for a module by the simulator.

In Verilog-A, each noise source is, by definition, independent. Correlation effects between noise sources can be modeled through linear combinations of real variables which are functions of the independent sources. For example, suppose that we would like to have a source $n1$ with power $P1$ and source $n2$

[1]The noise_table function may not be supported for some types of RF noise analysis. Its use should be avoided if possible.

with power $P2$, correlated to each other with a coefficient of K. This can be achieved by introducing three independent noise sources:

```
A = white_noise(K);
B = white_noise(P1-K);
C = white_noise(P2-K);
```

and then linearly combining them into the desired noise sources $n1$ and $n2$ as:

```
n1 = A+B;
n2 = A+C;
I(a, b) <+ n1;
I(c, d) <+ n2;
```

2.5. Analog Functions

Analog functions – sometimes referred to as user-defined functions – are directly analogous to their counterparts in conventional programming languages. Their primary role is to improve the readability and structure of a given analog block by encapsulating potentially complicated mathematical functionality. In some cases, using analog functions (instead of macros, for instance) can also lead to a smaller memory footprint.

Analog functions take as input a sequence of real or integer arguments, and return a real or integer value. In the 2.1 version of the standard, the arguments and return value are restricted to be scalar, and the arguments are passed by value. The 2.2 language standard allows the arguments to be arrays, and also allows them to be passed by reference (i.e., the function can effectively return values to the caller).

As an example of a typical analog function definition, we consider the following excerpt from a bipolar transistor model.

```
analog function real I_of_T;
   input IS, T, T_NOM, EG, N, Vth, XTI, XTB;
   real IS, T, T_NOM, EG, N, Vth, XTI, XTB;
   real ratioT;

   begin
     ratioT = T/T_NOM;
     I_of_T = IS / pow(ratioT, XTB) * exp((ratioT-1)*EG/
              (N * Vth))*pow(ratioT, XTI/N);
   end
endfunction // I_of_T
```

This function would be called from the module with the syntax

```
ISE_T = I_of_T(ISE, T, T_NOM, EG_T, NE, Vt, XTI, XTB);
ISC_T = I_of_T(ISC, T, T_NOM, EG_T, NC, Vt, XTI, XTB);
```

2.6. System Tasks

The Verilog-A language provides a set of “system tasks” which allow modules to interact with the simulation environment and with the input/output system. Each system task statement begins with the “$” character, followed by the name of the task and a parenthesized argument list. System tasks of interest to the compact model developer include the *$strobe* and *$debug*[2] calls for textual output to the display:

```
$strobe("V(coll) = %e", V(coll));
$debug("V(base)  = %e", V(base));
```

The *$strobe* task outputs its results at every converged solution point. In contrast, the *$debug* task generates output at every single Newton iteration, and (as its name suggests) is thus useful for debugging purposes. Numerous other system tasks are of course present in the language, and the reader is advised to consult [1] for a detailed list.

In addition to “system tasks”, Verilog-A also provides several “system functions” (also occasionally referred to as system calls). These are distinct from system tasks in that they are function calls returning real-valued expressions, whereas the system tasks are statements that do not provide a return value. Of particular interest to compact model developers are *$temperature* (which returns the temperature in Kelvin), *$vt* (which returns the thermal voltage), and *$abstime* (which returns the current simulation time). Although compact models should not explicitly rely on time for their current-voltage characteristics, the *$abstime* call can be very useful for data display and debugging.

2.7. Conditional Statements, Looping Constructs, and Genvars

Conditional statements and for-loops are very useful language constructs for device modeling. Their usage in Verilog-A is similar to their usage in standard programming languages such as C, with one important distinction – analog operators (Section 2.3) may be used inside the body of these constructs only if the controlling expression is not a function of the state variables.[3] The restriction exists because analog operators must store their state internally, and thus need to monitor their arguments during the course of an entire analysis. To illustrate the restriction, consider the following simple code snippet:

```
if(V(ctrl) > 0) begin
   x = V(a, b);    // this is legal
```

[2]The $debug task is only present in the 2.2 (and later) versions of the standard.
[3]In the language of the standard, this is referred to as a “genvar expression”.

```
      I(c, d) <+ V(a, b); // this is legal as well
      x = ddt(V(a, b)); // this is illegal:
                        // analog operator prohibited here
   end
```

The entire code fragment above would be legal if V(ctrl) was replaced by a parameter type.

For-loops, although not as widely used as conditional statements, are still quite commonplace in compact modeling applications. They are particularly useful for such tasks as looping through the fingers of a multi-finger device or iterating through the emitters of a multiple-emitter transistor. For those situations where analog operators are needed within the body of a for-loop, the language introduces the "genvar " type of integer-valued variable. Variables which are declared as *genvar* may only be initialized within the controlling expressions of a *for*-loop statement, and may only be functions of static expressions (i.e., ones which are not functions of state variables, and are thus not dependent on bias). As an example of this concept, the code fragment below would insert a linear parallel RLC network into each of the *NF* fingers of a Verilog-A device:

```
electrical [1:`NF] no, ni;
real vk;
real [1:`NF] R, C, L;
// Code to initialize R/L/C arrays goes here...
genvar k;
for(k = 1; k <= `NF, k = k+1) begin
   vk = V(no[k], ni[k]);
   I(no[k], ni[k]) <+ vk/R[k] + C[k]*ddt(vk) + L[k]*idt(vk);
end
```

2.8. Hierarchical Module Instantiation

Verilog-A modules can be hierarchical in nature – each module may itself instantiate an arbitrary number of sub-modules. For example, if a detailed Verilog-A model for a bias-dependent junction capacitor has been written, a MOS model could instantiate instances of it with the statements

```
juncap #(.TRJ(TRJ1), .DTA(DTA1), ...  ) JUNCAPsource(BS, S);
juncap #(.TRJ(TRJ2), .DTA(DTA2), ...  ) JUNCAPdrain(BD, D);
```

The syntax above indicates that two juncap devices will be placed hierarchically within the parent module. The first of these would be named *JUNCAPsource*, and attached between nodes *BS* and *S*, while the second (named *JUNCAPdrain*) would be connected to nodes *BD* and *D*. Parameters are passed from the parent module to the child sub-device through a comma-separated list after the ‘#’ symbol. The values may themselves be functions of other parameters:

```
parameter real L = 0.1u from (0, inf];
parameter real W = 0.5u from (0, inf];
some_device #(.Area(L*W)) dev(n1, n2, n3);
```

2.9. Events and Memory States

As a general-purpose modeling language, Verilog-A includes some behavioral constructs that are best avoided in compact modeling applications. Chief among these are events (e.g., *@(cross)* and *@(timer)*) and the use of “memory states”, which are further explained below.

The Verilog-A language standard mandates that local variables are initialized to zero at the beginning of the simulation, and that they retain their value after a given time point has converged. If a variable is used before it is assigned in a given module, it takes on the value from the previously-converged time point. We refer to such variables as “memory states”. (In other literature [8], such variables may be referred to by other names, such as “hidden states”.) Although such variables can be very useful for behavioral modeling applications, they clearly have very limited utility in compact model development. One possible use would be to limit the display of diagnostic information. For example, to print a warning only once, we could structure the code as follows:

```
integer warn_flag;
if(!warn_flag && R < 0) begin
   $strobe("Negative resistance in module %m");
   warn_flag = 1;
end
```

In addition to representing a questionable formulation from the standpoint of physical reality, modules with memory states can pose problems for RF simulation algorithms like harmonic balance and periodic shooting [9]. Indeed, for methods such as harmonic balance – which do not rely on conventional time-marching algorithms at all – the whole concept of memory states is particularly problematic. In the area of compact modeling most memory states can usually be attributed to an inadvertent mistake in variable usage, and compact model compilers should automatically warn the model developer of their presence [10].

3. Compact Model Development

3.1. Numerical Considerations

Circuit simulation algorithms are generally iterative in nature, and are typically based on the classical Newton-Raphson technique. To facilitate robust convergence, semiconductor device models should have smooth, differentiable characteristics, and should guard against floating point exceptions that can occur during the course of iterating to a solution. It is important to remember that state variables can assume non-physical values during the course of the iterative process, and that "reasonable" nodal values are only guaranteed once the system has converged to a valid solution.

3.2. Model Topology

The program-flow aspect of the Verilog-A language tends to be straightforward, intuitive, and very similar to the languages that compact model developers are accustomed to. The contribution statements specifying model topology – while also fairly intuitive – do not have any direct counterparts in conventional programming languages, and consequently merit some additional discussion.

Most circuit simulators use Modified Nodal Analysis (MNA) [11] or something very similar to formulate the circuit equations. Consider a simple linear resistor, connected between nodes $n1$ and $n2$, represented by the constitutive equation $I = V/R$ (or $I(n1, n2) < +V(n1, n2)/R$ in Verilog-A). A conventional circuit simulator would have state variables corresponding to nodes $n1$ and $n2$, and all components connected to these nodes would contribute terminal currents to the relevant Kirchoff's Current Law (KCL) equations. The resistor component would simply add the current V/R to one of the nodes, and subtract V/R from the other node. No additional equations or state variables would be necessary.

In contrast, consider a voltage source placed between nodes $n1$ and $n2$. The Verilog-A constitutive relation for this component takes on the form $V(n1, n2) < +Vdc$, and in this case cannot be expressed in a "voltage-controlled" formulation. To handle this scenario, typical circuit simulators proceed to create a new variable representing the current through the source, and then add the equation $Vn1 - Vn2 - Vdc == 0$ as an extra row in the system.

The formulation method is important for compact model developers, since it can impact the size of the matrix. Because performance is a key issue in compact model development, it is important to have a good understanding of when "extra" state variables may be introduced by the various language constructs. The general rule of thumb (described in Section 2.2) is that contributing *to* a voltage (i.e., having $V(\ldots) < +$ on the left-hand side) or sensing a current (i.e., using $I(\ldots)$ within a right-hand side expression) can lead to the insertion of extra state variables.

One issue that frequently comes up in compact modeling work is the presence of optional parasitic resistors on the device terminals. These cannot be portably implemented using the standard contribution statement:

```
I(e, ei) <+ V(e, ei)/Re;
```

because the resistor value may be zero. The obvious solution – that is, using the voltage contribution:

```
V(e, ei) <+ Re*I(e, ei);
```

is portable and robust. However, as we have seen previously, this formulation will introduce an extra state variable for the resistive branch. For the case where the resistor value is zero, this results in the creation of not one but two extra state variables – one for the internal node *ei*, and one for the current through the voltage branch between nodes *e* and *ei*.

To overcome the aforementioned problem, modern compact model compilers will often make a special allowance for the following idiom:

```
if(Re > 0.0)
   I(e, ei) <+ V(e, ei)/Re;
else
   V(e, ei) <+ 0;
```

So long as *Re* is a static expression (i.e., one that does not depend on the values of the state variables) the model compiler will "collapse" the nodes *e* and *ei* into a single state variable if the parasitic resistance value is zero. In the case of implementations which do not special-case this construct, the code fragment will introduce a "switch branch" but will still execute correctly and be fully compliant with the language standard.

3.3. Compact Modeling Extensions

The recent 2.2 release of the language standard [1] has added several features of interest to compact model developers [7, 12]. Facilities have been added for explicit derivative access, portable output of local variables, efficient and convenient representation of parameter sets, more flexible specifications of user-defined analog functions, as well as several other common tasks. Table-based modeling support has also been added as a standard feature, enabling compact models to utilize table-driven characteristics.

Because the new standard has only recently been released, support for these features is not yet widely available across the various Verilog-A distributions. If portability is important, models utilizing the new feature set can check the predefined macro 'VAMS_COMPACT_MODELING; implementations that support the compact modeling extensions will have this definition present. Constructs that are dependent on the 2.2 feature set can be placed within an *'ifdef* for backward compatibility with earlier implementations.

A full detailed discussion of the new feature set is beyond the scope of this chapter; for more information, the reader is directed to the language standard [1]. Here, we present a brief overview of some of the more useful 2.2 functionality.

3.3.1. The ddx operator

Although Verilog-A compilers must internally compute symbolic derivatives to ensure that the Newton-Raphson process exhibits robust convergence, the language standard prior to version 2.2 did not allow model developers direct access to symbolic derivative information. This situation has been remedied with the introduction of the *ddx* operator. The *ddx* operator takes two arguments – the expression to be differentiated, and the state variable with respect to which the differentiation should take place:

```
Id = Area*Is*(limexp(V(pos, neg)/(n*$vt))-1);
Qd = tt*Id + Area*V(pos, neg)
     *Cjo/pow((1-V(pos, neg)/Phi), m);
Gd = ddx(Id, V(pos));
Cd = ddx(Qd, V(pos));
```

It is important to keep in mind that the derivative is a true partial derivative – that is, all state variables (i.e., nodal voltages and branch currents) except the one being differentiated with respect to will be held fixed. The state variables being held fixed should be distinguished from the local module variables, which may of course vary with the "with respect to" state variable.[4]

An important point is that differentiation with respect to a voltage difference is not allowed. For example, it may be tempting to calculate transconductance for a BJT as:

```
Gm = ddx (Ic, V(b, e)); // error!
```

This formulation has two inherent problems. The first of these is that differentiation with respect to a voltage difference is forbidden by the standard (as we saw above). The second issue is that nodes *b* (base) and *e* (emitter) in most bipolar models will represent the external device nodes, connecting to the intrinsic model only through the parasitic lead resistors. As such, partial derivatives with respect to these nodes will not yield the derivative derivative value that the model developer desires, since these unknowns are independent of the intrinsic device's nodal values under partial differentiation.

3.3.2. Output variables

Built-in devices in spice-like circuit simulators are typically able to output internal information such as small-signal operating point values for a given

[4]Note that this variation is only conceptual in nature; there is no numerical limiting process, since the standard specifies that the derivative should be "exact" in a symbolic sense.

model instance. To enable portable output of model-specific data, the 2.2 standard now mandates that module-scope variables with description and unit attributes should be output to the data set. For example, if the real variable *Gd* of the preceding section was declared as

```
(* desc = "diode conductance", units = "mhos" *) real Gd;
```

then the value of *Gd* would be made available for output and plotting at every solution point (including every time point of transient analysis).

3.3.3. Limiting functions

Circuit simulators have traditionally employed solution algorithms which effectively “limit” the potentially sharp changes in nodal values that can occur during the Newton-Raphson iterative process due to strong device nonlinearities. Prior to the 2.2 standard, the *limexp* operator was provided to fulfill this role when dealing with exponential junction nonlinearities. However, the *limexp* facility did not have the full generality available to built-in junction limiting algorithms, and was clearly not applicable to other forms of limiting that are sometimes used for various compact models.

To address this situation, the *$limit* facility was introduced into the 2.2 language standard. In addition to providing a flexible interface for user-specified limiting algorithms, the standard also recommends that simulation environments provide default implementations of the common “pnjlim” and “fetlim” algorithms, which presumably are used by the built-in (native) devices. This allows compact models written in Verilog-A to employ limiting algorithms that are consistent with their native counterparts. For more detailed information, the reader is referred to [7] and Section 10.9 of [1].

4. Examples

Verilog-A enables an efficient and fast process for compact model developers to create and distribute models. Already many popular models are available in Verilog-A format from a variety of sources (see table below). However, for this process to have wide acceptance, the experience of the end-user of the model must be much the same as it is with the current model distribution methods. That is, the model must be available in all the available analyses, the simulation results must be identical, and the simulation performance must not be impaired. This section illustrates how both industry-standard and complex models implemented in Verilog-A perform with an identical use-model as far as the end-user is concerned.

Model Type	**Models**
BJT	SPICE-GP, HiCUM, MEXTRAM, VBIC
MOSFET	BSIM3, BSIM4, BSIM5, BSIMSOI, MOS11, PSP, EKV, RPI-Shur TFT
GaAs FET	Angelov, Curtice, Parker-Skellern, TOM1/3

4.1. Angelov-Chalmers GaAs FET Model

The Angelov-Chalmers GaAs FET model is a prominent compact model used in high frequency circuit designs [13]. It delivers good representation of high power behavior while also providing good prediction of harmonics.

It is a relatively straight-forward model to code; however, it is not available in all simulators since it is used by only a small segment of the design community. The model requires less than four hundred lines of Verilog-A code, including parameter definitions; actual behavioral expressions are less than two hundred lines of code.

The *I*–*V* characteristics shown in Figure 1 compare the results of a simulation using the Verilog-A model to the simulator's built-in version. As can be seen, the results are identical, as one would expect. From the user's perspective the model behaves and performs as though it were a natively coded model. Besides the advantage of easy-access to the code for modifications and extensions, a Verilog-A implementation of the Angelov-Chalmers model provides access to the model in simulators where the vendors have not provided a native version.

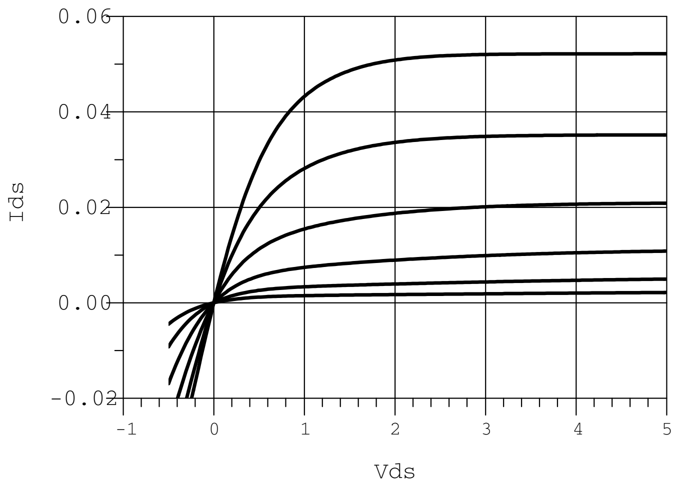

Figure 1. *I*–*V* characteristics of Built-in and Verilog-A versions of Angelov FET model.

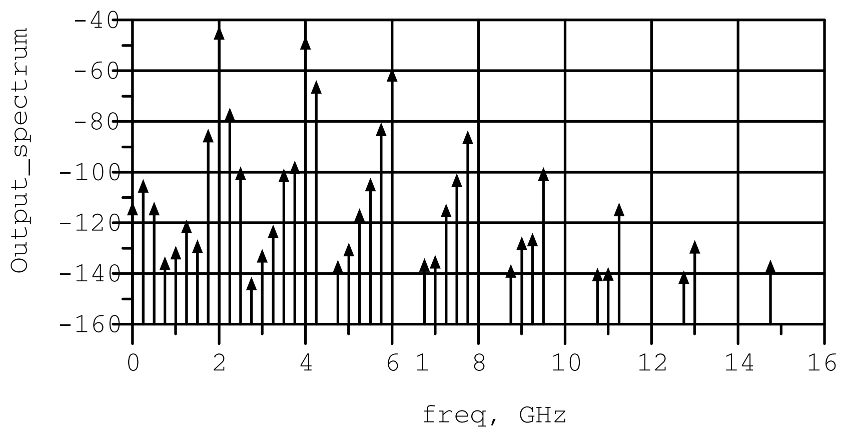

Figure 2. Output spectrum of EKV mixer analyzed in a commercial simulation program that does not natively support the EKV model.

4.2. EKV Model

The EKV model is another example of a popular MOSFET model that has been implemented in many, but not all, simulators. However, a Verilog-A version was released by the developers and this provides access to the model in simulators where it has not been implemented. Figure 2 illustrates a frequency domain simulation of an EKV mixer in a simulator that does not natively support the EKV model.

4.3. SPICE Gummel-Poon BJT

The SPICE Gummel-Poon BJT model is provided in virtually every analog simulator. Even though developed a half-century ago, until recently it has been general enough to sufficiently model advances in device technology. New compact models have been developed to address these improvements in topology and scaling. These recent models are more complicated, requiring more effort to extract the model parameters and using more simulation resources during analysis. However, in many cases minor modifications to the Gummel-Poon model would still be sufficient to accurately predict circuit performance. Verilog-A implementations of the BJT model allow users to add only the necessary behavior without adding unnecessary complications. For example, self-heating is an effect that is included in all of the next generation BJT models. It is important for devices used for power generation, or in materials with poor thermal characteristics. The self-heating effect can be modeled with just a few lines of Verilog-A code. A similar implementation in C-code, assuming the end-user had access to the code, would be much more involved as it would be up to the developer to provide the numerous associated thermal derivatives for the simulator.

To demonstrate how Verilog-A models can be used in any analysis type, including frequency domain simulations such as harmonic balance, a real-world circuit using both Verilog-A compact models, Verilog-A behavioral models, and native simulator models for a modulator and demodulator.

Figure 3 shows the circuit schematic layout while the associated output is presented in Figure 4. The magnitude of the output is plotted along with the output for the Verilog-A model when self-heating is enabled.

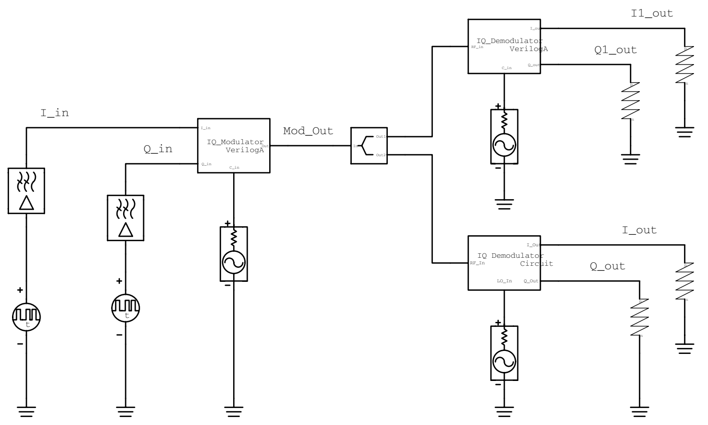

Figure 3. Schematic for a modulator-demodulator circuit employing Verilog-A for both compact models and behavioral models.

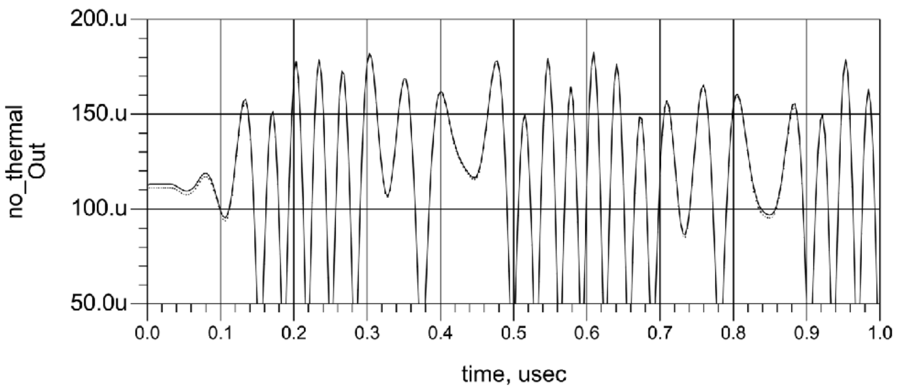

Figure 4. Output of demodulator in the time domain for a conventional SPICE Gummel-Poon model compared to the same model with a self-heating thermal circuit.

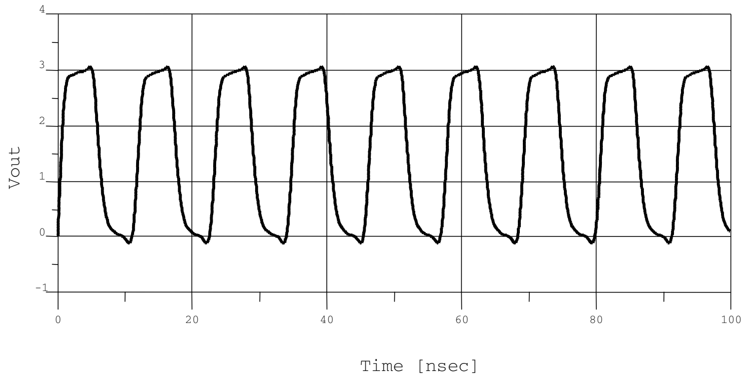

Figure 5. Ring oscillator output for both a native BSIM3 model and its Verilog-A equivalent.

4.4. BSIM3 MOSFET Model

The BSIM3 MOSFET model is the most extensively used compact model for analog and digital designs. It is the third generation model of the BSIM family and was developed with the intent of providing good fit to the underlying process as well as good mathematical behavior with respect to convergence. It is a complicated model with tens of thousands of lines of C-code and with hundreds of parameter values. In comparison, the Verilog-A implementation requires about one tenth the number of lines of code.

Since the model equations' derivatives are automatically generated, there is less chance of coding errors. This helps to accelerate the time it takes to get complex models out to the end-user. With shrinking geometries and novel device topologies, it is more difficult for any one compact model to accurately portray the device characteristics. Verilog-A allows new models to reach the end-user quicker; and for end-user feedback to return back to the model developer for model improvements.

Ring oscillator circuits are a simple way to exercise the model in a nonlinear manner. Small deviations in the models will result in large changes in the frequency of operation. Figure 5 shows the output of a ring oscillator for the built-in BSIM3 model and the Verilog-A equivalent model. As can be seen, the C-coded and Verilog-A models perform virtually identically.

References

[1] Verilog-AMS Language Reference Manual, Version 2.2, Accellera International, Inc.
[2] VHDL 1076.1 Language Reference Manual, IEEE.
[3] Lemaitre, L.; McAndrew, C.; Hamm, S. "ADMS – automatic device model synthesizer", *Proc. IEEE CICC*, **May 2002**, 27–30.

[4] Kundert, K. “Automatic model compilation – an idea whose time has come”, *The Designer’s Guide*, **May 2002**, (http://www.designers-guide.com).

[5] Mierzwinski, M.; O’Halloran, P.; Troyanovsky, B.; Dutton, R. “Changing the paradigm for compact model integration in circuit simulators using Verilog-A”, **February 2003**, 376–379.

[6] Troyanovsky, B.; O’Halloran, P.; Mierzwinski, M. “Portable high – performance models using Verilog-A”, *IEEE Conf. MTT*, **June 2003**.

[7] Coram, G.J.; “How to (and how NOT to) write a compact model in Verilog-A”, *Proc. BMAS 2004*. CA: San Jose, **October 21-22, 2004**, 97–106.

[8] Kundert, K. “Hidden State in SpectreRF”, *The Designer’s Guide*, **May 2003**, (http://www.designers-guide.com).

[9] Kundert, K.; White, J.; Sangiovanni-Vincentelli, A. *Steady-State Methods for Simulating Analog and Microwave Circuits*. Kluwer Academic Publishers, **1990**.

[10] Troyanovsky, B.; O’Halloran, P.; Mierzwinski, M. “Analog RF model development with Verilog-A”, *IEEE Radio Frequency IC Sympos.*, **June 2005**.

[11] Kundert, K. *The Designer’s Guide to SPICE and Spectre*. Kluwer Academic Publishers, **1995**.

[12] Lemaitre, L.; Coram, G.; McAndrew, C.; Kundert, K. “Extensions to Verilog-A to support compact device modeling”, *Proc. BMAS*, **October 2003**, 134–138.

[13] Angelov, I.; Zirath, H.; Rorsman, N. “A new empirical nonlinear model for HEMT and MESFET devices”, *IEEE Trans. Microwave Theory and Tech.*, **December 1992**, *40(12)*.