Well-Posed Device Models for Electrical Circuit Simulation

A Guide to Creating Robust Device Models

A. Gokcen Mahmutoglu, Tianshi Wang, Archit Gupta and Jaijeet Roychowdhury

March 25, 2017

Synopsis

This document provides guidelines for creating computational device models that work well in simulation. We build our discussion around the mathematical notion of "well-posedness". We show that the requirements for a model to be well-posed stem from the internal working mechanisms of simulators. Therefore, our main aim is to provide insight into the numerical procedures used by simulators in order to help model developers avoid ill-posedness issues. We start our discussion with an example that shows how an ill-posed Verilog-A model can produce different simulation results in different simulators. We then provide a step-by-step simulation case study. In this case study, we illustrate the role of device models in simulations by examining the steps a simulator goes through, from taking a netlist as input to producing a simulation result as output. Finally, we distill our discussion in a functional definition of a well-posed model. As an extension to our theoretical discussion, we also provide practical guidelines that should be followed by Verilog-A models in order to avoid ill-posedness issues.

This document is published as a part of the Nano-Engineered Electronic Device Simulation (NEEDS) initiative. NEEDS is an NSF-funded initiative whose charter includes the development of tools and techniques for the production of high-quality device models[1]:

> *"NEEDS has a vision for a new era of electronics that couples the power of billion-transistor CMOS technology with the new capabilities of emerging nano-devices and a charter to create high-quality models and a complete development environment that enables a community of compact model developers.*
>
> *NEEDS Team: Purdue, MIT, UC Berkeley, and Stanford."*

[1] For more information about NEEDS please visit https://nanohub.org/groups/needs/.

1 Introduction

Circuit designers rely on accurate simulators and simulators rely on accurate device models. For a device model, being accurate means featuring the right physical characteristics, *i.e.*, consisting of equations that capture every relevant physical effect. However, it also means offering an accurate and *robust numerical representation* of the physical device inside the simulator. Broadly speaking, we call device models *well-posed* if they are numerically robust representations of actual physical devices. In this document we unpack the concept of well-posedness with the aim of helping model developers to create device models that work well in various types of simulations and analyses.

Just like device models mimic physical objects, the simulator mimics the rules of physics. However simulators use numerical algorithms to perform this task and, therefore, they are subject to numerical limitations that are absent from the actual physical world. This in turn imposes limitations on the numerical techniques that can be used by device models. This ultimately means that simply putting the correct physical equations together is not enough to create a good device model. In this document, we explain what these limitations are and why they are important by providing insight into their origins.

Most ill-posedness problems in device models can be avoided by having a better understanding of different aspects of the role of a device model in simulation. In the subsequent sections we examine device models from three different perspectives.

i. Mathematical: We show that in the core of every device model there is a system of Differential Algebraic Equations (DAEs).
ii. Numerical: We demonstrate how the model DAEs are used in numerical simulations and analyses.
iii. Practical: We give examples of device code which cause ill-posedness issues and provide recommendations on how to avoid them.

The structure of the rest of this document is as follows. In Section 2 we start with a short introduction to the Verilog-A behavioral modeling language and also introduce software tools, the Berkeley Model and Algorithm Prototyping Platform (MAPP) and the Berkeley Verilog-A Parser and Processor (VAPP), which we will use throughout this document. Section 3 introduces the concept of an ill-posed model without delving into numerical issues. We demonstrate how the most fundamental feature of the Verilog-A language (the contribution operator) can be used in a way that produces an ambiguous, ill-posed device model. In Section 4, we take the first step towards understanding what device models do inside simulators by demonstrating how a simple single transistor circuit is converted into a set of DAEs before any kind of numerical treatment is performed by the simulator. This conversion process and the subsequent numerical methods used by the simulator are key to understanding the necessary qualities for a model to be well-posed. We give a functional definition of well-posedness in Section 5 by listing the properties of a well-posed model. Section 6 offers a comprehensive guide on how to use Verilog-A to create models that are well-posed and compliant with the requirements of the NEEDS initiative.

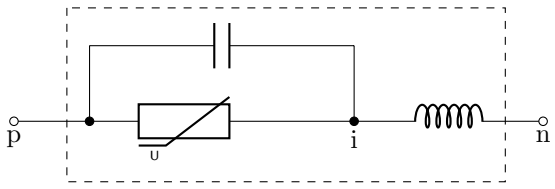

Fig. 1: A nonlinear resistor with a parallel capacitor and a series inductor.

2 Practical Device Modeling with Verilog-A and MAPP

In this section, we provide a short review of some practical aspects of device modeling using Verilog-A and introduce software tools for model developers that were developed within the framework of the NEEDS initiative: MAPP and VAPP.

2.1 The Verilog-A Modeling Language

Verilog-A is a domain specific, behavioral modeling language which has been adopted by the device modeling community as the default tool for developing and distributing compact device models. Detailed guides and tutorials for Verilog-A are readily available [1–3]. Here we will provide only a short introduction into the Verilog-A syntax and constructs that are essential for the rest of this document[2].

Listing 1: Verilog-A model code for the circuit shown in Figure 1.

```
1  `include "disciplines.vams" // disciplines, electrical etc.
2  module nlresistor(p, n);    // model name and terminals
3    electrical p, i, n;       // nodes (internal ones too)
4    branch (p,i) br_res;      // call p->i branch br_res
5    branch (i,n) br_ind;      // call i->n branch br_ind
6    // define circuit parameters
7    parameter real R0 = 1e3 from [0:inf); // resistance
8    parameter real C = 1.0e-12 from (0:inf); // capacitance
9    parameter real L = 1.0e-15 from (0:inf); // inductance
10
11
12   analog function real nlfunc; // nonlinearity function
13       input v_in, G;        // define input variables
14       output i_out;         // define output variable
15       real v_in, i_out, G;
16
17       begin
18         i_out = G*pow(v_in,3); // function core
19       end
20   endfunction
21
```

[2] For further information on Verilog-A, the interested reader is referred to the relevant literature [1, 4].

```
22      analog begin
23        I(br_res) <+ nlfunc(V(br_res), 1/R0); // resistive
24        I(br_res) <+ ddt(C*V(br_res));        // contributions
25        V(br_ind) <+ ddt(L*I(br_ind)); // capacitive contrib
26      end
27    endmodule
```

The example device model in Listing 1 is provided to illustrate the core concepts of Verilog-A. This piece of code represents a nonlinear resistor with an ideal capacitor connected to it in parallel and with an ideal inductor in series (shown in Figure 1). Lines 3 through 6 in Listing 1 define the model topology (terminals and internal nodes). The user defined function in lines 13–21 prescribes the nonlinear behavior of the device, and the model core in lines 23–27 associates this behavior with the previously defined model topology using the Verilog-A's contribution operator (<+).

The widespread adoption of Verilog-A by the circuit simulation community is not a coincidence. Verilog-A provides developers of circuit simulators with a way of importing device models that were not specifically developed for that simulator. This, in turn, enabled circuit designers and other end users of simulation software to share and use the same device models across different environments. At the end, the widespread adoption of Verilog-A has worked for the benefit of developers and end users of simulation software. However, the same cannot be said about the developers of the device models themselves. Although Verilog-A offers advantages to model developers for the distribution of new device model code, it still has no bearing on the actual model development cycle. This is because the use of Verilog-A requires an external simulator in which the developer imports and runs the model code. Employing such a method during the device model development phase is not practical because commercial simulators

- will run the model code only in a circuit and not in a stand-alone way,
- do not provide access to their numerical code, and
- do not offer programming tools such as a debugger.

For these reasons, many researchers in device modeling choose to develop their models in a programming environment like MATLAB®. The Berkeley Model and Algorithm Prototyping Platform (MAPP), developed within the framework of the NEEDS initiative, picks up on this predilection and offers a model development platform in MATLAB®, complete with a circuit simulator as well as access to various simulation algorithms and executable model code [5]. MAPP is accompanied by the Berkeley VAPP which is a separate software tool to import Verilog-A models into MAPP. Both MAPP and VAPP are freely available under GPL[3]. We offer a brief introduction into these tools in Section 2.2.

2.2 MAPP — The Berkeley Model and Algorithm Prototyping Platform

MAPP is a software tool entirely written in MATLAB® that is specifically targeted for the development of device models and simulation algorithms [5]. MAPP uses a simple yet powerful

[3]MAPP and VAPP both can be downloaded from Github at https://github.com/jaijeet.

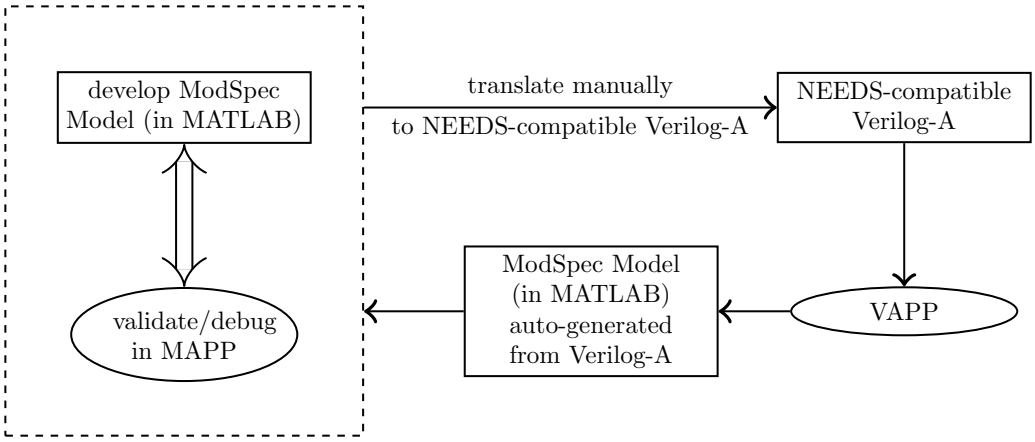

Fig. 2: Model development flow using MAPP and VAPP.

device model format called ModSpec [6]. Using ModSpec, instances of device models can be created (as MATLAB® data structures), added to circuits and simulated. It is also possible to use the model instance alone to perform simple analyses such as computing characteristic curves. Figure 2 depicts the steps in a typical model development flow using MAPP and VAPP. The development of a new device model starts at the upper left corner of the diagram and follows the steps indicated by the arrows.

Listing 2: Nonlinear resistor model with series capacitor implemented in ModSpec.

```
function MOD = nlresistor()
    MOD = ee_model();
    MOD = add_to_ee_model (MOD, 'modelname', 'nlresistor');
    MOD = add_to_ee_model (MOD, 'terminals', {'p', 'n'});
    MOD = add_to_ee_model (MOD, 'explicit_outs', {'ipn'});
    MOD = add_to_ee_model (MOD, 'internal_unks', {'vin', 'iin'});
    MOD = add_to_ee_model (MOD, 'parms', {'parm_C', 1e-12,...
                                'parm_L', 1e-6,...
                                'parm_R0', 1000});
    MOD = add_to_ee_model (MOD, 'fqei_all', @fqei_all);
    MOD = finish_ee_model(MOD);
end

function [fe, qe, fi, qi] = fqei_all(S)
    vbr_ind = S.vin;
    ibr_ind = S.iin;
    vpi = S.vpn - S.vin;
    vbr_res = vpi;
    fe(1,1) = nlfunc(vbr_res, 1/S.parm_R0);
    fi(2,1) = -nlfunc(vbr_res, 1/S.parm_R0);
    qe(1,1) = S.parm_C*vbr_res;
    qi(2,1) = (-S.parm_C*vbr_res);
    qi(1,1) = S.parm_L*ibr_ind;
    fi(1,1) = (-vbr_ind);
```

```
25      fi(2,1) = fi(2,1) + ibr_ind;
26  end
27
28  function i_out = nlfunc(v_in, G)
29      i_out = G*v_in^3;
30  end
```

The first step in Figure 2 is to implement the device equations in MATLAB® and validate them via benchmarks and tests using the exploration and simulation infrastructure provided by MAPP. For instance, the nonlinear resistor model given in Listing 1 can be implemented in ModSpec as in Listing 2. The model properties are defined in the function `nlresistor` whose output is a model object. This object has fields such as `modelname`, `terminals` etc. that are defined in lines 3 through 7 in Listing 2. Lines 16–29 implement the model behavior and the auxiliary function, `nlfunc`, at the end defines the nonlinear behavior of the resistor just like in the Verilog-A model. As opposed to its Verilog-A counterpart, this ModSpec model can immediately be instantiated with

```
MOD = nlresistor();
```

The model instance can be called directly by providing values for the input variables (`vpn`, `vin` and `iin`) in order to examine its outputs, e.g.,:

```
vpn = 2;
vpi = 1;
iin = 1e-5;
out = MOD.fi(vpn,[vpi;iin],{},{},MOD);
```

or it can be included in a circuit object by first creating the circuit

```
ckt.cktname = 'nlres_test_circuit';
ckt.nodenames = {'p'};
ckt.groundnodename = 'gnd';
```

and then adding the model object to this circuit

```
ckt = add_element(ckt, nlres, 'R', {'p', 'gnd'}, {}, {});
```

Listing 3: Transient simulation of the nonlinear resistor model in MAPP.

```
1  ckt.cktname = 'nlres_test_circuit';
2  ckt.nodenames = {'p'};
3  ckt.groundnodename = 'gnd';
4
5  vM = vsrcModSpec();
6  nlres = nlresistor();
7
8  tranfuncargs.f0 = 1e6;
9  tranfuncargs.A = 3;
10 tranfunc = @(t, args) (args.A*sin(2*pi*args.f0*t));
11
12 ckt = add_element(ckt, vM, 'vsrc1', {'p', 'gnd'}, {},...
```

```
13                                      {{'tran', tranfunc, tranfuncargs}});
14 ckt = add_element(ckt, nlres, 'R', {'p', 'gnd'}, {}, {});
15 ckt = add_output(ckt, 'p'); % node voltage of 'p'
16 ckt = add_output(ckt, 'i(vsrc1)', 100);
17 DAE = MNA_EqnEngine(ckt);
18
19 % DC analysis
20 dcop = dot_op(DAE);
21 feval(dcop.print, dcop);
22
23 % run transient simulation
24 xinit = feval(dcop.getsolution, dcop);
25 T0 = 1/tranfuncargs.f0; tstart = 0; tstep = T0/100; tstop = 2*T0;
26 TRobj = dot_transient(DAE, xinit, tstart, tstep, tstop);
27
28 % plot DAE outputs (defined using add_output inside vsrcRCL_ckt.m)
29 feval(TRobj.plot, TRobj);
```

We can then add outputs to the circuit, create an equation system out of it and perform, for example, a transient simulation. The entire code for this flow is provided in Listing 3. The important parts of the script occur in the following lines:

1–3: create the circuit object,

5–6: create a voltage source and the nonlinear resistor,

8–16: define circuit element properties and the circuit topology,

17: create differential algebraic equation (DAE) object using the circuit,

19–16: run initial DC and transient simulations,

29: display the result.

Device models should be tested with a multitude of simulation settings. They should be subjected to various types of simulations/analyses with various test inputs. After making sure that our model performs well in different types of tests, we would continue with the steps in Figure 2. To deploy our model, we would create its Verilog-A version. In order to make sure that our Verilog-A code produces the same output as the original model, we can translate the Verilog-A code back into ModSpec using VAPP. VAPP is still under heavy development but it already can produce simulation ready ModSpec models via its simple user interface. Model translation using VAPP can be as simple as making the following function call

```
va2modspec('nlresistor.va');
```

VAPP can also be used for importing existing Verilog-A models into MAPP for testing and development purposes or for building prototypes of small experimental circuits and simulating them.

As mentioned above, both MAPP and VAPP are freely available under GPL. Users are encouraged to get involved with the development by filing bug reports and contributing to the code base. We recommend that readers download and install MAPP and VAPP in order to be able to run the provided pieces of test code while going through this document.

3 A Verilog-A Model with Inconsistent Behaviors in Different Simulators

In this section we go through a case study using a Verilog-A device model that produces different results in two different simulators, Spectre and HSPICE. This case study shows that it is relatively easy to create an ill-posed model in Verilog-A. The example model we use is a simple two terminal device which does not have any complicated numerical components. We simply combine two basic features of Verilog-A to create a model which can be interpreted in more than one way. This causes the model output to be *not uniquely defined* and therefore the model to be ill-posed.

Listing 4: An ill-posed two terminal Verilog-A model.

```
1  `include "disciplines.vams"
2  module implicit_model (p, n);
3      inout p, n;
4      electrical p, n;
5      analog begin
6          I(p,n) <+ 0.5*I(p,n)*I(p,n);
7          I(p,n) <+ 0.5;
8      end
9  endmodule
```

When the DC operating point of the model in Listing 4 is computed with Spectre, the answer is `I(p,n) = 1A`. However, when the same analysis is performed with HSPICE, we obtain a different answer: `I(p,n) = 0.5A`. In the remainder of this section we explain why this is the case.

The main feature of the Verilog-A modeling language is the *contribution operator* (<+). In Verilog-A, the physical structure of a model is given by defining *nodes* and *branches* and the contribution operator enables us to assign (contribute) a voltage or current to a branch. For example, a resistor is defined as follows.

Listing 5: Resistor model in Verilog-A.

```
1  `include "disciplines.vams"
2  module implicit_model (p, n);
3      inout p, n;
4      electrical p, n;
5      analog begin
6          I(p,n) <+ V(p,n)/1e3;
7      end
8  endmodule
```

In Listing 5 (line 6) a current value is computed (`V(p,n)/1e3`) and then this value is added (contributed) using the contribution operator to the current between the `p` and `n` nodes. This is a very intuitive way of defining models. However, a programming language has to be logically precise and in this case, intuition might not always be helpful.

A prominent example of a language feature being extended beyond its immediate meaning is the notion of an *implicit contribution* in Verilog-A. Consider the model given in Listing 6.

Listing 6: Implicit diode model from the Verilog-AMS Language Reference Manual [7].

```
1  module implicit_diode (p, n);
2      inout p, n;
3      electrical p, n;
4      parameter real IS = 1e-12 from (0:inf);
5      parameter real R = 1 from (0:inf);
6      analog begin
7          I(p,n) <+ IS*(limexp((V(p,n) - R*I(p,n))/$vt) - 1);
8      end
9  endmodule
```

This model describes a simple diode and it was directly taken from the Verilog-AMS Language Reference Manual [7, Section 5.6.6]. The expression in line 7 defines the current `I(p,n)` by placing it on the left-hand side (LHS) of the contribution operator. However, the same current value (`I(p,n)`) appears on the right-hand side (RHS) of the contribution as well. What does that mean? Obviously, we cannot compute the RHS of the expression first and add it to `I(p,n)` because the RHS depends on the value of `I(p,n)`. The intuitive notion of a contribution does not apply here.

Statements like the one in Listing 6, line 7 are called *implicit contributions*. This means that the simulator interpreting the model has to bring both sides of the contribution statement together and solve it by setting the whole expression equal to zero. For the model in Listing 6:

```
I(p,n) - IS*(limexp((V(p,n) - R*I(p,n))/$vt) - 1) = 0
```

The Verilog-A language allows this usage simply because it is syntactically possible. However, the behavior of models with implicit contributions vary across simulators. In what follows, we will examine the behavior of a simple model with an implicit contribution in HSPICE and Spectre[4].

The model we will experiment with is similar to the one given in Listing 6 but, for simplicity, we replace the diode equation with a simpler equation. This new model is given in Listing 7

Listing 7: Simplified implicit model.

```
1  `include "disciplines.vams"
2  module implicit_poly (p, n);
3      inout p, n;
4      electrical p, n;
5      analog begin
6          I(p,n) <+ 0.5*I(p,n)*I(p,n)+ 0.5;
7      end
8  endmodule
```

[4]HSPICE and Spectre version used are as follows. HSPICE: Version H-2013.03 64-BIT. Spectre: Version 7.2.0 64bit.

The contribution in line 6 in Listing 7 forces the model to compute an `I(p,n)` value that satisfies the polynomial equation

$$x^2 - 2x + 1 = 0. \tag{1}$$

Hence, we expect that if we simply connect this device to a voltage source and compute the operating point (DC analysis), we should obtain

```
I(p,n) = 1.
```

This should be the case independent of the value of the voltage source.

When simulated with Spectre, we obtain

```
I(p,n) = 998.047m.
```

By tightening the value of `reltol` to `1e-9` we get the expected answer

```
I(p,n) = 1A.
```

However, if the same circuit is simulated with HSPICE, we receive a different answer.

```
I(p,n) = 493.8272m
```

In fact, when a sweep is performed, HSPICE seems to generate different results for different voltage source values.

volt	current vs
0.	-518.5185m
1.00000	-386.8313m
2.00000	-401.4632m
3.00000	-399.8374m
4.00000	-400.0181m
5.00000	-399.9980m
6.00000	-400.0002m
7.00000	-400.0000m
8.00000	-400.0000m
9.00000	-400.0000m
10.00000	-400.0000m

With a small modification to the example model in Listing 7, we obtain the model given in the beginning of this section. The model given in Listing 4 is reproduced below in Listing 8 for convenience.

Listing 8: Modified simplified implicit model.

```
1  `include "disciplines.vams"
2  module implicit_model (p, n);
3      inout p, n;
4      electrical p, n;
5      analog begin
6          I(p,n) <+ 0.5*I(p,n)*I(p,n);
```

```
7           I(p,n) <+ 0.5;
8       end
9   endmodule
```

Note that this model is similar to the simplified implicit model in Listing 7 except that the implicit contribution in line 6 does not contain the additive constant `0.5`. Instead, the model contains a separate contribution to `I(p,n)` in line 7 with the value of the missing constant in the implicit contribution. This contribution adds a constant value to `I(p,n)`. If `I(p,n)` was an explicitly defined current, we would interpret this extra contribution as "add `0.5` to the value of `I(p,n)`". However, in this case `I(p,n)` does not have an explicit value. So, how should the simulator interpret line 7 in Listing 8? One way to interpret this is to construe the second contribution as an addition to the implicit equation and solve exactly the same equation given in (1). This is indeed how Spectre interprets it. However, HSPICE seems to do something different. It ignores the implicit contribution altogether and produces

```
I(p,n) = 500.0000m.
```

This simple example shows that even syntactically correct, expressions in Verilog-A might not always be meaningful or they can be interpreted differently by different simulators. The only resolution for this problem is to restrict valid Verilog-A code to a subset of its current form. This subset must be able to support all the features a well-posed compact model might require. However, it also should exclude any expressions that might cause the model to be *ill-posed*. To do this, we will need to answer the following question: *what task does a computational model perform inside a simulator?* We answer this question with a step-by-step example in Section 4.

4 The Role of Compact Models in Simulation — a Case Study

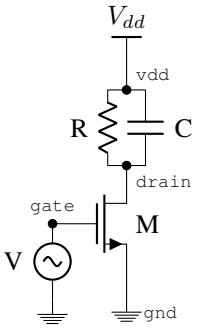

Fig. 3: CS amplifier.

In this section, the simple circuit in Figure 3 will be used to illustrate basic concepts about compact device models and their role in circuit simulation. Replicating the procedure followed by a simulator, we will start with a network of devices and arrive at a set of DAEs that describe the circuit dynamics. Our aim is to make clear how device models are used by a circuit simulator.

Figure 3 depicts a common source amplifier with four circuit elements: a voltage source (V), a MOSFET (M), a capacitor (C) and a resistor (R). A circuit is described to the simulator in terms of *nodes*, *branches* and *devices*. Listing 9 shows how this is done for the circuit in Figure 3. A text file similar to the one in Listing 9, defining the devices in the circuit and their connections, *i.e.*, the circuit topology, is traditionally called a *netlist*.

Lines 1–3 in Listing 9 set the name of the circuit and the names of its nodes. lines 5–9 create *instances* of devices. One of these devices, the voltage source, is added between the nodes `vdd` and `gnd` in lines 11–16 and its DC voltage is specified to be 1 volt (line 16). Lines 18–20 add the transistor to the circuit.

The second voltage source in the circuit is added in lines 22–28 while the final lines, 30–31, add the parallel resistor and capacitor pair.

Listing 9: A netlist for the circuit in Figure 3 (executable in MAPP).

```
1  cktnetlist.cktname = 'cs amp';
2  cktnetlist.nodenames = {'vdd', 'drain', 'gate'};
3  cktnetlist.groundnodename = 'gnd';
4
5  MOSFET = SH_MOS_ModSpec();
6  VS_VDD = vsrcModSpec();
7  VS_IN = vsrcModSpec();
8  RES = resModSpec();
9  CAP = capModSpec();
10
11 vdd = 1;
12 cktnetlist = add_element(cktnetlist, VS_VDD, ...
13                                       'Vdd',...
14                                       {'vdd', 'gnd'},...
15                                       {},...
16                                       {{'E', {'DC', vdd}}});
17
18 cktnetlist = add_element(cktnetlist, MOSFET,...
19                                       'NMOS',...
20                                       {'drain', 'gate', 'gnd'});
21
22 vinoft = @(t, args) args.A*sin(2*pi*args.f*t);
23 vinargs.A = 1; vinargs.f = 1;
24 cktnetlist = add_element(cktnetlist, VS_IN, ...
25                                       'Vin',...
26                                       {'gate', 'gnd'},...
27                                       {},...
28                                       {{'tr', vinoft, vinargs}});
29
30 cktnetlist = add_element(cktnetlist, RES, 'R', {'vdd', 'drain'}, 1e5);
31 cktnetlist = add_element(cktnetlist, CAP, 'C', {'vdd', 'drain'}, 1e-12);
```

The definition of a simple linear resistor in MAPP is given in Listing 10. When called with a voltage value as input, the function in lines 15–17 will query the resistance value of that specific resistor instance and return the current flowing through the device. When we talk about device models in simulators, what we primarily refer to is this input-output relationship function.

Listing 10: Model code for a resistor in MAPP. The function for the voltage-current characteristic of the resistor is given as `f(S)` in lines 15–17.

```
1  function MOD = resModSpec()
2      MOD = ee_model();
3
4      % define branch quantities vpn, ipn
5      MOD = add_to_ee_model(MOD, 'terminals', {'p', 'n'});
6      % ipn is available explicitly
7      MOD = add_to_ee_model(MOD, 'explicit_outs', {'ipn'});
8      % define device parameters
```

```
9       MOD = add_to_ee_model(MOD, 'parms', {'R', 1e3});
10      % define the input-output function (given below)
11      MOD = add_to_ee_model(MOD, 'f', @f);
12      MOD = finish_ee_model(MOD);
13  end
14
15  function ipn = f(S) % vpn and R are defined through S
16      ipn = S.vpn/S.R;
17  end
```

Device code can be considerably more complex (thousands of lines) for devices like transistors. In general, a device has also more than one output. For instance, an ideal capacitor might look like the code given in Listing 11. This function tells the simulator that when there is a voltage `vpn` across the terminals of the capacitor, a charge of magnitude `q` accumulates at each of those terminals while there is no direct current flow (`f = 0`) through them (the charge component can, of course, lead to its own current).

Listing 11: Model code for a capacitor in MAPP. Note the definition of the `fq` function in line 10 instead of only `f`.

```
1   function MOD = capModSpec()
2     MOD = ee_model();
3     MOD = add_to_ee_model(MOD, 'terminals', {'p', 'n'});
4     MOD = add_to_ee_model(MOD, 'explicit_outs', {'ipn'});
5     MOD = add_to_ee_model(MOD, 'parms', {'C', 1e-12});
6     MOD = add_to_ee_model(MOD, 'q', @fq);
7     MOD = finish_ee_model(MOD);
8   end
9
10  function [f, q] = fq(S)
11    f = 0;
12    q = S.C*S.vpn;
13  end
```

For devices with more than two terminals, the output variables of the device function will be vector valued. This is illustrated in Listing 12 for the Shichman-Hodges transistor model [8].

Listing 12: Shichman-Hodges transistor model. Note that the output is a 2×1 vector.

```
1   function MOD = SH_MOS_ModSpec()
2       MOD = ee_model();
3       MOD = add_to_ee_model(MOD, 'terminals', {'d', 'g', 's'});
4       MOD = add_to_ee_model(MOD, 'explicit_outs', {'igs', 'ids'});
5       MOD = add_to_ee_model(MOD, 'parms', {'Beta', 1e-3, 'vth', 0.5});
6       MOD = add_to_ee_model(MOD, 'f', @f);
7       MOD = finish_ee_model(MOD);
8   end
9
10  function iout = f(S)
11      igs = 0;
12      if S.vgs < S.vth
13          ids = 0; % cutoff region
```

```
14        else
15            if S.vds - S.vgs > -S.vth
16                % active
17                ids = 0.5 * S.Beta * (S.vgs - S.vth)^2 ;
18            else
19                % triode
20                ids = S.Beta * S.vds * (S.vgs - S.vth - 0.5*S.vds);
21            end
22        end
23        iout = [igs; ids];
24    end
```

Using the information about the circuit topology, the device equations are put together to obtain a larger set of equations describing the whole circuit. There is more than one way of generating circuit equations. However, all of them rely on the same principles: combine *constitutive equations* with *conservation equations*. In our case, the constitutive equations will be obtained by invoking device functions and the conservation equations will be built via Kirchhoff's Current Law (KCL) and Kirchhoff's Voltage Law (KVL).

The network in Figure 3 has three non-ground nodes (`vdd`, `drain` and `gate`). Two of these nodes, `vdd` and `gate`, have fixed voltages because they have voltage sources attached to them. In the end, we are left with only one node that has an unknown voltage: the drain of the transistor. The systematic form of this analysis, where the number and the type of unknowns are determined, is called the *nodal analysis*.

Since there is only one unknown in the circuit (the drain voltage, $v_{drain}(t)$), the size of the equation system will be one. In order to create this single equation, the simulator will go through the devices in the circuit. It will calculate the outputs of their device functions and combine them using KCL, with appropriate signs as follows:

$$-\underbrace{\frac{(v_{dd}-v_{drain}(t))}{R}}_{\texttt{RES.f}(v_{dd}-v_{drain}(t))}-\frac{\mathrm{d}}{\mathrm{d}t}\underbrace{[C(v_{dd}-v_{drain}(t))]}_{\texttt{CAP.q}(v_{dd}-v_{drain}(t))}+\underbrace{I_{ds}(v_{drain}(t),v_{gate})}_{\texttt{MOSFET.}f(v_{drain}(t),v_{gate})}=0 \tag{2}$$

In equation (2), the term under the time differentiation operator is a function of the independent variable, $v_{drain}(t)$. An alternative way of writing this equation is to represent the charges in the circuit as independent variables which are defined by algebraic equations.

$$\begin{aligned}\frac{\mathrm{d}Q(t)}{\mathrm{d}t} &= -\frac{(v_{dd}-v_{drain}(t))}{R}+I_{ds}(v_{drain}(t),v_{gate})\\ Q(t) &= v_{dd}-v_{drain}(t)\end{aligned} \tag{3}$$

Notice how the set of equations in (3) contains one differential and one algebraic equation. This is why circuit equations are generally called Differential Algebraic Equations (DAEs).

The circuit in Figure 3 can be simulated in MAPP by first running the script given in Listing 9 to construct the circuit, and then calling the code in Listing 13 to run a transient simulation. The result of this simulation is shown in Figure 4.

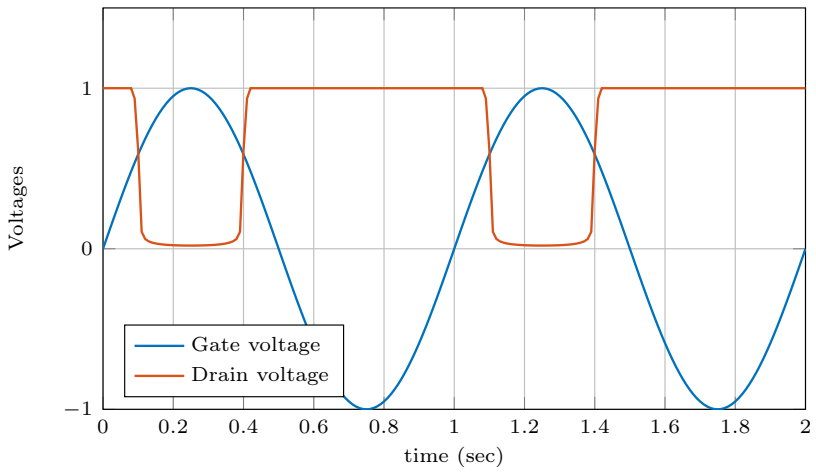

Fig. 4: The result of the transient simulation in Listing 13.

Listing 13: Transient simulation and plotting code for the netlist in Listing 9 (executable in MAPP).

```
1   DAE = MNA_EqnEngine(cktnetlist);
2
3   xinit = zeros(feval(DAE.nunks, DAE), 1);
4   xinit([1,2]) = vdd;
5   tstart = 0; tstep = 1e-2; tstop = 2;
6   LMSobj = transient(DAE, xinit, tstart, tstep, tstop);
7
8   % plot selected state outputs
9   souts = StateOutputs(DAE);
10  souts = souts.DeleteAll(souts);
11  souts = souts.Add({'e_gate', 'e_drain'}, souts);
12  feval(LMSobj.plot, LMSobj, souts);
```

In going through the case study above, our aim was to clarify the role of compact models in simulations. In summary, the simulator evaluates the device function for each compact model and combines their outputs using KCL and KVL in order to construct a set of DAEs for the whole circuit. This DAE system is then either numerically integrated as in the case of transient simulation[5] or used in other numerical algorithms such as shooting and harmonic balance. Therefore, for a successful simulation, the simulator needs to be able to construct a DAE system that is both well-defined and sufficiently well-behaved for the subsequent numerical analysis. Since the DAE system is composed of its smaller constituents, *i.e.*, the device models, this requirement can only be satisfied if every device model in the circuit is well-posed.

[5]This procedure is further described in Appendix A.

5 Well-Posedness

The mathematical notion of well-posedness applies to problems posed in form of differential equations and partial differential equations [9]. A mathematical problem is said to be well-posed if

- a solution to the problem exists,
- the solution to the problem is unique,
- the solution varies smoothly with respect to the parameters in the problem.

We borrow this notion and associate it with device models. *We call device models well-posed if they produce well-posed circuit DAEs.* Well-posedness imposes restrictions on device models from both a numerical and a programming standpoint. In this section, we discuss the key properties of a well-posed model.

We start by showing how a single device model itself can be expressed in a DAE formulation. As we have seen in equation (2), the simulator constructs the circuit DAEs by receiving the differential and algebraic parts of model equations from each device in the circuit. The most general form of model equations can be written as follows [5].

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}t}\vec{q}_e(\vec{x}(t),\vec{y}(t),\vec{u}(t)) + \vec{f}_e(\vec{x}(t),\vec{y}(t),\vec{u}(t)) &= \vec{z}(t) \\ \frac{\mathrm{d}}{\mathrm{d}t}\vec{q}_i(\vec{x}(t),\vec{y}(t),\vec{u}(t)) + \vec{f}_i(\vec{x}(t),\vec{y}(t),\vec{u}(t)) &= \vec{0} \end{aligned} \tag{4}$$

Where $\vec{x}(t)$ contains the terminal inputs and $\vec{z}(t)$ the terminal outputs of the device. The internal unknowns of the device are represented by $\vec{y}(t)$, and $\vec{u}(t)$ contains the time varying inputs to the device such as independent voltage and current sources. The four functions $\vec{q}_e$, $\vec{f}_e$, $\vec{q}_i$, $\vec{f}_i$ characterize the dynamical behavior of the device. The functions $\vec{q}_e$ and $\vec{q}_i$ represent the differential part of the device equations and $\vec{f}_e$ and $\vec{f}_i$ the algebraic ones.[6]. ModSpec uses this structure to define device models. We have been explicitly prescribing the $\vec{f}$ and $\vec{q}$ functions of the ModSpec models we have seen so far (*e.g.*, lines 19-25 in Listing 2). The DAE system in equation (4) has to be clearly and uniquely defined for every device model and this condition lies in the heart of creating well-posed models.

The restrictions for device models to be well-posed stem from multiple sources. The most important of these is the fact that simulators work in an iterative manner. At each step, the simulator checks how far away it is from satisfying the system equations by evaluating a residual function[7]. Then it computes its next guess by taking a step in the direction of decreasing error. This process is known as the *Newton-Raphson (NR) algorithm* and it requires the residual function to be well-defined for a wide range of input values as well as to be continuous and smooth. Because the residual function is computed hierarchically using individual device models, these conditions apply to device models as well. Well-posedness requirements that stem from this numerical procedure are as follows.

[6] For a detailed explanation of this equation system please see [5] and [6].

[7] Please see Appendix A for a brief explanation of how this is done.

1. **Finite and unique outputs**: The device functions ($\vec{q}_e$, $\vec{f}_e$, $\vec{q}_i$, $\vec{f}_i$) must have mathematically valid outputs for any mathematically valid input. In particular, this output should be unique and finite. Functions like $\frac{1}{a-x}$ become unbounded for $x = a$. Other functions are not defined for certain values of x. For example when $\sqrt{x}$ or $\log(x)$ is used, one has to make sure that their arguments are positive.

2. **Continuous and smooth outputs:** When finding the zero of a function, the NR algorithm uses the derivative of that function in order to take a step in the direction of decreasing function value. For this reason, the $\vec{q}_e$, $\vec{f}_e$, $\vec{q}_i$, $\vec{f}_i$ functions of a device must be continuous, *i.e.*, they must not contain any sudden jumps. They must also be smooth at least up to first order, *i.e.*, their first order derivatives with respect to the terminal inputs, $\vec{x}$, and the internal unknowns, $\vec{y}$, must exists. Higher order derivatives are highly desirable and, in the ideal case, the device functions should be in $\mathcal{C}^\infty$, *i.e.*, all orders of derivatives should exist [10].

3. **Wide input range:** Device model functions must produce real, finite outputs for a wide range of input and internal unknown values. The NR algorithm might probe devices with input values that are not physically possible under normal device operation. For example, a real transistor would burn if we applied a $1000\,\mathrm{V}$ potential across its drain and source terminals. However, the NR algorithm might very well call the device functions with a $1000\,\mathrm{V}$ input value while it is searching for a solution. Therefore, device models must be able to supply real and finite output values for a wide range of terminal voltages/currents and internal unknown values.

Apart from "low-level" numerical requirements, there are also "high-level" requirements for a model to be well-posed. These requirements are related to the model specification format in which the model is described. Today, most electrical device models are described using the Verilog-A behavioral modeling language. Although the Verilog-A language offers an intuitive model description framework with features such as its node/branch structure, time differentiation/integration operators, functionality for collapsing nodes etc., many of these features allow statements that are problematic from a well-posedness standpoint. Such problematic statements, when translated to a lower level, executable model description such as ModSpec, create either ambiguous or nonsensical models.

The most crucial high-level requirement for a model is that it has to *work in all analysis types*. Different analyses use the device functions in different ways. For example, transient simulation computes circuit variables at consequent time points, one at a time. AC analysis, on the other hand, linearizes the circuit around a DC operating point, performs computations using phasors and does not directly involve time. Therefore, it is erroneous to assume that when the device functions are called, it will always be at a well-defined time point.

The simple way of preventing high-level ill-posedness issues is to always formulate the device equations in form of equation (4). This, however, does not mean that it is sufficient to simply return some $\vec{f}$ and $\vec{q}$ functions for a device to be well-posed. In addition to formulating device equations in DAEs, one must adhere to the following rules.

1. The size of the DAE must stay constant during the entire simulation/analysis. Any statement that tries to change the size of the model equations during an analysis causes the model to

be ill-posed. A bias dependent node collapse is a typical example of this type of statement.

2. All device dynamics should be expressed using the DAE formulation. This means that no explicit time integration/differentiation may be performed inside the device code itself.
3. The device must communicate with the simulator only through device function calls. The device should be agnostic to the past, present or future state of the simulator. This means events such as crossing a voltage value or starting a certain type of analysis must not trigger any special event control routines in the device code.
4. Device code must not make any reference to absolute simulation time. It has to be able to run in analyses, such as AC and DC, where time is not directly involved.
5. The device DAEs must have a deterministic core that consistently returns the same outputs for the same inputs. All randomness information in the device (such as noise) must be implemented through special functions that tell the simulator what kind of randomness is to be added to the device equations, *e.g.*, white/flicker noise. There must not be any explicit pseudorandom number generation within the device code.

The rules in the above list are general guidelines that will ensure the correct operation of device models in every simulation/analysis type. The rather abstract nature of these rules will be more concretely explained in Section 6 where we discuss how to avoid ill-posedness issues in Verilog-A models.

6 NEEDS Compatible Device Modeling Using Verilog-A

The chief *raison d'etre* for device models is that they should work properly in simulators; a large portion of their merit stems from how well they achieve this goal. In this section, we discuss best practices to be observed in implementing compact device models in Verilog-A. We propose best practices for developing device models that are well-posed and minimize the risk of running into problems during numerical simulations and analyses. The guidelines and rules laid out here are mandated by the NEEDS initiative; accordingly, device models that follow these guidelines will be called "NEEDS compatible".

In Sec. 6.1, we list recommended practices for developing device models in Verilog-A and rules for producing NEEDS compatible models. Section 6.2 provides an overview of the most important tests device models should be subjected to before they are considered robust enough for a public release. We also offer a five-step guide on general model development methodology using Verilog-A in Section 6.3.

6.1 Verilog-A: Best Coding Practices

In this section, we will provide a list of dos and don'ts for coding *simulation-ready* device models in Verilog-A. As mentioned earlier, models which follow the recommendations in this list will be deemed "NEEDS compatible". The NEEDS initiative stipulates that device models published on

the NEEDS website must be NEEDS compatible. An overview of the recommendations and rules for NEEDS compatibility is given in the list below; the rest of this section expands on these.

1. DO NOT use a global ground node.
2. DO use branches.
3. DO declare and initialize all variables and DO NOT use memory states.
4. DO NOT use event control statements.
5. DO NOT use analysis dependent functions.
6. DO use `ddt`, but only in allowed ways.
7. DO NOT use `idt`.
8. DO NOT use time-varying functions.
9. DO NOT use random number generators.
10. DO take great care when using implicit equations.
11. DO NOT allow any nodes in your model without having at least one branch with a well-defined contribution attached to it.
12. DO NOT use bias-dependent switch branch and node collapse conditions.
13. DO use parameter ranges.

Table 1: List of "do's and dont's" for producing NEEDS compatible Verilog-A models.

1. DO NOT use a global ground node.

Example 1 shows part of a model which has an access function with only one node as argument (`V(p)` in line 5). This means that the model relies on a *global ground* (which represents a zero volt reference). This usage is not NEEDS compatible. To see why, consider the scenario where the potentials at every node of the device is increased by the same amount. Since in this case the potential differences across the device remain the same, we would expect no change in the outputs of the device. However this principle does not hold for the model in Example 1, because `V(p)` and hence `I(br_res)` would change[8].

[8] If the device really needs a connection to the global ground, it should add an extra terminal which should be connected (at the circuit netlist level) to the global ground.

EXAMPLE 1

NOT NEEDS compatible

```
1  module exampleModel(p, n);
2    electrical p, n;
3    branch (p,n) br_res;
4    ...
5    I(br_res) <+ G*V(p);
6    ...
7  endmodule
```

NEEDS compatible

```
8  module exampleModel(p, n);
9    electrical p, n;
10   branch (p,n) br_res;
11   ...
12   I(br_res) <+ G*V(br_res);
13   ...
14 endmodule
```

One may still use probes/sources in Verilog-A with a single node argument while being physically correct, *e.g.*,

```
1    I(br_res) <+ G*(V(p) - V(n));
```

Even so, using probes/sources with single node arguments is not recommended, because it can easily lead to mistakes in more complicated expressions. Instead of using single node arguments in probes/sources, NEEDS recommends using branches as shown in Example 1, line 12 and discussed below in Rule 2.

2. DO use branches.

Verilog-A supports access functions (`V()`, `I()`, `Pwr()` etc.) in the following equivalent forms:

EXAMPLE 2

NOT NEEDS compatible

```
1 V(node1) - V(node2)
```

Accepted by NEEDS but not recommended

```
2 V(node1, node2)
```

NEEDS compatible

```
3 branch (node1, node2) br_label;
4 V(br_label)
```

NEEDS compatibility requires that all probes and sources use branches. With this convention, the risk of running into problems such as the one discussed above in rule 1 will be eliminated.

3. DO declare and initialize all variables and DO NOT use memory states.

EXAMPLE 3

NOT NEEDS compatible

```
1  //The calculation of charges:
2
3  aa=1+gamma/(2*sqrt((phib-Vbb+dvg)));
4  qi=(2/3)*(1+x+pow(x,2))/(1+x);
5  Qinvq=Qinv*qi;
6  Vxint=V(Vx)+(tipe*dir)*dvg;
7  Vyint=V(Vy)-(tipe*dir)*dvg;
8
9  ...
10
11 //NVsat charge model: for long channel drift/diffusion
       ↪ without saturated          drift velocity
12 x=(1-Fsatq);
13     den=15*pow((1+x),2);
14     qsc=(6+12*x+8*pow(x,2)+4*pow(x,3))/den;
15     qdc=(4+8*x+12*pow(x,2)+6*pow(x,3))/den;
16 //end of NVsat charge model
```

A common mistake in Verilog-A models is leaving the variables undeclared/uninitialized, *e.g.*, under the assumption that the compiler will create a new variable and initialize it

automatically to zero. This is a dangerous programming practice. Example 3 was taken from a real life model where an uninitialized variable was the source of a bug. Notice that the variable `x` is used (on line 4) before it is set (on line 12).

Uninitialized variables can also cause so-called "hidden states" or "memory states". For instance, a hidden state occurs if a variable is assigned a value during one evaluation of the device function and is accessed during consequent evaluations without being assigned a new value. This causes problems in analyses such as shooting [11].

Conditional statements (*e.g.*, `if/else`) that are *bias-dependent* are usually the culprit for hidden states. Example 4 shows the relevant part of a model where this problem occurs[9].

EXAMPLE 4

NOT NEEDS compatible

```
1 real i_out;
2 analog begin
3   if (V(in) < 1) i_out = 0;
4   if (V(in) > 2) i_out = 1;
5
6   I(out) <+ i_out;
7 end
```

In Example 4, the `i_out` variable is declared but has no default value. In the model body, this variable gets a value assigned to it only when the input voltage value is outside the interval $[1, 2]$. It is not prescribed what value `i_out` is to assume when the input voltage satisfies `V(in)` $\in [1, 2]$. The expectation in the model in Example 4 is that the simulator would retain the value of `i_out` from one call of the device function to the other. NEEDS compatibility requires that, for any given input to the device function, the values of all variables in the model are uniquely defined without depending on previous input values. For this reason, the usage of hidden states is not NEEDS compatible[10].

4. DO NOT use event control statements.

Event control statements in Verilog-A start with the '`@()`' delimiter and tell the simulator to perform tasks at certain points in the simulation. The code in Example 5 line 1, for instance, tells the simulator to set the value of `register_a` to `register_b` at the positive edge of a clock cycle. Similarly, the model code in lines 2–4 performs a certain task when the input voltage crosses the 1 Volt threshold. Other possible event control statements in Verilog-A are listed in [7, Section 5.10].

[9] Seen on designers-guide.org (http://www.designers-guide.org/Forum/YaBB.pl?num=1170146863). In this model, hidden states are used as a "technique" to create hysteresis-like behavior. For a proper way of implementing hysteresis in compact models see [12].

[10] Moreover, discovering hidden states in larger models is a nontrivial task. Model developers are strongly advised to keep their models free of hidden states.

EXAMPLE 5

NOT NEEDS compatible

```
1 @(posedge clk_a) register_a = register_b;
```

NOT NEEDS compatible

```
2 @(cross (V(in), +1)) begin
3   crossings = crossings + 1;
4 end
```

NEEDS compatibility forbids the use of event control statements for the simple reason that they do not make sense in every simulation and analysis performed by the simulator. For example there are no threshold crossings in DC analysis or clock edges in AC analysis. As explained in Section 4, models are used to construct DAEs. Whatever analysis the simulator performs is performed on this DAE system and the DAE system must stay the same independent of the type of analysis. This means models should not be aware of what type of analysis is performed.

5. DO NOT use analysis dependent functions.

Analysis dependent functions in Verilog-A are used to make models behave differently in different analyses such as AC, DC, transient, *etc*. The model in example 6 uses the `analysis()` function to assign different values to the `out` variable in different types of analyses [13]. If the simulator computes a DC operating point or initial conditions, `out` gets the negative sign of the input voltage.

EXAMPLE 6

NOT NEEDS compatible

```
1 if (analysis("dc", "ic"))
2   out = ! V(in) > 0.0 ;
3 else
4   @(cross (V(in),0)) out = ! out
```

NEEDS compatibility does not allow the use of analysis dependent functions for the same reason it disallows event control statements: *model code should have no knowledge of which analysis is performed by the simulator*. Models should be able to work consistently in every analysis by providing the same DAE to the simulator. Converting and modifying the DAE is the simulators task and models should be transparent with regard to different types of

simulations and analyses. This transparency is one of the primary objectives of ModSpec. For more information about this subject, we refer the interested reader to [5, 6].

6. DO use `ddt`, but only in allowed ways.

The `ddt` function is Verilog-A's way of denoting that an expression should be placed under the time derivation operator in the circuit DAEs. For instance, in Section 4, the capacitor charge in equation 2 should be denoted in Verilog-A as in Example 7.

EXAMPLE 7

NEEDS compatible

```
1 I(br_pn) <+ ddt(C*V(br_pn));
```

NEEDS compatibility has two rules regarding `ddt` usage:

1. The `ddt` function can only be used on the RHS of a contribution operator.
2. The entire charge (or flux) expression must go into the `ddt` function.

The purpose of both rules is to ensure that the system DAE can be properly constructed. Rule 1 enforces the principle that `ddt` is not used to compute the time derivative of a quantity in the model code but only indicates that the expression in the `ddt` function should be placed in the differential part of the DAE. Again, this procedure is handled by the simulator and not by the model.

EXAMPLE 8

NOT NEEDS compatible

```
1 i_disp = ddt(C*(V(br_pn));
2 I(br_pn) <+ i_disp;
```

NEEDS compatible

```
3 charge = C*V(br_pn);
4 I(br_pn) <+ ddt(charge);
```

The first case in Example 8 is not NEEDS compatible because it assigns the time derivative of a charge to a variable and then adds this variable to the branch current as a contribution. The second case (lines 3–4) shows how to do this in a NEEDS compatible manner. First the charge is computed and stored in a variable and then it is used in the `ddt` function in the contribution (line 4).

To understand the second rule above, consider a nonlinear capacitor with a quadratic voltage charge relationship ($Q = C\,V^2$). This means that the model represents the following differential equation.

$$\frac{\mathrm{d}}{\mathrm{d}t}(C\,V^2) - I = 0 \tag{5}$$

EXAMPLE 9

NOT NEEDS compatible

```
1 I(br_pn) <+ C*pow(ddt(V(br_pn)), 2);
```

NEEDS compatible

```
2 I(br_pn) <+ ddt(C*(pow(V(br_pn), 2));
```

Line 1 in Example 9 shows a wrong attempt at implementing this model. In fact, this expression makes no mathematical sense either. The literal translation of this expression into a differential equation format would be

$$C\left[\frac{\mathrm{d}V}{\mathrm{d}t}\right]^2 - I = 0 \tag{6}$$

which is not only different from equation (5) but also does not conform to the format of the circuit DAE formulation (see Section 4). The second line in Example 9 shows the correct Verilog-A implementation. In order to avoid this type of problem, NEEDS forbids the use of `ddt` *inside* other functions.

The best practice, when using a `ddt` statement, is to separate the `ddt` part of the contribution and collect all quantities that are part of the `ddt` statement together inside the argument of the `ddt` function. This is illustrated in Example 10 for an inductor model with a series resistor. The `ddt` contribution to `V(br_pn)` is placed on a separate line and the constant `L` is written inside the `ddt` function.

EXAMPLE 10

NOT NEEDS compatible

```
1 V(br_pn) <+ R*I(br_pn) + L*ddt(I(br_pn));
```

NEEDS compatible

```
2 V(br_pn) <+ R*I(br_pn);
3 V(br_pn) <+ ddt(L*I(br_pn));
```

7. DO NOT use `idt`.

Besides `ddt`, Verilog-A offers another function that can be used to specify the dynamical behavior of the model: `idt`. While `ddt` supplies the differential parts of the model equations, `idt` is meant to contribute to the *integral* parts. Therefore using `ddt` functions in a model would result in the following equation system[11].

$$\frac{\mathrm{d}}{\mathrm{d}t}\vec{q} + \vec{f} + \int \vec{g}\,\mathrm{d}t = \vec{0} \tag{7}$$

However this formulation is not in the DAE format of equation (4). NEEDS compatibility forbids the use of `idt`. We explain the reasons below.

Every well-posed Verilog-A model can be defined without using the `idt` function. Consider a simple inductor model. The main model equation can be written using `idt` but it can also be written using `ddt` as shown in Example 11. To achieve this, one only needs to exchange the potential and flow access functions (`V()` and `I()`) on either side of the contribution statement.

EXAMPLE 11

NOT NEEDS compatible

```
1 I(branch1) <+ idt(V(branch1)/L)
```

NEEDS compatible

```
2 V(branch1) <+ ddt(L*I(branch1));
```

The use of `idt` also poses a problem from an analysis standpoint. Consider DC analysis. In DC analysis, we solve for the steady state of a circuit. That means we assume that all time derivatives are zero and therefore drop the $\vec{q}$ function in the DAE system. However, if the circuit equations were of the form in (7), what would that mean for the integral part of the equation? For instance, if the $\vec{g}$ function has a constant nonzero term, the integral will be infinite and the entire system will be ill-posed. Therefore, in order to specify the dynamical model behavior, NEEDS compatibility requires models to exclusively use the `ddt` function.

8. DO NOT use time-varying functions.

[11] The distinction between $\vec{f}_e$, $\vec{f}_i$ and $\vec{q}_e$, $\vec{q}_i$ in equation (4) is dropped here for clarity.

EXAMPLE 12

```
1 analog begin
2   ...
3   V(branch1) <+ sin(2 * `M_PI * freq * $abstime);
4   ...
5 end
```

Example 12 shows a code snippet from a model that has a time-varying voltage assigned to one of its branches. This model uses the `$abstime` simulator variable in order to reference the simulation time within the model. This makes the model unsuited for DC and AC analyses because "absolute time" does not make sense in these analysis types. In DC analysis, the circuit is assumed to have reached a steady state where the currents and voltages in the circuit do not change with time anymore. Therefore the concept of "absolute time" is not required in DC analysis. In AC analysis, the circuit is linearized around its DC solution and all subsequent computations are performed using *phasors* instead of time-varying quantities.

What a given simulator might do for these analyses is unclear — *e.g.*, `$abstime` variable may be assigned a value that is arbitrarily decided by the simulator, or the simulation may fail. The proper way to specify time-varying sources for each type of analysis is at the circuit netlist level. Verilog-A does not provide a mechanism for embedding time-varying sources into models. However, ModSpec provides a simple, general and elegant solution to this problem[5]. It is in this context that NEEDS compatibility requires that model developers do not use time-varying functions in their Verilog-A models. We do plan, however, to propose Verilog-A extensions for proper and general input support within models and to support them in our open-source toolchain (VAPP and MAPP).

9. DO NOT use random number generators.

All device models must have a clearly defined deterministic part (in form of equation (4)) to be used in *all* analyses. Therefore NEEDS compatibility forbids the use of random number generator functions (such as `$dist_uniform`, `$dist_normal` *etc*.). Noise and parameter variability related analyses are exceptions to this rule, however in those cases only the form of randomness should be specified in the model (*e.g.*using `$white_noise`, `$flicker_noise` *etc*.) and all explicit random number generation should be left to the simulator. Putting random functions within the model description breaks the deterministic replicability of the model, and also its suitability for consistent use by every (deterministic) analysis in the simulator.

A good example of this situation can be found in the RRAM model presented in [14]. An excerpt of this model is given in Example 13.

EXAMPLE 13

Excerpt from the RRAM model presented in [14].

```
143 gap_ddt = - Vel0 * exp(- q * Ea / kb / T_cur) * sinh(
        ↪ gamma * a0 / tox * q * Vtb / kb / T_cur);
144 // gap time derivative - variation part
145 deltaGap = deltaGap0 * model_switch;
146 gap_random_ddt = $rdist_normal(rand_seed, 0, 1) *
        ↪ deltaGap / (1 + exp((T_crit - T_cur)/T_smth));
147 gap = idt(gap_ddt+gap_random_ddt, gap_ini);
```

The `gap` variable in line 147 in Example 13 corresponds to a physical length in the device which grows over time. Instead of modeling the dynamics of this quantity with differential equations which are numerically integrated by the simulator, the developers of the model in [14] chose to perform the integration in the device model itself[12]. Moreover, this integration involves a random number that is also determined inside the device. The numerical integration of a random process is an intricate operation and the variance of the pseudorandom numbers used in this operation are directly connected to the integration time step. This is only one of the many reasons why the model code given in Example 13 will lead to problems during simulation and will possibly generate erroneous results. NEEDS compatibility requires that models do not contain any explicit random number generation code.

10. DO take great care when using implicit equations.

Verilog-A allows implicit equations to be defined with the regular contribution format. While this is a feature that can be very useful if properly exercised, Verilog-A's syntax and constructs for implicit equations can be opaque and confusing to those unfamiliar with the underlying process of converting Verilog-A equations into DAEs. It is also possible to write Verilog-A code for implicit equations that makes no sense, as we have demonstrated in Section 3, which different simulators interpret differently and inconsistently. The first rule about implicit equations is: *try to avoid them if you can*. Therefore, it is strongly recommended that implicit equations in Verilog-A be avoided, unless the modeler has a good understanding of the mapping between Verilog-A model code and the underlying DAEs they correspond to. If implicit equations are desired, the following examples lay out how they can be implemented in a safe and unambiguous way.

We again turn to the simple implicit diode example given in Section 3, Listing 6. This series circuit, consisting of an ideal diode and a resistor, is shown in Figure 5. In fact, this model can be described by explicit equations, as shown in Example 14.

[12] For an exposition on how to do this properly, *i.e.*, using an extra internal unknown in the model, please see [12].

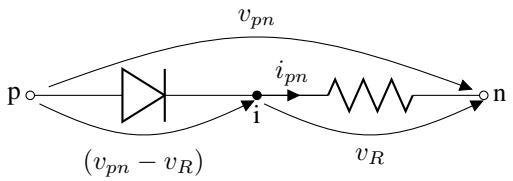

Fig. 5: Diode model with and ideal diode and a series resistor.

EXAMPLE 14

Figure 5 modeled with explicit equations.

```
1  module explicit_diode (p, n);
2    inout p, n;
3    electrical p, i, n;
4    branch (p, i) br_dio;
5    branch (i, n) br_res;
6    parameter real IS = 1e-12 from (0:inf);
7    parameter real R = 1 from (1e-3:inf);
8    analog begin
9      I(br_dio)) <+ IS*(limexp(V(br_dio)/$vt) - 1);
10     I(br_res) <+ V(br_res)/R;
11   end
12 endmodule
```

The Verilog-A model in example 14 consists of two contributions given in lines 9 and 10. Both of these are are regular (explicit) contributions. The first one in line 9 quantifies the current flow through the ideal diode, between the nodes `p` and `i` in Figure 5. The second contribution in line 10 does the same for the current through the series resistor between the `i` and `n` nodes. However, because we have

$$v_{pi} = (v_{pn} - v_R) = (v_{pn} - R\, i_{pn}), \tag{8}$$

we can also construct this model as in Example 15.

EXAMPLE 15

Figure 5 modeled with an implicit equation.

```
1  module implicit_diode (p, n);
2    inout p, n;
3    electrical p, n;
4    branch (p, n) br_pn;
5    parameter real IS = 1e-12 from (0:inf);
6    parameter real R = 1 from (1e-3:inf);
7    analog begin
8      I(br_pn) <+ IS*(limexp((V(br_pn) - R*I(br_pn))/$vt) - 1);
9    end
10 endmodule
```

There are two things to note in the model given in Example 15. First, the internal node `i` has been eliminated. Second, the current `I(p,n)` appears on both sides of the contribution in line 8, which makes it implicit. The explicit model in Example 14 and the implicit model in Example 15 are mathematically equivalent. However, implicit equations have to be handled specially by the simulator. As noted above, it is easy to come up with constructs that make no sense. If implicit equations are used, it is important to abide by the following two NEEDS compatibility rules:

(a) **DO NOT use a second contribution to an implicitly defined quantity**.

EXAMPLE 16

NOT NEEDS compatible

```
1 I(br_pn) <+ IS*(limexp((V(br_pn) - R*I(br_pn))/$vt) - 1);
2 I(br_pn) <+ R2*I(br_pn);
```

The expression in Example 16 is undefined because it makes no sense from the perspective of DAE construction. Once `I(br_pn)` is defined implicitly, it is not clear to what quantity the second contribution should be added. The first possible interpretation would be to add the second contribution to the first one and solve the equation

$$i_{pn} - I_s\,[\text{limexp}\,((v_{pn} - R\,i_{pn})/v_t) - 1] - R_2\,i_{pn} = 0. \tag{9}$$

This is how Spectre handles it as we have seen in Section 3. The second possibility is to view `I(br_pn)` as completely determined by the implicit contribution and ignore the explicit contribution in line 2 in Example 16. This would mean the simulator will solve the original implicit equation

$$i_{pn} - I_s\,[\text{limexp}\,((v_{pn} - R\,i_{pn})/v_t) - 1] = 0. \tag{10}$$

The third possible interpretation is to ignore the previous implicit contribution if there is an explicit contribution to the same branch. This is indeed how HSPICE interprets it. There is no formal reason to favor either one of these three possibilities since adding a second contribution to an implicitly defined quantity is inherently nonsensical. Therefore, NEEDS compatibility forbids a second contribution to a quantity once it is defined implicitly.

(b) **If you are using a "dummy probe", place it as the first expression on the right hand side of the implicit contribution.**

EXAMPLE 17

NOT NEEDS compatible

```
1 V(branch1) <+ f(I(branch1) + I(branch2)) + V(branch1);
```

NEEDS compatible

```
2 V(branch1) <+ V(branch1) + f(I(branch1) + I(branch2));
```

Since Verilog-A contributions cannot have a zero on their LHS, `V(branch1)` in Example 17 acts as a "dummy probe" that facilitates the definition of the implicit equation for `I(branch1)` and `I(branch2)`. Both expressions in Example 17 are equivalent to the following equation

$$f(i_{branch1} + i_{branch2}) = 0.$$

However, for clarity, NEEDS compatibility requires that the dummy probe appear as the first quantity on the RHS of an implicit contribution.

NEEDS recommends strongly that device modelers use implicit equations only if they know exactly what they are doing. Model developers are encouraged to confirm their equations are correct by examining the ModSpec code generated by VAPP on their model.

11. DO NOT allow any nodes in your model without having at least one branch with a well defined contribution attached to it.

Some models can contain nodes that naturally do not have any contributions to any of the branches involving that node. This is usually the case for controlled sources that use one of the nodes in the model as a probe to obtain information from the outside world, *i.e.*, the rest of the circuit.

If there are no currents flowing into (or out of) a node, this should explicitly be stated by writing down a zero contribution to one of its branches. Ideally, the second node of this branch will be the reference node of the model.

Consider the model in Example 18.

EXAMPLE 18

```
1 module controlled_cap(n,qex,p);
2 inout p, qex, n;
3 electrical p, qex, n;
4 branch (p, n) br_pn;
5 branch (qex, n) br_qexn
6 analog begin
7     V(br_pn) <+ pow(V(br_qexn), 2);
8   end
9 endmodule
```

This is a voltage/charge controlled capacitor model that uses the `qex` node to receive voltage/charge information from some other node in the circuit. Although this model is complete in the above form, one should nevertheless explicitly state that there is no current flowing flowing through the branch `br_qexn`. This is easily achieved by including a zero contribution to the model.

```
8   I(br_qexn) <+ 0;
```

Although the sensible choice for a compiler implementation is to handle these type of nodes as zero contribution nodes, it is always safer to state this fact explicitly. This way, the risk of running into undefined or compiler dependent behavior is minimized.

12. DO NOT use bias-dependent switch branch and node collapse conditions.

EXAMPLE 19

NOT NEEDS compatible

```
1 analog begin
2   ...
3   if (V(ps,ns) > thresh)
4     V(p,n) <+ ron*I(p,n);
5   else
6     I(p,n) <+ goff*V(p,n);
7   ...
8 end
```

Example 19 shows the use of a *switch branch* (taken from [1, Section 6.1]). Switch

branches use `if/else` statements in order to determine if a current contribution or a voltage contribution is to be made to a branch. However, the conditional expression of this `if/else` statement (`V(ps,ns) > thresh`) is *bias-dependent*, *i.e.*, it changes during the operation of the circuit. The subsequent two contribution statements will result in a different number of unknowns in the circuit. This is because one of them probes a current variable and the other one a voltage variable. This means depending on the value of `V(ps,ns)`, the number of variables in the device state vectors, *i.e.*, $\vec{x}(t)$ and $\vec{y}(t)$ in equation (4), have to be changed mid-simulation. This will break analyses such as shooting and harmonic balance that depend on combining system matrices under different bias conditions. Therefore, NEEDS compatibility forbids the use of switch branches with bias-dependent conditions.

After the construction of the DAE system from the input netlist, the numerical algorithms employed by the simulator operate on the equations in this system without any regard to the underlying circuit. For this reason, changing the number of current variables, for instance, at any time during a simulation will interfere with the operation of the numerical simulation algorithms. A NEEDS compatible device model will contribute the same number of equations with the same number of unknowns to the DAE no matter what its bias point is.

EXAMPLE 20

NOT NEEDS compatible

```
1 analog begin
2   ...
3   if (V(ctrl) > thresh)
4     V(out) <+ 0;
5   ...
6 end
```

A similar problem occurs if bias-dependent `if/else` conditions are used in combination with a *node collapse* statement. This is illustrated in Example 20 (taken from [7, Section 5.6.5]) Line 4 in Example 20 indicates that the `out` branch is to be collapsed if the condition in line 3 is satisfied. However, similar to Example 19, this condition is bias-dependent and will cause similar problems when constructing circuit DAEs. Therefore NEEDS compatibility does not allow node collapse statements with bias-dependent conditions.

The node collapse feature aims to achieve increased simulation performance by changing the number of nodes in a device in the case when a branch has no current or voltage contributions to it. By removing a node from the list of nodes in the device, one changes the size of the system DAEs. Since the system size has to stay constant during analyses, NEEDS compatibility only allows node collapse statements that have *bias-independent* conditions. Example 21 (excerpt from BSIM-4 [15]) shows a NEEDS compatible node collapse statement. In this case, the condition `(RDSMOD == 0)` depends solely on a parameter value, *i.e.*, is *bias-independent*.

EXAMPLE 21

NEEDS compatible

```
1 ...
2 parameter real RDSMOD = 0 from [0:2];
3 analog begin
4   ...
5   if (RDSMOD == 0)
6     begin
7       V(source, sourcep) <+ 0;
8       V(drainp, drain)   <+ 0;
9     end
10  else
11    begin
12      I(drain, drainp)   <+ type * gdtot * vded;
13      I(source, sourcep) <+ type * gstot * vses;
14    end
15  ...
16 end
```

13. DO use parameter ranges.

EXAMPLE 22

NEEDS compatible

```
1 parameter real R=1000 from (0:inf);
2 parameter real K=10 from (-inf:inf) exclude 0;
```

Parameter ranges are not a hard requirement for well-posed models but using them is good modeling practice. Using the `from` and `exclude` keywords as shown in Example 22 enables the simulator to check if the user assigned parameter values are within allowed limits. In Example 4, a resistance parameter, `R`, is restricted to strictly positive values and another parameter, `K`, is permitted to have negative values but is required to be nonzero. This way, the assignment of nonphysical values to parameters can be prevented. NEEDS compatibility requires that the range of every parameter is declared even if it is a trivial one such as `(-inf:inf)`.

6.2 Testing and Documentation Best Practices

An important requirement for NEEDS compatibility is that device models are tested in a number of ways, outlined below, the results of the testing documented, and the test scripts provided along

with the model so that any user can verify that the model passes the tests. NEEDS offers tools for easy and efficient testing of device models: NEEDS compatibility testing should be performed using the following NEEDS tools (additional testing with the commercial simulator Spectre, while not required, is recommended).

- VAlint: to assess the NEEDS compatibility of Verilog-A models.
- VAPP: to convert Verilog-A models into MAPP's ModSpec format.
- MAPP: to evaluate the performance of device models in various simulation algorithms.

The main NEEDS model development flow was outlined in Section 2, Figure 2. The crucial part of this flow is the first two steps where the model is developed in ModSpec format and iteratively tested using MAPP. The most important tests a model should be subjected to before it can be deemed *simulation ready* are listed below.

1. Make sure that your model produces sensible values for DC analysis at zero bias. DC analysis is used in all simulations/analyses in order to find consistent initial conditions for the circuit. Therefore, it is supremely important that your model produces sensible values for DC bias conditions. Non-zero voltage/current output values at zero bias are typically indicative of a modeling problem.

   MAPP provides a script with the name `model_dc_exerciser` which takes a model as input and constructs a circuit with it by connecting all of its terminals to voltage/current sources. Using `model_dc_exerciser`, one can easily find out if a model has any problems at zero bias. For a hypothetical four terminal device, `mos_model_under_test`, with terminal names `d`, `g`, `s` and `b`, we can simply run the script given in Listing 14.

Listing 14: Example usage of MAPP's model_dc_exerciser script.

```
1  MOD = mos_model_under_test(); % create an instance of the device
2  MEO = model_dc_exerciser(MOD); % initiate model exerciser
3  MEO.Id(0,0,0,0,MEO); % check drain current at zero bias
4  MEO.Ig(0,0,0,0,MEO); % check gate current at zero bias
5  MEO.Is(0,0,0,0,MEO); % check source current at zero bias
6  MEO.Ib(0,0,0,0,MEO); % check bulk current at zero bias
```

   By running the code in Listing 14, we can find out if all of the terminal currents are in fact zero at zero bias. For instance, if the above code is executed, replacing the hypothetical model with the modified MVS model, version 1.0 from [16] , we quickly discover that this model does not converge to a DC solution at zero bias and MAPP quits with the error.

```
1  NR: Jacobian is singular, aborting.
2  QSSsolve: failed after 2 iterations
```

   This problem was discovered while testing the modified MVS model with MAPP and was fixed in a later version, v1.1.1, of the model[13].

[13] This updated version can be downloaded from https://nanohub.org/publications/95/3.

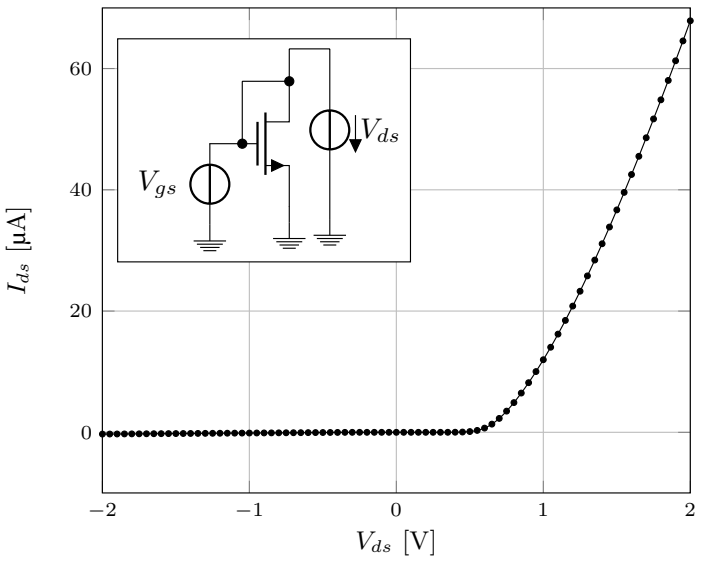

Fig. 6: DC sweep test using MAPP for the circuit shown in the inlay. The graph shows the drain-source current of the transistor plotted against the drain-source voltage.

2. Conduct tests in which you connect all possible subsets of terminals together.

   Test your model in all possible configurations, *e.g.*, diode connect a transistor model (as shown in the inlay of Figure 6) and run simulations for that configuration. For a general device with n terminals, exercise every possible combination of connecting the terminals together and run DC sweeps over both positive and negative voltage values.

Listing 15: DC sweep in MAPP for the circuit in Figure 6.

```
% define nodes
cktnetlist.cktname = 'diode connected NMOS';
cktnetlist.nodenames = {'drain'};
cktnetlist.groundnodename = 'gnd';
% add voltage source to the circuit
cktnetlist = add_element(cktnetlist, vsrcModSpec(), 'Vdd',...
                         {'drain', 'gnd'}, {});
% create instance of transistor model and add it to the circuit
MOD = mos_model_under_test();
cktnetlist = add_element(cktnetlist, MOD, 'NMOS',...
                         {'drain', 'drain', 'gnd', 'gnd'}, {});
% create DAE from the circuit netlist
DAE = MNA_EqnEngine(cktnetlist);
% run dc sweep
dcSwObj = dcsweep(DAE, [], 'Vdd:::E', 0:0.01:2);
% plot results
[Vins, sols] = dcSwObj.getSolution(dcSwObj);
ipnIdx = DAE.unkidx('Vdd:::ipn', DAE);
```

```
19  plot(Vins, -sols(ipnIdx, :));
```

Listing 15 shows a MATLAB script that uses MAPP to test the circuit given in Figure 6 for a hypothetical transistor model, `mos_model_under_test`. This piece of code first constructs the circuit (lines 2–11) and then runs a DC sweep on it where the voltage between the drain and source terminals of the transistor is increased from $-2\,\mathrm{V}$ to $2\,\mathrm{V}$. The result of running this analysis with the BSIM-3 model is shown in Figure 6.

3. If possible, observe residual values that are produced by the simulator when conducting tests and make sure that they are close to zero within machine precision. Some simulators might use insufficiently loose tolerance values (*e.g.*, `abstol`, `reltol`) in order to make simulations converge. This can create a "garbage in, garbage out" situation. To prevent this, make sure that circuit equations are satisfied with sufficient accuracy.

   This is generally not possible in commercial simulators but is very easy to establish in MAPP which is specifically designed for model development purposes. After writing down a circuit such as the one in Listing 15 and constructing the DAE system as in line 13, one would simply compute a solution at a bias point.

```
1  drainVoltage = 1;
2  DAE = DAE.set_uDC(drainVoltage, DAE); % set bias point
3  dcObj = op(DAE, []); % run dc analysis at bias
```

Afterwards, one can examine the residual values easily.

```
1  sol = dcObj.getsolution(dcObj); % get solution from dc analysis
2  res = DAE.f(sol, drainVoltage, DAE) % display the residual value
```

This operation will display the output of the KCL/KVL equations, which should vanish (within machine precision) for a valid solution of the circuit which is indeed the case here.

```
1  res =
2
3  1.0e-17 *
4
5                  0
6                  0
7  0.164657492856171
8  0.000360890656224
```

4. Make sure your model does not contain any discontinuities at least up to first order derivatives[14]. Figure 7 shows a plot of the drain current in the BSIM-3 model computed by varying the drain and gate voltages between $0\,\mathrm{V}$ and $4\,\mathrm{V}$. As the graph depicts, the drain current exhibits a discontinuity around $V_{gs} = 3.2\,\mathrm{V}$, which is an unphysical artifact. Because of this

[14] Ideally models should be C^∞ continuous [10].

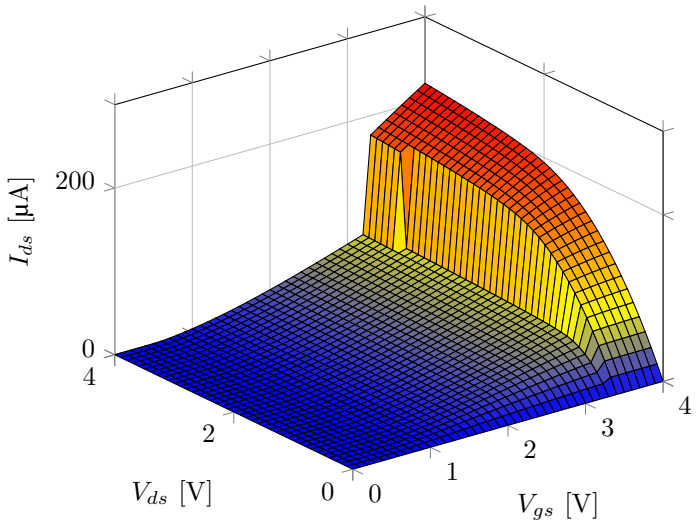

Fig. 7: The discontinuity in the drain current of the BSIM-3 model.

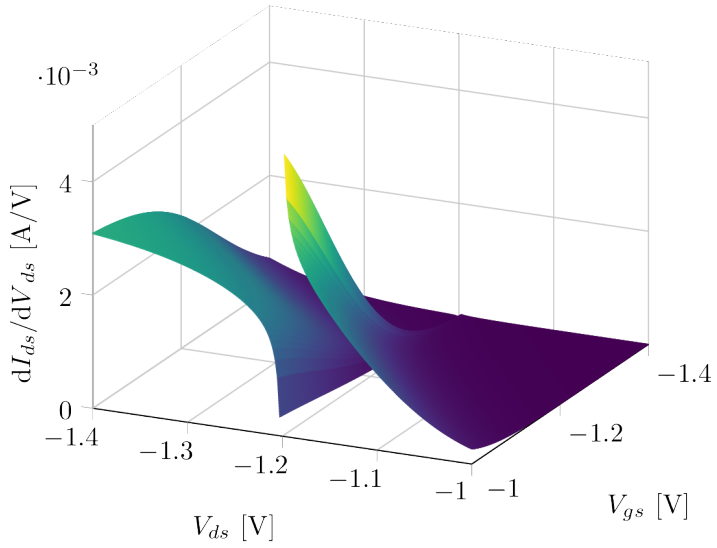

Fig. 8: The discontinuity in the drain current derivative of the MVS v1.0.1 model.

jump in the current value, simulators will have convergence problems around this value of the gate voltage. Even if the transistor is biased far from this particular value, the simulator might still probe the current value around where it is discontinuous, and fail because it will not be able compute the derivative at this point.

5. Test your models to their limits using MAPP and see where they break. Remember, simulators can try out voltage/current values that don't make sense physically but are required in numerical algorithms. Make sure that your model responds well to a reasonably broad set of input values.

   As an example we demonstrate a discontinuity in the derivate values of the MVS model, version 1.0.1 [17]. When tested with MAPP, applying negative voltages to both the drain and the gate terminals, it was discovered that the model had convergence problems at around $V_{ds} = -1.2$. We have then investigated the derivatives around this point using MAPP's model exerciser. The result is shown in Figure 8. The derivative of I_{ds} with respect to V_{ds} is clearly discontinuous. Upon investigation, the highlighted expression in the model code excerpt in Listing 16, line 143 was found to be the culprit. Because the default value of the parameter `phib` is 1.2, and because of the drain source inversion, the value of `Vbs` becomes also 1.2. This makes the argument to the `sqrt` function zero and causes a discontinuity in the derivatives.

Listing 16: Excerpt from MVS v1.0.1 Verilog-A code.

```
141   //Correct Vgsi and Vbsi
142   //Vcorr is computed using external Vbs and Vgs but internal Vdsi, Qinv and
          ↪ Qinv_corr are computed with uncorrected Vgs, Vbs and corrected
          ↪ Vgs, Vbs respectively.
143   Vtpcorr = Vt0 + gamma * (sqrt(abs(phib - Vbs)) - sqrt(phib)) - Vdsi * delta;
144   eVgpre = exp(( Vgs - Vtpcorr )/ ( aphit * 1.5 ));
145   FFpre = 1.0/ ( 1.0 + eVgpre );
```

6. Test your model in multiple simulators.

   In order to make sure that your model does not exhibit simulator dependent behavior, test your model in more than one simulator and make sure that the results are consistent. After initial testing in MAPP, NEEDS recommends Spectre and Xyce for further testing.

   Different simulators can and do use different strategies to make simulations converge. As an example, we explore how Spectre and HSPICE handle the problem in the modified MVS model we have discussed in point 1 above. A DC operating point analysis was run in both simulators with the diode connected modified MVS negative capacitance model [16]. The netlist for these analyses is given in Listing 17.

Listing 17: Netlist for the diode connected modified negative capacitance MVS model.

```
1   simulator lang=spectre
2   // global 0
3
4   parameters
5
```

```
6   ahdl_include "mvs_5t_mod.va"
7   ahdl_include "neg_cap_3t.va"
8
9   simulator lang=spice
10
11  V1 1 0 0
12  X0 1 gn qg_as_vn neg_cap_3t
13  X1 1 gn 0 0 qg_as_vn mvs_5t_mod
14
15  .op
16
17  .end
```

When the netlist in Listing 17 is run in Spectre, the simulator quits with the following error message.

```
Error found by spectre during DC analysis 'opBegin'.
ERROR (SPECTRE-16080): No DC solution found (no convergence).
```

However, if the same netlist is run in HSPICE, the operating point calculation converges and produces the output

```
**info** dc convergence successful at Newton-Raphson method
...
element  0:v1
volts      0.
current   -2.1786n
```

Notice how applying a zero voltage bias across the device produces a nonzero current through the voltage source, which is of course nonphysical. This example shows how important it is to conduct the standard tests listed in this section in every stage of device model development.

6.3 A Five Step Guide to Writing a Verilog-A Model

The five step guide given below lays out a procedure to construct Verilog-A device models starting from a given internal node topology of the model. We provide this guide with the hope that a clear, step by step procedure will help model developers create better device models. The individual steps that will be discussed in the rest of this section are as follows.

1. Determine the nodes in the model.
2. Determine the branches in the model.
3. Identify the unknowns of the model.
4. Pick half of the unknowns as "sources".
5. Write differential equations for the "sources", using "probes" and other variables.

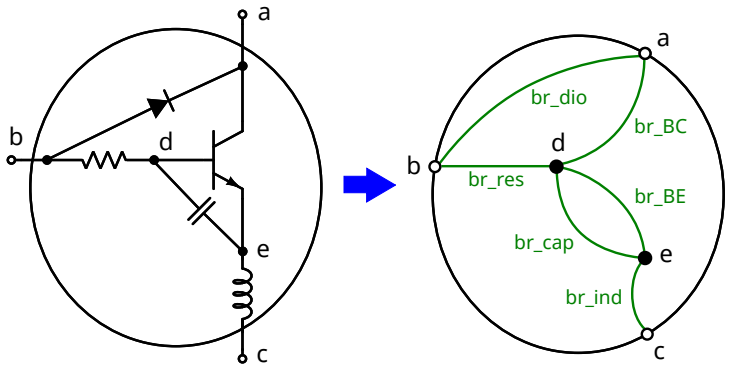

Fig. 9: An example device model and its node-branch representation.

In the Verilog-A language, device models are written with an internal circuit topology, *i.e.*, with terminals, internal nodes and branches defined just like in a subcircuit. The variables in a Verilog-A model, the "sources" and "probes", are potentials and flows specified based on this topology. Coming from this subcircuit perspective, Verilog-A is a reasonably simple language to use. For most devices, developers can write their Verilog-A models following the recommended steps listed below. We illustrate these steps on the example device shown in Fig. 9.

- *Step 1: Determine the nodes in the model.*

  Nodes in a Verilog-A model can be partitioned into two groups: *terminals* and *internal nodes*. Terminals have connections to the outer world (other devices in the circuit) while internal nodes do not. Terminals are declared using the "`inout`" keyword. This naming convention may not seem intuitive at the first glance. It comes from the name for bi-directional pins in the Verilog hardware description language. Apart from "`inout`", there are other types of terminals, such as "`in`" and "`out`". These other types are used to model signal-flow systems[15]. While they may be useful in some cases, these naming conventions can also get confusing for the novice in the field of analog compact modeling. Therefore NEEDS recommends that only "`inout`" declarations be used for terminals. In other words, "`inout`" should be synonymous with "terminal". After declaring terminals using "`inout`" statements, their *disciplines* have to be specified (most likely "`electrical`"). Available natures and disciplines are described in a standard definition file usually named "`disciplines.vams`".

  Internal nodes are often used to model parasitics, or to declare internal unknowns in the model. They are commonly specified as "`electrical`" nodes (just like terminal nodes), although other natures and disciplines are possible. For the device shown in Fig. 9, terminals and internal nodes can be declared using the following lines.

```
1  inout a, b, c;
2  electrical a, b, c, d, e;
```

[15] In signal-flow systems, terminals declared with "`in`" have infinitely large input impedances, while those labelled as "`out`" have zero output impedances. In this way, voltage signals "flow" from device to device with no loading effects.

- *Step 2: Determine the branches in the model.*

  After specifying the terminals and internal nodes of the model, the next step is to associate them with branches. Some experienced Verilog-A developers may not declare branches in their model code; branch declarations may also seem unnecessary for some common devices with a well-defined topology, such as MOSFETs and BJTs. However, NEEDS recommends that developers draw or write down the branches and declare them explicitly in their models. This will eliminate many common modeling mistakes, such as unbalanced numbers of unknowns and equations, disconnected models, non-branch-based quantities being used in the model, *etc*.

  For the device example in Fig. 9, we can build an abstraction of the device with a network of 6 branches. These branches can be declared in Verilog-A using `branch` statements:

```
1  branch (b, a) br_dio;
2  branch (b, d) br_res;
3  branch (d, e) br_cap;
4  branch (e, d) br_ind;
5  branch (d, a) br_BC;
6  branch (d, e) br_BE;
```

  Note that more than one branch can be declared to connect two nodes, *e.g.*, `br_cap` and `br_BE` both connect `d` and `e`. Moreover, branches can form loops, such as `br_res`, `br_BC` and `br_dio`.

- *Step 3: Identify the unknowns of the model.*

  With the nodes and branches declared, the next step is to identify the unknowns (free variables) in order to write down model equations. In the Verilog-A language, for each branch, there is a *potential drop across it* and a *current flow through it*. In the case of electrical devices, the potential drop is the branch voltage and the flow is the electrical current. For example, for `br_dio`, there will be two variables available: `V(br_dio)` and `I(br_dio)`. Although the electrical potential is a "node quantity", model developers should only use potential differences, *e.g.*, branch voltages, when writing their models. Because the behaviour of a model should only depend on the relative potentials, *i.e.*, the branch potential drops, instead of absolute potentials of the nodes.

  The potential and flow variables are the unknowns in the device's equation system. By unknowns, we mean that these variables will be *solved for* during simulation. Apart from the unknowns, *i.e.*, potentials and flows, models contain quantities called *parameters*. The major difference between parameters and unknowns is that parameters are not solved for during simulation. They can be construed as constants. Parameters are declared using the "`parameter`" keyword. For example,

```
parameter (* desc="resistance" unit="V/A" *) real R = 1.0e3 from (0:inf);
```

  The parameter declaration statements include the name and type of the parameter, as well as its default value, range and optionally a description.

Other than unknowns and parameters, one can also declare general variables in Verilog-A, much like one would do in C. Usually, scalar variables are declared with the type "real". Note that these variables are internal to the model. They can be thought of as intermediate quantities used in the evaluation of the model equations. Unlike unknowns, they are not solved for in simulation. And unlike parameters, their values cannot be set up in the circuit netlists. Since they cannot become unknowns in system equations, they must not contain any contribution statements or `ddt` expressions. Further, because they are internal to the device, they must be initialized before any calls to the device evaluation function. If a variable is used before initialization, it is considered a so-called "memory state", also called a "hidden state", and can create difficulties in many analysis algorithms.

- *Step 4: Pick half of the unknowns as "sources".*

  Usually, some of the unknowns, *i.e.*, potentials and flows, can be written as explicit outputs on the left hand side of model equations. In Verilog-A, these unknowns are called "sources", as they are analogous to the outputs of controlled sources in SPICE subcircuits; the other type of unknowns are called "probes" (they are similar to the inputs of controlled sources). In its simplest form, a Verilog-A model calculates the values of the "sources" based on inputs from the "probes".

  If a model has n branches, there are $2n$ unknowns associated with them: n branch potentials and n branch currents. Usually, you will be able to pick n of these unknowns as "sources", *i.e.*, one for each branch. After that, the other n unknowns are considered "probes". For example, for the resistor branch in Figure 9, we can choose `I(br_res)` as the "source" and `V(br_res)` as the "probe". This way, you will need to write expressions that express the `I(br_res)` in terms of the other available non-"source" variables. For the resistor, this can be simply achieved by writing "`V(br_res)/R`", then contributing this term to `I(br_res)`. Similarly, you can also choose "`V(br_res)`" as "source" and contribute "`R*I(br_res)`" to it. But `V(br_res)` and `I(br_res)` cannot both be "sources" or both be "probes" simultaneously.

  Similarly, for the three-terminal component, one can choose `I(br_BC)` and `I(br_BE)` as the "sources" for `br_BC` and `br_BE` branches, respectively. When expressing the value of `I(br_BC)`, both `V(br_BC)` and `V(br_BE)` can be used. In other words, multiple "probes" can be used in a contribution statement to contribute to a single "source". This makes Verilog-A much more powerful and flexible than controlled sources in SPICE subcircuits.

  For any branch, the answer to the question of whether to choose its potential or flow as a "source" depends mainly on which of the two options is more convenient when writing the constitutive equation of the branch. However, if possible, it is recommended that the branch flow be picked as the "source", since this will result in a smaller equation system in most simulators.

- *Step 5: Write differential equations for the "sources", using "probes" and other variables.*

  After identifying the sources and probes in the model, we are left with n sources and n probes. The relationship between the unknowns of the model are specified by the model equations. To this end, sources are expressed as the explicit outputs of differential equations, in terms of the probes, parameters and intermediate variables. The general form of such differential equations can be written as in equation (4).

For instance, in order to characterize the resistor branch, we can write

```
1   I(br_res) <+ V(br_res)/R;
```

The constitutive relationships of capacitors and inductors involve differential terms, *i.e.*, the q function. In Verilog-A, the q part of the differential equation is specified using the `ddt` operator on the RHS of a contribution sign.

```
1   I(br_cap) <+ ddt(C * V(br_cap));
2   V(br_ind) <+ ddt(L * I(br_ind);
```

Using contribution statements and `ddt` operators, differential equations can easily be expressed in Verilog-A. To do this, one has to first identify the f and q parts of the constitutive equation of a branch and assign them to its source variable.

For the model to work well in simulation, the f and q functions must be both continuous and smooth. Common numerical problems, such as division by zero, numerical overflow and underflow must be avoided.

Putting everything together, for the toy-example device in Fig. 9, we list its Verilog-A code in Listing 18. The five steps laid out above can be summarized as follows: First, we determine the topology of the model, including nodes (terminals and internal nodes) and n branches. Then we pick n unknowns, one for each branch, as sources, and use the other unknowns (probes) and parameters to write well-behaved differential equations to express the sources. By following these steps, model developers can avoid many common mistakes in Verilog-A models, and have a greater chance of developing robust device models.

While the above steps apply to writing Verilog-A models for most devices, there can be exceptions. First, when writing equations for the internal three-terminal BJT component, what if we needed to use the voltage between node `C` and `E`? Experienced developers would often directly use `V(C, E)` in their code. Does this break the node-branch sub-circuit topology we have been assuming in the above steps?

If `V(C, E)` is used in the model, it indicates that there is an unnamed branch `(C, E)`, and associated with this branch are two unknowns `V(C, E)` and `I(C, E)`. Since `V(C, E)` is used as a probe, `I(C, E)` automatically becomes a source. And by default, this source is a zero current source, *i.e.*, `I(C, E) <+ 0` is implied. The fact that model developers can directly use the expression `V(C, E)` doesn't mean a device can be modeled without an explicitly declared topology. It only means that the device now has a different topology from what we have declared in Listing 18, with one more branch `(C, E)`.

Listing 18: Verilog-A code for the model in Figure 9.

```
1   `include "disciplines.vams"
2   `include "constants.vams"
3
```

```
4   module RLCdioBJT(a, b, c);
5       inout a, b, c;
6       electrical a, b, c, d, e;
7
8       branch (b, a) br_dio;
9       branch (b, d) br_res;
10      branch (d, e) br_cap;
11      branch (e, d) br_ind;
12      branch (d, a) br_BC;
13      branch (d, e) br_BE;
14
15      parameter real R      = 1.0e3   from (0:inf);
16      parameter real L      = 1.0e-9  from [0:inf);
17      parameter real C      = 1.0e-6  from [0:inf);
18      parameter real IS     = 1.0e-12 from (0:inf);
19      parameter real alphaR = 0.99    from (0:1);
20      parameter real alphaF = 0.5     from (0:1);
21
22      real Id, Id_BC, Id_BE, IBC, IBE;
23
24      analog begin
25          Id    = IS*(limexp((V(br_dio)/$vt) - 1));
26          Id_BC = IS*(limexp((V(br_BC)/$vt) - 1));
27          Id_BE = IS*(limexp((V(br_BE)/$vt) - 1));
28          IBC   = Id_BC - alphaR * Id_BC;
29          IBE   = Id_BE - alphaF * Id_BE;
30
31          I(br_dio) <+ Id;
32          I(br_res) <+ V(br_res)/R;
33          I(br_cap) <+ ddt(C * V(br_cap));
34          V(br_ind) <+ ddt(L * I(br_ind));
35          I(br_BC)  <+ IBC;
36          I(br_BE)  <+ IBE;
37      end
38  endmodule
```

In fact, you can keep the topology unchanged and simply replace `V(C, E)` with `(V(br_BE)-V(br_BC))`. However, instead of using `V(C, E)` as a probe, the recommended practice is to go back to Step 2 and redeclare the branches in the model. For example, if `br_CE` is declared as a branch and `I(br_BE)` specified as a source with the contribution `I(br_CE) <+ 0` written out explicitly, the model will become much cleaner.

The steps we recommend assume that half of the unknown variables can be modeled as sources, *i.e.*, they can be written as explicit outputs on the LHS of the model equations. While this will work for most device models, some models require the use of implicit equations. In certain cases, neither the potential nor the flow of a branch can be expressed explicitly. Therefore, implicit equations are needed in order to construct the constitutive relationship of that branch. In this scenario, we can write an implicit equation in Verilog-A by assigning either the potential or the flow as a dummy source and then adding this dummy source to the RHS of the contribution along with the actual implicit equation. For instance, if the constitutive relationship for the resistor branch is given by

$$g(V_{\mathtt{br\_res}}, I_{\mathtt{br\_res}}) = 0, \tag{11}$$

in Verilog-A we can write

```
1   I(br_dummy) <+ I(br_dummy) + g(V(br_res), I(br_res));
```

where `br_dummy` can be any branch in the device except for `br_res` and the function `g` can be defined as an "`analog function`" in the Verilog-A module. Note that this way of expressing implicit equations uses the same concept of sources and probes, making the intended use clear to the Verilog-A compiler.

Another exception to the "make half of the unknowns sources and the remaining half probes" rule is that sometimes the unknowns one wants to model are neither voltages nor currents. The most likely cause of these scenario is that the model unknowns are quantities from other physical disciplines than *electrical* such as a *physical distance*, a *temperature*, *etc*. They can be modeled by first declaring nodes and branches from disciplines other than electrical and then following the same procedure for writing the model equations. However, sometimes the model can have general internal unknowns that are neither potentials nor flows. In this case, it is recommended to create dummy nodes and branches for these variables This way these variables can be treated as potentials and flows in the Verilog-A representation. This makes sure that these variables will be recognized by Verilog-A compilers as system unknowns and the model will work more robustly across different simulation platforms. Although some simulators support the use of "real" variables as unknowns, the resulting compact models will not work in most other simulators. Moreover, depending on the particular simulator, such models may only work in certain types of analyses and fail in others. For the model to work consistently across different simulators and with various analysis types, only potentials and flows should be used as system unknowns.

Note that most of the "awkwardness" that is associated with modeling implicit equations and using general internal unknowns in Verilog-A stems from the fact that Verilog-A is a high-level behavioural language for describing devices with subcircuit structures. However, we believe that using the workarounds we have discussed above, in conjunction with the "five step procedure" we have presented, even a complete beginner will be able to develop well-posed compact models in the Verilog-A language for a broad range of device types.

Appendices

A Numerical Integration and the Newton-Raphson Algorithm

In Section 4, we have discussed how a simulator takes a netlist as input and converts it to a set of DAEs. We expand that discussion with a brief explanation of how the simulator converts the circuit DAEs into an algebraic residual function and finds the zeros of this function using the Newton-Raphson (NR) algorithm.

Once the simulator generates the circuit DAEs, the next step is to *solve* them. Solving the circuit

equations means providing values for the unknown quantities in the circuit. In the case of the formulation given in equation (2), this means determining the value of v_{drain}. Depending on the way we want to analyze the circuit behavior, we would use different solution methods, *i.e.*, different *analyses*. In the end, every type of analysis uses numerical techniques to convert the circuit DAEs into a *linear equation system* which can be solved by a computer. This usually happens in two distinct steps:

(i) converting nonlinear differential equations into nonlinear algebraic equations,

(ii) converting nonlinear algebraic equations into linear algebraic equations.

We again apply these steps on the common source amplifier example given in Figure 3 for the case of transient simulation.

First, we are going to convert the differential equation in (2) into an algebraic equation. This step must involve some sort of approximation that estimates the value of the time derivative of the `q` function using the function values themselves. To this end, we first rewrite equation (2) as

$$\frac{\mathrm{d}q(x,t)}{\mathrm{d}t} + f(x,t) = 0. \tag{12}$$

To arrive at equation (12) we have simply made the substitutions

$$q = C(v_{dd} - v_{drain}(t)) \tag{13}$$

$$f = (v_{dd} - v_{drain}(t))/R - I_{ds}(v_{drain}(t), v_{gate}), \tag{14}$$

and used the independent variable $x = v_{drain}$. Now suppose we are given an initial value $x_0(t_0)$ for the independent variable x. Since the differential equation (12) describes the time evolution of the circuit variables, we should be able to find the value of those variables in a later time point $t_1 = t_0 + \Delta t$. Indeed, if we assume that Δt is "small", we can approximate the derivative in (12) with

$$\frac{\mathrm{d}q}{\mathrm{d}t} = \frac{q(x_1,t_1) - q(x_0,t_0)}{\Delta t}.$$

Substituting this approximation into (12) we obtain[16]

$$q(x_1,t_1) - q(x_0,t_0) + f(x_1,t_1)\ \Delta t = 0. \tag{15}$$

Since we know the values of t_0, t_1 and x_0, equation (15) is a *nonlinear algebraic equation* in x_1. In short, we can write (15) as

$$g(z) = 0 \tag{16}$$

with

$$g(x_1) = q(x_1,t_1) - q(x_0,t_0) + f(x_1,t_1)\ \Delta t \quad \text{and} \quad z = x_1. \tag{17}$$

Therefore, if we solve (16) we will obtain the next value of x. We can use this value to construct another equation for $x_2 = x_1 + \Delta t$ and continue in this manner to obtain the full time evolution of the circuit state.

[16]The use of $f(x_1,t_1)$ in equation (15) instead of $f(x_0,t_0)$ is significant in terms of numerical properties of the approximation which is a topic of too great a depth to be covered in this document. The interested reader is referred to [18] for a circuit simulation specific treatment.

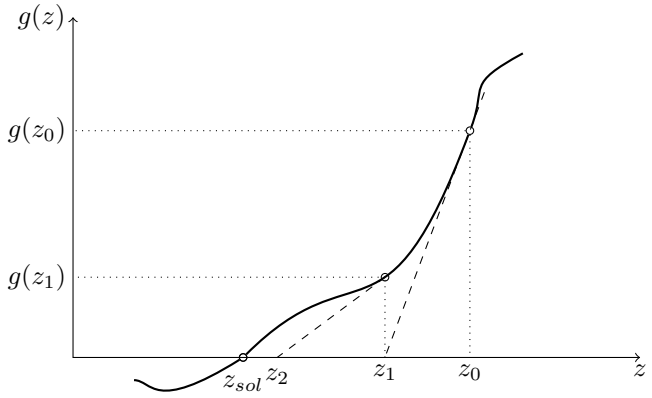

Fig. 10: Sketch of how the Newton-Raphson algorithm finds the zero of a function. The algorithm starts with an initial guess, z_0. It then approaches the solution, z_{sol}, with iterative steps $\{z_1, z_2, \ldots\}$.

The solution of (16) is obtained using the *Newton-Raphson algorithm*. The Newton-Raphson algorithm is an iterative method for the solution of general nonlinear equations. It starts with an initial guess and takes successive steps until it reaches the solution of the equation. Figure 10 shows how these steps are computed. The algorithm starts with an initial guess, z_0. It looks up where the function value lies for z_0, *i.e.*, $g(z_0)$ and "draws" the tangent line starting with the point $(z_0, g(z_0))$. The point where that tangent line crosses the z-axis becomes the next step in the algorithm, *i.e.*, z_1. The same procedure is applied to z_1 to obtain z_2. The algorithm proceeds in this manner until it reaches the final solution point z_{sol}.

Using the Newton-Raphson algorithm requires computing the derivative of the function to obtain the slopes of the tangent lines. This is why device models need to have well behaved derivatives (continuous, smooth *etc*.). Another characteristic of the Newton-Raphson algorithm is that the input values at which the device functions are evaluated are dictated by a mathematical procedure. Hence, a simulator may probe a device well beyond its normal operating point. This means, for instance, a transistor model should be able to supply well defined current outputs even if the voltage values provided for its terminals are such that they would make a real transistor smoke.

References

[1] K. Kundert and O. Zinke, *The designer's guide to Verilog-AMS*. Springer Science & Business Media, 2004.

[2] C.C. McAndrew, G.J. Coram, K.K. Gullapalli, J.R. Jones, L.W. Nagel, A.S. Roy, J. Roychowdhury, A.J. Scholten, Geert D.J. Smit, X. Wang and S. Yoshitomi, "Best Practices for Compact Modeling in Verilog-A," *IEEE J. Electron Dev. Soc.*, vol. 3, no. 5, pp. 383–396, September 2015.

[3] G. Coram, "How to (and how not to) write a compact model in Verilog-A," in *Behavioral Mod-*

eling and Simulation, 2004. BMAS 2004. Proceedings of the 2004 International Workshop on. IEEE, 2004, pp. 97–106.

[4] D. FitzPatrick and I. Miller, *Analog behavioral modeling with the Verilog-A language*. Springer Science & Business Media, 2007.

[5] T. Wang, K. Aadithya, B. Wu, J. Yao, and J. Roychowdhury, "MAPP: The Berkeley Model and Algorithm Prototyping Platform," in *Proc. IEEE CICC*, September 2015, pp. 461–464, DOI link.

[6] D. Amsallem and J. Roychowdhury, "ModSpec: An open, flexible specification framework for multi-domain device modelling," in *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*. IEEE, 2011, pp. 367–374.

[7] "Verilog-AMS Language Reference Manual Version 2.4.0," May 2014. [Online]. Available: http://www.accellera.org/images/downloads/standards/v-ams/VAMS-LRM-2-4.pdf

[8] H. Shichman and D. A. Hodges, "Modeling and simulation of insulated-gate field-effect transistor switching circuits," *IEEE Journal of Solid-State Circuits*, vol. 3, no. 3, pp. 285–289, September 1968.

[9] J. Hadamard, "Sur les problèmes aux dérivées partielles et leur signification physique," *Princeton university bulletin*, vol. 13, no. 49-52, p. 28, 1902.

[10] C. C. McAndrew, "Practical modeling for circuit simulation," *IEEE Journal of Solid-State Circuits*, vol. 33, no. 3, pp. 439–448, March 1998.

[11] K. Kundert, "Hidden state in SpectreRF," *The Designer's Guide*, 2003. [Online]. Available: http://www.designers-guide.org/analysis/hidden-state.pdf

[12] T. Wang and J. Roychowdhury, "Well-Posed Models of Memristive Devices," *arXiv:1605.04897 [cs]*, May 2016, arXiv: 1605.04897. [Online]. Available: http://arxiv.org/abs/1605.04897

[13] "Cadence Verilog-A Language Reference, Version 6.1," December 2006.

[14] Z. Jiang, S. Yu, Y. Wu, J.H. Engel, X. Guan and H.-S. P. Wong, "Verilog-A compact model for oxide-based resistive random access memory (RRAM)," in *International Conference on Simulation of Semiconductor Processes and Devices (SISPAD)*. IEEE, 2014, pp. 41–44.

[15] N. Paydavosi, T. H. Morshed, D. D. Lu, W. M. Yang, M. V. Dunga, X. J. Xi, J. He, W. Liu, M. C. Kanyu, X. Jin *et al.*, "BSIM4v4.8.0 MOSFET Model."

[16] M. A. Wahab and M. A. Alam, "A Verilog-A compact model for negative capacitance fet," Nov 2015. [Online]. Available: https://nanohub.org/publications/95/1

[17] S. Rakheja and D. Antoniadis, "MVS Nanotransistor Model (Silicon)," https://nanohub.org/publications/15, Oct 2014.

[18] K. S. Kundert, *The Designer's Guide to Spice and Spectre*. Norwell, MA, USA: Kluwer Academic Publishers, 1995.